



Delft University of Technology

## WebDSL

### Linguistic Abstractions for Web Programming

Groenewegen, D.M.

#### DOI

[10.4233/uuid:fb0cc4b7-a67b-474b-9570-96eb054a39ec](https://doi.org/10.4233/uuid:fb0cc4b7-a67b-474b-9570-96eb054a39ec)

#### Publication date

2023

#### Document Version

Final published version

#### Citation (APA)

Groenewegen, D. M. (2023). *WebDSL: Linguistic Abstractions for Web Programming*. [Dissertation (TU Delft), Delft University of Technology]. <https://doi.org/10.4233/uuid:fb0cc4b7-a67b-474b-9570-96eb054a39ec>

#### Important note

To cite this publication, please use the final published version (if applicable). Please check the document version above.

#### Copyright

Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

#### Takedown policy

Please contact us and provide details if you believe this document breaches copyrights. We will remove access to the work immediately and investigate your claim.

# WebDSL: Linguistic Abstractions for Web Programming

---

DISSERTATION

for the purpose of obtaining the degree of doctor  
at Delft University of Technology  
by the authority of the Rector Magnificus Prof.dr.ir. T.H.J.J. van der Hagen  
chair of the Board of Doctorates  
to be defended publicly on  
Friday 10 November 2023 at 12:30 o'clock

by

Danny Maria GROENEWEGEN

Master of Science in Computer Science,  
Delft University of Technology, the Netherlands  
born in Nootdorp, the Netherlands

This dissertation has been approved by the promotor.

Composition of the doctoral committee:

Rector Magnificus	chairperson
Prof.dr. A. van Deursen	Delft University of Technology, promotor
Prof.dr. S.T. Erdweg	Johannes Gutenberg University Mainz, promotor

Independent members:

Prof.dr.ir. A. Bozzon	Delft University of Technology
Prof.dr. J. Cheney	University of Edinburgh
Prof.dr. J. Vinju	Eindhoven University of Technology
Prof.dr. R. Lämmel	University of Koblenz-Landau
Dr. M.T.J. Spaan	Delft University of Technology
Prof.dr. M.M. de Weerd	Delft University of Technology, reserve member

Prof.dr. E. Visser (Delft University of Technology) was the original promotor and supervisor of this research until his untimely passing on April 5th, 2022.

The work in this dissertation has been carried out at the Delft University of Technology under the auspices of the research school IPA (Institute for Programming research and Algorithmics).



Copyright © 2023 Danny M. Groenewegen

Cover: Kijkduin beach - Photo © 2017 Danny M. Groenewegen

IPA Dissertation Series: 2023-11

Printed and bound by: Gildeprint - <https://www.gildeprint.nl/>

ISBN: 978-94-6419-976-5

# Summary

---

Information systems store and organize data, and manage business processes concerned with that data. Information systems aim to support operations, management and decision-making in organizations. Web applications are ideal for implementing information systems. Although existing web frameworks provide abstractions for creating web applications, there are three major issues with current web frameworks. Insufficient or leaky abstraction: web programming concerns are not sufficiently covered or abstractions contain accidental complexity. Lack of static verification: application faults are not removed during development. Security flaws: web application security issues are not sufficiently addressed in the framework, web programmers are exposed to many possible security faults.

How can the benefits of web frameworks be provided for web programming while avoiding the major issues of abstraction, static verification, and security? We propose a domain-specific language (DSL) solution. The challenge is to design a language that provides abstractions for all kinds of web programming tasks with the web framework issues in mind. We designed multiple sublanguages to address web programming concerns, and integrated them to form the WebDSL web programming language. WebDSL incorporates better abstraction for web programming concepts, has static checks on the application code with accurate error reporting, and automatically addresses security concerns in the code generation and runtime.

The primary concerns in web programming are user interfaces and data handling. Which features do we need from a user interface language? These features include both the rendering of data persisted in the database, as well as providing input-handling components to enter new data and update existing data. Additionally, data invariants need to be enforced by the system. How can a DSL provide these features in an integrated way? These are language-design challenges that are investigated in this dissertation. The user interface sublanguage of WebDSL contains several unique improvements compared to existing approaches: form submits that are safe from hidden data tampering; prevention of input identifier mismatch in action handlers; safe composition of input templates; automatic enforcement of Cross-Site Request Forgery protection; expressive data validation; and partial page updates without explicit JavaScript or DOM manipulation.

Access control is essential for the security and integrity of interactive web applications. Existing solutions for access control often consist of libraries or generic implementations of fixed policies. These rarely have clear interfacing capabilities, and they require manual extension and integration with the application code, which is error-prone. WebDSL provides a declarative access control sublanguage, which is entirely integrated with other language components and automatically weaves checks into the application code. Errors

related to inconsistent application of access control checks are avoided. The access control language shows that various policies can be expressed with simple constraints, allowing concise and transparent mechanisms to be constructed.

Our work on abstractions for web programming resulted in several scientific and software contributions: The design and implementation of a linguistically integrated domain-specific language for web programming that combines abstractions for web programming concerns covering transparent persistence, user interfaces, data validation, access control, and internal site search. Sub-languages for the various concerns are integrated through static verification to prevent inconsistencies, with immediate feedback in the integrated development environment (IDE) and error messages in terms of domain concepts. WebDSL is the largest programming language created with the Stratego program transformation language and the Spoofox language workbench, in which the DSL compiler and IDE have been iteratively developed. This iterative development is a recurring pattern of discovering new abstractions, domain-specific language abstraction, and reimplementing using new core abstractions tailored to the language. To validate WebDSL, we have created several real-world applications in the domain of research and education for external clients.

In our research we aim to create solutions for problems in web engineering and language engineering by developing concepts, methods, techniques, and tools. We aim to create more than just prototypes by continuing maintenance and development beyond the proof of concept. For over 10 years, we have developed WebDSL, and created and operated practical applications for external clients. For example, EvaTool is a course evaluation application that supports processes for analyzing student feedback by lecturers and other staff. WebLab is an online learning management system with a focus on programming education (students complete programming assignments in the browser), with support for lab work and digital exams, used in dozens of courses at TU Delft. Conf Researchr is a domain-specific content management system for creating and hosting integrated websites for conferences with multiple co-located events, used by all ACM SIGPLAN and SIGSOFT conferences. MyStudyPlanning is an application for composition of individual study plans by students and verification of those plans by the exam board, used by multiple faculties at TU Delft.

# Samenvatting

---

Informatiesystemen bewaren en organiseren data, en beheren bedrijfsprocessen die over deze data gaan. Informatiesystemen hebben als doel het ondersteunen van activiteiten, management, en besluitvorming in organisaties. Web applicaties zijn ideaal voor het implementeren van informatiesystemen. Hoewel bestaande web frameworks abstracties leveren voor het maken van web applicaties, zijn er drie grote problemen met huidige web frameworks. Onvoldoende of lekkende abstractie: web programmeerbelangen zijn onvoldoende gedekt, of abstracties bevatten accidentele complexiteit. Gebrek aan statische verificatie: applicatie fouten worden niet opgelost tijdens ontwikkeling. Beveiligingsfouten: web applicatie beveiligingsproblemen worden onvoldoende verholpen in het framework, web programmeurs worden blootgesteld aan te veel mogelijke beveiligingsfouten.

Hoe kunnen de voordelen van web frameworks gebruikt worden voor web programmeren terwijl de problemen met abstractie, statische verificatie, en beveiliging worden voorkomen? Wij stellen een domein-specifieke taal (DSL) oplossing voor. De uitdaging is om een taal te ontwerpen die abstracties geeft voor allerlei web programmeertaken met de problemen van web frameworks in gedachten. We hebben meerdere subtalen ontworpen voor web programmeerbelangen, en hebben deze geïntegreerd in de WebDSL web programmeertaal. WebDSL integreert betere abstractie voor web programmeerconcepten, heeft statische controles op de applicatie code met precieze foutmeldingen, en neemt automatisch beveiligingsbelangen mee in de code generatie en runtime.

De voornaamste belangen in web programmeren zijn user interfaces en behandeling van data. Welke features hebben we nodig voor een user interface taal? Deze features omvatten het tonen van data die opgeslagen is in de database, maar ook invoercomponenten voor de mogelijkheid om nieuwe data in te voeren of bestaande data aan te passen. Daarnaast moeten data invarianten beschermd worden door het systeem. Hoe kan een DSL deze features geven op een geïntegreerde manier? Dit zijn taalontwerp uitdagingen die in deze dissertatie worden verkend. De user interface subtaal van WebDSL bevat meerdere unieke verbeteringen ten opzichte van bestaande aanpakken: formulier invoer die beveiligd is tegen manipulatie van verborgen data; voorkomen van invoer identificatie problemen in actie afhandeling; veilige compositie van invoer templates; automatische handhaving van Cross-Site Request Forgery beveiliging; expressieve data validatie; en gedeeltelijke pagina updates zonder expliciete JavaScript of DOM manipulatie.

Access control is essentieel voor de beveiliging en integriteit van interactieve web applicaties. Bestaande oplossingen voor access control bestaan vaak uit libraries of generieke implementaties van rigide policies. Deze hebben zelden duidelijke functionaliteit voor koppelingen, en vereisen manuele uitbreiding en integratie met de applicatie code, wat foutgevoelig is. WebDSL biedt een

declaratieve access control subtaal, die volledig geïntegreerd is met andere taalcomponenten en automatisch de checks in de applicatie code weeft. Fouten gerelateerd aan inconsistente toepassing van access control checks worden voorkomen. De access control taal laat zien dat verscheidene policies uitgedrukt kunnen worden met simpele constraints, die het mogelijk maken beknopte en transparante mechanismen te construeren.

Ons werk aan abstracties voor web programmeren heeft verschillende wetenschappelijke en software bijdragen geleverd: Ontwerp en ontwikkeling van een linguïstische geïntegreerde domein-specifieke taal voor web programmeren die de belangen combineert voor transparante persistentie, user interfaces, data validatie, access control, en interne zoekfuncties. Subtalen voor de verschillende belangen zijn geïntegreerd door statische verificatie om inconsistenties te voorkomen, met directe feedback in de ontwikkelomgeving en rapportage van problemen in termen van domeinconcepten. WebDSL is de grootste programmeertaal die ontwikkeld is met de Stratego programma transformatie taal en de Spoofox language workbench, waarin de DSL compiler en ontwikkelomgeving iteratief ontwikkeld zijn. Deze iteratieve ontwikkeling is een terugkerend patroon van ontdekken van nieuwe abstracties, domein-specifieke taal abstractie, en herimplementatie via nieuwe kern abstracties op maat gemaakt voor de taal. Om WebDSL te valideren, hebben we verschillende realistische applicaties gemaakt in het domein van onderzoek en onderwijs voor externe klanten.

In ons onderzoek hebben we als doel om oplossingen te creëren voor problemen in web applicatie ontwikkeling en programmeertaal ontwikkeling door het maken van concepten, methoden, technieken, en tools. We proberen meer dan alleen prototypes te maken door onderhoud en ontwikkeling verder door te zetten na een proof of concept. Al ruim 10 jaar hebben we WebDSL ontwikkeld en praktische applicaties voor externe klanten gemaakt en beheert. EvaTool is bijvoorbeeld een vakevaluatie applicatie die processen ondersteunt voor het analyseren van studentfeedback door docenten en andere medewerkers. WebLab is een online leeromgeving met een focus op programmeeronderwijs (studenten maken programmeeropgaven in de browser), met ondersteuning voor practica en digitale tentamens, dat gebruikt wordt bij tientallen vakken van TU Delft. Conf Researchr is een domein-specifiek informatiebeheersysteem voor het maken en hosten van geïntegreerde websites voor conferenties met meerdere co-located evenementen, in gebruik door alle SIGPLAN en SIGSOFT conferenties. MyStudyPlanning is een applicatie voor het samenstellen van individuele studeertrajecten door studenten, en controle van deze plannen door de examencommissie, in gebruik bij meerdere faculteiten van TU Delft.

# Preface

---

What is this WebDSL thing, how did I get here, and why did it take so long? While I did not learn programming at a young age, I grew up playing games on the NES and SNES game consoles. When I went to high school in 1996, we finally got our first PC at home, with a dial-up connection to the Internet. Since I used this PC mostly for playing games, I tried online games for the first time. They were pretty unplayable because of way too much lag, and most players were cheating anyway. Investigating how they cheated was my first exposure to computer program internals. I found some website explaining how to edit game files with a hex editor to gain additional capabilities. Upon further investigation, I discovered memory editing tools, which enabled searching and changing values in the memory of the running process, which involved learning a little bit about data structures and their memory layout. I found it intriguing how game state was managed on both my own PC and the game server, and fun to discover what happens when they are not synchronized.

At TU Delft I received an introduction to programming with Java, and in my free time I learned PHP and tried to recreate a persistent browser-based game our group of friends liked playing. This was my first experience with web application programming. When it was time to do our BSc end project in 2005, Joost Heijkoop, my lab partner for all the university assignments, knew a small refurbishment company that needed to upgrade their inventory management system. We thought this project would be doable in the 3 months allocated for it. Since we wanted to avoid installation hassle and did not mind learning another programming language, we decided to implement this system in C++, which seemed the best technology choice at the time given our limited experience. Although it looked like a basic administrative application on paper, we went a few months overtime to work out all details and finish the implementation. In particular, choosing a Graphical User Interface (GUI) framework was hard, as there were many options with varying levels of maturity and tool support. In the end, we built a tiny executable that did not require any installation and ran on every Windows machine, connecting directly to a MySQL server on the local network. The application is still in use at the company today, although a different technology stack would probably have been better for maintenance.

My observation from various projects at this point was that many web applications had usability and security issues, whether it was leaking information through direct URL access, view state leaking between tabs, or short server-side session timeouts with insufficient detection at the client (which is still the case in 2023 for the TU Delft webmail client which redirected me back to login and dropped my email's text when I pressed send). I also experienced that building information systems is tedious, regardless of the programming language used. You would either end up using the minimum number of



libraries (writing data model classes to generate queries and wasting time on deciding what primary key to use for representing relations between objects in the database), or you would choose a persistence library, and end up digging through documentation because you wanted to do something slightly different than the library authors expected, and get confusing errors. Similarly, for the user interface you would pick a GUI framework or web framework, and as soon as it did not do exactly what you needed, you had to dive into the implementation, learn the implementation patterns and figure out how to patch them to do what you wanted (wishing you had just done it from scratch).

I met Eelco Visser in 2006. I was looking for a MSc thesis project and visited several professors to discuss potential projects. Eelco, just having started at TU Delft, immediately stood out, because he seemed mostly interested in taking pictures of the sunset above Delft, from the view of his office in the high-rise EEMCS building. It soon became clear to me that he had a long history of research projects behind him already, which he continued to build upon. He showed me the first fragments of WebDSL, an improved programming language for web applications, which he had started working on. I appreciated how SDF enabled you to specify syntax freely without immediately having to think about algorithm details like left recursion conflicts. Stratego expressed abstract syntax tree node transformation and construction very cleanly, using pattern matching and concrete object syntax, which enabled describing transformation rules based on readable WebDSL and Java code. This was a big contrast to the Compiler Construction course I followed earlier in the MSc curriculum, which was focused on low-level aspects and more of an exercise in C programming, which prevented me from understanding the concepts well. The WebDSL language idea also appealed to me, as it seemed to address many of the frustrations I had experienced building information systems and web applications.

Eelco was inspired by discussions with fellow researcher William Cook to look into modeling access control. My MSc thesis project became an investigation of what kind of access control policies are out there, how they are implemented in applications, and how we can make the WebDSL language support this in a novel way. Eelco encouraged me to write a paper together about this work, which was accepted at the ICWE'08 conference held in New York, and won the best paper award. This was both my first time attending a conference and my first time flying. I trusted Eelco's traveling experience, and he selected the perfect seats. These turned out to be the worst seats of the plane, next to the toilet, with no leg space from a protruding emergency exit, and right next to airconditioning. We joked about it afterwards, and I never trusted his seat-picking skills again. Over dinner we talked about continuing working together, and the topic of my PhD project became: to extend the WebDSL language and reimplement the runtime to make it more efficient and reliable.

At the start of my PhD, the WebDSL runtime was hardly practically applicable, as requesting simple pages took a full second and sometimes gave errors, it would write gigabytes of log files per hour without having any actual users,

and the servlet container would crash often after a day. I started working on an entirely new runtime, replacing all the existing web framework code. During my PhD time I continued working on the new runtime as well as adding new language features, and supervising MSc students that worked on various aspects of WebDSL (see Section 1.11 for more details on the supervised projects). Eventually, I was maintaining all parts of the WebDSL system, such as the build files, front-end static analysis, and IDE. I also took over server management from the NixOS team when their Buildfarm project funding ended. I felt that I needed to master all aspects of WebDSL and application deployment to be able to diagnose any problem that came up affecting the reliability of the applications.

Around 2011, Eelco and I were considering starting a company to develop WebDSL applications. Through a colleague I explored creating software for the healthcare domain in Sudan. Although it was quite an experience going there, and somehow I was able to demonstrate the application on a severely outdated computer and download the required software through embargo blockages, I also realized that this was never going to work due to our lack of domain knowledge and the distance we had to the clients. Discussing this with Eelco, we came to the conclusion that we would be more effective in domains we already knew: education and academia. Our focus shifted to developing the tools discussed in Chapter 7 of this dissertation: WebLab, Conf Researchr, MyStudyPlanning, and EvaTool.

The best stress tests for WebDSL were the exams in the WebLab online learning environment with hundreds of students simultaneously doing programming assignments. I remember a few exams in the first years of WebLab that did not go smoothly, where either the web application stalled due to being overloaded or some bug in the runtime or application code, or a guest program going rogue in the back-end code runner without getting detected correctly. Elmer van Chastelet and I would be constantly monitoring the server and intervening where necessary during those initial exams. Besides issues in our own software, there were other issues we had less control over, such as university networking issues or software package installations on the lab computers. Eventually, we addressed the major problems and the exams started going more smoothly.

Eelco stated in his own PhD dissertation: “one of the great contributors to this thesis is the deadline” [Visser 1997]. One of the great contributors to the delay of mine was the lack of a deadline. By continuing my work on extending and maintaining WebDSL and its applications, it never felt done. Whenever I made some progress on the dissertation draft, I identified some limitation in WebDSL and thought to myself: “I can write about this limitation or just fix it”. Additionally, deciding on an evaluation strategy for the dissertation was hard. Examples that fit in a paper do not really say much about the practical usage of the language for creating real-world applications. As we gained more application users over time, at some point it became hard to even find time to work on WebDSL itself. WebDSL was stable enough and application features and support always seemed more urgent. Daniël Pelsmaeker and

Max de Krieger joined our team and took on some of our workload. During the COVID-19 pandemic lockdown we initially spent some time adjusting our applications to cope with the new reality. We improved WebLab to run online exams from home, with plagiarism scanning, randomized question variants, and timed questions. We extended the Conf Researchr conference management application with support for virtual conferences with mirrored sessions, such that virtual attendees in other timezones can still catch a talk at an acceptable time. The number of support requests for our applications reduced during this period as well, because study program and course managers avoided making large changes. I taught a course on Web Programming Languages for the first time in 2021, which was taking up quite some time. At the end of 2021, during the next lockdown, I finally started working on completing the dissertation draft.

In March 2022 I handed in my dissertation draft to Eelco. We had our last Zoom session two weeks before his untimely passing. We talked about the dissertation and looked forward to celebrating the defense together. He also had another application idea that he wanted to explore, and an hour later we were in a conference call with potential academic workflow clients in the US. Later that year, Arie van Deursen and Sebastian Erdweg kindly took over the task of being my promotors, provided detailed feedback on the dissertation draft, and assisted with the remaining steps towards the defense.

## Acknowledgements

The WebDSL project, our academic workflow applications, and this dissertation would not have existed without Eelco Visser. Eelco's vision and strong drive inspired me to work hard and get impactful results. Without him, I probably would not have thought about doing a PhD, and missed out on expanding my view of the world through traveling and getting to know a diverse group of colleagues. When he passed away, I lost not only my mentor, but also a friend who looked out for me and had my back.

My current promotors Arie van Deursen and Sebastian Erdweg helped me get to the finish line. Arie invited Eelco to come to his research group in Delft in 2006, and introduced me to Eelco when I was searching for MSc projects. Although Arie has been head of my department for a long time, I had not worked directly with him before. I appreciate his ability to pinpoint the strong and weak points in the dissertation draft with great clarity, based on many years of supervision experience. Sebastian Erdweg worked in our research group as an assistant professor from 2016 to 2019. After I started developing MyStudyPlanning in 2016, he was the first MSc coordinator to test and use the system for approving individual study programs. Besides this collaboration, we had many interesting discussions during coffee and lunch breaks, and we often went for last-minute dinners and joined board game evenings.

The committee members took the time to review my dissertation which is not of the typical 'papers stapled together' form: Alessandro Bozzon, James Cheney, Jurgen Vinju, Ralf Lämmel, Matthijs Spaan, and Mathijs de Weerd.

The members of my Academic Workflow Engineering (AWE) team have made the real-world impact possible: Elmer van Chastelet, Daniël Pelsmaeker, and Max de Krieger. We work together on designing and implementing the applications, assisting WebLab exams, and handling all the support issues that come along. Elmer is the most prolific WebDSL application programmer in the world. He is the primary developer of our largest applications, the Conf Researchr and WebLab systems. Daniël focuses on WebLab and maintains the Docker-based backend, which has been instrumental in getting WebLab adopted by more course creators. Max develops features for all our applications, and takes care of many support requests. I value their work and our friendships greatly.

During my initial PhD study years, I enjoyed working together and hanging out with Zef Hemel, Lennart Kats, Sander Vermolen, Maartje de Jonge, Sander van der Burg, Eelco Dolstra, Rob Vermaas, and with visiting researchers Karl Trygve Kalleberg and Nicolas Pierron. Back then, IRC channels were used heavily for work discussions, jokes, and coffee break calls. I shared an office with Zef and learned a lot from him about programming and writing. I try to keep in contact with them occasionally online, and I met Lennart and Maartje several times in person over the years.

In 2012, our research group attended a summer school in Romania. Vlad Vergu invited Gabriël Konat, Guido Wachsmuth, and me for a week-long road trip in the Carpathian Mountains. He showed us some beautiful sceneries and villages. I will never forget his efficient driving style, which even ambulances could not keep up with.

I had many interesting conversations with Luís Eduardo de Souza Amorim (Eduardo) about cultural differences and relationships. We also experienced several late nights going out for drinks in the weekend.

I shared an office with Daco Harkes for a while, where we collaborated on applying the IceDust language to WebDSL applications. Over coffee we had many good discussions about work-life balance, and career options for MSc and PhD graduates in academia and industry.

There are many past and current colleagues in the Programming Languages group that I haven't mentioned yet, who have made working there an enjoyable experience: Jeff Smits, Casper Bach Poulsen, Jesper Cockx, Sven Keidel, Jasper Denkers, Arjen Rouvoet, Hendrik van Antwerpen, Peter Mosses, Robbert Krebbers, Benedikt Ahrens, Alexander Chichigin, Bohdan Liesnikov, Aron Zwaan, Cas van der Rest, Dennis Sprokholt, Lucas Escot, Luka Miljak, Arjan Mooij, Soham Chakraborty, Xulei Liu, Jaro Reinders, Kobe Wullaert, Michael Steindorfer, Jules Jacobs, Paolo Giarrusso, Augusto Passalacqua, and Pierre Néron. Additionally, I want to mention some of the MSc students that I supervised or had nice interactions with: Michel Weststrate, Chris Gersen, Nathan Bruning, Chris Melman, Jesse Tilro, Sverre Rabbelier, Martijn Dwars, Nick ten Veen, Jonathan Dönszelmann, Maarten Sijm, Taico Aerts, Chiel Bruin, Bram Crielaard, Olaf Maas, Wiebe van Geest, Oskar van Rest, Ivo Wilms, André Vieira, Ricky Lindeman, Ruben Verhaaf, Jippe Holwerda, and Wouter Mouw. Since 2022, our AWE team joined forces with the Computer

Science & Engineering Teaching Team, which had been using WebLab for many years in course labs and exams: Andy Zaidman, Christoph Lofi, Otto Visser, Stefan Hugtenburg, Taico Aerts, Thomas Overklift, Frank Mulder, Bart Gerritsen, Amira Elnouty, Ivo van Kreveld, Ruben Backx, Mathijs Molenaar, Elena Congeduti, Chivany van der Werff, Azqa Nadeem, Christos Koutras, and Ivar de Bruin. Our secretaries, Roniet Sharabi, Marsha Ginsberg, and Shelly Dawn Stok have been very kind and helpful. In particular they have helped our AWE team in dealing with everything related to invoicing for our applications.

Since my first year of university in 2002, I have played Magic the Gathering regularly with a group of fellow students: Ruben Wieman, Boaz Pat-El, Joost Heijkoop, Daniël van Gelderen, Michel Deconinck, Reinier Zwitserloot, and Martijn Bleeker. At various points in time this grew into other fun activities like computer games, board games, building a pinball machine from scratch (Ruben is a pinball fanatic), and cinema nights. I have often invited and brought along colleagues and students to our game evenings. I still play weekly with Ruben, Gabriël Konat, Daniël Pelsmaeker, and Jeff Smits.

Ronald van der Kraan has been my best friend ever since elementary school. We frequently have long chats in which we exchange life updates and talk about progress or frustrations at work. He is an intelligent self-taught self-employed programmer who keeps up with interesting technical discussions and provides invaluable feedback. Discussions with him helped me put things into perspective and get clearer views.

My sister Ramona has provided me with support over many years. She has been looking forward to the defense day, and is excited to be one of my paranymphs.

My parents always encouraged me to stay in school as long as possible. My father was young when he took over his father's agricultural land and started working. He would have been proud to experience the defense, just like my mother is now. In our summer holidays, we would go on fishing trips on the North Sea. Sometimes we went several consecutive days, and each day he would invite different friends to come along. Spending hours floating on a small boat with no coast in sight was sometimes very quiet and relaxing, and other times led to interesting conversations about the things that really mattered to people. After he passed, I continued this ritual my father established for a while. I invited friends to come along on several occasions: Vlad, Elmer, Reinier, Daniël and Eduardo, and Sven and Nick. This activity inspired the cover choice for this dissertation.

Meeting my partner Mozhan in 2017 had a huge impact on my my life and made it more enjoyable and meaningful. Although she comes from a different part of the world, our values and goals in life align perfectly. She encouraged and supported me to finally finish this dissertation. Thank you for being the love of my life.

Danny Groenewegen  
October 23, 2023  
The Hague

# Contents

---

<b>Summary</b>	<b>iii</b>
<b>Samenvatting</b>	<b>v</b>
<b>Preface</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Web Information Systems . . . . .	1
1.2 Problems in Web Programming . . . . .	2
1.3 Insufficient or Leaky Abstraction . . . . .	3
1.3.1 Design Coverage . . . . .	3
1.3.2 Design Fragmentation . . . . .	4
1.3.3 Forced MVC structure . . . . .	6
1.4 Lack of Static Verification . . . . .	6
1.4.1 Linguistic Separation . . . . .	6
1.4.2 Typical Verification Problems . . . . .	6
1.4.3 Error Reporting Quality . . . . .	7
1.4.4 Statically and Dynamically Typed Languages . . . . .	8
1.4.5 Verification Related Work . . . . .	8
1.5 Security Flaws . . . . .	8
1.5.1 Web Application Security Vulnerabilities . . . . .	9
1.5.2 SQL Injection . . . . .	10
1.5.3 Cross-Site Scripting . . . . .	12
1.5.4 Cross-Site Request Forgeries . . . . .	13
1.5.5 Reflection and Run-Time Code Manipulation . . . . .	13
1.6 Thesis . . . . .	14
1.7 WebDSL Design Principles . . . . .	16
1.8 Research Methodology . . . . .	17
1.9 Contributions . . . . .	19
1.10 Structure of this Dissertation . . . . .	21
1.11 Origin of Chapters . . . . .	23
<b>2 The WebDSL Web Programming Language</b>	<b>27</b>
2.1 Introduction . . . . .	27
2.2 Language Concepts . . . . .	28
2.3 Example Application . . . . .	29
2.3.1 Data Model . . . . .	30
2.3.2 Web Interface . . . . .	30
2.4 Transparent Data Persistence . . . . .	32
2.4.1 Memory and Storage . . . . .	32
2.4.2 Entity Objects . . . . .	34

2.4.3	Associations . . . . .	36
2.4.4	Object Identity . . . . .	37
2.4.5	Inheritance and Polymorphism . . . . .	38
2.4.6	Invariants . . . . .	39
2.4.7	Discussion Data Persistence . . . . .	39
2.5	Expressions and Functions . . . . .	41
2.5.1	Functions . . . . .	43
2.5.2	Safe Query Embedding . . . . .	45
2.5.3	Java Interoperability . . . . .	46
2.5.4	Discussion Functions . . . . .	46
2.6	Pages and Templates . . . . .	49
2.6.1	Pages and Navigation . . . . .	50
2.6.2	Templates . . . . .	50
2.6.3	Template Calls . . . . .	54
2.6.4	Template Content . . . . .	56
2.7	User Interface Input . . . . .	57
2.7.1	Forms and Input . . . . .	60
2.8	User Management and Access Control . . . . .	60
2.9	Search . . . . .	63
2.10	Discussion . . . . .	64
2.11	Conclusion . . . . .	66
<b>3</b>	<b>User Interface Templates</b>	<b>67</b>
3.1	Introduction . . . . .	67
3.2	Design Goals . . . . .	68
3.3	Request Lifecycle . . . . .	69
3.4	User Input . . . . .	71
3.4.1	Multi-phase Evaluation . . . . .	72
3.5	Language Primitives for Input Implementation . . . . .	72
3.5.1	Phase Function Code . . . . .	72
3.5.2	Ref Types . . . . .	74
3.5.3	Tempate Identifier Generation . . . . .	76
3.6	Data Validation . . . . .	77
3.6.1	Value Well-Formedness . . . . .	78
3.6.2	Data Invariants . . . . .	79
3.6.3	Form Input Validation . . . . .	81
3.6.4	Function Assertions . . . . .	81
3.6.5	Messages . . . . .	82
3.6.6	Validation Phase in Request Lifecycle . . . . .	83
3.6.7	Library Input With Validation . . . . .	83
3.7	Partial Page Updates . . . . .	85
3.7.1	Nested Page . . . . .	87
3.7.2	Inline Refresh . . . . .	90
3.7.3	Inputs with Immediate Validation . . . . .	91
3.8	Conclusion . . . . .	93

<b>4</b>	<b>Access Control</b>	<b>95</b>
4.1	Introduction . . . . .	95
4.2	Access Control . . . . .	97
4.2.1	Authentication . . . . .	98
4.2.2	Restricting Access . . . . .	100
4.2.3	Administration . . . . .	102
4.3	Access Control Policies . . . . .	105
4.3.1	Mandatory Access Control . . . . .	106
4.3.2	Discretionary Access Control . . . . .	107
4.3.3	Role-Based Access Control . . . . .	109
4.4	Transformational Semantics . . . . .	111
4.4.1	Policy Normalization . . . . .	111
4.4.2	Rule Weaving . . . . .	112
4.5	Related Work . . . . .	114
4.5.1	Language Design . . . . .	114
4.5.2	Policy Languages . . . . .	115
4.5.3	Frameworks . . . . .	116
4.6	Discussion . . . . .	117
4.6.1	Future Work . . . . .	118
4.7	Conclusion . . . . .	119
<b>5</b>	<b>The WebDSL Language Evolution Pattern</b>	<b>121</b>
5.1	Introduction . . . . .	121
5.2	A Generic Language Evolution Pattern . . . . .	124
5.2.1	Evolution of User Interface Templates . . . . .	126
5.2.2	Evolution of Persisted Data Models . . . . .	129
5.2.3	Evolution of Static Code Template Expansion . . . . .	130
5.3	Evolving Compiler Extensions . . . . .	132
5.3.1	Evolution of Email Notifications . . . . .	134
5.3.2	Evolution of Files and Images . . . . .	134
5.3.3	Evolution of Internal Site Search . . . . .	136
5.4	Discussion . . . . .	137
5.5	Conclusion . . . . .	139
<b>6</b>	<b>The WebDSL Compiler, IDE, and Runtime</b>	<b>141</b>
6.1	Introduction . . . . .	141
6.2	Compiler Pipeline . . . . .	142
6.3	Front-End Analysis . . . . .	143
6.4	Front-End Transformations . . . . .	147
6.5	IDE support . . . . .	148
6.5.1	IDE Caching . . . . .	151
6.6	Back-End . . . . .	152
6.6.1	Code Generation . . . . .	152
6.6.2	WebDSL Request Lifecycle . . . . .	154
6.6.3	Runtime System . . . . .	156
6.7	Compiler Caching Strategies . . . . .	156



6.7.1	Code Generation Cache . . . . .	159
6.7.2	Compile Unit Cache . . . . .	159
6.8	Application Deployment . . . . .	160
6.9	Discussion . . . . .	160
6.10	Conclusion . . . . .	162
<b>7</b>	<b>WebDSL in Practice</b>	<b>163</b>
7.1	Introduction . . . . .	163
7.2	EvaTool . . . . .	164
7.3	WebLab . . . . .	167
7.4	Conf Researchr . . . . .	170
7.5	MyStudyPlanning . . . . .	173
7.6	Robustness Engineering Experiences . . . . .	175
7.7	Performance Engineering Experiences . . . . .	177
7.8	Security Engineering Experiences . . . . .	178
7.9	Reflections on Experiences . . . . .	180
7.10	Threats to Validity . . . . .	186
7.11	Conclusion . . . . .	187
<b>8</b>	<b>Related Work</b>	<b>189</b>
8.1	Introduction . . . . .	189
8.2	Conventional Full-Stack Web Frameworks . . . . .	189
8.2.1	Django . . . . .	190
8.3	Multi-tier Web Programming Languages . . . . .	195
8.3.1	Ur/Web . . . . .	196
8.3.2	Links . . . . .	201
8.3.3	Hop.js . . . . .	205
8.4	Modeling and Low-Code Tools . . . . .	206
8.4.1	WebML . . . . .	207
8.5	Conclusion . . . . .	207
<b>9</b>	<b>Conclusion</b>	<b>209</b>
9.1	Thesis Revisited . . . . .	209
9.2	Design Principles Revisited . . . . .	210
9.3	Directions for Future Work . . . . .	214
	<b>Bibliography</b>	<b>217</b>
	<b>Curriculum Vitae</b>	<b>229</b>
	<b>List of Publications</b>	<b>231</b>
	<b>Titles in the IPA Dissertation Series since 2020</b>	<b>233</b>

# Introduction

---

# 1

My thesis: **New web programming abstractions integrated in a domain-specific language improve web programming by avoiding boilerplate code, providing timely and accurate feedback on problems in application source code, and ensuring reliability (robustness, performance, scalability, and security) of applications. The design and implementation of such a language is feasible and has practical applicability.**

## 1.1 Web Information Systems

Information systems store and organize data, and manage business processes concerned with that data. Information systems aim to support operations, management and decision-making in organizations. For example, in a university context, information systems are used to track student progress, store grades, manage individual study program selection, and create overviews for university staff. The study program and course selection workflow we encountered at our university grew organically. Initially, students handed in paper forms with physical signatures, which the university staff transferred into spreadsheets saved in a shared drive. This process was very error-prone, forms could get lost, duplicate entries could be made in the spreadsheets, errors occurred in the data due to bad handwriting. Additionally, it took a long time because of print-sign-scan loops, it costs a lot due to the staff work involved, and it is bad for the environment because of printing. It makes sense to convert such workflows into a digital system. However, implementing tailormade software for information system workflows is not trivial. It is easy to underestimate the amount of work required for getting close to a user-friendly, intuitive, and secure system, which is as error-free as can be. And when it starts getting used, it requires continuous maintenance of the code and deployment to keep running and handle new requirements.

Web applications are ideal for implementing information systems. Web applications organize and persist all data in a database, protect data by only allowing specific operations, do not require installation on client computers, work on all operating systems and devices, and can be upgraded without interruption. Unfortunately, web application development is complex due to its heterogeneous nature. It involves multiple programming languages with their own programming models (e.g. DOM updates with client-side JavaScript, server-side Java code to execute operations, SQL database queries), and separate software systems in a network (browser, proxy server, application server, database server) that all need to work together. Additionally, there are non-functional requirements inherent to the web platform such as protecting against request tampering and injection attacks.

## 1.2 Problems in Web Programming

The abundance of web frameworks in existence (e.g. Spring [2023] framework for Java, Django [2023] framework for Python, Ruby on Rails [2023], Laravel [2023] for PHP, Sails [2023] for Node.js) indicates a need for abstraction when programming web applications. There are many duplicated features in web applications, such as rendering data, data persistence, authentication, access control, and styling. Web frameworks assist programmers in organizing the complexity of web programming by enforcing standard patterns. The frameworks provide facilities for commonly used features that would otherwise be duplicated. Common design patterns are captured in the semantics of a framework, allowing application writers to focus on the specifics of their application.

Although existing web frameworks provide abstractions for creating web applications, there are several problems that remain. Among these problems are having to write boilerplate code to glue together components, late integration checks between framework components, and weak IDE support for framework concepts [Hemel et al. 2011; Hemel 2012]. The problems are partially caused by the lack of collaboration between the programming language design and the framework design. Frameworks are written in general-purpose programming languages like Java, C#, Python, and JavaScript. General-purpose programming languages can have features that are not necessary for typical web application code. These features can actually introduce abstraction problems and vulnerabilities when the language is used for the purpose of web applications. For example, where a typical framework provides convenient ways to escape values in queries, query injection attacks can be prevented in a safer way if the programming language is made aware of queries, because the developer can forget about the problem entirely (see Section 1.5.2). The problems we identify in web programming frameworks are:

1. *Insufficient or Leaky Abstraction* When abstraction is too narrow, web programming concerns require encoding into low-level commands, leading to error-prone boilerplate code. When abstractions are leaky, underlying complexity shimmers through the abstraction.
2. *Lack of Static Verification* Application faults are not discovered during development. Problems become clear at a later point when deploying or running the application, which means they can be missed entirely or solving them becomes more costly than if they were discovered while writing the code.
3. *Security Flaws* Web application security issues are not sufficiently addressed in the abstractions, web programmers are exposed to many possible security faults.

## 1.3 Insufficient or Leaky Abstraction

There are several problems that can be classified as either insufficient or leaky abstraction. Insufficient abstraction means that application concerns require a lot of work to implement in the programming language. It requires encoding a concern into low-level commands, leading to error-prone boilerplate code. A leaky abstraction is an abstraction that contains a detail where the underlying complexity is exposed. Whether an abstraction is leaky or not is not an absolute measure. The law of leaky abstractions as coined by Joel Spolsky [2002] states “All non-trivial abstractions, to some degree, are leaky”. Although web programming abstractions are also non-trivial and leaky to some degree, we can identify unnecessary leaks and fix them.

### 1.3.1 Design Coverage

Web frameworks provide abstractions for common technical concerns in web programming. However, the provided abstractions are often insufficient to conveniently describe higher-level application concerns, such as access control policies and data validation requirements. These end up scattered throughout the source code, with custom code or third-party libraries. The framework is insufficiently aware of these concerns, other features are unaware of the intended meaning and there is no support with static analysis to avoid consistency problems. We can identify several web programming concerns for which web frameworks provide insufficient abstraction:

*Access Control* Access control governs access to the components of an application based on the user’s access rights. Access control is essential for the security and integrity of interactive web applications. It should be possible to express various access control policies clearly and concisely. An access control policy often ends up being woven manually into the application source code. Because of its crosscutting nature, access control code gets scattered across the codebase. If one check is accidentally omitted when making changes to the access control policy, the application has a security problem. It also becomes hard to inspect and verify the implemented access control policy, because it requires searching the codebase for all occurrences.

*Data Validation* Data validation prevents incorrect or inconsistent data from entering the system. Data validation should cover multiple aspects like data wellformedness, data model invariants, form validation, and general error message handling. Each of these is usually implemented as separate features in a framework, introducing unnecessary complexity.

*User Interface Templates* Template languages abstract from details related to user interface specification. However, their design is typically not integrated with the underlying general-purpose language. For example, a template language has poor encapsulation when it does simple file includes and relies on global variables to pass arguments. Template languages can have different control flow constructs than the general-purpose language, such as for branching

and conditionals. Expressions inside templates can have different syntax and semantics, as can be seen with the Expression Language in JSP and JSF [JavaEE 2023]. These small and subtle differences are accidental complexity in web programming.

*Object-Relational Mapping* Object-relational mapping solutions abstract from code related to persistence and database querying, however, they have many abstraction leaks. For example, the programmer has to configure table and column mappings. The Hibernate ORM [2023] framework for Java requires annotations on simple Java classes to determine mapping strategies. This requires an understanding of framework internals, such as master and slave columns for inverse properties.

*Client-Server Partitioning* Web applications can run partly on the client and partly on the server. Running code on the client is never secure from tampering, as the client has complete control over the execution environment. Although client code cannot be trusted, running harmless code such as user interface updates on the client can reduce load for the server. Specific features such as animations and real-time updating games actually need to run in the client, as the latency for server requests would be too large. The separation between running code on client and server is typically strict, because the client-side code is written in JavaScript, and the server code in any general-purpose language. Even in the case of Node.js [Node.js 2023], a server-side runtime environment based on JavaScript, the available APIs and programming patterns required are different between server-side and client-side components. Besides having to learn multiple languages, APIs, and programming patterns, the communication between these components is error-prone, requiring low-level marshalling of data of one format into the other. Several web programming languages have been proposed that abstract over client- and server-side code, including UR/Web [Chlipala 2010], Links [Cooper et al. 2006], and Hop [Serrano 2007]. These systems provide the web programmer with one language that can be compiled to transparently run on the client or on the server. For security-critical code, these solutions require an annotation or other marker to force code to be run on the server.

### 1.3.2 Design Fragmentation

Design fragmentation refers to general-purpose languages and web programming frameworks being separately designed. General-purpose languages have different design goals than web programming frameworks, favoring flexibility and extensibility over other concerns. Due to building on top of an existing general-purpose language, the user of a web framework has to deal with accidental complexity inherited from the underlying programming language and tooling.

For example, consider the mismatch between database identity and object identity. With database persisted data, identity of an object means the primary key field in the database. However, equality operators in general-purpose lan-

languages will compare references, or fields, or require a custom implementation for each object. For web programming it would be more convenient if there is a default that compares primary keys. Some languages even have different equality operators to add to the possible confusion, e.g. `==` in PHP will do type conversions before comparison, while `===` is comparison without conversion. Although PHP was initially designed for web programming, it does not provide reusable abstractions for current web programming concerns such as persistence, user interfaces, form handling and databinding, access control, and data validation. Consequently, PHP has more in common with general-purpose languages, serving as base language for many web programming frameworks.

The Hibernate ORM [2023] framework for Java uses proxies to provide lazy loading of objects. When an initial object is loaded from the database, fields that are references to other objects get instantiated with a generated proxy class. This proxy class is a subclass of the declared type of the field, and all its fields are null. If any of the proxy class methods are invoked, the actual object is loaded from the database. The proxy delegates method invocations to the actual object. The goal of these proxies is to provide transparent persistence, the application code does not have to be concerned about when to load an object. Because Hibernate is developed separately from the Java language, it has to resort to complex methods to achieve this transparency. Unfortunately, the implemented abstraction is leaky. Accessing fields directly on a proxy object, instead of the getter method, will return a null-value. Querying the type of a proxy with `instanceof` will report a generated subclass of the declared field type. However, the type of the actual object might be a subclass of that field type due to polymorphism. This information will not be available anymore, leading to possible application faults. The broken property access and inheritance checks are additional examples of leaky abstractions in ORM.

In a web setting with world-wide reach and internationalization concerns, string values should be in the UTF8 character set, while many tools in the chain will default to ISO-8859-1 or Latin-1. PHP's built-in string operations will not work with UTF8, and require the programmer to invoke string operations from the `mbstring` extension. MySQL's default character set is Latin-1 and requires configuration options when creating tables to get the UTF8 character set for text values. Additionally, HTML pages need `charset` options to make the browser work correctly with UTF8 values. Since web applications deal with many string values, pitfalls like these cause unnecessary debugging overhead and distract from implementing the actual features.

Project structures become complex due to boilerplate code for integration of framework and libraries. Many web applications use a similar set of libraries, however, setting up a project often requires using a project template or scaffolding. The code and libraries of such a template cannot be differentiated from custom implemented code and thus adds to the complexity of the project.

These are examples of low-level friction between a web framework abstraction and the framework's host language. Such issues are often not easy to address in a framework setting. Backwards compatibility is a major concern for

these languages, and invasive changes can cause separation in the community.

### **1.3.3 Forced MVC structure**

The Model-View-Controller [Reenskaug 2003] pattern is adopted heavily in web programming frameworks. Model contains data and business logic, view generates the user interface, and controller receives events from the user interface to interact with the model. Also the top-level code hierarchy in many web frameworks is forced to model, view, and controller separation. The problem is that this model is too simple for current web application concerns. Persistent data, functions, user interface templates can be clearly separated, but where do access control, data validation, internal site search and other concerns go? A project could be better off being structured according to the semantics of the application rather than the technical components. The model, view, and controller subdivision does not give any useful information to understand a specific application. The common Ruby on Rails advice to have “fat models, skinny controllers” [Buck 2006] also indicates a problem with the role of the controller. A thin controller is often tightly integrated with a view and is mostly boilerplate code.

## **1.4 Lack of Static Verification**

Lack of static verification is typically an undesirable consequence of linguistic separation. This means that web programming uses multiple languages that each have separate compilers and analysis tools. Inter-language consistency is a common source of faults, e.g. checking that data properties exist when they are referred to from a user interface template. The quality and timeliness of reported errors also depends on the completeness of analysis.

### **1.4.1 Linguistic Separation**

Web application platforms consist of many languages. There are common base languages in web applications, such as HTML5, CSS, and SQL. The languages are linguistically separated because the compiler and analysis tools for one language are not aware of the other languages. An application spread over multiple languages can have consistency faults which can lead to failures at run-time. While it is conceptually appealing to have different languages that address specific technical concerns, the lack of integration results in poor static verification.

### **1.4.2 Typical Verification Problems**

In web programming, the technical solution components are specific to application concerns. For example, a template language handles the user interface, and an ORM takes care of data persistence. There can be faults inside an application concern, such as a data model property with an unknown type, or

a navigation link that refers to a non-existing page. There can also be faults in the links between application concerns. The user interface renders a view based on the persisted data. If this view contains a form, that form connects through its input identifiers to a controller that handles the form request. If the connection between components is inconsistent, the application will not work correctly.

To get an overview of static verification opportunities in web programming, we list common intra- and inter-aspect faults:

- *Invalid data model references in user interface templates, application logic, data validation checks, and access control rules*, e.g. referring to a non-existing property, or a property with an unexpected type.
- *Properties of non-existing types*, the data model uses types that are not defined.
- *Invalid template references*, the use of tags and controls that do not exist.
- *Invalid HTML element nesting*, incorrectly nesting tags and controls, e.g. list items outside a list, or rows outside a table.
- *Broken page links*, links to non-existing application pages, or links that provide the wrong type of parameters.
- *Broken action links*, actions to be triggered, e.g. when clicking a submit button, do not exist or are invoked incorrectly.
- *Invalid redirect from actions*, the user is redirected to pages within the application that do not exist.
- *Form parameter mismatch*, values in a form submit are ignored due to a mismatch between rendered parameter names, and expected parameter names in the form submit action.

Early in the WebDSL development we performed a case study which analyzed the quality of verification in web programming [Hemel et al. 2011]. We used fault seeding to discover when and how web programming frameworks manifested failures. We observed that while certain frameworks report some application inconsistencies at compile-time, many are only discovered later, at deployment or run-time. Additionally, the reported error messages were often confusing and not in terms of domain concepts, e.g. generic type errors in Java or Scala to indicate several different types of consistency problems.

### 1.4.3 Error Reporting Quality

Verification errors are more useful if they are immediately visible, without the developer having to switch context from the IDE to the browser. Static analysis will report application faults as early as possible. An IDE can mark erroneous fragments while the programmer is writing the application code.



Error messages should not be leaky, it should make sense at the level of abstraction in which the programmer is developing. For example, a broken navigation link should report the problem using domain terminology such as “page” and “link” rather than “constant” or “method”.

#### **1.4.4 Statically and Dynamically Typed Languages**

Both statically and dynamically typed languages suffer from lacking static verification. In dynamically typed programming languages such as Python and Ruby, type errors and framework usage failures occur at run-time. Faults will not be caught unless the code is run. While this is not a problem during initial development, subsequent maintenance changes requires that all the code is run again to find regressions. Without a test suite with sufficient coverage, faults can easily go undetected.

In a statically typed language, guarantees the typechecker gives do not provide guarantees at the level of the framework. For example, the JavaServer Faces [Burns and Kitain 2006] components for building web interfaces with Java include an expression language that is similar, but not equivalent to expressions in Java. The Java compiler does not check definitions in this expression language for type correctness or consistency with other defined components.

#### **1.4.5 Verification Related Work**

Various solutions have been proposed in the literature to address the lack of static verification in web programming practice. Halfond et al. [Halfond and Orso 2008] describe static analysis to identify the implicit interfaces in web application components implemented in a conventional web framework, and statically verify the correctness of these invocations. Braband et al. [Brabrand, Møller, and Schwartzbach 2001] describe a dataflow analysis to determine whether an application always generates correct HTML, e.g. ensures that list items are enclosed in a list. Chlipala [Chlipala 2010] introduces a language with static consistency checks between form view components and form handlers.

Static analysis removes faults from applications that are not prevented through better abstractions. Static analysis becomes simpler when the abstractions are better, because fewer faults have to be prevented. They are both essential to improving reliability in web programming.

### **1.5 Security Flaws**

Security is related to every other aspect of web programming. Improving abstraction and static analysis both improve security. A fault can easily result in a security flaw and become a vulnerability. Besides the general complexity of building full-stack web applications, security in web programming is complex due to inherent accessibility and multi-user aspects. Online web applications are accessible to anyone with an internet connection, which includes

potential hackers. Web applications need to handle multiple users simultaneously, and correctly identify the user and provide the intended capabilities. Web application security is a broad concept. Security can mean designing authentication checks and an access control policy, modeling data validation, as well as preventing request tampering and injection attacks. Access control and data validation are specific to an application and thus require explicit specification in the application implementation. Functional security features like access control are problematic when there is insufficient abstraction in the web programming language, as discussed in Section 1.3.

The other type of security is a non-functional requirement. An application should be protected from malicious hackers. While this goes beyond just application security, as gaining access to the server can be attempted in many ways, the application itself is the most vulnerable component. For example, common vulnerabilities such as SQL injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and request tampering are all attacks on the application itself.

The risks of security flaws in web applications are severe, an attacker could potentially:

- execute arbitrary code;
- steal identity of another user;
- cause denial of service;
- access or destroy privileged data;
- deform sites;
- and redirect users to malicious phishing sites.

This section provides an overview of common web programming security vulnerabilities, in order to understand the complexity involved in web programming and the challenges for web programming solutions.

### 1.5.1 Web Application Security Vulnerabilities

The Open Web Application Security Project (OWASP) maintains a list of the top ten security risks in web applications based on analyses of thousands of applications [Open Web Application Security Project 2017]. We can group these risks into the following main categories:

- *Tier and Language separation*: Code is shipped between tiers and blindly interpreted. When user input is part of this code, there is a potential for abuse. The most prominent examples are (SQL) *Injection*, *Cross-Site Scripting (XSS)*, and also *XML External Entities (XXE)* and *Insecure Deserialization* fall into this category.

- *Authentication*: Because HTTP is a stateless protocol, state is encoded in URL and form parameters, and cookies sent with each request. The most important use of a cookie is to identify an authenticated user. There are many possible cases of *Broken Authentication*. Parameters and cookies can be tampered with by a malicious user to attempt to gain more access. Cookies that are used to decide on access can be stolen by monitoring an unsafe network channel. Passwords that are too simple can be brute force guessed. Passwords that are insufficiently encrypted could get stolen through other security flaws like XSS. A forged request in a separate browser tab can attempt to use a cookie temporarily to execute a request on behalf of a different user, which is *Cross-Site Request Forgery (CSRF)* (in the top 10 of the 2013 version of the OWASP list).
- *Access Control*: The problem of *Broken Access Control* refers to issues where the access control policy is incorrectly enforced in the code. This can easily happen when access control has to be encoded in generic code instead of being identified as an implementation concept in the source language. Besides having to deal with accidental implementation errors, it can be challenging to design the correct access control policy for a specific application. Application developers can make errors in the logic for determining access, e.g. by not considering all possible cases.
- *Maintenance and Operations*: An existing application can become insecure by incorrect configuration of the server or by additional exploits getting discovered in the technology stack. Security flaws in existing frameworks and platforms get discovered (zero-day vulnerabilities) and are made public with a security update. Existing applications then need to be updated in a timely manner or the vulnerability can easily be exploited. Security standards rise as computers get more powerful, and so do the requirements for encryption. It requires an active role to keep the server configuration up-to-date, and correctly configuring proxy servers that handle outside requests. The security risks in this category are *Sensitive Data Exposure*, *Security Misconfiguration*, *Using Components with Known Vulnerabilities*, and *Insufficient Logging & Monitoring*.

Web applications are hard to make secure, and non-functional security features require development effort by the framework or application developer. Ideally, all such non-functional security features are taken care of automatically by the web programming framework or platform, because they are not specific to an application. In the following section we show examples of the security risks SQL injection, XSS, CSRF, and Insecure Deserialization.

### 1.5.2 SQL Injection

Injection flaws can occur when untrusted user data is sent to an interpreter as part of a command or query. The most common type is SQL injection, which, if successful, enables a malicious user to read and write potentially all the data controlled by the application. Halfond et al. [Halfond, Viegas, and Orso 2006]

analyzed different types of SQL query injection attacks, and categorized them in tautologies, illegal/logically incorrect queries, union query, piggy-backed queries, stored procedures, inference, and alternate encodings.

To give an example, a tautology-based attack injects code in a SQL where-clause so that it always evaluates to true. An unsafe login query might be implemented as checking whether the following returns any result:

```
query = "select * from users where username='" + username + "' and pass='" + password + "';"
```

A malicious hacker could enter the following username:

```
' or 1=1 --
```

Then, the executed query becomes:

```
select * from users where username='' or 1=1 --' and pass=''
```

Because `1=1` is always true and anything after `--` is considered a comment, this would return all the users. The login would succeed if the condition is that the query yields results, without knowing any valid credentials. There are several solutions for the problem of SQL injection.

- *Escaping functions* Consistent use of escaping functions for unsafe input going from user data to a query. Escaping functions filter the user data to escape special query syntax. This is not a safe solution because escaping calls can easily be forgotten by the application programmer and cause an injection flaw. For example, when researchers implemented static analysis to search for code patterns of typical escaping bugs [Livshits and Lam 2005], they discovered SQL injection vulnerabilities in several open-source applications and libraries.
- *Use API to build queries* An API on top of prepared statements helps the programmer avoid injection issues. However, blindly trusting the API is dangerous, as it can still be used incorrectly.

For example, a safe Hibernate Query Language fragment:

```
session.createQuery(  
    "FROM User WHERE username=:name and password=:pass;")  
    .setParameter("name", request.getParameter("name"))  
    .setParameter("pass", request.getParameter("pass"))  
    .list();
```

becomes unsafe when used in the following way:

```
session.createQuery(  
    "FROM User WHERE username = "  
    + request.getParameter("name")  
    + " and password = "  
    + request.getParameter("pass")  
    + ";")  
    .list();
```

This second variant would still work, however, it also introduces an injection vulnerability because the parameter will not be escaped.

- *Integrate SQL into Language* Integrating SQL into the syntax of a language provides a way to automatically enforce correct construction of SQL queries with parameters. This way there is no risk of accidentally omitting

an escaping function or using the API in the wrong way. This solution was proposed in the Stringborg [Bravenboer, Dolstra, and Visser 2010] approach.

### 1.5.3 Cross-Site Scripting

Cross-Site Scripting (XSS, as not to be confused with Cascading Style Sheets) is the situation where page content crafted by a malicious user is shown to victim users without proper escaping. The content will be seen as coming from the web application itself and assumed valid. It might contain JavaScript that executes a request to the malicious user's web server and pass the cookie data along. This means the hacker can assume the identity of the victim on the web application. It could also replace the content of the application and show different information or completely deform the site.

There are several variants of XSS, it can be divided in server- and client-side, and both can be either stored or reflected. Server-side means the malicious content went through the server application code, while client-side means it never left the browser. Stored XSS is when the content is stored in a database such as MySQL on the server or HTML5 storage on the client. Each request that uses the server data without escaping to generate a page will be vulnerable. Reflected means that the content is not stored but passed along with the request, e.g. with a specially crafted URL that sets request parameters.

Without HTML escaping on the values of a request parameter, it can easily insert page content and JavaScript code. An example of a server-side reflected XSS vulnerability in PHP is the following:

```
<?php
  $user = $_GET['user'];
  echo "Username: $user";
?>
```

The server accepts any `user` argument and inserts it into the page without escaping. The following request would steal the session cookie of the logged in user by sending it to the hackers own website:

```
index.php?name=
<script>new Image().src="http://example.com/store?c="+encodeURIComponent(document.cookie);</script>
```

An XSS worm plagued Twitter on September 21, 2010. A crafted tweet would simply show a blacked out box, with an active mouseover JavaScript trigger. The JavaScript would use the logged in session of the viewer to tweet the same message and spread the worm. Another variant made the size of the box as big as the page, so the user would not even have to mouse over the blacked out text. This reflected server-side XSS bug was caused by insufficient escaping in the output, anything after an `@` symbol in the URL was not escaped:

```
http://t.co/@"style="font-size:9999999999px;"onmouseover="$.getScript('http://\u002f ...
```

The `$.getScript` is a JQuery JavaScript function that retrieves JavaScript code and executes it, in this case the JavaScript code would simulate the user creating a tweet with the malicious URL in there to spread the worm.

### 1.5.4 Cross-Site Request Forgeries

Cross-Site Request Forgery (CSRF) [OWASP 2022] is an attack that attempts to execute unwanted actions on a web application where the victim is authenticated. The hacker attempts to trick the victim into loading a page, while the browser will automatically send along cookie data such as the currently authenticated session. The harmful request URL can be inserted in an IMG or IFRAME tag so that navigating to a crafted web page of the hacker will automatically execute commands on the victim's behalf on the targeted web application. In particular, web applications in which a users stays logged in often and keeps a tab open in the browser are vulnerable.

A simple example is an image tag with a source url that would execute a delete action:

```

```

If the user is logged in at `example.com` and receives an email and opens a link to this page, it will execute the delete action even if the user did not trigger it themselves. Because the browser will include cookie data of `https://example.com` in the request, the actions is allowed as if the user would have executed it.

The remedy is to make sure that action request URLs and parameter names are not guessable and unique for each user. A common way to do CSRF protection in frameworks like Django [2023] is to rely on adding an additional CSRF token to all the forms. The token is a random secret value associated with a user session that needs to be submitted with the request parameters to perform the action. Although it makes protection convenient, it is still something that a developer can forget to include, or cause confusion if it is used incorrectly and blocks a submit unintentionally.

### 1.5.5 Reflection and Run-Time Code Manipulation

An interesting class of vulnerabilities that occurs in web programming is related to reflection and run-time code manipulation. The abstraction and safety provided by a framework is weakened by flexibility of the underlying general-purpose language.

Ruby on Rails is a web framework that is built on top of the Ruby language. Ruby features such as `eval`, evaluating an arbitrary String value as code, and monkey patching, replacing code at run-time without altering the original source code, can easily lead to security flaws. Handling data in such a dynamic way also happens in the framework itself. The mass assignment function provides a convenient way for loading all inputs into an entity object directly from the request parameters. However, it is also a security leak since 'POST'-data tampering allows setting any property in the entity. To work around this issue, a property can be shielded from mass assignment in the data model definition, something which is easily neglected and can break other action handlers that rely on this feature.

In January 2013, a major vulnerability in Ruby on Rails was discovered [National Vulnerability Database 2013]. The YAML [2023] format has a documented feature to deserialize into arbitrary objects, much like the mass assignment function. The default `YAML.load()` function does not prevent untrusted data from creating arbitrary objects. This unsafe loading was used in many places in Rails, e.g. to parse incoming JSON arguments. This enables remote attackers to perform object-injection attacks and execute arbitrary code.

The challenge is to provide web programmers mechanisms for all kinds of web programming tasks while avoiding security flaws. The programmer should not be burdened with remembering to include all the required safety mechanisms, such as CSRF tokens in forms, or escaping values that are inserted in SQL or in HTML. The easy way to handle a certain task should also be the correct and safe way. An example of this is database querying where concatenating strings to build a query is convenient to use but unsafe, using an API is inconvenient but safe, and providing query support in the programming language is both easy to use and safe.

## 1.6 Thesis

In the previous sections we have identified 3 main categories of problems in web application programming:

*Insufficient or Leaky Abstraction* Common web application concerns are insufficiently supported by abstractions provided in web frameworks. Boilerplate code is required for setting up ORM mapping persistence to map objects with fields to tables and columns in the database. User interface components encode input names that need to be processed in separate action handler definitions, consistency between these names is not enforced. Cross-cutting concerns such as access control policies are not easily specified in abstraction features provided by most general-purpose programming languages. Data validation is not well integrated with the user interface, requiring synchronization between error checking and rendering errors at the right locations.

Web frameworks are designed within the constraints of a general-purpose programming language. Design choices in the programming language can cause notational overhead and potential confusion when used in a web programming context. For example, equivalence between objects has a different meaning when they are backed by database persistence. Strings are handled in various ways internally in programming languages, which causes friction when working with strings coming from HTML form inputs and being placed in output HTML. To work within the constraints of the programming language, boilerplate code to set up the framework and glue components together becomes part of each application codebase. Many web frameworks enforce a coarse-grained Model View Controller file layout, which prevents organizing web applications according to their logical structure or dividing the source code based on application features.

*Lack of Static Verification* An application spread over multiple languages can have consistency faults which can lead to failures at run-time. There can be faults contained in a specific web application concern, such as a data model property with an unknown type, and faults can be in the links between application concerns. Some typical intra- and inter-aspect faults that can occur in web programming are: broken navigation links, broken submit action links, incorrect data access in rendering, and form parameter mismatch in controllers. Error reporting is limited by the type system of the programming language. Typically, not all constraints can be expressed, and error messages can be confusing when they are based on technical programming language concepts instead of concerns related to web programming. Dynamically typed languages increase complexity of code maintenance, by not assisting the programmer to ensure consistency when changing the program, and trusting the programmer's testing practices. Sound and complete verification becomes hard in a language that has too much flexibility, in this case abstractions that are closer to application concepts can simplify verification.

*Security Flaws* General-purpose programming languages are not designed with web security concerns in mind, even simple features like string interpolation can encourage wrong programming patterns that introduce injection vulnerabilities. Vulnerabilities such as SQL injection, Cross-Site Scripting, and Cross-Site Request Forgeries should be avoided without having to rely on rigorous programming practice by application developers. Programming languages providing reflection and run-time code manipulation, e.g. Ruby and PHP, are not ideal when it comes to security in a web environment. Features in the programming language that interpret text as code are especially vulnerable to invasive attacks.

Based on the problems we identify the following goals for improving web programming:

1. We need to improve web programming by designing better abstractions to cover web application concerns. These abstractions should as much as possible avoid boilerplate code, avoid opportunities for mistakes, and remove potential for accidental security faults.
2. The solution should provide timely and accurate feedback across web application programming concerns. Analysis of application code should be fast so that the feedback can be provided to the programmer in an IDE without separate verification steps or delays. Feedback should point directly to the inconsistency or mistake, and describe the problem in understandable terms. Resulting applications should not have remaining broken links or combine incompatible features.
3. We want to use this solution in practice, it should be feasible to build the designed solution, and use it in practice. Developed web applications should not crash, respond fast enough, scale to a large number of concurrent users, and avoid security problems.



Our solution for the problems of abstraction, static verification, and security in web programming is a language-based approach. We designed and implemented the WebDSL web programming language, a domain-specific language (DSL) for the development of full-stack web applications that integrates sub-languages for web programming concerns such as data persistence, user interfaces, access control, data validation, and full text search. The abstractions provided by the language are designed to only express essential complexity, and avoid accidental complexity by generating required boilerplate code. The sub-languages are linguistically integrated to provide cross-language consistency checking and ensure that a web application is created without broken links or incompatible features. Static analysis is provided live in an IDE during programming, and error messages are described in domain concepts, to provide timely and accurate feedback. Abstractions are designed to avoid security problems, and the runtime automatically enforces common exploit countermeasures. We evaluated this language over 10 years by developing, operating, and maintaining multiple real-world web applications used by thousands of users.

The identified problems and goals in web programming, language-based solution, and evaluation in practice form the basis for my thesis: **New web programming abstractions integrated in a domain-specific language improve web programming by avoiding boilerplate code, providing timely and accurate feedback on problems in application source code, and ensuring reliability (robustness, performance, scalability, and security) of applications. The design and implementation of such a language is feasible and has practical applicability.**

## 1.7 WebDSL Design Principles

Based on the problems we observed in web programming languages, and our experiences in designing and developing the WebDSL language and applications, we identify 5 core design principles for WebDSL:

1. **Linguistic abstractions should enable direct expression of intent.** Boilerplate code is generated or hidden in the runtime. Accidental complexity is removed, only essential complexity is expressed. Design language concepts with the right flexibility required to express the essential complexity.
2. **Linguistic abstractions should ensure reliability and security.** Applications should keep working when deployed in a real setting. This means the runtime should ensure robustness, performance, scalability, and also security, protecting against malicious web technology exploits (e.g. Cross-Site Scripting or remote code evaluation). Exploit countermeasures are enforced in the runtime without adding complexity to application code.
3. **Static checking should present errors in terms of the domain.** Design from the ground up with static analysis and cross-language consistency

checking in mind. The IDE and compiler can analyze the code and immediately report errors. Use explicit syntactic constructs for language concepts, so that semantic errors can be precise and messages in terms of the domain concepts.

4. **Extensibility should be explicit.** Avoid abstractions from becoming leaky, in cases where knowledge of the generated code is required to complete the application. Extension with external components is done through explicit foreign function interfaces in the language, such as for invoking server-side Java or client-side JavaScript libraries.
5. **Lessons learned should be consolidated in the language.** Language and applications should co-evolve, reflecting experiences from requirements engineering and application development in the language design. General problems found and fixed in applications should become language or library improvements, so that other applications automatically reap the benefits.

## 1.8 Research Methodology

In our research we aim to create solutions for problems in web engineering and language engineering by developing concepts, methods, techniques, and tools. We aim to create more than just prototypes by continuing maintenance and development beyond the proof of concept. Our tools are free and open source, which enables outside contributions as well as encourages adoption and building a community. The larger application case studies we perform (see Chapter 7) are not all open source software. The reason for this is that funding is acquired for developing these applications, which requires protecting the value that these applications provide and prohibit sharing it freely. The funding for these applications also sustains maintenance and further development of WebDSL itself.

The WebDSL project has contributed valuable insights into DSL development in general, and provides a platform to develop sizeable web applications. This is a longitudinal study that involves applying it in practice, as well as using it for further research and education. This dissertation does not contain a detailed formalization of the WebDSL language with proofs of correctness, typically found in Programming Languages research. Although we have made an initial attempt to model the dynamic semantics of WebDSL, making such a model complete and finding the right properties to prove is considered future work. Our focus has been on practical application of the language on real-world applications, the validation method used for our work on WebDSL is Technical Action Research.

Wieringa and Morali [2012] propose Technical Action Research (TAR) as a validation method in information systems design science. In TAR the starting point is artifact design. The researcher initially designs the artifact for use in a class of situations, as they are imagined by the researcher. The artifact is first tested on toy problems in idealized circumstances. The artifact is iteratively

improved and the tests are scaled up to solve more realistic problems, until it is scaled up enough to be tested in real-world conditions that occur in concrete client organizations to solve concrete problems. This allows the researcher to develop knowledge about the behavior of artifacts in practice, with the goal of increasing relevance without sacrificing rigor.

WebDSL (the artifact) was initially applied to create small web applications for internal use, where various requirements such as usability, performance and scalability were relaxed (idealized circumstances). Additionally, many automated compiler tests were created to test specific features of the language in isolation. As confidence in the artifact increased, we started looking for particular client problems to solve using the artifact. These problems were found in the international academic community, where the costly (researchers wasting time to build websites) creation and maintenance of conference websites was a problem. We also found particular problems in the university organization, in the workflows around education quality control and the approval of individual study programs. Finally, we found problems in programming education, where the setup of programming environments formed unnecessary time sinks for students, and the examination of programming courses was done on paper without testing performance of the actual skill. Our validation of WebDSL is based on applying it to design and implement web information systems in the academic domain. This gave us the opportunity to learn lessons from the practical application of WebDSL.

TAR consists of two engineering cycles to solve improvement problems. In the first engineering cycle a researcher aims at improving a class of problems (develop a useful artifact), in the second improving a particular problem (help a client). The third cycle is an empirical cycle, where knowledge problems are investigated about the artifact under study, which connects the two engineering cycles. In a TAR project, these cycles become three roles that must be kept separate, in our case these are: (1) Technique developer: develop WebDSL to improve creation of web information systems. Chapters 2-6 of this dissertation describes the design and implementation of WebDSL, and focus on the role of technique developer. (2) Client helper: use WebDSL for applications to help real clients, in particular the Researchr conference system, WebLab, MyStudy-Planning, and EvaTool. Chapter 7 contains description and usage analysis of the applications developed for clients. (3) Empirical researcher: draw lessons learned from this use, to discover the practical applicability of WebDSL in a real setting. Chapter 7 describes the lessons learned from solving real client problems.

The set of units of study make up a population. It is difficult to state exactly what the population is for the design of WebDSL. Continued research may be required to provide a more precise delimitation of the population. In our case, the explored population involves academic workflow applications. The application sizes in number of users are hundreds in high-intensity workloads (e.g. 500 students taking a WebLab exam at exactly the same time), and thousands in low-intensity workloads (e.g. 2000 conference visitors browsing the program occasionally spread over many days). The TAR technique recom-

mends choosing a client company which clearly falls inside the imperfectly known population. In our case we expected WebDSL to scale up sufficiently for the number of users found in these academic workflow situations. In terms of complexity we did not expect problems for building academic workflow applications, although in the case of WebLab there is additional complexity in the LabBack code evaluator, which is separately developed from the web application.

A major threat to validity is that the researcher, who developed the artifact, is able to use it in a way that no one else can. To mitigate this threat, others have been taught WebDSL and have contributed to building the client web applications. For example, the initial version of EvaTool was developed by a student that was introduced to WebDSL in a course on model-driven software development. The first version of WebLab was created in collaboration with a student, who was investigating safe execution of guest code as part of a MSc thesis. Most members of our academic workflow engineering team joined in later phases of the project, and were not involved in the entire artifact design and implementation process. More recently, we successfully taught WebDSL in a course on Web Programming Languages. In this course, students with different backgrounds, some having done no web programming at all, were able to construct varied web applications. The applications included several web programming concerns such as user management and access control. The applications were free from typical web security vulnerabilities like SQL injection, XSS, CSRF, and broken access control. Some of these students are now contributing to the development of client applications as freelancers.

## 1.9 Contributions

Our work on abstractions for web programming resulted in several scientific and software contributions:

1. Design and implementation of a linguistically integrated domain-specific language for web programming that combines abstractions for web programming concerns covering transparent persistence, user interfaces, data validation, access control, and internal site search. Sublanguages for the various concerns are integrated through static verification to prevent inconsistencies, with immediate feedback in the IDE and error messages in terms of domain concepts. The abstractions in WebDSL are designed to avoid boilerplate code as much as possible, only essential complexity of web application concerns is expressed. The design of the main abstractions is explained in Chapter 2.
2. Design and implementation of a web application user interface sublanguage that integrates automatic data binding, provides safety from data tampering, prevents input identifier mismatch in action handlers, enables safe composition of input templates, automatically enforces Cross-Site Request Forgery protection, allows expressive data validation, and supports partial page updates without explicit JavaScript or DOM manipulation.

The user interface sublanguage covered in Chapter 3 is the most important abstraction in the WebDSL language, it is typically the largest part of a web application codebase.

3. Design and implementation of an access control sublanguage in which various policies can be expressed with simple constraints, allowing concise and transparent mechanisms to be constructed. Enforcement of access control is automatic through weaving checks into the application code. The explicit declaration of access control through language elements, allow assumptions to be made about the related parts in the application, such as hiding inaccessible navigation links. The access control sublanguage of WebDSL is the subject of Chapter 4.
4. WebDSL is the largest programming language created with the Stratego program transformation language and the Spoofox language workbench, in which the DSL compiler and IDE have been iteratively developed. This iterative development is a recurring pattern of discovering new abstractions, domain-specific language integration, and reimplementing using new core abstractions tailored to the language. The generic evolution pattern for the iterative design of the WebDSL language is explained in Chapter 5.
5. The WebDSL compiler front-end combines program transformation with static analysis in a modular way. The implementation of static analysis is shared between compiler and IDE. The back-end code generator and runtime library avoid unsafe client-side operations, and employs countermeasures for security vulnerabilities. The IDE provides syntax highlighting, immediate error feedback, reference resolving, content completion, and automatic deployment of development builds. The design and implementation of the WebDSL compiler and IDE is discussed in Chapter 6.
6. To evaluate applicability and reliability of WebDSL, we have created real-world applications in the domain of research and education for external clients. The practical applicability of WebDSL depends on the reliability of these applications, how robust, performant, scalable, and secure they are. Chapter 7 reflects on the development of the applications, and in particular the experiences in engineering reliability. The applications are:
  - **EvaTool [2012]**: a course evaluation application that supports processes for analyzing student feedback by lecturers and other staff.
  - **WebLab [2012]**: an online learning management system with a focus on programming education (students complete programming assignments in the browser), with support for lab work and digital exams, used in multiple courses at TU Delft.
  - **Conf Researchr [2014]**: a domain-specific content management system for creating and hosting integrated websites for conferences

with multiple co-located events, used by all ACM SIGPLAN and SIGSOFT conferences.

- **MyStudyPlanning [2016]**: an application for composition of individual study plans by students and verification of those plans by the exam board, used by multiple faculties at TU Delft.

## 1.10 Structure of this Dissertation

This dissertation is structured as follows:

- We start with an introduction of the WebDSL language in Chapter 2. The WebDSL language consists of several new abstractions that work together in the specification of a complete web application. The three core language concepts in WebDSL are *data model* entities with automatic persistence to the database, *user interface templates* with safe HTML output and data binding in forms, and *functions* to implement operations on data. Using an example application, this chapter also introduces language concepts for access control, partial page updates, and search. These features are revisited in later chapters. This chapter concludes with a reflection on important design decisions in the core concepts and how these abstractions reduce boilerplate code when programming web applications.
- User interfaces are the most important language concept in WebDSL. Providing user interfaces is typically the largest part of a web application codebase, and many of the problems found in web programming relate to abstraction and security when creating user interfaces. Therefore, Chapter 3 covers the core language concept of user interface templates in WebDSL. At the core of the user interface templates is the request processing lifecycle. User interface code is processed in multiple phases to handle form inputs and validation. Data validation is supported in various forms, specific to certain user interface elements or generic for all entity data instances. The standard library provides default definitions for input templates of each data type. WebDSL's features for partial page updates provide opportunities to improve the user experience without sacrificing convenience for the developer.
- A real-world application has many users with different roles and capabilities, tied to specific data in the system. WebDSL provides built-in authentication features and an access control policy modeling sublanguage, which is explained in Chapter 4. The access control sublanguage enables expressing a wide range of access control policies concisely and transparently as a separate concern. The sublanguage is realized by means of a transformational semantics that reduces separately defined aspects into an integrated implementation.
- The WebDSL language design went through several iterations. In particular, the user interface language improved significantly from the initial

implementation with rigid input components defined in the code generator. The current implementation provides customizable library-defined components for input and output templates using a small set of language primitives. The generic evolution pattern that has been applied in the development of WebDSL language features is described in Chapter 5. Several examples of the pattern applied to evolution of language features are covered, including persistence, static code templates, email generation, files, and internal site search. This chapter concludes the discussion of WebDSL language features.

- A large programming language with several sublanguages is complex to create. The implementation of such a language benefits from reusable patterns for language analysis and code generation. The design and implementation of the WebDSL compiler explores the interaction between analysis and transformations, in a high-level transformation language based on the paradigm of rewrite rules with programmable strategies (Stratego). Chapter 6 provides an overview of the WebDSL compiler, IDE, and language runtime. The compiler pipeline consists of analysis, transformations, and code generation. The analysis of WebDSL is divided into the following steps: parsing and imports, declare global definitions, name resolution, and check constraints and report errors. Analysis is reused for compiler and IDE. Using caching of analysis results, the IDE provides timely and accurate feedback on application faults. Transformations provide reuse in the WebDSL language semantics by encoding language features into other language features. The WebDSL compiler generates Java code that together with a runtime library forms the deployed web application. Code generation benefits from several caching strategies to improve rebuild times. The discussed implementation strategies provide insight into the implementation effort and the feasibility of our language-based approach to improving web programming.
- We have used WebDSL to design and implement several real-world applications with thousands of users. Chapter 7 evaluates applicability and reliability of WebDSL, by analyzing four applications and reporting on practical experiences. These applications are: EvaTool, WebLab, Conf Researchr, and MyStudyPlanning. We evaluate reliability of WebDSL by covering robustness, performance, and security experiences. Furthermore, we evaluate practical applicability of WebDSL by reflecting on our experiences of the whole design and implementation process of these applications: requirements gathering, prototyping, testing, deployment, version migrations, bug fixing, and feature additions.
- There are many technical solutions available for web programming. In Chapter 8, we compare WebDSL against conventional full-stack web programming, and language-based solutions proposed by other researchers. In particular, a detailed code comparison showing UI and persistence is made with the Django Python framework, the Ur/Web programming language, and the Links programming language.

- Chapter 9 concludes this dissertation by revisiting the thesis and design principles, and discussing future work opportunities.

## 1.11 Origin of Chapters

This dissertation contains content from peer-reviewed publications as well as new material to create a complete coverage of WebDSL research. Content from published papers has been adapted and extended to create a monograph.

- The 2013 Software and Systems Modeling journal paper *Integration of Data Validation and User Interface Concerns in a DSL for Web Applications* [Groenewegen and Visser 2013] and its predecessor, a Software Language Engineering (SLE) 2009 short paper [Groenewegen and Visser 2009a] contain information about the request lifecycle of user interfaces and data validation. This is included as a part of Chapter 3.
- The International Conference on Web Engineering (ICWE) 2008 best paper *Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns* [Groenewegen and Visser 2008] is included in the discussion of access control in Chapter 4. This paper was based on my own MSc thesis project *Declarative Access Control for WebDSL* [Groenewegen 2008].
- The International Workshop on Programming Technology for the Future Web (Proweb) 2020 paper *Evolution of the WebDSL runtime: reliability engineering of the WebDSL web programming language* [Groenewegen, Van Chastelet, and Visser 2020] is the basis for the experience report in Chapter 7.
- The 2023 Eelco Visser Commemorative Symposium papers *Eating Your Own Dog Food: WebDSL Case Studies to Improve Academic Workflows* [Groenewegen et al. 2023a] and *Conf Researchr: A Domain-Specific Content Management System for Managing Large Conference Websites* [Groenewegen et al. 2023b] are reflections on our activities working with Eelco as Academic Workflow Engineers and designing, developing, and operating our WebDSL applications. The EvaTool, MyStudyPlanning, WebLab, and Conf Researchr applications are also discussed in Chapter 7.

During my own WebDSL journey, I collaborated closely with other PhD students, as well as supervising MSc students on their thesis projects. This work informed several of the observations and conclusions drawn in this PhD dissertation. After the student projects finished, I continued further maintenance and development of these areas of the WebDSL project.

- The IEEE Software 2010 article *Separation of Concerns and Linguistic Integration in WebDSL* [Groenewegen, Hemel, and Visser 2010] and the co-authored Journal of Symbolic Computation 2011 paper *Static Consistency Checking of Web Applications with WebDSL* [Hemel et al. 2011]



contain motivation and development guidelines that are addressed in the WebDSL work as a whole. An analysis of static analysis problems in web frameworks, and implementation techniques for implementing static analysis and compiler transformations are part of Zef Hemel’s PhD dissertation titled *Methods and Techniques for the Design and Implementation of Domain-Specific Languages* [Hemel 2012]. WebDSL is the largest programming language created with Stratego and Spoofox. The design and implementation of the Spoofox language workbench are part of Lennart Kats’ PhD dissertation titled *Building Blocks for Language Workbenches* [Kats 2011]. An early case study of WebDSL, the YellowGrass issue tracker, was part of research into coupled evolution in Sander Vermolen’s PhD dissertation titled *Software Language Evolution* [Vermolen 2012].

- Chapter 6 contains ideas from the co-authored journal publication in *Software and System Modeling* *Code generation by model transformation: a case study in transformation modularity* [Hemel et al. 2010], as well as insights into the further development and application of those ideas.
- The co-authored Software Language Engineering (SLE) 2013 paper *A Language Independent Task Engine for Incremental Name and Type Analysis* [Wachsmuth et al. 2013] is motivated by insights and experiences from implementing and using the WebDSL IDE. That paper explores a different implementation approach for the static analysis of the WebDSL language. The original and current implementation approach is described in Chapter 6.
- The OOPSLA Workshop on Domain Specific Modelling (DSM) 2008 paper *When Frameworks Let You Down: Platform-Imposed Constraints on the Design and Evolution of Domain-Specific Languages* [Groenewegen et al. 2008b] marks the evolution from generating code for a framework to a custom back-end tailored to our needs. Chapter 5 provides a more detailed analysis of the evolution stages of the WebDSL project.
- The co-authored ECOOP 2016 paper *IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs* [Harkes, Groenewegen, and Visser 2016] describes a language for derived value computations, and different implementation strategies in persistent object graphs. The IceDust compiler can generate WebDSL code, which was used to run performance benchmarks for different calculation strategies in persisted object graphs. This was a collaboration with Daco Harkes who designed and implemented IceDust as part of his PhD dissertation titled *Declarative Specification of Information System Data Models and Business Logic* [Harkes 2019]. A case study of IceDust was performed on the WebLab application [Harkes, Van Chastelet, and Visser 2018]. Chapter 7 provides more details about WebLab. In a more recent project, MSc thesis *Comparing Static Semantics Specifications for the IceDust DSL: A Case Study of Statix* by Jesse Tilro [Tilro 2023], we examined the migration of the

static semantics implementation of IceDust from NaBL2 [Van Antwerpen et al. 2016] to Statix [Van Antwerpen et al. 2018].

- MSc thesis *Abstractions for Asynchronous User Interfaces in Web Applications* by Michel Weststrate [Weststrate 2009]: the initial implementation for partial page updates in WebDSL, this is a core component of current WebDSL and has been further expanded to improve integration with other language features, increase reliability, and cover additional use cases. Partial page updates using Asynchronous JavaScript And XML (AJAX) are described in Section 3.7. Michel is the author of the popular open-source JavaScript projects MobX [Weststrate 2023b] and Immer [Weststrate 2023a], and several other open-source packages.
- MSc thesis *A Domain-Specific Language for Internal Site Search* by Elmer van Chastelet [Van Chastelet 2013]: the search language in WebDSL, also in active use in several real-world applications. Elmer joined me in the WebDSL project, developing applications, providing support for applications, maintaining the language, and managing server operations. A discussion of our largest applications is provided in Chapter 7.
- MSc thesis *ORM Optimization through Automatic Prefetching in WebDSL* by Chris Gersen [Gersen 2013]: optimization of the runtime by automatically deriving prefetching instructions based on program structure. This project was inspired by the work on query extraction from Java programs in Wiedermann, Ibrahim, and Cook [2008]. Part of this work also involved adding support for manual prefetch statements, which is something used in current applications as well to address performance issues when they arise. Experiences regarding application performance and improvements made in this area are discussed in Chapter 7.
- MSc thesis *Separate Compilation as a Separate Concern: A Framework for Language-Independent Selective Recompile* by Nathan Bruning [Bruning 2013]: this work is the basis for several compiler caching strategies, in particular the Java code generation cache and IDE analysis cache. These were a major improvement to the usability of the compiler and editor of WebDSL. Compiler caching strategies are described in Section 6.7.
- MSc thesis *A Generative Approach for Data Synchronization between Web and Mobile Applications* by Chris Melman [Melman 2013]: this project was about generating services to integrate WebDSL with a client-side Mobl [Hemel and Visser 2011] application. It provided a useful case study to explore service features of WebDSL and automatic generation of web APIs. Support for creating web services in WebDSL still has a lot of room for improvement. The requirements have also changed since our initial exploration. Web service components now play a more crucial role as a back-end for client-side user interfaces. This is an area of future work (see Section 9.3).

- MSc thesis *Modernizing the WebDSL Front-End: A Case Study in SDF3 and Statix* by Max de Krieger [De Krieger 2022]: this work explores migration of WebDSL, the largest language implemented in Spoofox, to new technologies in the Spoofox ecosystem. The syntax definition is migrated from SDF2 to SDF3, tackling issues where features in the SDF language were changed or dropped. Using the syntax specification in SDF3, the static analysis previously specified in Stratego code is migrated to a declarative style in Statix. Chapter 6 provides an overview of the WebDSL compiler and IDE pipeline, explains the static analysis approach, and outlines the structure of the generated application code, however, it does not go into details of SDF and Stratego implementation code. Max started as a part-time student assistant working with us on our applications, and is now a full-time member of our WebDSL team.

# The WebDSL Web Programming Language

---

# 2

## 2.1 Introduction

In this chapter we provide an overview of WebDSL, covering the language and implementation considerations. WebDSL was originally proposed in 2007 [Visser 2007]. Since then, it has been in constant use in various production applications (see Chapter 7), and under constant development. This chapter provides an integral overview of the version of WebDSL as it is at the time of writing.

This chapter contains small comparisons to existing web programming solutions that motivate specific design decisions. A more detailed related work comparison of the WebDSL language to other solutions is postponed to Chapter 8. In that chapter, we compare features for persistence, user interface definitions, and functions, by taking examples from other solutions and explaining the differences with a WebDSL equivalent.

Section 2.2 provides an overview of the sublanguages in WebDSL, starting with the core concepts of data models, user interface templates, and functions. Section 2.3 introduces the example application that will be used in the rest of the chapter to give examples for various application components. Section 2.4 first explains the conceptual model for handling persistence. Then, entity definitions and property types are introduced. Several design considerations related to persistence are explained in more detail, namely bidirectional associations, object identity and equality checking, performance in Object-Relational Mapping, and database management. Section 2.5 introduces the sublanguage for specifying functions to handle operations on data. This section reflects on the approach of introducing an entirely custom language for expressions, and covers specific topics such as safe query embedding, Java interoperability, dispatch mechanisms, null values and runtime errors, and alternate back-ends. Section 2.6 presents the user interface language containing pages with navigation, template definitions and template calls, and templates elements for output of data. Section 2.7 continues the introduction of the user interface language with forms and input templates for updating data. Forms with data input and the request processing workflow in WebDSL will be covered in more detail in Chapter 3, because it constitutes an important part of the novel web programming features in the WebDSL language. Section 2.8 provides a first glance of user authentication and access control, which is the topic of Chapter 4. Section 2.9 provides an example of search definitions and partial page updates. Section 2.10 contains a general discussion of WebDSL, reflecting on questions about domain coverage, practical applicability, and security. Section 2.11 concludes this chapter.

Table 2.1 WebDSL Language concepts and their interactions

Concept	Functionality
<b>data model</b>	data entity objects with database persistence primitive, reference, and collection types load/save functionality for objects
<b>ui templates</b>	pages connected by navigation links render HTML tags and <b>data model</b> values forms with databind update <b>data model</b> objects
<b>functions</b>	general-purpose object oriented language actions triggered from <b>ui templates</b> update <b>data model</b> objects with assignments
queries	query <b>data model</b> objects in <b>functions</b>
email	email templates, based on <b>ui templates</b> send email trigger in <b>functions</b>
data validation	validation phase after databind in <b>ui templates</b> <b>data model</b> invariants, <b>functions</b> assertions render messages in <b>ui templates</b>
access control	rule-based sublanguage to create security policy declare principal <b>data model</b> object rule checks can use expressions from <b>functions</b> rule needs to refer to existing <b>ui templates</b>
native classes	declare interface of Java code in <b>data model</b> create objects and invoke methods in <b>functions</b>
services	<b>ui templates</b> page request to generate JSON read incoming JSON request data in <b>functions</b>
search	search field mapping in <b>data model</b> search queries in <b>functions</b>
JS CSS embed	embed JS and CSS fragments in <b>ui templates</b>
AJAX updates	update subset of <b>ui templates</b> inside page

## 2.2 Language Concepts

The WebDSL language consists of several concepts or sublanguages that work together in the specification of a complete web application. The three core language concepts in WebDSL, listed at the top of Table 2.1, are *data model* entities with automatic persistence to the database, *user interface templates* with safe HTML output and data binding in forms, and *functions* to implement operations on data. The rest of the table lists extensions to the core concepts and provides insight into the interactions between concepts. These interactions determine the static consistency checking the compiler and IDE perform and how error messages about failed consistency checks are phrased. For example, user interface templates require well-typed access to the data model; access control rules refer to defined user interface components for weaving in checks; the principal entity declaration needs to refer to entities and properties that have been declared elsewhere. Access control rules define the accessibility to

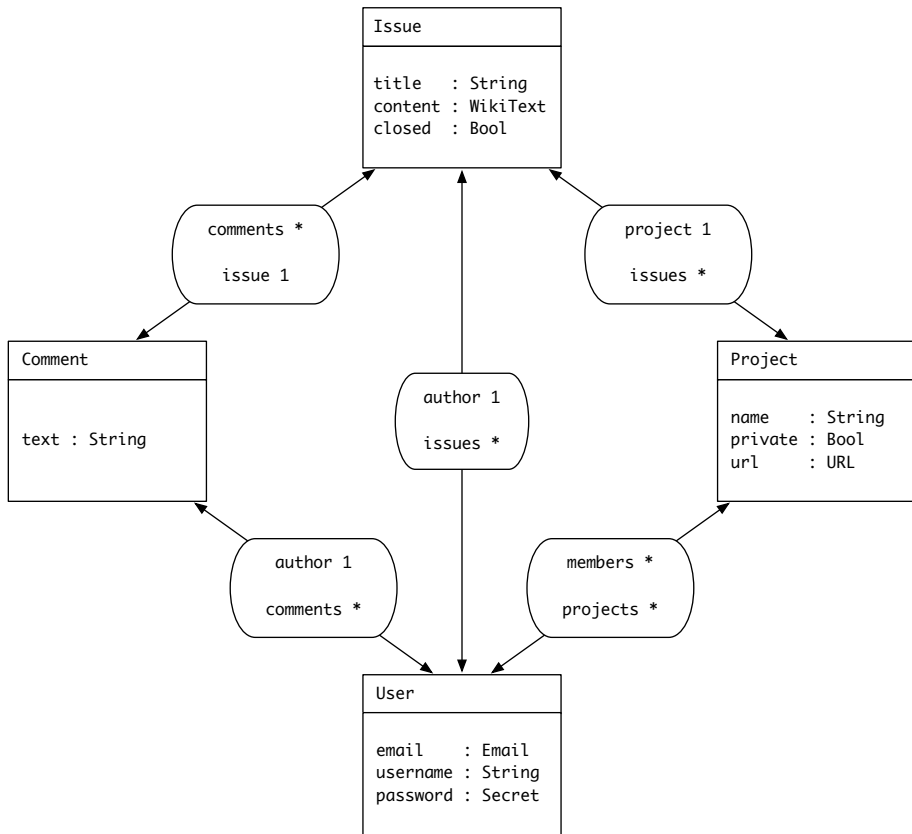


Figure 2.1 Example application data model.

pages and templates. If a page does not have an associated access control rule, a warning is given to notify the developer that the page is inaccessible. In addition to static consistency checking, the links between language concepts also influence the runtime of the language. For example, email rendering reuses template rendering, but stores rendered content in an email queue instead of returning it in a response to the browser. The number of sublanguages that can be added to WebDSL is open-ended. If multiple applications have a specific pattern or idiom, it is often worth investigating whether it can be captured in a library component or new sublanguage.

## 2.3 Example Application

To get a better understanding of WebDSL, we will use a simple project issue tracker as running example in this chapter. The application organizes issues under projects, and allows users to create and manage projects and issues. The Bootstrap [2023] CSS library is used for styling. This section introduces the data model and explains the usage scenarios.

### 2.3.1 Data Model

The data model is shown in Figure 2.1. It consists of `Issue`, `Project`, `User`, and `Comment` entities. They store primitive data such as the text value `Issue.title`, and the boolean value `Project.private`. Entities also have relations or associations to other entities, shown in rounded boxes. In this example all the relations are bidirectional, meaning that both sides can access and edit the relation. This allows easy navigation through the object graph, simplifying lookup of data. The relation name and multiplicity closest to the entity is how that entity refers to it. For example, `Issue` has a `single project` and `Project` has `multiple issues`.

### 2.3.2 Web Interface

In this section we will step through the various pages of the issue tracker example application using screenshots and descriptions of the web interface.

---

**The Project Manager** [Sign In](#) [Create Project](#) [Create Account](#)

**Freely manage all your projects!**  
Project issue management with tagging, commenting and search.

[Start your project today](#)

**Projects**

Spoofox	<a href="http://spoofox.org">http://spoofox.org</a>
Yellowgrass	<a href="http://yellowgrass.org">http://yellowgrass.org</a>
Researchr	<a href="http://researchr.org">http://researchr.org</a>
WebDSL	<a href="http://webdsl.org">http://webdsl.org</a>

The frontpage shows the list of managed projects. The navigation bar contains links to `Sign In`, `Create Project`, and `Create Account` pages. The large button also links to the `Create Project` page. The list of projects is shown in a table with alternating white and grey background.

---

**The Project Manager** [Sign In](#) [Create Project](#) [Create Account](#)

**Create Project**

Project name  
  
Please enter a project name.

Project website

private project

[Create Project](#)

This is the `Create Project` page. The application validates input data and displays errors in the page. In this example the `Create Project` button was pressed while there was no name specified yet.

## Spoofox

<http://spoofox.org>

[Report issue](#)

[Search issues](#)

The project view page shows its reported issues, these link to the issue view page. Additional navigation links are provided that go to the [Create Issue](#) and [Search Issues](#) pages.

### Issues

<a href="#">Update icon for Spoofox editor</a>
<a href="#">SDF3 operator precedence redundancy</a>
<a href="#">Make nightly-archive the update site for nightly builds</a>
<a href="#">Reference resolving to files outside workspace</a>
<a href="#">Hook for desugaring before transformation</a>
<a href="#">Analysis not cancelled in cree mode</a>
<a href="#">NullPointerException while getting tasks for a selection</a>
<a href="#">Concrete syntax ambiguity crashing JSGLR</a>
<a href="#">Semantic highlighting</a>
<a href="#">Add support for quick fixes</a>

### Report issue for project Spoofox

Title

Description  
Specification of associativity on the grammar rule  

```
templates
Exp.Addition = <<Exp> + <Exp>> (left)
and in priorities
...
{ left:
  Exp.Addition
  Exp.Subtraction
}
```

Description Preview  
Specification of associativity on the grammar rule  

```
templates
Exp.Addition = <<Exp> + <Exp>> (left)
and in priorities
...
{ left:
  Exp.Addition
  Exp.Subtraction
}>
...
```

On the [Create Issue](#) page, issues can be created for projects by supplying a title and description. The description can use Markdown, a preview is shown below the issue content entry text box. This preview updates automatically whenever a change is made.

## Spoofox

### Add support for quick fixes

Quick fixes could be used to fix semantic errors such as missing import declarations and even syntactic errors.  
To implement non-local quick fixes, a combination of pretty printing and layout-preserving unpadding is necessary. We also need that for refactoring.  
Submitted by Lennart on 09/07/2014 0:09

Comment

[Add Comment](#) [Close Issue](#)

The issue view page shows the issue title and description. The comments are listed below the issue details. Buttons are provided to add a comment or close the issue.



## Sign Up

**Username**

**Email**

**Password**

User accounts can be created on the `Sign Up` page. Usernames must be unique. The password value has a minimal length of 12 characters.

## Sign In

**Username**

**Password**

**Stay Logged In**

The `Sign In` page is the page for logging into the system. Access to private projects and its issues and comments is only allowed to members.

**Search query**

[editor sdf3](#)  
[editor selection](#)  
[editor site](#)  
[editor spooifax](#)  
[editor syntax](#)

**Results**  
[Empty editor is not analyzed](#)  
[Update icon for Spooifax editor](#)  
[SDF editor report unreachable productions](#)  
[Code folding hides errors in editor](#)  
[Ctrl-click fails on terms followed by a \\* in SDF editor](#)

Issues can be searched on the `Search Issues` page. Search is scoped to a project. Word completions based on the indexed issues are provided. The completions and results update automatically when typing a search query. The results are links to their corresponding `issue` view page.

---

This concludes the overview of features in the running example. In the following sections the core language concepts of WebDSL are introduced, illustrated with source code snippets of the example application.

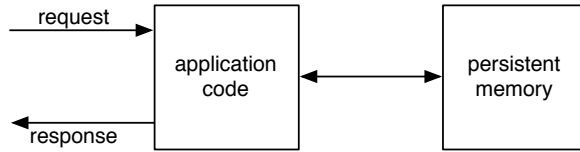
## 2.4 Transparent Data Persistence

Data persistence is a common concern in web programming. In this section we look at the conceptual model for data persistence and its implementation in WebDSL based on entity definitions.

### 2.4.1 Memory and Storage

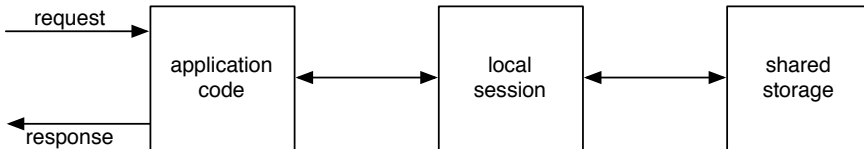
There are several ways to look at persisting data from an application developers point of view. Ideally, the persistence is handled completely transparently and

the model for the developer is working directly with persisted memory. The application tier contains the data and there is no separate instance for storage:

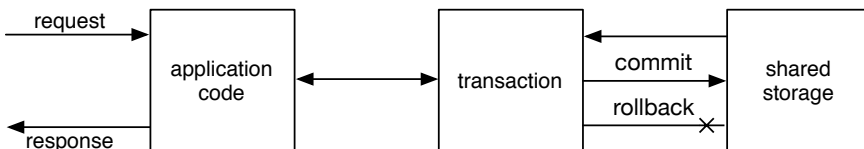


There are two main issues with this simplified model. Firstly, there is no notion of concurrency, which, for applications with little or no possibility of conflicting user writes, can be sufficient to be able to develop the application. However, for applications with more interaction between users the simplified model is insufficient. Secondly, a typical implementation does consist of a separate application and storage system, where transferring data in between systems means overhead, resulting in friction in the simple model when it comes to performance. Transferring a large amount of data from a DBMS to the application server can cause an undesirable delay. Optimizations can be performed in the mapping to an actual implementation, but a naive application can still request excessive amounts of data.

A refinement of the model is to introduce the notion of an in-memory session that keeps track of data retrieved from the shared storage. Data that is requested is pulled from the storage, in order to be presented to the user. Modifications made in the session are flushed back to the shared storage. This model makes the concurrency aspect more explicit by differentiating between shared and local data:

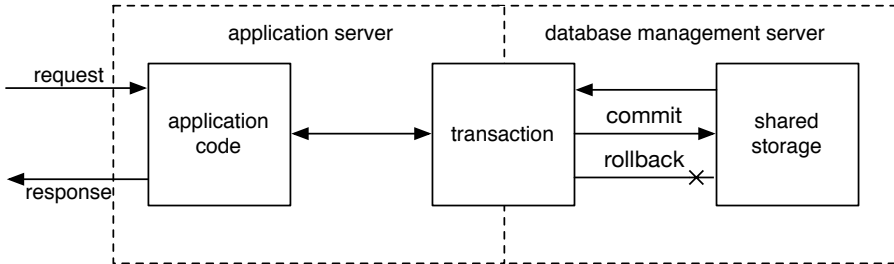


Another refinement can be made by seeing the session as the transaction of a single user interacting with the web application, and the flush to the storage as the transaction commit. A rollback would be equivalent to throwing away the session data instead of pushing it back to storage:



Looking at a typical implementation platform in the form of an application server with DBMS, sessions and transactions are mapped to both the application and database system. The DBMS handles transactions and concurrency,

and the application server handles each request in a new session and controls the transaction. Possibly this involves pulling in more data and flushing changes to the transaction managed in the DBMS, and eventually either committing or rolling back the transaction:



Transactions in WebDSL are implicit, each server request is a single transaction. In this transaction, objects are loaded from the database (often starting from the page arguments), edited in-memory, changes flushed back to the database, and finally the transaction is committed. The goal is to provide the developer a familiar object-oriented data model abstraction with simple saving and loading, without requiring knowledge of the underlying database or ORM intricacies.

## 2.4.2 Entity Objects

Entity definitions declare objects with persistence. They implicitly define a database schema for storage. Entity definitions contain properties, which describe the data contained in the entity. The `Issue` entity from our example project has six properties. Properties can have primitive types (`title`, `closed`, and `text`), entity references (`project` and `author`), and set or list collections (`comments`). The complete data model for the example application is shown in Figure 2.2. This section will further discuss entities, types, and property annotations. Section 2.5 explains WebDSL expressions and functions.

*Primitive Types* The primitive types are `String`, `Int`, `Bool`, `Float`, the date types `Date`, `Time`, `DateTime`, and the `String`-like types `Text`, `WikiText`, `Secret`, `Email`, and `URL`. The `String`-like types have all the `String` functions and the same runtime type. They differ in default input and output widget, additional functions, and validation rules that restrict the possible values. The date types `Date`, `Time`, `DateTime` are also related in this manner. Conversion between such equivalent types is implicit when doing assignment of values to variables or entity properties. These types support overloading, for example, there are specific template implementations for `input (WikiText)`, `output (WikiText)`, `input (Secret)`, and `output (Secret)`.

*Entity Types* Entity types are types that refer to defined entities. The default value of entity type properties is `null`. The default input of an entity property is a drop-down list input (`select tag` in HTML) which shows the names of the selectable entities.

```

1 entity Issue {
2   project : Project ( inverse = issues )
3   title   : String
4   comments : {Comment} ( inverse = issue )
5   closed  : Bool
6   author  : User ( inverse = issues )
7   content : WikiText
8 }
9 entity Project {
10  name : String ( id
11    , iderror = "Project name already exists."
12    , idemptyerror = "Please enter a project name." )
13  issues : {Issue}
14  url    : URL
15  members : {User}
16  private : Bool
17 }
18 entity User {
19  username : String ( id, name )
20  password : Secret ( validate( password.length() > 12, "Password too short" ) )
21  email    : Email
22  comments : {Comment}
23  issues   : {Issue}
24  projects : {Project}
25 }
26 entity Comment {
27  text : String
28  author : User
29  issue : Issue
30 }

```

---

Figure 2.2 Example application entity definitions.

*Collection Types* WebDSL has two collection types: `{Set}` and `[List]`. These collections are homogeneous, and parameterized with the type of elements they contain. Collection properties are initialized to an empty collection. The default input of a collection property uses checkboxes for selection of the options. Alternatives are also provided in the standard library, such as a drop-down list input that allows multiple selections (based on the `select` HTML tag with `multiple` attribute enabled). Because the `for` iteration construct supports sorting, filtering and mapping, the `Set` collection can be used most of the time. A `Set` avoids having to store ordering information in the database, which the `List` does store. Automatic entity persistence and query integration in WebDSL provide applications with an advanced “collection type”, in the form of the database. This means that in practice there is less need for more complex collection types besides the simple built-in `Set` and `List`. However, additional collection types could be added to WebDSL for convenience. This would involve implementing the application logic for implicit persistence of such collections in the WebDSL compiler.

*Default Values* Default values for primitive types are `"` for String-like types, `0` for number types, and `false` for `Bool`. The date types are initialized with no value, they are set to `null`. The `default` annotation can be used to change the default value for a property in a newly instantiated entity. This overrides the implicit default values for types.

*Implicit Properties* There are a few properties that are implicitly defined for each entity, either for convenience or because they are required for mapping to database tables. The implicit `id` property, which is used as primary key in database tables, is explained in Section 2.4.4. Other automatic properties are `created`, `modified`, and `version`, which describe when and how often an entity has been updated. The `name` property is used to describe the entity instance when displayed on a page. It is used in many input templates to display entities, e.g. as the label of radio buttons. The default `name` property is read-only and shows the value of the `id`. This default property can be customized, and made writable, by declaring an explicit `name` property or `name` annotation. The `name` annotation changes the value of the implicit `name` property to the value of the annotated property.

### 2.4.3 Associations

Entities can have relations to other entities. In WebDSL such associations can be created with properties having entity or collection types. An entity definition does not require any additional form of annotation or configuration to handle persistence of associations. While this seems trivial, it avoids accidental complexity that occurs in current web programming solutions when implementing associations.

In Ruby on Rails [2023] Active Record persistence, associations require specifying the relation type such as `belongs_to`, `has_one`, and `has_many`. Additionally, since Ruby does not have types declared, the developer must use the correct name for a relation. An example from the online manual [Rails Guides 2022]:

```
class Customer < ActiveRecord::Base
  has_many :orders
end

class Order < ActiveRecord::Base
  belongs_to :customer
end
```

Using any other name than `customer` for the `Customer` entity will make the association incorrect. This results in a generic error message “uninitialized constant Order::Customers”, because of the conventions for name pluralization. These design decisions are basically encoding types, however, without the benefit of static checks.

In a Java Hibernate ORM [2023] project, a developer must annotate the properties in domain objects with annotations such as `@ManyToMany` and `@ManyToOne`. This repeats part of the information of the property type, e.g. a reference to a single property will never need `@ManyToMany`.

*Bidirectional Associations* Configuration is even more intricate for bidirectional associations. These require matching column names in two domain entity objects, as well as deciding on the master and slave sides. The Master-side is the property that Hibernate uses to update database rows. Debugging such

mapping annotations typically means running the application and manually inspecting the generated tables. In-memory data for bidirectional associations has to be updated by the application developer. For example, a bidirectional association between two collection properties requires the developer to correctly implement that both collections are updated when an item is added or removed on one side. Only after committing the data, Hibernate will correctly load both collections with the same data automatically.

The `inverse` annotation links properties of entities and automatically keeps them synchronized. Inverses can be created for several kinds of bidirectional associations: one-to-one (both entity references), one-to-many (entity reference and collection), or many-to-many (both collections). The project manager example has three inverse relations. Setting the `project` field of `Issue` automatically updates the `issues` property of `Project` and vice versa. Similarly, the `author` property of `Issue` and `issues` property of `User` are linked. The annotation is only needed on one side, and can be placed on either side. The distinction between a master and a slave side for persistence is part of the abstraction and not exposed to the web programmer. The in-memory objects are automatically updated on both sides.

#### 2.4.4 Object Identity

Object identity and equality checking is a common source of confusion in many programming languages due to complex and overlapping mechanics. In Java `==` compares object references, i.e. pointers to object data in memory, while `.equals` invokes the object's `.equals` method. Primitive values can be compared using `==` without problems. Strings are a special case where compiler-constant strings can be compared with `==` while in general `.equals` should be used. The default implementation for an object equality is the same as doing `==`, which is typically not what is required. Objects, and in particular data objects, require the developer to overwrite the `equals` operator to perform a meaningful comparison of significant property values [Bloch 2008]. Additionally, the `hashCode` method used for generating deterministic hash values must be defined in tandem, to support the requirement that `hashCode` must generate equal values for equal objects. In PHP there is both a type-coercing `==` comparison and a stricter `===` comparison operator.

Besides equality checking issues, developers also have to choose what value to use as primary key for rows in the database. Usually this is just an autoincrementing number provided by the database. However, when using ORM abstractions this can cause problems. In a Java Hibernate application, a non-persisted entity can be used as value in a HQL query. The primary key is then requested with an additional query first, as the database has the control over identifiers. Besides some overhead and unnecessarily incrementing the `id` values, it is problematic when the unstable `id` value is used for equality checking and hashcodes. The point at which the `id` changes is determined implicitly by the Hibernate runtime. If a transient object is stored in a `HashSet`, the automatic loading of an `id` value can cause the entity to no longer be

retrievable from the set.

In WebDSL entities, the implicit `id` property is a unique identifier for an entity instance. A new value for the `id` is randomly generated when an entity is created. This value is used internally as primary key in the database. The type is a universally unique identifier (UUID). UUIDs are 128 bit numbers that can be created without querying a centralized authority. This is typically the database, as would be the case with autoincrement identifiers. This approach also simplifies running multiple instances of the application for load balancing. While in theory a duplicate could be generated, in practice, it is sufficiently improbable. A major benefit is that all entity instances can be given an identifier, even those that are not persisted to the database. Equality can thus be determined based on matching identifiers. Also, because each entity instance has a unique identifier value, the implementation can safely use hashed collections, using the identifier as hash value. No `equals` or `hashCode` methods have to be defined by the web application developer to support equality checking or hashed collections.

*Entity URL Name* What URLs look like is an important aspect for usability. Besides easy bookmarking, being able to see what information a link is likely to display helps the user decide whether to follow that link. Representing an entity as a URL component requires a uniquely identifying value for the entity. While the UUID `id` is a safe default, it does not look nice in the browser address bar. The `id` annotation provides a solution for this problem. `id` annotated properties are used to automatically create readable URLs for pages. Such a property is used as uniquely identifying value of the entity instance. The `id` annotation also adds validation for checking whether the values are defined and unique. The messages for these validation rules can be customized with `iderror` and `idemptyerror` annotations. The `Project` entity in the example project has a `name` property with `id` annotation. For example, the page `page project(p:Project)` for the `Project` entity with `id 'c70efffc-2069-4doe-8f60-5be18822c99a'` is without specifying an `id` associated with the url:

```
https://example.com/project/c70efffc-2069-4d0e-8f60-5be18822c99a
```

This URL is stable and bookmarkable, however, it does not look very nice, it is hard to remember, and web search engines will index meaningless text. By specifying the `name` property as `id` the URL becomes:

```
https://example.com/project/Spoofax
```

## 2.4.5 Inheritance and Polymorphism

Entities support inheritance: they can be declared as a subtype of another entity, incorporating properties and functions of its supertype. Subclass entities can

be assigned when their super type is expected (polymorphism). There is only single inheritance: one super entity can be declared per entity. Checking the dynamic type of an entity is performed using `is a` and casting is performed using `as`. In an overriding method definition the overridden method can be called with `super`. Using these object-oriented programming features in WebDSL is optional. There is no fixed framework structure that needs to be subclassed, which is often the case in Java web frameworks. Reuse of properties and functions is often better achieved through composition of entities.

### 2.4.6 Invariants

A high-level web programming solution should provide a uniform and declarative data validation model that integrates with the other application concerns. In addition to ensuring data consistency by enforcing data validation, the integration of data validation in a web application requires a mechanism for reporting constraint violations to the user. It should indicate the origin of the violation in the user interface with a sensible error message and consistent styling.

Data invariants are constraints on entity properties, checked both as input validation in a form, as well as at the end of an action that updated or created an entity. A simple invariant found in the example application is the `password` length check. Additionally, the `id` annotation implicitly adds invariants for non-empty and unique values. Invariants can be arbitrary WebDSL expressions, however, they are implicitly executed multiple times, so typically they should not have side effects. Data validation and invariants, and their handling in user interface templates is further examined in Section 3.6. Section 2.5 will continue with explaining expressions and functions in WebDSL.

### 2.4.7 Discussion Data Persistence

*Object-Relational Mapping* Performance is a major concern in the mapping of an object-oriented model to the relational model. WebDSL uses Hibernate for ORM, and out of the box the queries that get generated are not efficient in most cases. There are many options for tweaking this behavior, e.g. by specifying custom joins or prefetching paths to combine queries. However, in the setting of WebDSL a generic solution is needed in order for all applications to benefit. As described in Section 2.4.1, the model of a local session and transaction abstracts over the actual implementation in application server and DBMS. When code can run entirely in the DBMS using a single query, but instead all the data is loaded into the application server, processed there, and sent back, the performance suffers. We have seen this especially with collection types, where loading a big collection from the database into memory could be avoided in cases where the operations is a simple addition or removal. A similar situation occurs when a for loop iterates over a filtered collection, where the filtering could have been done in the database to avoid sending unnecessary entities to application server memory. We have investigated several of these



issues and implemented query optimization based on static analysis of the application to alleviate these problems [Gersen 2013]. Our findings were that the default query behavior generated many simple queries, however, these queries remained fast. In some cases these remained faster than an “optimized” version with fewer queries. Reliable measurements are hard to get due to many different caching and indexing schemes implemented in databases. A more dynamic optimization scheme that takes the data at runtime into account could be a solution for handling the complex performance characteristics of the database.

Besides issues with inefficient execution of queries, another concern is applications that retrieve unwieldy amounts of data. How many entities and properties become unwieldy depends on many factors. There is overhead in the data loaded from the database which is related to the automatic management of persistence. Also, some servers simply have less memory available. A possible solution for such a case could be monitoring the running application for request handling threads that consume too much memory and report by displaying a warning or take action by stopping the thread.

Another issue with the ORM abstraction lies in mismatch of operators and constraints in the application server object model with those in the database relational model. Whether two strings are equal in a database depends on the collation setting of the database table or the query itself, while Java has its own collation settings (typically the default binary collation is used). This is especially important when there is a uniqueness constraint in the database, where a difference in equality can lead to unexpected commit failures. A related issue is that in many database management systems, e.g. MySQL, uniqueness is checked in a running transaction already, where nothing is final yet. This means the transaction is aborted abruptly when such a uniqueness violation occurs. In most situations checking uniqueness or other constraints with a query gives the most reliable result, because in that case the decision is deferred to the database, which also makes the final decision.

Referential integrity in a relational database prevents deletion of rows that are referenced by foreign keys. In the object model this means persisted entities that are referred to by other entities cannot be deleted, however, all the entities that are relevant for this kind of check are typically not loaded into memory, so deleting an entity can fail at the final point of commit. This deletion issue, combined with the choice for archiving data instead of permanently deleting it, has resulted in most applications rarely deleting entities, unless the entity is explicitly used as a temporary data structure (in which case there are no references by design). A better solution for deletion would involve taking into account referential integrity and removing references to deleted entities. This would require some input by the application developer to determine how much related data should get deleted.

*Database Management* There are many technology options for filling the database storage role. In the case of WebDSL, we have mainly used MySQL in production environments, and the light-weight H2 [H2 Database Engine 2023] database engine for testing. Because of the choice for Hibernate in the

implementation, which abstracts over the differences between DBMS systems, switching to other databases requires minimal effort.

A database provides default indexing for primary keys, meaning that a lookup of a referenced entity is fast. Some queries do not benefit from the default indexes, because they use a combination of fields, or sort the result set based on a property value. Custom indexes can provide increased performance for queries that get executed many times. Creating and working with such indexes is very specific to the chosen DBMS. Also the type of index can influence this performance. For index choices and configuration WebDSL applications rely on the tools that come with the database. The Google App Engine platform provides a better integrated approach where performance of pages can be reviewed and indexes created from a web interface.

Backing up the database is also accomplished with the tools provided by the DBMS. MySQL has command-line utilities for dumping and loading the contents of a database using a text file with SQL statements. These backups can be automated with simple shell scripts and cron jobs.

Migrating the database schema is handled in Hibernate for simple cases. Adding tables and columns is done fully automatic. Changing existing data and migrating from old tables or columns to new cannot be done fully automatic. There are many possible situations for migrating existing data to a new schema. There has been research on the coupled evolution of data in the database with new versions of an application, applied to a WebDSL application [Vermolen 2012]. This work resulted in a prototype tool which can handle complex migrations. In our production applications we have done manual migrations, using update queries to set values, or handling migration in the application by including both old and new models, and migration code (which then becomes a live and incremental migration).

In general, with database operations like index creation, backing up, and migration, incrementalizing and being able to run in the background without noticeable interference is essential for usability. The level of maturity for these features can vary a lot between DBMS technologies.

## 2.5 Expressions and Functions

In most web frameworks, functions are specified in the host language of the framework. This host language has no specific knowledge of the web framework, preventing static consistency checking to assist the developer. For example, action parameters are often passed to the action in a generic Map structure instead of typed arguments. Another issue that arises is that hooks into the web framework require knowledge of its internal structure, such as the libraries used. For example, if a query needs to be described, this can be regular SQL or some SQL-like language such as HQL (Hibernate Query Language). Since such queries are encoded in a String there are no checks to support developers in creating the right type of query.

Handling application operations and modifying data in WebDSL is done with an object-oriented sublanguage. The WebDSL programming language

defines its own expressions, statements, and functions. The syntax and semantics should be familiar for programmers that know other object-oriented programming languages such as Java and C#. The object-oriented sublanguage is statically typed and can refer to other WebDSL elements defined in the application, such as entity types, local and global variables. The expressions are used in several WebDSL contexts, e.g., boolean conditions describe conditions in `if` templates and access control rules, and the arguments in user interface template calls are expressions just like the arguments in function calls. Static typing and integration of languages enables instant editor feedback of errors and possible completions, and makes an application robust when editing and refactoring.

Defining a completely customized language for handling function code in web programming has several advantages:

- *Better static analysis* The integration of the complete function language in the compiler enables consistency checking with respect to other definitions like entities. Language features that break static guarantees can be excluded entirely.
- *Non-functional security features in generator* Boilerplate code to avoid security issues, such as using a query API and prepared statements to prevent SQL injections, can be generated and automatically enforced.
- *Avoid pitfalls and accidental complexity of general-purpose languages* By being less tied to an existing language, it becomes possible to address some of their problems, such as those related to object identity and equality checking.
- *Portability* By having a complete definition of the function language, the code generator could be retargeted to a different language and runtime environment.

Disadvantages are:

- *Introducing another new language to learn* Especially when it comes to features already found in general-purpose languages, learning a new language can be a hurdle for web programmers.
- *An off the shelf general-purpose language is likely to provide more features* Providing a complete language replacement is more work than creating a framework. Integration with existing libraries is not automatic.

This section first explains WebDSL functions in more detail. Then the issue of safe queries is examined. The disadvantage of making it harder to integrate with existing libraries is important, how this is addressed in WebDSL is covered in Section 2.5.3.

```

1 extend entity Issue {
2   function addComment( t: Text, a: User ){
3     var c := Comment{ text := t author := a };
4     comments.add( c );
5   }
6   predicate isAccessAllowedTo( u: User ){
7     ! project.private || u in project.members
8   }
9 }
10 extend entity Project {
11   predicate isAccessAllowedTo( u: User ){
12     ! private || u in members
13   }
14   function getRecentIssueTitles: [String] {
15     return [ i.title | i : Issue in issues
16             where i.created.after( now().addMonths( -1 ) )
17             order by i.created desc ]; }
18 }
19 init {
20   Project{ name := "WebDSL"      url := "http://webdsl.org" }.save();
21   Project{ name := "Spoofax"    url := "http://spoofax.org" }.save();
22   Project{ name := "Researchr"  url := "http://researchr.org" }.save();
23   Project{ name := "Yellowgrass" url := "http://yellowgrass.org" }.save();
24 }
25 function allprojects: [Project] {
26   return from Project as p order by p.name asc;
27 }

```

---

Figure 2.3 Functions in example application.

### 2.5.1 Functions

Functions in WebDSL can be defined globally and as entity methods. A function definition has a list of formal arguments, an optional return type, and a body of statements. Function calls are expressions that contain the name of the function followed by a list of expressions that are the actual arguments. All functions defined in the example application are shown in Figure 2.3. Entity methods can use `this` to refer to the entity itself, and the properties can be accessed as variables directly. The `return` statement exits the function and returns the value. The `predicate` is a syntactic variant of function that only contains a boolean expression and returns its result. The `extend entity` feature allows specifying an entity in multiple fragments and files, which are merged by the WebDSL compiler.

*Literals* Each primitive type has a literal to construct it from a constant value. Lists can be created with square brackets, and sets with curly brackets. Figure 2.4 shows examples of literals and collections of primitives values.

*Entity Creation* The expression for instantiating new entity objects allows setting initial values for its properties. The syntax is the entity name followed by an optional list of property assignments between curly brackets. An example is shown in Figure 2.5.

*Variables* A variable declaration consists of a name, a type, and an expression for the initial value. Either the type or the initial value can be omitted. If the type is not explicit, the variable will receive the type resulting from the

```
String: "This is a string"
Int:    42
Float:  1.2
Bool:   true
        false
List:   [1,2,3]
Set:    {1,2,3}
Null:   null
```

---

Figure 2.4 Example showing literals.

```
Project {
  name := "WebDSL"
  url  := "http://webdsl.org"
}
```

---

Figure 2.5 Example showing entity creation.

expression (local type inference). Figure 2.6 shows examples of variable declarations and variable references.

```
function testFunction {
  var number := 0;
  var user   := User{};
  user.name := "WebDSL";
  var users  := [ user ];
  var webdsl := users[ number ];
  log( webdsl.name );
}
```

---

Figure 2.6 Example showing variable declarations and variable references.

Global application variables are defined at the top level together with entities, templates, and functions. They always have an expression for initialization. These variables are added to the database once when the application is deployed. At that point the database is checked to see whether all global variables have been created already and, if not, they are added. The global variable reference itself is immutable. It is designed to be the same entity instance, with mutable properties, at all times. Typical use cases are application configuration settings and entities that describe enumeration options.

*String interpolation* Variable content can be spliced into strings using string interpolation. This is performed with the `~` operator. Simple expressions such as a variable reference or property access do not require any additional markers, complex expressions (or disambiguation if necessary) can be performed with parentheses. Figure 2.7 shows examples of string interpolation expressions.

```
"content of x: ~x"
"~u.show() calls an entity function"
"global function result: ~getResult()"
"~(select count(*) from User) is the query result"
"disambiguation: ~(x)cm"
```

---

Figure 2.7 Example showing string interpolation.

*Function Overloading* WebDSL functions can be overloaded, i.e., functions can have the same name, but different number of arguments or different argument types. The most specific matching function signature is statically determined, taking into account subclassing in entities.

*Method Overriding* Entity methods allow overriding in subclasses. Calling an entity method performs single dispatch, the runtime type of the entity on which the method is called determines which override of the method implementation is invoked.

*Extend Function* Extend function enables the extension of functions with extra statements. These function extensions cannot have a return value and the order in which they are combined is undefined (as in there should be no dependencies between them). This feature enables crosscutting concerns to be expressed in a single location, similar to extend entity.

*List Comprehension* List comprehension expressions allow the construction of new lists by applying transformations to another list. This transformation can involve filtering on a certain property, reordering the old list, or mapping functions over the elements. The expression at lines 15-17 in Figure 2.3 returns all issue titles `i.title` from `issues` where the time created `i.created` is greater than a certain date one month back, ordered by `i.created` in descending order.

*Save and Load* Saving an entity is done through an explicit `save` method call, or through save cascading. Save cascading happens when persisted entities, either loaded from the database or newly persisted with a call to `save`, refer to transient entities. Those transient entities will automatically be persisted as well. This happens transitively, so any transient entity reachable from a persisted entity will be saved.

Loading entities is done using `loadX` global functions that accept an `id` value of UUID type and retrieve the object of entity type `X` corresponding to the `id`. Typically this happens automatically by using entity types as page arguments. In these cases the WebDSL runtime will expect an `id` for a specific type, and tries to load that object before handling the rest of the request.

### 2.5.2 Safe Query Embedding

WebDSL applications typically load entities starting from the page arguments, and traversing its properties recursively to load other related entities. Another way to load entities is by writing database queries. The queries are part of the syntax, so it does not allow queries represented by an arbitrary String, which would enable SQL injection vulnerabilities. Any value that is used in a query is automatically escaped to prevent SQL injection attacks.

WebDSL follows the Stringborg [Bravenboer, Dolstra, and Visser 2010] approach of embedding query syntax in the language. This method has both the benefits of natural looking queries as with String concatenation, but also safety by generating code that invokes a query API. This method ensures safe queries by construction. In the Java back-end of WebDSL, queries translate to

calls to the Hibernate Query Language (HQL) API. Line 26 in Figure 2.3 shows a query where all `Project` entities are retrieved ordered by name.

### 2.5.3 Java Interoperability

Integration with existing libraries is not automatic when defining a new language to handle function code. A particular web application can call into some existing library that is not specifically related to web applications, e.g. invoking a compiler for an online IDE. To enable such unanticipated library usages, WebDSL can invoke Java classes and libraries. The typechecker of WebDSL needs to be informed of the interface that is available. Native class declarations declare new types, with constructors, (static) properties and (static) methods, by referring to their fully qualified Java classname and specifying the name to be used in WebDSL. The interface has to use the Java equivalent of primitive types in WebDSL, meaning Java's primitive `int` or `Integer` object for WebDSL's `Int`, `boolean` or `Boolean` for `Bool`, `float` or `Float` for `Float`, `String` for `String`. Autoboxing and unboxing in Java takes care of conversion between primitive and object types. Additionally, the `List` and `Set` collection interfaces can be used with WebDSL. The invoked methods should not have checked exceptions. If the library does not conform to these requirements, Java code can be added in the project to wrap the library and make the interface accessible to WebDSL. The downside of using Java escapes is that this part of the application does not have the benefits of a custom defined function language, namely analyzability, security, portability, and avoiding common pitfalls in the general-purpose language.

### 2.5.4 Discussion Functions

*Dispatch* Function overloading, having functions with the same name but different type signatures, is statically resolved at compile-time. Function overriding, a subclass entity can override the function of its superclass entity, is dynamically resolved at runtime. For function or method calls on *entities*, WebDSL inherits single dispatch from the Java target. The most specific override of the function is based on the runtime subtype of the entity on which the call is invoked. For applications that heavily use inheritance and function overriding, multiple dispatch could avoid writing custom dispatching code. In multiple dispatch all the argument runtime types to the function would be considered for determining the closest matching signature. For readability of the code and static analysis, having binding resolved at runtime is problematic. When reading the code, a large number of definitions need to be taken into account, and similarly it becomes more difficult to create an exact static analysis of the code that will be executed.

Currently, WebDSL does not support dispatch for *template calls* in a user interface definition. In some of our applications that used entity subclassing, this required boilerplate code switches with checks (`x is a Foo`), and casts (`x as Foo`). To reduce this boilerplate code, and preserve static analyzability,

we introduced a `typecase` construct. The example in Figure 2.8 illustrates this construct. The alias `x` in the example has the specific subtype that the block is specified for, which also enables accurate reference resolving in the IDE to improve readability of the code again. This fragment makes immediately clear to the reader that control flow based on the dynamic type happens at this point in the code, instead of dispatch behavior that is implicitly defined by the structure of classes and method overrides.

```
entity Super {
    superprop : String
}
entity Sub : Super {
    subprop : Int
}
template show( arg: Super ){
    typecase( arg as x ){
        Sub { ~x.subprop } // x has type Sub in this block, calls output( Int )
        Super { ~x.superprop } // x has type Super in this block, calls output( String )
    }
}
```

---

Figure 2.8 Typecase in template example.

Since WebDSL provides a web framework built into the language, it is often not necessary to use inheritance at all. Entities are used for data, where code reuse is more often required than polymorphic behavior. In these cases composition can be used instead of inheritance, i.e. capture the common functionality in another entity and include a reference to it whenever the functionality is needed. In object-oriented programming, this design principle is called composition over inheritance [Gamma et al. 1995].

*Null Values* All types currently can have `null` as a value. Although this immediately raises The Billion Dollar Mistake [Hoare 2009] question, and this part of WebDSL could be improved, there are some mitigating factors. Primitive values in variables and entity properties are by default initialized to default values (`0` for `Int`, `false` for `Bool`, `""` for `String`). Set and List collections are by default initialized to empty collections. When dereferencing a null value in the context of templates (the majority of WebDSL application code is in user interface templates), the specific template element will be skipped without interfering with other template elements, and produce a warning in the log. Any action that triggers a null-pointer exception is considered failed and aborted, there is no risk of accidental incomplete database updates.

A null-value is currently used to represent the value of an invalid input field. In the case of a String-like type having an empty input is not an issue, the value simply becomes the empty string. However, in the case of an `Int` or `Date` type no sane value can be constructed from an empty input. Using the default value here would be ambiguous, since there is a difference between a user entering `0` for an `Int` field, or entering nothing.

One convenience introduced in other programming languages for dereferencing potential null values is a safe dereference operator, e.g. the expression `variable?.field` in Groovy results in the value `null` if the `variable` is `null`.



A more thorough solution is a wrapper type that can represent either a value or nothing. Such a wrapper type can be enforced by static analysis, and warn the developer when a potential missing value case is not handled. Examples of this are the `Option` type in Scala, which can be `Some(value)` or `None`, and the `Maybe` type in Haskell, which can be `Just value` or `Nothing`. Different types can be used to explicitly denote a type as nullable (can have value null) or non-nullable. For example, C# provides both regular and nullable variants of types, e.g. `int` for non-nullable, and `int?` for nullable. A downside of introducing more strict empty value handling is that the code becomes larger when potential missing values always need to be handled explicitly, as opposed to the current skip template element behavior. Connecting with Java libraries would require more boilerplate code if WebDSL introduces a specific variant of empty value handling, instead of null values.

*Runtime Errors* The back-end of WebDSL translates an application to the Java platform. A running Java application can fail with errors that might not be clear to the WebDSL application developer. An error can occur in the generated code of the application, while not being reported by static analysis. Some errors are not caught at compile-time because they depend on dynamic types or values. Properly handling such errors in a way that makes sense at the WebDSL abstraction level will improve the effectiveness of the language.

An infinite recursion of function calls causes a `stackoverflow`. These and other exceptions that occur in the code can be hard to trace back to the application code, because they will point to the generated Java code. Tracing these problems back to the original application code requires keeping track of the WebDSL stack in the run-time.

The JVM has a setting for the maximum heap memory size. In older JVM and Tomcat versions, when the application reached this hard limit, it could get stuck trying to access additional heap space and garbage collect. This issue was quite hard to resolve, without killing the JVM process. In the past, it has been the cause of a crashing Tomcat server for our production applications several times. This situation is handled better in current JVM and Tomcat versions.

*Alternate Back-ends* In addition to the Java back-end, we have experimented with an alternate Python back-end targeting the Google App platform (which at that point only supported Python). A major issue that came up was the diverging semantics of the Java and Python back-ends, especially in the storage behavior, where the Google App platform did not allow variations besides its own NoSQL store. Reflecting on the problem of diverging semantics and considering our limited resources, we decided to focus on maintaining one reference back-end. To the user of a web application it does not matter what platform the server-side application is running on, as long as it is available and fast enough.

Tight integration with Java using direct class references (Section 2.5.3) make development of the WebDSL language more incremental. Language features can be developed as a library for WebDSL applications before fully integrating

a feature in the compiler with syntax and static checks. This approach was used in implementing advanced search (discussed in Section 5.3.3). The downside is that direct class references bind the language to this particular platform. Such a language feature makes the migration to other platforms harder. Direct class references should primarily be used for calling into external Java libraries, or for internal functions that call into the WebDSL runtime framework. A typical application should not have to implement application logic using such escapes. An example of when we needed Java code is the back-end of WebLab (see Section 7.3), a component that runs Scala, C, and JavaScript guest code in a locked down environment. Another example is the checkout and data import of subversion and git repositories in Codefinder, a source code repository search engine.

## 2.6 Pages and Templates

A typical web application is divided into several pages. These pages can relate to specific parts of data or functionality, such as managing users or content. They can also be tied to the tasks that a specific user has to perform, such as an administrator or moderator. The mapping of application features to pages is at the liberty of the designer of the application. Navigation is performed through clicking on links in pages, which triggers the browser to move to a new page. Pages are also directly accessible by entering the URL address. This means that pages are bookmarkable, users can store the locations of their favorite pages, or refer others to a specific page by sending just the URL. If an application has separated functionality over many pages, these pages can usually be quite static and simple, i.e. only having a few forms or buttons to invoke actions. On the other hand, web pages can also be very dynamic, a whole web application can even be contained in a single page. In that case, transitioning to other areas of the application is done using the browser's JavaScript engine to send requests to the server, and based on the response the engine inserts, replaces, or removes parts of the current web page.

Most of the complexity in web application development lies in creating user interfaces. The user experience is determined by the choices made in user interface design, where functionality is intertwined with aesthetics. These choices are somewhat limited by having the browser as client application, which is tied to specific user interface technology. HTML is used for page structure and content, CSS for layout and styling, and JavaScript for dynamic behavior. The application server generates the user interface for web applications in the form of HTML, CSS, and JavaScript. Generating these files can be done in any programming language that runs on the server, which is very unrestrictive compared to the client-side situation.

This section introduces user interface template definitions in WebDSL. Forms with data input and the request processing workflow in WebDSL will be covered in more detail in Chapter 3, because it constitutes an important part of the novel web programming features in the WebDSL language.

### 2.6.1 Pages and Navigation

Web applications consist of one or more pages with links between them. The example in Figure 2.9 shows three pages from the example application with their navigation links. The corresponding WebDSL code is shown in Figure 2.10. The home page in the middle contains most links, it can point to both the `createproject` page on the left, and the `view project` page on the right. The rounded boxes at the top indicate that a browser can directly request each page, e.g. if the user types the URL in the address bar. Some pages require arguments. In the example, the `project` page requires the identity of the project to be shown. In the corresponding WebDSL example fragment the pages only contain their navigation links. The `root` page is special in WebDSL, it indicates what page to show when the application root url is accessed without a page name or arguments. Each application requires one `root` page.

Page links in HTML are a textual reference to a page and its arguments. Because such links are a textual representation of a reference, it is possible to create broken links. An abstraction is required to check the validity of navigation links, and generate the textual representation. Page navigation links in WebDSL are checked in the static analysis of the compiler and IDE, and their textual representation is generated.

WebDSL pages are an entry point to the application server, while templates are only accessible through pages and other templates. This distinction between page and template is especially important for security and access control. Before responding to a page request, the server needs to verify correctness of the arguments and decide whether access is allowed based on the authentication data in the request.

A navigation link becomes part of the page HTML and needs to have an encoding of the arguments, suitable for sending to the client. In the case of primitive value types, like `Int` or `String`, this is simply the value itself converted to a `String`. For entity references the `id` property is used in arguments. Handling a page request is done by the WebDSL runtime, which decodes the page arguments from their client representation. In the case of an entity, the entity is loaded from the database using the `id` that is passed as parameter. That also means a navigation link is only valid with entity instances that are persisted. Otherwise the entity with that `id` would not exist anymore when clicking the link to request the page. If an entity was deleted, trying to retrieve the page with that entity as argument results in a 404 error.

### 2.6.2 Templates

Template definitions are reusable parameterized fragments of page content in a WebDSL application. Templates can be divided in smaller templates to improve modularity, readability, and reusability. By simplifying templates and reducing their responsibilities, it becomes easier to abstract over subtemplates and understand everything that is happening on a page. They are syntactically

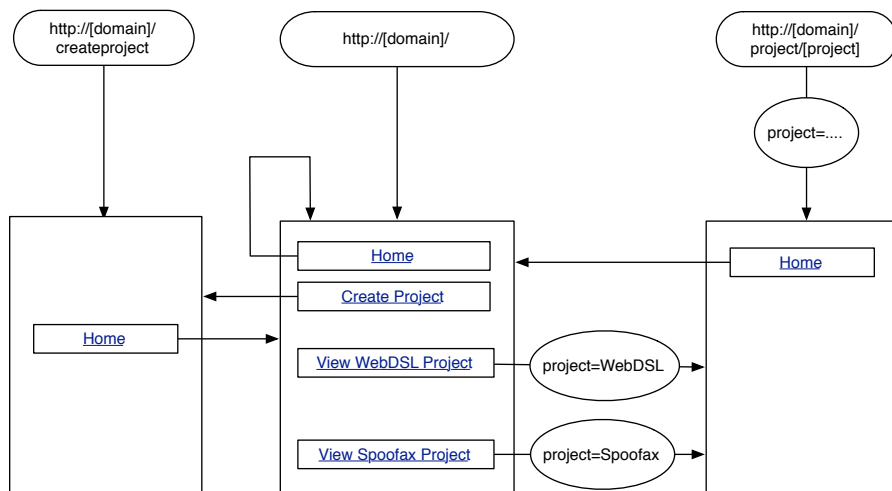


Figure 2.9 Example pages with navigation. This diagram shows the application root page, the create project page, and the view project page, together with their interconnecting page links.

```

1 page root {
2   navigate root { "home" }
3   navigate createproject { "Create Project" }
4   for( p: Project ){
5     navigate project( p ){ ~p.name }
6   }
7 }
8 page createproject {
9   navigate root { "home" }
10 }
11 page project( p: Project ){
12   navigate root { "home" }
13 }

```

Figure 2.10 Corresponding WebDSL code for page navigation example.

and semantically checked for consistency with the rest of the application. For example, template formal arguments use the same WebDSL types as the expression language, and the actual arguments in template calls consist of the same WebDSL expressions that are available to regular functions. The main responsibility of these templates is rendering information to the HTML format. Most CSS and JavaScript content is generic for all generated HTML. Besides rendering, some templates also process information and invoke commands from `form` submits. Input templates and their additional request handling phases are discussed in Section 2.7.

An example of the project view page composed of templates is shown in Figure 2.11. On the left the page is shown with its concrete instantiations of templates. The page argument is the `Spooifax` project, which is loaded from the database. The `showIssues` template retrieves the three issues of the project

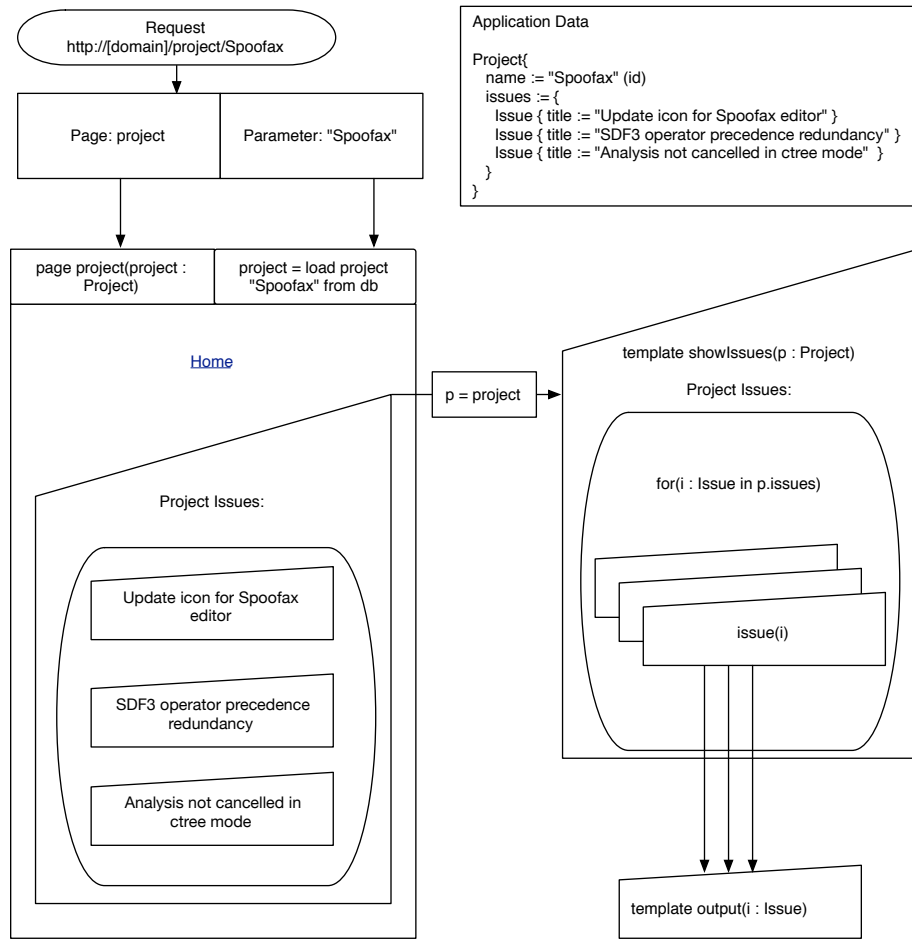


Figure 2.11 Example template composition.

```

1 page project( project: Project ){
2   navigate root { "home" }
3   showIssues( project )
4 }
5 template showIssues( p: Project ){
6   "Project Issues: "
7   for( i in p.issues ){
8     issue( i )
9   }
10 }
11 template issue( i: Issue ){
12   output( i.title )
13 }

```

Figure 2.12 Corresponding WebDSL code for template composition example.

```

1 page root {
2   main {
3     jumbotron {
4       h1 { "Freely manage all your projects!" }
5       lead { "Project issue management with tagging, commenting and search." }
6       par { navigate createproject { "Start your project today" } }
7     }
8     gridrow { gridcolMiddleThird { projects } }
9   }
10 }
11 template projects {
12   pageHeader { "Projects" }
13   tablestriped {
14     for( pr in allprojects() ){
15       row {
16         column { navigate project( pr ){ ~pr.name } }
17         column { ~pr.url }
18       }
19     }
20   }
21 }
22 template main {
23   head {
24     title { "Project Manager" }
25     includeJS( "bootstrap.min.js" )
26     includeCSS( "bootstrap.min.css" )
27   }
28   div[ class = "container" ]{
29     masthead {
30       navigate root {
31         image( "/images/logo-pm.jpg" )[ class = "logonav" ]
32       }
33       navbarRight {
34         navigate signin { "Sign In" }
35         navigate createproject { "Create Project" }
36         navigate signup { "Create Account" }
37       }
38     }
39     messages
40     elements
41   }
42 }
43 htmlwrapper {
44   container div[ class = "container" ]
45   masthead div[ class = "masthead" ]
46   lead par[ class = "lead" ]
47   gridrow div[ class = "row" ]
48   jumbotron div[ class = "jumbotron" ]
49   navbarRight div[ class = "btn-group pull-right" ]
50   gridcolMiddleThird div[ class = "col-lg-4 col-lg-offset-4" ]
51 }

```

---

Figure 2.13 Pages and templates for main layout in example application.

and invokes the `issue` template for each of them. The corresponding WebDSL code is shown in Figure 2.12.

### 2.6.3 Template Calls

Templates are composed with template calls. Templates are parameterized with typed arguments, these parameters are instantiated in the call. The example application page and template code for the main layout of the application is shown in Figure 2.13.

*Elements* Template calls pass on an implicit template `elements` part, which is the defined block enclosed in curly brackets `{}` after the call. This part is scoped within its definition context and is passed on as a closure. A template definition can call `elements` to include these template elements. There are some variations of template call syntax. If there are no arguments the parentheses can be omitted, similarly, if there is no template elements block the curly brackets are optional.

*Attributes* Besides the regular arguments and template elements, templates can be passed extra attributes. These can contain e.g. custom class attributes for one of the HTML tags. Template call attributes are enclosed in square brackets. An element can insert these elements into an HTML tag using `all attributes` as shown in Figure 2.14.

```
template gridrow {
  div[ class = "row" ]{ elements }
}
template div {
  <div all attributes> elements </div>
}
htmlwrapper {
  div div
  myblock div[ class = "my-block" ]
}
```

---

Figure 2.14 All attributes example.

There are some accessor variants to provide more fine-grained control over this inclusion of attribute. For example, `all attributes except "class"` selects all attributes except the one for "class", and an `attribute("class")` function call only retrieves the value of the "class" attribute. The definition of a simple wrapper template that only wraps an HTML tag with some attributes is so common that a shorthand is provided with `htmlwrapper`. It specifies a list of entries with template name, the HTML tag to wrap, and the added attributes. The example `div` template and `div htmlwrapper` entry in Figure 2.14 are equivalent.

*Overloading* While pages must have unique names, templates can be overloaded. The overloading is resolved compile-time, based on the static types of the arguments. A typical overloaded template is the `output` template. For primitive values, it renders the HTML-escaped value. For entities it shows the

name value, which can be changed by defining a custom output for an entity type as shown in Figure 2.15.

```
template output( i : Issue ){
  navigate issue( i ){ ~i.title } // ~i.title is shorthand for output( i.title )
}
```

---

Figure 2.15 Template overloading example.

*Global Overriding* Template and page definitions can be globally overridden using the override modifier, e.g. how to override a built-in page such as `pagenotfound` is shown in Figure 2.16.

```
override page pagenotfound {
  "Unknown project or issue, return to " navigate root { "homepage" }
}
```

---

Figure 2.16 Global override example.

*Local Overriding* Template definitions can be redefined locally in a page or template, to change their meaning in that specific context. All uses are transitively replaced in templates called from the redefining template.

*Multiple Elements Arguments* If multiple template elements arguments are required, they can be added in the list of regular arguments using the `TemplateElements` type. Additionally, an argument list containing a tuple of connected arguments can be declared for passing a variable number of related arguments, including template elements. A typical use case is a template that provides an abstraction for tabbed content, as shown in Figure 2.17.

```
template showStudentCohorts( cohorts: [Cohort] ){
  tabs([ // call tabs template with application-specific content
    ( "home", true, "Home", { panel( "Home" ){ instructions } } )
    , for( sc in cohorts ){ // this for loop is flattened into the argument list
      ( sc.key, false, sc.label, { panel( sc.label ){ showCohort( sc ) } } )
    }
  ])
}
// generic template definition for tabbed content
template tabs( elems: [key: String, active: Bool, label: String, content: TemplateElements] ){
  tabHead {
    for( e in elems ){
      tabLink( e.key, e.active, e.label )
    }
  }
  tabBody {
    for( e in elems ){
      tabContent( e.key, e.active ){ e.content }
    }
  }
}
```

---

Figure 2.17 Variable number of connected arguments example.



## 2.6.4 Template Content

*Output template* The `output` template is defined in the library for every type, and is the default way to insert data values in the page. The text in such static fragments is HTML-escaped, which means replacing `<`, `>`, `"`, and `&` with `&lt;`, `&gt;`, `&quot;`, and `&amp;` respectively. Because `output` is the most commonly used template name, a shorthand is available. The notation `~expr` is equivalent to calling `output(expr)`. Output template definitions are typically overloaded for specific entity types to provide a convenient way to print the entity property data.

*Static text and string interpolation* Static text can be inserted in template definitions using double quotes. These fragments are HTML-escaped. The `~` can be used for string interpolation, e.g. `"name:~ref"`, `"text:~ref.prop"`, `"~call(expr) is the result"`, or `"Result:~(any-expr)"`.

*HTML tags* HTML tags with attributes are specified with tag literals. Attributes can be added, e.g., to set a class value for the tag. The attribute values can be any expression that produces a string. Attributes can also be added conditionally with an `if` construct.

*Choice* Choice is represented with an `if` template element, the `else` part is optional. When there are multiple options based on the value of a specific expression, the `case` template element can be used instead of nesting multiple `if` elements.

*Loops* `for` loops iterate over a collection of entities or primitives. The `for` template element allows optional filtering of elements and an optional separator. See example in Figure 2.18.

```
for( pr in allprojects() ){
  row {
    column { navigate project( pr ){ ~pr.name } }
    column { ~pr.url }
  }
}
```

---

Figure 2.18 For loop example.

*HTML Escaping* Escaping HTML tags and markup is necessary to prevent HTML and JavaScript injection attacks. When user content is not HTML-escaped before displaying it on a page, all the tags that are in the content are processed as regular HTML. First of all this can cause page rendering to break, because of tags in the content transforming the page into an incorrect HTML document. A broken page is the least of your worries though. A malicious user can inject JavaScript code into an unescaped value, meaning that whenever another user navigates to a page with that output, some JavaScript code is executed which is crafted by the malicious user. This JavaScript code can do many things like hijack the session cookie to gain access to the account of another user, or invoke application actions directly on behalf of the other user.

To further mitigate the risk of session hijacking, the `HttpOnly` flag can be used for cookies, which disallows the JavaScript engine access to cookie data.

In some cases, the HTML should not be escaped, e.g. when entities are used to cache actual page content. This should only be allowed if the users with access to such features are trusted (e.g. an admin) or the feature is used by the application internally. The `rawoutput` templatecall renders its argument without escaping. Using `rawoutput` with anything else than a String literal is marked with a warning to make developers cautious when using this feature.

*Styling* Templates `main` and `logo` in the example are part of a standard template that is used for all pages in this application. The `elements` call in `main` is where the template elements passed to `main` are inserted. `navigate` template elements construct a page url from a page call. Templates shown in Figure 2.19 such as `jumbotron`, `header`, `lead`, `gridrow`, `gridcolthird`, `container`, `masthead`, `justifiednav`, and `navitem` are small templates that construct an HTML div or span tag with specific class attributes for Bootstrap CSS styling rules.

```
htmlwrapper {
  container div[ class = "container" ]
  masthead div[ class = "masthead" ]
  lead     par[ class = "lead" ]
  gridrow  div[ class = "row" ]
}
```

---

Figure 2.19 Defining templates to assist styling.

## 2.7 User Interface Input

The template elements discussed until now cover rendering of static and dynamic page content. Template compositions provide flexibility in constructing application pages. They are no less expressive than regular function calls, but also automatically take into account escaping of values inserted into the page. This section covers another core aspect of templates in WebDSL, which is user input and action invocation.

In order to handle user input in a web application, pages must render forms with input tags. The name attribute of these tags is used to pass the entered values in a request to the application as name-value pairs. The application must analyze these name-value pairs and decide on an appropriate function to execute. The main problem here is the correct reconstruction of the arguments. The server must look for the right names in the submitted data. If the matching is implemented by the application developer, there is an opportunity for faults. In many frameworks that force an MVC separation of view and controller, the developer must correctly match names for inputs in view and controller. Besides being error-prone, it also adds to the amount of code required to get a functional form. In Ruby on Rails, this can be addressed by having form helper functions (e.g. `form_for` and `fields_for`) that automatically put submitted data back into model objects (input names need to match object

property names). Unfortunately, because controllers have no direct relation to views, they will accept any argument as long as it matches some property of the model object. This means that it is easy for a client to make a fake action request that sets additional data that was not available for entering as input. To address this problem an extra step is needed in the controller for whitelisting parameters, which they named strong parameters. The solution to avoid defining input names is in this case causing accidental complexity of its own.

Existing solutions to avoid input name matching, such as form helper functions, have another downside. They make it harder to reuse template code related to forms. By having a direct relation between form inputs and model objects in the input names, they cannot be easily combined. For example, if part of a form is repeated for each object in a collection, each name will need to receive a unique identifier in addition to its model property name. The controller also becomes more complex because it will receive different name parameters and needs to reconstruct the collection from it. Helper functions for collection properties exist in Ruby on Rails (`collection_select`, `collection_check_boxes`, `collection_radio_buttons`), but these only edit the collection itself and not properties of the objects in the collection. It is not possible to reuse a form component for editing a single object as a repeated form component for objects in a collection, because input names are generated in each helper function separately. Each iteration will use the same input names. A new helper function would need to be constructed that is aware of the other input names, potentially for each combination of inputs in a form.

Another problem is that readable and consistent input names are a security vulnerability. XSS and CSRF attacks can abuse the guessable nature of input names to construct fake action requests. The delivery of fake action requests is different between these attacks. XSS abuses unescaped display of user-inputted values. By inserting JavaScript commands, a victim user visiting the page would run this JavaScript code in the browser. This could make a request to some malicious server to steal the login cookie or other user data, but can also make a request to the application itself. A CSRF attack abuses the browser behavior of automatically including the session cookie with any request to the application. If a crafted url with action parameters is inserted in a malicious web page, just visiting that page would trigger the action request with the credentials of the currently logged in user. To limit the danger of both types of attacks it is important that input names are not guessable. That means no meaningful names and different for each user. Having such uniqueness in input names is an improvement, although for XSS it does not provide complete protection as such a vulnerability is much more dangerous than CSRF. For CSRF protection a single unique form input or authentication token can also be used. With XSS just having unique input names is not enough to prevent the attack, as JavaScript code could also scan the page for input names.

```

1 page newproject {
2   var p := Project{}
3   mainMiddleRow {
4     h2 { "Add Your Project" }
5     form {
6       formgroup {
7         label( "Project name" ){
8           input( p.name )[ placeholder = "Enter project name" ]
9         }
10      }
11     formgroup {
12       label( "Project website" ){
13         input( p.url )[ placeholder = "Enter project website URL" ]
14       }
15     }
16     formgroup {
17       label( "" ){ input( p.private ) " private project" }
18     }
19     submit action {
20       p.members.add( principal );
21       p.save();
22       message( "Your project has been created!" );
23       return project( p );
24     }{ "Add project" }
25   }
26 }
27 }
28 page newissue( p: Project ){
29   request var i := Issue{}
30   mainMiddleRow {
31     h2 { "Report issue for project ~p.name" }
32     form[ role = "form" ]{
33       formgroup {
34         label( "Title" ){ input( i.title )[ placeholder = "Enter issue title" ] }
35       }
36       formgroup {
37         label( "Description" ){
38           input( i.content )[ placeholder = "Enter issue description"
39             , oninput = replace( preview ) ]
40         }
41       }
42       formgroup {
43         label( "Description Preview" ){
44           placeholder preview {
45             ~i.content
46           }
47         }
48       }
49       submit action {
50         i.project := p;
51         i.save();
52         message( "Your issue has been reported!" );
53         return issue( i );
54       }{ "Report issue" }
55     }
56   }
57 }

```

---

Figure 2.20 Example application create project and create issue pages.

### 2.7.1 Forms and Input

Form template elements handle user input. Figure 2.20 shows the `create project` and `create issue` pages for the example application. The `form` element acts as a scope to group `input` elements together. `input` template elements create user input fields, text areas, checkboxes, radio buttons, drop-downs and other HTML input widgets. The type of widget is influenced by the type of the argument, using overloading of the `input` template definition. Inputs handle the rendering of the widget, displaying the current persisted value, and binding the data back to the data model in a form submit. Validation rules are also enforced for input elements. `submit` and `submitlink` elements render a submit button which can trigger a form submit and finalize the changes with a function. Input and submit names are generated automatically, so that they are unique, deterministic, and hard to guess for CSRF attacks. The submit action code can perform additional operations, pass along messages to be shown on the next page, and set the redirect page for when the request is completed. Submit action code can also be defined separately inside a template using an `action` declaration, and called from the submit element. If no additional actions are necessary besides form data binding and updating input values, the action code part can be omitted from the submit.

*Input Templates* Input templates perform data binding. When rendering the input, the current persisted value is shown. Before invoking an action, the entered values are inserted into the data model in the transaction. After the data binding phase, the action is invoked, for which the submit was triggered.

The `input` templates create input form elements appropriate for each WebDSL type. The default widgets are text field for `String` and `Int`, text area for `Text`, checkbox for `Bool`, dropdown select for entities, checkbox selection for sets of entities, a list editor for lists of entities (allows adding and removing items, and reordering), and a date/time picker for date types. These templates can be overridden, or a variant can be defined, to customize the input widget used for databinding of a value. Some of the mentioned inputs map directly to a single HTML input tags, while others need some additional template elements to handle an input. For example, date/time pickers for date types consist of several HTML elements and some JavaScript for creating a popup when the input field is clicked.

*Template Variables* Template definitions can contain variable declarations. These template variables are used to create a new entity instance or store temporary template data. They can be used as the argument to input templates enabling users to edit the values through data binding.

## 2.8 User Management and Access Control

Access control policies determine whether a specific user has access to a specific resource. The current user is represented by a principal entity in the

system. An entity can be marked as principal entity through the `principal is` declaration.

```
principal is User with credentials username, password
```

---

Figure 2.21 Principal declaration, enables access control, marks user-identifying entity and authentication properties.

A default authentication form is generated for the indicated properties of that entity. For example, the principal entity in this application is `User`, and the fields used for authentication are `username` and `password`. This generates an `authenticate` function that takes care of password hash comparison for properties of type `Password`. It also generates `authentication`, `login`, and `logout` templates to quickly prototype an application.

Access control rules specify the policy for pages and templates. When access control is enabled (by specifying the principal), pages are by default not accessible without a rule. The reason is that they have direct URLs associated with them, which could easily become security leaks if access was allowed by default. Templates can only be included in pages and other templates, these are accessible by default. Allowing access to templates by default is done for convenience, because applications typically have many small templates. A page rule determines whether access is allowed, possibly based on the page arguments. A template rule will exclude templates from pages when the rule evaluates to false. Any form input or submit action on a page is only accessible if the page or template itself is accessible, because databinding and action handling are handled while traversing the page and template control flow. The server decides whether these parameters that change data and invoke submits are actually available to the current user. For this reason access control rules for pages are enough to cover a complete application. In many web frameworks, where actions have separate URLs, access control checks would be required in all form action handlers separately as well. Figure 2.23 shows the example application `Sign Up` and `Sign In` pages, as well as the page access control rules.

*Session Entity* A special type of entity is the session entity, used for storing data in the user session. These are similar to regular entity definitions. However, they declare a variable as well as an entity type. Most importantly, a session entity is used to store the currently logged in user. The session entity that is implicitly defined through the principal declaration in the example app is shown in Figure 2.22.

```
session securityContext {  
    principal : User  
}
```

---

Figure 2.22 Implicit session entity that holds the principal entity of the current user.

The session entity name is a globally visible variable in the application code. The entity object is automatically instantiated and saved, one for each browser

```

1 page signup {
2   var u := User{}
3   mainRowMiddle {
4     h2 { "Sign up" }
5     form {
6       formgroup {
7         label( "Username" ){
8           input( u.username )[ placeholder = "Enter your desired username" ]
9         }
10      }
11     formgroup {
12       label( "Email" ){
13         input( u.email )[ placeholder = "Enter your email address" ]
14       }
15     }
16     formgroup {
17       label( "Password" ){
18         input( u.password )[ placeholder = "Enter your password" ]
19       }
20     }
21     submit action {
22       u.password := u.password.digest();
23       u.save();
24       principal := u;
25       message( "Your account has been created! You have been logged in! ");
26       return root();
27     }{ "Create your account" }
28   }
29 }
30 }
31 page signin {
32   main {
33     authentication
34   }
35 }
36 principal is User with credentials username, password
37 access control rules
38 rule page root { true }
39 rule page signin { true }
40 rule page signup { true }
41 rule page projects { true }
42 rule page newproject { true }
43 rule page newissue( p: Project ){ p.isAccessAllowedTo( principal ) }
44 rule page issue( i: Issue ){ i.isAccessAllowedTo( principal ) }
45 rule page project( p: Project ){ p.isAccessAllowedTo( principal ) }

```

---

Figure 2.23 Example application access control.

session accessing the application. Typically, session data is used for keeping track of authentication state, but it can also be used for temporarily storing data for anonymous users. For the generated `securityContext` session entity specifically, a shorthand `principal` can be used to refer to `securityContext.principal` in application code.

## 2.9 Search

SQL queries provide a basic facility for searching entities, through specifying an exact value or subvalue of a text property. This type of search is quite limited, e.g. it does not find values that differ only slightly because of inflection of words. Additionally, there is no convenient way to give priority to certain properties such as the name, when searching in multiple properties at once. Typical search features such as spell corrections and auto complete suggestions would have to be build from scratch.

The search user interface minimally requires an input field with `onkeyup` event or submit button. Typically there is some usage of partial page updates to show results and suggestions without unnecessary page reloads for the user who is entering a search query. The user interface could show search refinement options through faceting. Faceting is a dynamic way of presenting navigation and search filter options when browsing a collection of items, e.g. filtering closed issues, issues from a certain date range, issues with a certain amount of comments.

To address search application concerns, we designed and implemented an abstraction for search in WebDSL. Search mappings describe what entity properties can be searched. They also configure what kind of tokenization and normalization will be performed on the values, such as stemming and removing stop words. Entity properties can be analyzed in multiple ways, e.g. as whole matches, splitting the words, or matching individual n-grams of letter combinations. Different analyzers can be combined to get search results, giving higher ranking to results that match on multiple analyzers. Analyzers get applied on the indexed data, and on the query that searches the data. The features in the search sublanguage allow for customization based on the search requirements. For example, we investigated search optimized for source code, where parentheses and other meaningful characters in code are part of the search query and indexed data instead of getting filtered out [Van Chastelet 2013]. The live version of this application is a helpful programming tool we still actively use [Reposearch 2011]. Autocompletion can also be customized based on specific analyzers, e.g. choosing either to complete words or complete titles. Namespaces are used to separate search indexes, to be able to have local search specific to the context that is being viewed. Figure 2.24 shows the search mappings and issue search page of the example application. In this example, the project is used as namespace to avoid polluting the completion and results with issues from other projects.



```

1 extend entity Issue {
2   projectname : String := project.name
3   search mapping {
4     title using none as originaltitle // exact match (no analyzer)
5     title (autocomplete) // default analyzer (filtering and split words)
6     title using ngrams as titleletters // ngrams is custom analyzer defined below
7     content using ngrams
8     namespace by projectname // separate search and completion indexes per project
9   }
10 }
11 analyzer ngrams { // search individual letters and groups of letters
12   tokenizer = StandardTokenizer
13   token filter = LowerCaseFilter
14   token filter = NGramFilter( minGramSize = "1", maxGramSize = "30" )
15 }
16 page search( p: Project, query: String ){
17   request var s := query;
18   main {
19     form {
20       label( "Search query" ){
21         input( s )[ onkeyup = replace( completions, results ); ]
22       }
23     }
24     placeholder completions {
25       for( result in (Issue completions matching title: s in namespace p.name limit 5) ){
26         div { navigate search( p, result ){ ~result } } // complete on words in title
27       }
28     }
29     label( "Results" ){
30       placeholder results {
31         for( result in results from search Issue in namespace p.name matching s ){
32           div { navigate issue( result ){ ~result.title } } // search results
33         }
34       }
35     }
36   }
37 }

```

---

Figure 2.24 Example application search page.

*Partial Page Updates* The WebDSL language integrates a mechanism for updating a page partially through an AJAX request. The `placeholder` keyword (lines 24 and 30 in Figure 2.24) marks a page fragment for which the rendering can be updated without updating the rest of the page. As an action, placeholders can be rerendered through a `replace` call (line 21). Entering characters in the search input field triggers a request to update the `completion` and `result` placeholders. The `request var` on line 17 marks a variable for use in rendering with transient data, the results will render with the value of `s` equal to the search query. This language feature is further discussed in Section 3.7.

## 2.10 Discussion

This chapter contains an overview of the WebDSL web programming language that addresses problems encountered in the domain of web programming. This section discusses concerns that can be raised with the solution and encountered issues when applying WebDSL in practice.

*What about other sublanguages?* The set of sublanguages is not necessarily complete or set in stone, since common concepts in web programming evolve over time. The core WebDSL system can be considered as data models, user interfaces, and functions. Access control, data validation, and search were created as extensions originally. Not every application will require all of these subsystems, as it is perfectly possible to create an application without persisted state or a service app with only a JSON interface.

Features that are not available in WebDSL are easy to add in. Java classes and methods can be made accessible from WebDSL to benefit from Java's library ecosystem. Client-side JavaScript code can be wrapped in WebDSL templates to be able to reuse JavaScript libraries and widgets.

*Does it work in practice?* There are several additional concerns that need to be taken into account to make this solution practical. In particular the ease of use to get your program to run, and compiler performance have been important aspects which require engineering effort. Given that many web programmers are used to the deployment speed of scripting language interpretation, i.e. immediately deployed, any time spent waiting for compilation and deployment is a deterrent for practical use. Chapter 6 discusses implementation aspects of the WebDSL compiler, and several measures for improving the speed of compilation. We have successfully used WebDSL for creating several real-world applications with thousands of users. Chapter 7 shows application statistics and provides a discussion of our experiences with the practical applicability of WebDSL.

*Are applications written in WebDSL completely secure?* While WebDSL avoids common web programming security problems, application security depends on every piece of software that is on the production server of the deployed application. A misconfiguration of the server easily leads to security problems of the application. Therefore, it is hard to give guarantees about security. Even libraries that are as widespread as OpenSSL suffer from zero-day vulnerabilities, as became clear again with the Heartbleed [Synopsys 2020] vulnerability in 2014. On the positive side, because of the high-level specification of web applications in WebDSL, there is less code to comprehend when trying to grok an application. Furthermore, security fixes can be applied in the WebDSL compiler, while all applications only need a recompile with the patched compiler. In general, any application used in production requires continuous maintenance when it comes to security. In Section 7.8 we discuss more examples of security issues we have encountered in practice.

*How does it compare to other solutions?* In Chapter 1 we have looked at existing languages and frameworks for web programming and discussed their problems. The WebDSL design is based on these observations and attempts to address common problems. In Chapter 8 we compare WebDSL with other approaches found in literature.

## 2.11 Conclusion

In this chapter we provided an overview of WebDSL, covering the language and implementation considerations. The WebDSL web programming language and system is unique in its focus on linguistic abstractions for web programming. The language addresses the observed problems in web programming (Chapter 1), providing new abstractions for web programming concerns, static verification for those abstractions to find inconsistencies, and taking security into account in the design of the language and the resulting generated code and runtime. In the remaining chapters of this dissertation, we will cover key elements of WebDSL, and assess its usage in a number of production applications serving thousands of users (Chapter 7). Chapter 3 dives deeper into user interface definitions in WebDSL, explaining the request processing lifecycle with multi-phase evaluation, because it constitutes an important part of the novel web programming features in the WebDSL language.

# User Interface Templates

---

# 3

## 3.1 Introduction

Parameterized HTML templates are a common component to encode pages in web application frameworks, however, templating libraries are often lacking in flexibility compared to regular functions in a general-purpose language. Another issue is the lack of integration with the expressions of the host programming language. How WebDSL handles the issues related to template composition and integration with the expression language is explained in Chapter 2. That chapter covers the output of data with automatic escaping of HTML tags in the data to prevent Cross-Site Scripting. Input handling is more complex than just rendering data.

This chapter examines in more detail the design of user input handling in templates. User interaction occurs in the form of button presses, form data entry, and other input events such as mouse hover. Data validation is tightly connected to user input. The user must be notified and the operation blocked when the entered data is not valid. Furthermore, partial page updates using AJAX requests increases the possibilities for user input and immediate feedback.

Section 3.2 provides a general overview of the improvements over other solutions in the design of the WebDSL user interface language. WebDSL's approach for handling user interface templates is unique and not found in other solutions. Instead of using separate action handler functions with manually specified input identifiers, WebDSL provides abstraction over the construction of input names, and links actions to the context of the user interface definition. This approach avoids any inconsistency issues that can occur with separately defined action handlers and manually constructed input names, as well as avoiding inconsistencies in enforcing access control when handling form submits. The related work for user interface languages is further discussed in Chapter 8. In that chapter, a detailed analysis of a user interface definition is made for a small application written in the Django framework in Section 8.2.1, which is compared to the equivalent WebDSL code. In Section 8.3.1 several user interface examples from the Ur/Web language are compared to WebDSL, and in Section 8.3.2 the Links language is discussed in a similar way.

Section 3.3 explains the request processing lifecycle, consisting of databind, validate, action, and render, which is essential for understanding the semantics of user interface templates. Section 3.4 provides an example of the multi-phase evaluation, and how the template elements involved with user input, namely forms, actions, input templates, and template variables, are handled in each phase. Section 3.5 explains the language primitives required for expressing various input templates as reusable and reconfigurable library

components. These are phase function hooks, ref types, and template identifier generation. Section 3.6 covers the abstractions for various forms of data validation: value well-formedness, data invariants, form input validation, and function assertions. This section also extends the example of multi-phase evaluation with data validation, and gives an example of library input code that manages data validation. Section 3.7 goes over two different modes for partial page updates, namely nested pages and inline refresh. Additionally, an example library input with validation using partial page updates is discussed. Finally, Section 3.8 concludes this chapter on user interface templates.

## 3.2 Design Goals

WebDSL user interface templates define what information is visible on pages and how the persisted state can be altered through user input forms. The semantics of templates is based on multiple phases of execution: databind, validate, execute action, and render. User input is handled through form and input template elements. Input names used for submitting data are generated by the WebDSL compiler. The data validation checks can be value well-formedness checks, data invariants, input assertions, and action assertions. Partial page updates are supported to create dynamic pages with AJAX updates. The standard library contains output and input components, with on-submit or immediate validation feedback.

We set out to achieve the following goals with the WebDSL user interface:

- *Form submits that are safe from hidden data tampering* Input templates define exactly what values can be edited on a page. Entities are never automatically created from blindly accepted input parameters. The control flow of the page and templates determines what inputs and submits can be accepted. This avoids problems related to tampering, where additional values are inserted and accepted because an entity is automatically filled based on request parameter names. This aspect is based on a request processing lifecycle (Section 3.3) in which user interface template code is traversed in multiple phases to handle form inputs and actions (Section 3.4).
- *Prevention of input identifier mismatch in action handlers* Application developers do not have to specify name identifiers for HTML inputs, these are generated automatically. Input template definitions take care of matching the identifiers when reading submitted data and performing data binding to update values in the data model. This prevents faults related to mismatching identifiers in action handlers entirely. Instead of separate action handling code, actions are designed to be executed in the context of a user interface definition, as part of the multi-phase evaluation (Section 3.4).
- *Safe composition of input templates* Because input names are generated by the WebDSL runtime, individual input templates do not have the problem

of causing overlapping names in the generated page. Inputs can be safely repeated in `for` iterations without any changes. The construction of unique input names is provided through language primitives, described in Section 3.5.

- *Automatic enforcement of Cross-Site Request Forgery protection* Input names are generated from the specific data they edit, and on the identity of the logged in user. This aspect avoids guessable input names that could be used to construct malicious CSRF requests. This aspect is provided through the primitives for template identifiers (Section 3.5.3).
- *Expressive data validation* Data validation is expressed in terms of the WebDSL models for entities and expressions. Validation rules are specified in a coherent way, whether they are form input validations, data invariants, or function assertions. Validation rules can depend on multiple properties, traverse the entity graph, and request additional information from the database without restrictions. Rendering error messages is automatic, and can also be customized. Data validation is integrated in the request processing lifecycle and multi-phase evaluation of WebDSL (Section 3.6).
- *Partial page updates without explicit JavaScript or DOM manipulation* Partial page updates are incorporated in the model for request handling. Page replacements are specified in terms of placeholders and template elements. Application developers do not have to write additional JavaScript code to handle AJAX requests and DOM replacements. The design for integrating partial page updates is explained in Section 3.7.
- *Partial page updates that are safe from hidden data tampering* The abstraction for partial page updates does not send entity data directly to the client, instead it works with HTML fragments. This means there is no need for additional server-side validation checks. This would be necessary if entities are sent as JSON objects to be edited on the client, and then submitted as a whole without knowing what operations were performed. The current features for partial page updates handle updates and rendering on the server, and through access control rules check whether access to functions or data is allowed (Section 3.7).

### 3.3 Request Lifecycle

Handling a page request is performed in several phases. The phases are shown in Figure 3.1. Data validation failures can stop the processing of a form submit and continue with the render phase to show error messages.

*Convert Request Parameters* Users interact with web applications through the browser. This process consists of request and response strings being exchanged between the web server and the browser. A form is defined by a response string, which is interpreted by the browser to produce components that allow user interaction. A user can fill in data in a text field, and press

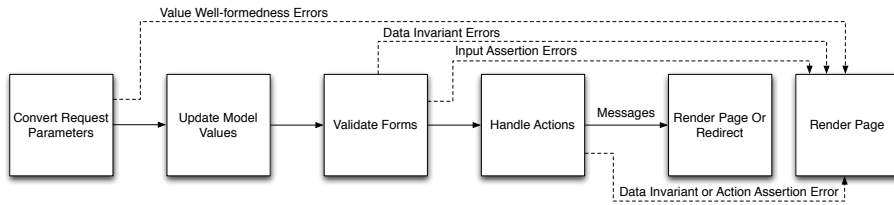


Figure 3.1 WebDSL request lifecycle.

the submit button. The browser first collects the data from the form input fields, and constructs a request string to send to the web server, which receives the request string and parses it. Values from input fields can be accessed separately but are represented as strings. The web application converts these strings to actual types to be used in further processing of the request. Since such conversions are common in web applications, they are typically directly supported in frameworks. The supported types are the native types available in the language used to build the framework. WebDSL extends the usual set of primitive types with domain-specific types such as `Email` and `Secret`. Implicit conversions from and to strings for these types are supported in the language itself.

Request parameter conversion is not possible if the incoming value is not well-formed. For example, a value of "3f" cannot be converted to an integer. Since a failed conversion invalidates any input it is not necessary to update the model before re-rendering the page with error messages. The `Value Well-formedness Errors` arrow indicates this situation. In a page render resulting from validation errors, input components that were submitted restore the submitted value instead of the original value. This allows a user to fix the entered data.

*Update Model Values* In the previous phase, parameters are decoded from strings. In the 'Update Model Values' phase, these parameters are automatically inserted in data model entities. WebDSL supports such *data binding* through `input` elements. For example, the `input(u.email)` element declares that an input field should be displayed with the current contents of the email property of variable `u` of type `User`. Furthermore, when a user submits the containing form with a new value in the email field, the new value will be assigned to `u.email`. An action finalizing this operation just needs to save the variable `u` in order to persist the new email address.

*Validate Forms* The changes made through data binding have to be validated, this is performed after data binding for the whole form is completed. When an entity property is being validated, each validation rule defined on that property is checked, possibly producing multiple error messages. Besides entity validations, there can also be validation rules in pages which need to be enforced. The 'Validate Forms' phase traverses the form that is submitted and checks any validation it encounters. When at least one validation fails during

this phase, further processing is disabled and errors are displayed, indicated by the `Data Invariant Errors` and `Input Assertion Errors` arrows.

*Handle Actions* When all validation checks in previous phases have succeeded, the selected action is executed. During the execution of an action there can be action assertions that validate the data in the current execution state of the action. Moreover, data invariants are still checked during this phase and can produce validation errors as well. If any validation check fails, the entire action is cancelled. This means all the changes to the data model are reverted and rendering is initiated (`Data Invariant` or `Action Assertion Error` arrow). Only one error can be produced at a time since action processing will not continue when a validation fails.

*Render Page or Redirect* Validation messages produced in the previous phases result in a re-render of the same page with error messages inserted. If all validations succeed, the action results in a redirect to the same or a different page, possibly sending messages along which describe successful execution of the action (`Messages` arrow).

*WebDSL Phases* In WebDSL, both request parameter conversion and update model values are performed in the databind phase. These steps do not depend on the surrounding elements and can be performed in one traversal of the templates. The four resulting phases that are used in the WebDSL runtime are:

- databind
- validate
- action
- render

## 3.4 User Input

The main components for user input, as introduced in Chapter 2, are forms, submit functions, input templates, and template variables.

*Forms* The `form` template element enables and scopes user input. It groups input elements that belong together when a user submits the form. Input template elements perform automatic data binding upon submit, which means the values of the entities are updated in-memory with the new values entered in the form inputs. Also, when rendering the form initially, the old values are shown. A form submit happens when a submit button or submit link is clicked.

*Actions* The `submit` and `submitlink` elements represent a submit button or link connected to an action. Actions perform further processing after handling databinding and validation of the form. Actions can be anonymous inline statement blocks or named definitions in the template. Alternatively, a form submit action can be invoked based on an HTML DOM event such as `oninput` or `onclick`.



*Input Templates* Input templates perform data binding, before invoking an action, the values entered are inserted into the data model in the transaction. After the data binding phase, the action is invoked, for which the submit was triggered.

*Template Variables* Template definitions can contain variable declarations. These variables are used to create a new entity instance or store temporary template data. They can be used as the argument to input templates enabling users to edit the values through data binding.

### 3.4.1 Multi-phase Evaluation

Figure 3.2 shows an example of multi-phase evaluation of templates. The Project entity together with initialization data is shown at the top. The project entity has two properties, a name and a set of subprojects. The init block contains application initialization that is run when the application is started with an empty database. The first request to the `editProject` page renders a form that shows the current values of the project names that are subprojects of the "WebDSL" project. The second request happens when the input values are edited and the save button is pressed. The request contains a map of the form input names and their values. In this example, the input `name` attribute values are simplified for clarity. On a form submit, `databind` is the first phase that gets executed. In this phase the in-memory data model is updated with the submitted values. Next, the action belonging to the submitted button is executed. When the action finished successfully, the updated values are persisted to the database and a redirect is sent to the client. By default a redirect will be to the same page. The new values will be visible in the rendered form of the third request.

## 3.5 Language Primitives for Input Implementation

To be able to define input templates, some additional language features are required. These are phase function code, `ref` types, and template identifier generation.

### 3.5.1 Phase Function Code

To implement input templates as library components, we need to be able to execute commands in specific phases of the request lifecycle. WebDSL provides this hook in the form of blocks inside templates, named `databind`, `validate`, `action`, and `render`. Input templates use the `databind` block to update the value of the input argument. For example, the input for `Int` values performs a `databind` in which the submitted value is attempted to be parsed as an integer. This is shown in Figure 3.3.

The utility `String.parseInt` method returns `null` when parsing fails. Variable `req` is initialized when the first phase, `databind`, is executed. `getRequest-`

<pre>entity Project {   name : String (id)   apps : {Project}   validate(name.length &gt; 0,     "Project name may not be empty") }</pre>	<pre>init {   var yg := Project{ name := "yellowgrass" };   var wl := Project{ name := "weblab" };   var cf := Project{ name := "codefinder" };   var re := Project{ name := "researchr" };   var webdsl := Project {     name := "WebDSL"     apps := {yg, wl, cf, re}   } }</pre>
---	---

Request URL 1: editProject/WebDSL

Render	<pre>page editProject(p:Project){   form{     output(p.name)     for(app in p.apps){       input(app.name)     }     submit save() {"save"}   } }</pre>	<pre>&lt;form name="c1" action=".../editProject/WebDSL"&gt;   WebDSL   &lt;input type="text" name="c2f1" value="yellowgrass"/&gt;   ...   &lt;input type="text" name="c2f4" value="researchr"/&gt;   &lt;input type="submit" name="c3" value="save"/&gt; &lt;/form&gt;</pre>
--------	---	--

Request URL 2: editProject/WebDSL

submit arguments: c1 = "1" c3 = "save"	c2f1 = "Yellowgrass" c2f2 = "Weblab" c2f3 = "Codefinder" c2f4 = "Researchr"
--	--

Databind	<pre>form{   output(p.name)   for(app in p.apps){     input(app.name)   }   submit save() {"save"} }</pre>	<pre>if(requestparam("c1") == "1"){   yg.name := requestparam("c2f1")   ...   re.name := requestparam("c2f4") }</pre>
Action	<pre>form{   output(p.name)   for(app in p.apps){     input(app.name)   }   submit save() {"save"} }</pre>	<pre>if(valid &amp;&amp; requestparam("c1") == "1"){   if(requestparam("c3") == "save"){ save(); } }</pre>

Redirect Request URL 3: editProject/WebDSL

Render	<pre>form{   output(p.name)   for(app in p.apps){     input(app.name)   }   submit save() {"save"} }</pre>	<pre>&lt;form name="c1" action=".../editProject/WebDSL"&gt;   WebDSL   &lt;input type="text" name="c2f1" value="Yellowgrass"/&gt;   ...   &lt;input type="text" name="c2f4" value="Researchr"/&gt;   &lt;input type="submit" name="c3" value="save"/&gt; &lt;/form&gt;</pre>
--------	--	--

Figure 3.2 Data input with multi-phase evaluation example.

```

1  template input( i: ref Int ){
2    var req := getRequestParameter( id )
3    < input
4      name = id
5      if( req != null ){ value = req } else { value = i }
6      all attributes
7    />
8    databind {
9      if( req != null ){ i := req.parseInt(); }
10   }
11 }

```

---

Figure 3.3 Example form input databind.

`Parameter(String)` returns for its argument name the value of the submitted parameters in the request. `id` retrieves the generated unique template identifier which is further explained in Section 3.5.3.

### 3.5.2 Ref Types

The `ref` marker on a type allows primitive types to be passed by reference. Regular template arguments without `ref` are not directly assignable. An entity property access passed as a `ref` argument, e.g. `project.name`, contains a reference to the owning entity. This can be used to inspect annotations and check validations declared on the property. The compiler checks whether the call contains an argument that can be a `ref` type. The value is automatically boxed at the call site. Assigning to a `ref` type argument has the effect of assigning to the actual property in the entity. The developer only has to declare the `ref` type in the formal argument of the template, the call site does not require a special marker. The example in Figure 3.4 shows a simple input template call and how the input updates the `ref` type value to assign the entered name to the `u.name` argument.

```

1  template exampleForm {
2    var u := User{}
3    form {
4      input( u.name )
5      submit action{ }{ "save" }
6    }
7  }
8  template input( s: ref String ){
9    ...
10   databind { if( req != null ){ s := req; } }
11 }

```

---

Figure 3.4 Example ref type argument.

Using phase hooks and `ref` types, a simplified version of the standard library inputs for primitive types (without validation) can be constructed. This is shown in Figure 3.5. In Section 3.5.3 we will discuss the generation of template identifiers, which are accessed through the `id` keyword.

```

1  template input( i: ref Int ){
2    var req := getRequestParameter( id )
3    < input
4      name = id
5      if( req != null ){ value = req } else { value = i }
6      all attributes
7    />
8    databind { if( req != null ){ i := req.parseInt(); } }
9  }
10 template input( s: ref String ){
11   var req := getRequestParameter( id )
12   < input
13     name = id
14     if( attribute( "type" ) == "" ){ type = "text" }
15     if( req != null ){ value = req } else { value = s }
16     all attributes
17   />
18   databind { if( req != null ){ s := req; } }
19 }
20 template input( b: ref Bool ){
21   var rnamehidden := id + "_isinput"
22   < input type = "hidden" name = id + "_isinput" />
23   < input type = "checkbox"
24     name = id
25     if(  getRequestParameter( rnamehidden ) != null
26         && getRequestParameter( id ) != null
27         || getRequestParameter( rnamehidden ) == null
28         && b ){
29       checked = "true"
30     }
31     all attributes
32   />
33   databind {
34     var tmp := getRequestParameter( id );
35     var tmphidden := getRequestParameter( rnamehidden );
36     if( tmphidden != null ){
37       if( getRequestParameter( id ) != null ){ b := true; } else { b := false; }
38     }
39   }
40 }
41 template input( i: ref Float ){
42   var req := getRequestParameter( id )
43   < input
44     name = id
45     if( req != null ){ value = req } else { value = i }
46     all attributes
47   />
48   databind {
49     if( req != null ){
50       i := req.parseFloat();
51     }
52   }
53 }

```

---

Figure 3.5 Library input templates for primitive types without validation.

### 3.5.3 Template Identifier Generation

A typical problem in existing web programming frameworks is caused by the composition of input templates into a single HTML structure. This structure contains identifiers and names that should be unique, however, the template libraries usually leave this entirely to the developer. The effect is that subtle issues arise when composing templates, e.g. iterating over a datepicker template multiple times and finding that only the first one is working. These identifiers are automatically generated in WebDSL. The application developer is not exposed to the low-level decisions for name parameters of inputs. To automatically generate identifiers for templates, several requirements need to be taken into account:

- *Deterministic*: subsequent requests should look for the parameter names that were generated during the preceding render.
- *Unique*: parameter names should be unique for each input element on a page.
- *Protect against parameter tampering*: forging requests should not enable more functionality to the user.
- *Protect against Cross-Site Request Forgery (CSRF)*: the identifiers should be different depending on the data referred to and the user requesting it, in order to avoid forged requests from malicious websites.

Any value that is sent to the client is considered user input that needs to be checked. WebDSL sends only minimal state to the client. Page arguments show up in the URL and as hidden inputs in the form. Access control rules are used to specify access based on these arguments. Inside the pages, the only values that are sent to the client are those inserted in templates. The name of an input element refers to the input element in the page, but not to the data model property it binds to. That binding happens on the server. If these input names are tampered with, the only thing the user can achieve is that it fills the data of another input on that page. Template identifiers are md5 hashed, to keep the length of the identifiers consistent. The user session entity id is included in the hash to provide protection against CSRF. Each user will have different template identifiers, even if they access the same page.

The template identifiers are based on the template inputs they correspond to, we divide this information into a static and a dynamic component. The static component is the position in the source definition, multiple calls in a row to the same template will have a different identifier. The dynamic component is relevant when a templatecall is inside a for loop. In that case the for loop identifier determines which iteration the input belongs to. The influence of each control flow element on the template identifier is shown in Figure 3.6. Figure 3.7 shows an example of how the template identifiers are generated for template elements.

Template Element	Template Id Component
<code>for(e in persistent entities)</code>	<code>e.id</code>
<code>for(e in transient entities)</code>	<code>iteration number</code>
<code>for(e in primitive values)</code>	<code>iteration number + e</code>
<code>templatecall(args) AST-location</code>	<code>AST-location</code>

Figure 3.6 Template unique identifier generation based on control-flow elements.

WebDSL	Generated Template Id
<pre> page showMessages(inbox : Inbox) {   output(inbox.user.name) .....   for(m in inbox.messages) {     output(m) .....   } } template output(m : Message) {   output(m.contents) ..... } </pre>	<pre> tcall1 for1+m1.id+tcall2, for1+m2.id+tcall2, ... tcall3 </pre>

Figure 3.7 Template unique identifier generation example.

The input name-value pairs that are expected depend on the existing data. If the data changes between the rendering of the form and the submit, e.g. through an update of another user, some of the submitted data might be ignored. For example, a for loop on a collection of entities is rendered. Between the render and the form submit one of the entities is removed. When the server handles the submit, the new values for the removed entity will simply be discarded, because those inputs do not exist anymore. There are several ways to remedy this. The developer can decide on a more explicit handling of conflicts by making a personal copy of the entity for the user, and then in a separate step deciding how to merge the edited entity with the original. Another way to handle potential conflicts is to detect concurrent changes asynchronously and report it to the user before the form is submitted. Similar to how Gmail reports that additional replies have been sent while you are writing your own reply. A more heavy weight solution is to implement Operational Transformation [Sun 2010] algorithms for collaborative editing, which provides immediate feedback on concurrent activity. For example, this would be required for a live shared code editor input component.

### 3.6 Data Validation

The core of a data-intensive web application is its data model. The web application must be organized to preserve the consistency of data with respect

```

1  entity User {
2  username : String ( id, name )
3  email    : Email
4  }
5  entity UserGroup {
6  name     : String ( id )
7  members : {User}
8  }

```

```

9  page editUser( u: User ){
10 form {
11   group( "User" ){
12     input( "Username", u.username)
13     input( "Email", u.email )
14     submit save { "Save" }
15   }
16 }
17 action save {
18   return user( u );
19 }
20 }

```



Figure 3.8 Example value well-formedness validation.

to the data model during updates, deletes, and insertions. The core consistency properties of a data model are formed by structural constraints. These are the primitive properties and relations between entities. Some consistency properties cannot be expressed as structural constraints, e.g. if they depend on additional information outside the persisted entities. Data validation rules constitute the constraints that data input and processing must adhere to in addition to the structural constraints imposed by the data model. WebDSL integrates declarative data validation rules with user interface models, unifying syntax, mechanisms for error handling, and semantics of validation checks. The validation rules cover value well-formedness checks, data invariants, form input validation, function assertions, and success messages.

### 3.6.1 Value Well-Formedness

*Value well-formedness* checks verify that a provided input value conforms to the value type. In other words, the conversion of the input value from request parameter to an instance of the actual type must succeed. This type of validation is usually provided by libraries or frameworks. However, it often has to be declared explicitly, and possibly at each input of a value of the type. In WebDSL, value well-formedness rules are checked automatically. WebDSL supports types specific for the web domain, including `Email`, `URL`, `WikiText`, and `Image`. Automatic value well-formedness constraints for all value types provides decent input validation by default. Note that validation rules are only used for input checks that require notification to the user. Checks and filtering to prevent post-data tampering and JavaScript injection are taken care of by the input and output components of WebDSL, such filtering should not have to be expressed in an application's validation rules. For example, in the case of `WikiText`, there is only a validation for the maximum length allowed. The `output(WikiText)` template already filters HTML elements based on a restrictive whitelist after processing Markdown.

The `editUser` page in Figure 3.8 consists of a form with labeled inputs for the `User` properties. The `save` action persists the changes to the database, provided that all validation checks succeed. Since well-formedness validation checks are automatically applied to properties, the `email` property is validated against its well-formedness criteria. The result of entering an invalid email address is shown in the screenshot. A message is presented to the user and the action is not executed.

### 3.6.2 Data Invariants

*Data invariants* are constraints on the data model, i.e. restrictions on the properties of data model entities. These validation rules can check any type of property, such as a single entity reference, a collection, or a primitive type. By declaring validation in the data model, the validation is reused for any input or operation on that data. Validation rules in WebDSL are of the form `validate(e, s)` and consist of a Boolean expression `e` to be validated, and a String expression `s` to be displayed as error message. Any globally visible functions or data can be accessed as well as any of the properties and functions in scope of the validation rule context. In the examples shown in this chapter, error messages are placed inline for conciseness. In general, error messages can also be placed inside a function or even be stored as entity property in the database, depending on the requirements for configuration and internationalization of the application.

Validation checks on the data model are performed when a property on which data validation is specified is changed and when the entity is saved or updated. Validation is connected to properties either by adding the validation in the property annotation or by referring to a property in the validation check. The validation mechanism takes care of correctly presenting validation errors originating from the data model. For form inputs causing data invariant violations the message is placed at the input being processed. When data model validation fails during the execution of an action, the error is shown at the corresponding button.

Figure 3.9 presents a `User` entity with several invariants and a `password` property. The `username` property has the `id` annotation, which indicates the property is unique and can be used to identify this entity type. The `password` property is annotated with validation rules that express requirements for a stronger password. By declaring validation rules in the entity, explicit checks in the user interface can be avoided.

Figure 3.10 shows more advanced validation rules, which express dependencies between the properties of an entity. The `UserGroup` entity is extended with an `owner` reference, a `moderators` set, and a `memberLimit` value. The `editUserGroup` page allows the owner to edit some of the `UserGroup` properties. The validation rule on the `moderators` set expresses that the owner should always be in this set of moderators. Similarly, the owner should always be a member. The `member` set is constrained in size based on the `memberLimit` value. Validation rules that cover multiple properties, such as the 'owner



```

1 entity User {
2   username : String ( id, name )
3   password : Secret
4   email    : Email
5 }
6 extend entity User {
7   validate( password.length() >= 8, "Password needs to be at least 8 characters" )
8   validate( /[a-z]/.find( password ), "Password must contain a lower-case character" )
9   validate( /[A-Z]/.find( password ), "Password must contain an upper-case character" )
10  validate( /[0-9]/.find( password ), "Password must contain a digit" )
11 }
12 page editUser( u : User ){
13   form {
14     group( "User" ){
15       input( "Username", u.username )
16       input( "Email", u.email )
17       input( "New Password", u.password )
20       submit save { "Save" }
21     }
22   }
23   action save {
24     return user( u );
25   }
26 }

```

Figure 3.9 Example data invariants on User entity.

```

1 entity UserGroup {
2   name      : String ( id )
3   owner     : User
4   memberLimit : Int
5   moderators : {User}
6   members   : {User} ( validate( membersWithinLimit(), "Exceeds member limit" ) )
7   validate( owner in moderators, "Owner must always be a moderator" )
8   validate( owner in members, "Owner must always be a member" )
9   predicate membersWithinLimit { members.length <= memberLimit }
10 }
11 page editUserGroup( ug : UserGroup ){
12   form {
13     group( "User Group" ){
14       input( "Name", ug.name )
15       input( "Member Limit", ug.memberLimit )
16       input( "Moderators", ug.moderators )
17       input( "Members", ug.members )
18       submit save { "Save" }
19     }
20   }
21   action save {
22     return userGroup( ug );
23   }
24 }

```

Figure 3.10 Example data invariants on UserGroup entity.

in moderators' check, are performed for all input components of properties the validation is specified on. However, the checks can be added to a single property as well, in order to specialize the error message. This is illustrated by the member limit check, which is added to the `members` properties. Note that although the check is only attached to the `members` property, a form and action that changes only `memberLimit` would still check invariants for the whole

```

1 page editUser( u: User ){
2   var p: Secret
3   form {
4     group( "User" ){
5       input( "Username", u.username )
6       input( "Email", u.email )
7       input( "New Password", u.password )
8       input( "Re-enter Password", p ){
9         validate( u.password == p
10                  , "Password does not match" )
11       }
12     submit action{ }{ "Save" }
13   }
14 }
15 }

```

Figure 3.11 Example form input validation.

entity before committing changes. Duplication in checks can be avoided by putting checks in predicates or functions.

### 3.6.3 Form Input Validation

*Form Input Validation* are necessary when the validation rule targets an input that is not directly connected to the persisted data model. These types of constraints are easy to address in the form environment itself. Validation checks in WebDSL pages have access to all variables in scope, including page variables and page arguments. Since databinding inputs happens before these validation rules are checked, the placement and order of validation rules does not influence the results of the checks. Visualization of errors resulting from validation in forms are placed at the location of the validation declaration. Usually such a validation rule is connected to an input, which can be expressed by placing the validation rule as a child element of `input`.

The example in Figure 3.11 shows an extra password input field in which the user must repeat the entered password. This validation cannot be expressed as a data invariant, since the extra password field is not part of the `User` entity. Therefore, the rule is expressed in the `form` directly, where it has access to the page variable `p`. This variable contains the repeated password whereas the first password entry is saved in the password field of `User` entity `u`. When entering a different value in the second field the validation error is presented, as can be seen in the screenshot.

### 3.6.4 Function Assertions

*Function assertions* are predicate checks at any point in the execution of functions for verification during the processing of inputs. If such an assertion fails, the function processing needs to be aborted, reverting any changes made, and the validation message has to be presented in the user interface. WebDSL supports this type of validation transparently using the `validate` syntax. The errors resulting from function assertion failures are displayed at the place

```

1 page createGroup {
2   var ug := UserGroup{}
3   form {
4     group( "User Group" ){
5       label( "Name" ){ input( ug.name ) }
6       label( "Owner" ){ input( ug.owner ) }
7       submit save() { "Save" }
8     }
9   }
10  }
11  action save() {
12    validate( ug.owner != null, "A group must have an owner" );
13    validate( ug.owner.email != "", "Owner has not provided an email address" );
14    ug.save();
15    email newGroupNotify( ug );
16    return userGroup( ug );
17  }
18  email newGroupNotify( u: UserGroup ){
19    from( "info@example.com" )
20    to( u.owner.email )
21    subject( "Usergroup created: ~u.name" )
22    "Your usergroup ~u.name has been created."
23  }

```

Figure 3.12 Example function assertions.

the execution originated, e.g., above the submit button which triggered the erroneous function.

Figure 3.12 provides an example of a function assertion. The `createGroup` page allows creating new `UserGroup` entities. The constraints expressed in the `save` function require that an owner is selected and that the owner has an email address specified. After validating these requirements, the group can be created. The `newGroupNotify` email definition retrieves an email address from its `UserGroup` argument, through `ug.owner.email`, and sends a notification email to the owner of the new group.

### 3.6.5 Messages

The discussed data validation features report erroneous behavior in functions. Related to such function assertions, is a generic messaging mechanism for giving feedback about the correct execution of a function. This requires a place to show messages, for instance by adding a default message template at the top of each page. This is done by calling the `messages` template. This takes care of retrieving the messages and displaying them. Display of success messages can be customized by overriding the `templateSuccess` template. Furthermore, a message has to be generated in a function. The `message(String)` function takes the success message and passes it along to the next page that will be rendered. An example of such messaging is shown in Figure 3.13. The `save` function of the `editUser` page passes a message to the page redirected to, namely `user`.

```

1 page editUser( u: User ){
2   //... form ...
3   action save {
4     message( "User information successfully changed" );
5     return user( u );
6   }
7 }
8 page user( u: User ){
9   group( "User" ){
10    label( "Username" ){ output( u.username ) }
11    label( "Email" ){ output( u.email ) }
12    navigate editUser( u ){ "edit" }
13  }
14 }

```

User information successfully changed

User	
Username	charlie
Email	charlie@charlie.xyz
	<a href="#">edit</a>

Figure 3.13 Example success messages.

### 3.6.6 Validation Phase in Request Lifecycle

The example in Figure 3.14 shows a multi-phase evaluation that includes the validation phase. This example illustrates what happens on a validation failure. The submitted values contain an empty name for one of the projects, which is not allowed according to the validation rule. After databind, the validate phase checks these validations for each of the inputs. When validation fails at this point, the action is not executed, indicated by the check of `valid`. Instead of redirecting the client to request a new page, the response returns the current page with validation errors inserted. The inputs contain the entered values, that way the user can fix the problem without having to re-enter previously entered information.

### 3.6.7 Library Input With Validation

Figure 3.15 shows the input template for `Int` values together with validation handling. The validation checks wellformedness of the value, which can fail if the value cannot be parsed as a number, or it is outside the range of allowed integers. This check shows the use of language embedded regular expressions. A regular expression can be included between slashes. This type of expression has several functions such as `find` to find an occurrence, and `match` for exact match. Errors are collected in a request var `errors`, which is a variable that does not get reinitialized when validation fails. That is why even after calling `cancel()`, the errors can still be rendered. `cancel()` is a built-in function to signal validation failure. It will prevent the execution of a submit function, and signals that a response needs to be rendered instead. It also sets the database transaction to be aborted after processing completes, to prevent any entity updates. `validationContext( id ){ elements }` invokes the nested validate elements of the input template and captures the resulting errors. These errors are retrieved using the `getValidationErrorsForId` function. `ref` type arguments provide a `getValidationErrors` method, which invokes the checking of entity invariants. Validation is implemented in a separate

Request URL 1: editProject/WebDSL		
Render	<pre> page editProject(p:Project) {   form{     output(p.name)     for(app in p.apps){       input(app.name)     }     submit save() {"save"}   } } </pre>	<pre> &lt;form name="c1" action="../../editProject/WebDSL"&gt;   WebDSL   &lt;input type="text" name="c2f1" value="yellowgrass"/&gt;   ...   &lt;input type="text" name="c2f4" value="researchr"/&gt;   &lt;input type="submit" name="c3" value="save"/&gt; &lt;/form&gt; </pre>

Request URL 2: editProject/WebDSL	
submit arguments:	<pre> c2f1 = "" c2f2 = "Weblab" c2f3 = "Codefinder" c2f4 = "Researchr" </pre>

DataBind	<pre> form{   output(p.name)   for(app in p.apps){     input(app.name)   }   submit save() {"save"} } </pre>	<pre> if(requestparam("c1") == "1"){   yg.name := requestparam("c2f1")   ...   re.name := requestparam("c2f4") } </pre>
Validate	<pre> form{   output(p.name)   for(app in p.apps){     input(app.name)   }   submit save() {"save"} } </pre>	<pre> if(requestparam("c1") == "1"){   yg.validateName();   ...   re.validateName(); } </pre>
Action	<pre> form{   output(p.name)   for(app in p.apps){     input(app.name)   }   submit save() {"save"} } </pre>	<pre> if(valid &amp;&amp; requestparam("c1") == "1"){   if(requestparam("c3") == "save"){ save(); } } </pre>
Render Validation Errors	<pre> page editProject(p:Project){   form{     output(p.name)     for(app in p.apps){       input(app.name)     }     submit save() {"save"}   } } </pre>	<pre> &lt;form name="c1" action="../../editProject/WebDSL"&gt;   WebDSL   &lt;input type="text" name="c2f1" value=""/&gt;   Project name may not be empty   &lt;input type="text" name="c2f2" value="Weblab"/&gt;...   &lt;input type="submit" name="c3" value="save"/&gt; &lt;/form&gt; </pre>

Figure 3.14 Validation in multi-phase evaluation example.

template from the actual input component, which means it can be replaced with other validation mechanisms if required. A variant with AJAX validation will be discussed in Section 3.7.3.

## 3.7 Partial Page Updates

Regular page requests retrieve the contents of an entire page. This is not the only way to update the view of the web application in the browser. From the JavaScript runtime additional requests can be made that do not have to retrieve an entire page. Instead these can retrieve a fragment of the displayed page to update only part of the page. The JavaScript runtime can also request just the data that is to be displayed, without transferring any additional rendering or layout information. Because only a fragment of the page is updated, the server response generation and delivery are faster. Also, the browser can process such a partial update faster than a full page update, which improves the user experience.

The technique for partial updates is called AJAX (Asynchronous JavaScript and XML). The asynchronous part refers to the fact that a page does not stop responding when an AJAX request is made, as opposed to regular page requests which block further interaction with the displayed page. A side-effect of updating page content dynamically is that the browser URL is not updated automatically. Additionally, the specific page with dynamically updated content is not bookmarkable by default either. This is typically seen as a downside, although these features can be reconstructed through updating the URL with the HTML5 History API after an AJAX update, and making the specific selection of content possible through the URL. Using XML as data format is not required, in fact, data is typically transferred in the JSON (JavaScript Object Notation) format. The JavaScript runtime has built-in support for parsing this format and converting it to JavaScript objects.

Partial page updates with AJAX enable single-page interfaces. A single-page interface loads all user interface updates and executes operations through AJAX requests. This application style avoids having to fully load a page after every action or navigation click. Navigation links can also be replaced with tabs or other visual indicators that contribute to the feeling of a single-page interface.

JavaScript widgets, reusable user interface components, can be created without having to supply all data in advance. For example, in a search field you could add autocompletions that are generated by the server based on the partially entered text. This requires an asynchronous request to retrieve the possible completions for the currently entered search term.

Implementing partial page updates directly in JavaScript can be done in many ways, however, it is typically quite low-level. Response handling is specified in callback functions. JSON data from the server has to be converted to DOM actions. Form submits have to construct a request string which contains all the input data. JavaScript libraries like JQuery can remove some of the boilerplate code, but also require an additional tool which adds to the

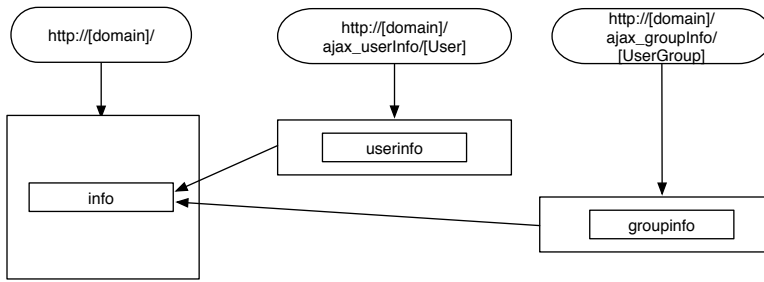
```

1  template input( i: ref Int ){
2    var req := getRequestParameter( id )
3    request var errors: [String] // request var is not reset when validation fails
4    if( errors.length > 0 ){
5      errorTemplateInput( errors ){
6        inputIntInternal( i, id )[ all attributes ]
7      }
8    }
9    else {
10   inputIntInternal( i, id )[ all attributes ]
11 }
12 validationContext( id ){ elements }
13 validate {
14   if( req != null ){
15     if( /-?\d+/.match( req ) ){
16       if( req.parseInt() == null ){
17         errors.add( "Outside of possible number range" );
18       }
19     }
20     else {
21       errors.add( "Not a valid number" );
22     }
23   }
24   if( errors.length == 0 ){ // no wellformedness errors
25     errors.addAll( i.getValidationErrors() ); // check invariants in entity
26     errors.addAll( getValidationErrorsForId( id ) ); // errors from nested elements
27   }
28   if( errors.length > 0 ){
29     cancel(); // marks request as validation failed
30               // no submit action is executed and transaction is aborted
31   }
32 }
33 }
34 template inputIntInternal( i: ref Int, tname: String ){
35   var req := getRequestParameter( tname )
36   < input
37     name = tname
38     if( req != null ){ value = req } else { value = i }
39     all attributes
40   />
41   databind {
42     if( req != null ){
43       i := req.parseInt();
44     }
45   }
46 }

```

---

Figure 3.15 Input for Int with validation.



```

1 page root {
2   sidebar( principal )
3 }
4 template sidebar( u: User ){
5   placeholder info { "Placeholder for user or group info" }
6   submitlink action{ replace( info, userInfo( u ) ); }{ "Show user info" }
7   submitlink action{ replace( info, groupInfo( u.group ) ); }{ "Show group info" }
8 }
9 ajaxtemplate userInfo( u: User ){
10  output( u )
11 }
12 ajaxtemplate groupInfo( g: UserGroup ){
13  output( g )
14 }

```

Figure 3.16 Partial page updates with AJAX templates.

complexity for the definition of the user interface. There is room for abstraction, since most updates consist of simple replace, add, or remove actions of DOM elements.

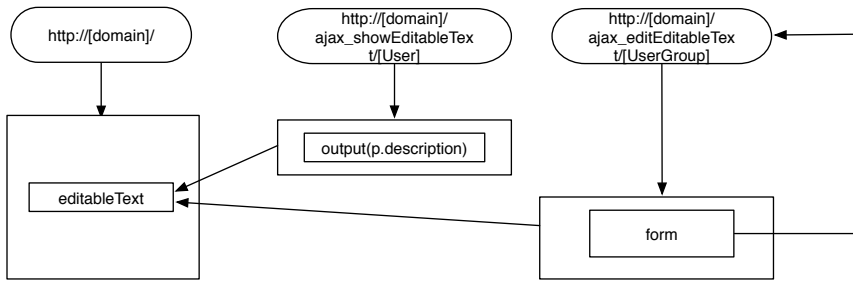
Finding a single abstraction that covers all cases of partial page updates, while also preserving the safety guarantees against tampering is difficult. In WebDSL we have implemented and experimented with two different styles for partial page updates: nested pages and inline refresh. There is room for improvement, current frameworks like React [2023], Angular [2023], and Elm [2023] perform client-side rendering where there is complete control over incremental interface updates.

### 3.7.1 Nested Page

The first type of partial page update abstraction is updating isolated templates contained in another page. This can be seen as a nested page inside a page. They are isolated because they handle subsequent requests separate from the containing page. The AJAX template is an entry point on the server with its own URL. The URL of an AJAX template is not meant to be used from the address bar and thus consists of a generated name. This concept of AJAX templates enables coarse-grained partial page updates. Part of a page is marked as a placeholder, which can be filled in with arbitrary AJAX templates. Filling in a template means that an action is invoked that replaces the placeholder contents.

The example in Figure 3.16 shows a nested page that outputs information.





```

1 entity Project {
2   description : Text
3   members : {User}
4   function canUserEdit( u: User ): Bool {
5     return u in members;
6   }
7 }
8 page root {
9   var p := Project{}
10  placeholder editableText {
11    showEditableText( p )
12  }
13 }
14 ajaxtemplate showEditableText( p: Project ){
15   output( p.description )
16   submitlink action{ replace( editableText, editEditableText( p ) ); }{ "[Edit]" }
17 }
18 ajaxtemplate editEditableText( p: Project ){
19   form {
20     input( p.description )
21     submit action{ replace( editableText, showEditableText( p ) ); }{ "Save" }
22   }
23 }
24 rule ajaxtemplate showEditableText( p: Project ){ p.canUserEdit( principal ) }
25 rule ajaxtemplate editEditableText( p: Project ){ p.canUserEdit( principal ) }

```

Figure 3.17 Forms in AJAX templates.

Through submit links, the shown data can be changed to user or group info. In Figure 3.17, a partial page with input form is shown. The submit inside `editEditableText` will not submit to the page, but uses the `ajaxtemplate` entry point. Access control rules are required to verify that the user is allowed to perform the value update.

The benefit of the nested page method is abstraction over low-level JavaScript requests and DOM manipulation. AJAX can be described in terms of an `ajaxtemplate` being placed or updated in placeholders. The WebDSL compiler generates the client-side and server-side code to handle these updates. Initially, we created an API with several commands, such as `replace`, `hide`, `show`, `append`, and `remove`. However, for typical scenarios, usually just the `replace` command is sufficient.

There are also limitations with this method. Firstly, because the AJAX templates are entry points to the server, they have to be included in an access control specification. This introduces some syntactic overhead when creating AJAX templates. Secondly, because the fragments are isolated, this type of

partial page update cannot be used to create a dynamic form that changes shape based on input. The AJAX template cannot be nested in a form of the enclosing page, as the page does not know about the contents of the placeholder. This indicates a weakness in the solution regarding static analysis. The content of a placeholder cannot be known in advance, any AJAX template can be nested in a placeholder.

An AJAX template is rendered differently than a regular template, it behaves more like a page. Generating template identifiers starts fresh when handling an AJAX template. The AJAX template cannot be part of a form that is outside its own definition. It is also a boundary for forms nested inside it. A form submit inside an AJAX template does not go through the surrounding page. Also any rerenderings, such as those for data validation errors, happen inside the placeholder only.

Placeholders consist of the `placeholder` keyword and an identifier, followed by a call to an AJAX template with the initial content. Placeholders declare the placeholder name as a template variable with `String`-like type `Placeholder`. This variable is assigned a name based on the declared name and the unique template identifier. The assigned placeholder name will be automatically unique on the page. The `replace(Placeholder, ajaxtemplatecall)` function performs the replacements, this can be invoked from a submit function connected to a button or other input event, such as `oninput`. The unique placeholder names enable such templates to be composed without interference caused by duplicate identifiers. As a low-level fallback, a placeholder name can be generated from an expression. This expression has to be reproduced when invoking the `replace` calls. Using this option requires that the application designer ensures these identifiers are unique on a page, however, in this case the placeholder name does not have to be passed around to be able to refer to it from different templates.

There were some issues we encountered when using the nested page AJAX templates. One issue is that certain configurations that occur due to dynamically changing the DOM get rejected and implicitly transformed by the browser. In particular these problems arise with nested forms, and incorrectly nested table elements. This breaks the abstraction of the isolated page inside a page. Another issue is that misbehaving JavaScript widgets can cause confusing errors. For example, a JQuery UI modal dialog widget conceptually fits well within an AJAX template. However, if the JavaScript code for dialogs leaves old closed dialogs hidden inside the DOM, these can cause unintended interactions with a new dialog. Datepickers stop working, because they are finding the hidden dialog fragment in the DOM instead of the active one. The workaround that was needed here was to explicitly clean up old hidden dialog elements.

Nested pages based on AJAX templates are problematic for modularization. When creating a reusable library of these components, access control rules cannot be included in that library, because these depend on the specific application. This means that a library will always need some boilerplate access control rules to enable AJAX templates.

Functions in WebDSL run on the server, and allow an escape to Java code

and invoking libraries. This Java class interoperability feature increases the coverage of the language, by adding a pragmatic way of doing something the language has no built-in support for. In the case of partial page updates and AJAX, escaping means invoking JavaScript code and similarly increases the coverage of the language. There are two ways to invoke JavaScript code, either by including a script tag inside a template, or sending JavaScript commands from the server similar to replacing a placeholder. Script tags embed JavaScript, with a tilde escape to splice in a String value from WebDSL. The `runscript` command sends the supplied JavaScript to the client and invokes it there.

### 3.7.2 Inline Refresh

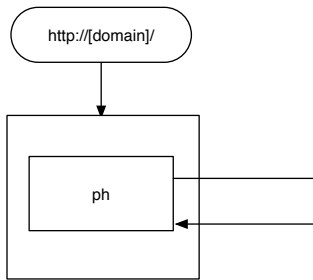
The nested page partial page updates approach in the previous section abstracts over low-level DOM operations and AJAX JavaScript request handling. While this already provides a major relief from custom AJAX implementations, there are a couple of downsides. In particular, the difficulty to dynamically alter forms based on provided input, and the necessity of declaring access control rules due to the global nature of replacements. Additionally, AJAX requires extra template definitions that might only be used in one context. These issues have been addressed in an alternative style of partial page updates implemented in WebDSL.

The inline refresh method for partial page updates is based on refreshing placeholder contents, instead of replacing them with a new template call. In this method the AJAX requests go through the page as they would with normal templates. This means they inherit the access control rule of the page and are not a separate entry point on the server. Also, the contents of the placeholder can be statically analyzed, because all the possible content is explicit in the definition.

The example in Figure 3.18 illustrates dynamic form behavior with inline refresh. The input for the first value can trigger the input for the second value to show up. In the case of simple triggers that show or hide form inputs, such form visibility could also be encoded client-side with JavaScript. However, besides breaking the abstraction, that solution does not easily extend to checks that require the server.

While syntactically a lot more light-weight, at runtime these inline updates do not get the performance benefit of only accessing the AJAX template. The page structure is analyzed to determine what components need to be updated. While being easier to analyze, this method is also more restrictive, because the nested page approach allowed any AJAX template to be inserted in any placeholder. In practice we have found use cases for both variants.

Besides inheriting access control specifications, these local placeholder refreshes can work in conjunction with form input elements. This is possible due to the contents of the placeholder being known in the page definition. The placeholder does not become a boundary of the form. Form elements can be made optional with a regular WebDSL `if` condition. The `request var` can be used to mark a transient values to not be reset to the persisted state when



```

1 entity Assignment {
2   type : AssignmentType
3   lang : Language
4 }
5 enum AssignmentType { programming( "Programming" ), multiplechoice( "Multiple Choice" ) }
6 enum Language { java( "Java" ), python( "Python" ) }
7 page root {
8   request var a := Assignment{} // request var is not reset when rendering in replace
9                                 // call, form input state used to decide 'if' on line 18
10  form {
11    input( a.type )[ oninput = replace( ph ) ] // trigger the inline refresh on change
12    /* replace is a built-in action variant for common pattern:
13       - run action even if form has validation errors that would normally block action
14       - content of placeholder(s) will be replaced with new render based on form data
15       - rollback any changes to persisted data, replace action is meant for UI update
16    */
17    placeholder ph {
18      if( a.type == programming ){
19        input( a.lang )
20      }
21    }
22    submitlink action{ a.save(); }{ "Save" }
23  }
24 }

```

Figure 3.18 Partial update model with inline refresh.

rendering (which is the normal behavior for `var`). The point at which the rerender is invoked is explicit here, typically these would be placed in DOM event actions that trigger when a previous input is changed. The order of the inputs starts to matter here, an input that influences a conditional fragment must come before it. The reason for this is that first phase, databind, already has to pick the right control flow path.

### 3.7.3 Inputs with Immediate Validation

Figure 3.19 shows the `Int` template that performs the regular on submit validation, as well as immediate AJAX validation when entering data. The validation is performed in the `validator` function, which is executed whenever the input changes. It performs the same validation as would happen on a form submit, but renders its messages by replacing the `validateph` placeholder. This is using the nested page abstraction. The `messages` AJAX template only displays the messages passed to it, so it is safe to make publicly accessible with access control rules. The `rollback` is introduced to be able to use submit

```

1  template input( i: ref Int ){
2  var req := getRequestParameter( id )
3  request var errors: [String] // initialized to empty list
4  inputIntInternal( i, id )[ oninput = validator() ] // invokes validator on any change
5  validationContext( id ){ elements } // regular nested validation on submit
6  placeholder validateph {
7    if( errors != null && errors.length > 0 ){
8      showMessages( errors ) // regular validation errors on submit
9    }
10 }
11 validate { // regular form submit validation
12   errors := checkIntWellformedness( req ); // shortened, shown in previous fragments
13   if( errors.length == 0 ){
14     errors.addAll( i.getValidationErrors() );
15     errors.addAll( getValidationErrorsForId( id ) );
16   }
17   if( errors.length > 0 ){
18     cancel(); // marks request as validation failed
19     // no submit function will be executed and transaction will be aborted
20   }
21 }
22 action ignore-validation validator { // ignore-validation: perform action
23   // even if other validations in form fail
24   errors := checkIntWellformedness( req );
25   if( errors.length == 0 ){
26     errors := i.getValidationErrors(); // validate entity property invariants
27     validateTemplate( validationContext( id ){ elements } ); // nested validate checks
28     errors.addAll( getValidationErrorsForId( id ) ); // in input template call
29   }
30   if( errors.length > 0 ){
31     replace( validateph, showMessages( errors ) );
32   }
33   else {
34     replace( validateph, noMessages() );
35   }
36   rollback(); // complete function normally but abort transaction,
37   // this action is only for generating replace calls to show errors
38 }
39 }
40 template inputIntInternal( i: ref Int, tname: String ){
41 var req := getRequestParameter( tname )
42 < input
43   name = tname
44   if( req != null ){ value = req } else { value = i }
45   all attributes
46 />
47 databind {
48   if( req != null ){
49     i := req.parseInt();
50   }
51 }
52 }

```

---

Figure 3.19 Input for Int with AJAX validation.

functions for validation only. It completes the function, but will abort the transaction to prevent any changes to persisted data. The `ignore-validation` modifier allows the function to be executed regardless of whether earlier validation failed. This template definition repeats the validation for the actual form submit, in order to guarantee that the final submit values are checked again before completing the form and saving the data.

### 3.8 Conclusion

WebDSL's approach for handling user interface templates is unique and not found in other solutions (see Chapter 8 for a comparison with other solutions). Instead of using separate action handler functions with manually specified input identifiers, WebDSL provides abstraction over the construction of input names, and links actions to the context of the user interface definition. This approach avoids any inconsistency issues that can occur with separately defined action handlers and manually constructed input names, as well as avoiding inconsistencies in enforcing access control when handling form submits. Form submits are safe from hidden data tampering. Input templates can be safely composed, and input identifier mismatch in action handlers is prevented through template identifiers provided by the runtime. Furthermore, these identifiers provide automatic enforcement of Cross-Site Request Forgery protection. WebDSL enables expressive data validation, which is integrated into the multi-phase execution semantics of templates. Partial page updates are supported without explicit JavaScript or DOM manipulation. Data updates are always performed on the server, and only if the relevant inputs and actions are available at the time of handling the update request. WebDSL provides an extensive standard library of input and output components that support both regular validation and immediate validation feedback with partial page updates.

This concludes the discussion of WebDSL user interface templates, this topic will be revisited in Section 5.2.1 where the evolution of this feature is described. Chapter 4 will cover access control, which is also an essential feature of the WebDSL language for creating real-world applications.



# Access Control

---

# 4

## 4.1 Introduction

Access control is essential for the security and integrity of interactive web applications. While a simple ‘all-or-nothing’ access control policy such as provided by Apache’s `.htaccess` file is sufficient for many web sites and applications, other applications such as conference management systems, social network services, and online auctions require more sophisticated policies to regulate the access to sensitive data and operations on those data. A simple access policy can be enforced by means of a generic ‘check at the gate’, verifying the identity of the user. More advanced policies grant or deny access based on the identity of the user, but also the particular entry point, the subsequent actions to be invoked, *and* the data to be viewed or modified. For example, in a social network service, certain pages may be only accessible to the members of a group, which requires checking the particular combination of page, group, group membership, and user identity. Definition of such policies requires close integration with the web application, tuned to its data model and operations. Furthermore, validation and verification of a policy requires a concise high-level specification, separate from the implementation details of the rest of the application. In practice, existing solutions for access control for web applications are either separate policy languages, which cannot be seamlessly integrated, or application frameworks, which do not support high-level definition of policies.

Access control policy languages such as Ponder and XACML [Damianou et al. 2001; Moses et al. 2005] are often implemented as an autonomic system that can be queried for access control decisions. This approach makes it hard to support flexible access control. All the information needed for a check has to be explicitly transferred to the policy engine. Furthermore, these frameworks do not aid in separating checks from the main application; the actual check invocations are still scattered across the application code. Finally, the complexity of these policy languages (e.g., the RBAC template for XACML [Anderson 2004]) decreases their readability, resulting in unclear policies. Web application frameworks such as Spring/Acegi [Johnson et al. 2005; Alex 2008] and Seam [Yuan and Heute 2007] support an aspect-oriented approach to access control, separating access control code from application code. These frameworks supply more or less fixed role-based [Sandhu 1998; Sandhu, Ferraiolo, and Kuhn 2000] and discretionary [Samarati and di Vimercati 2001; Sandhu and Samarati 1994] access control configurations. Extending the built-in policy or creating other types of policies requires manually implementing these as an extension of the underlying object-oriented framework. Such



framework extensions are hard to test and require knowledge of the framework to be able to understand the policies.

This situation is not unique for the domain of access control. The implementation of web applications comprises many other technical concerns, including data representation, querying, and modification, user input, data validation, user interface design, and navigation. These concerns are often addressed by separate languages. For example, in (one configuration of) the Java web programming platform we find the Java general purpose programming language, the SQL query language (or some dialect such as HQL), the JavaServer Faces (JSF) presentation language with the EL expression language for accessing data, the CSS stylesheet language, and other XML schemas for configuration such as page flow declarations.

While separation of concerns and ‘choosing the right language for the job’ are conceptually appealing, the amalgam of languages used in a single web application project are typically poorly integrated, with an adverse effect on productivity and software quality. For example, while Java is a statically checked language, the portions of a web application implemented in XML are outside the control of the Java compiler. Thus, the integration of XML data and Java classes is not statically verified, requiring run-time debugging and possibly errors that go undetected into production systems. The encoding of SQL queries as string literals entails that queries are only checked syntactically and semantically at run-time, and introduce the risk of injection attacks. Similarly, the use of *dependency injection* leads to leaner programs, but also shifts linking of program components from compile-time to deployment- or even run-time. Besides the loss of static verification, the languages are often redundant, i.e. need to address overlapping concerns. For example, the language of EL expressions in JSF is a poor subset of Java expressions used for accessing properties and methods of Java objects connected to a JSF page.

Generic aspect languages such as AspectJ [Kiczales et al. 2001] achieve separation of concerns while maintaining the benefits of linguistic integration, such as static analysis and reuse of overlapping functionality (e.g. statements and expressions). However, generic aspect languages are agnostic about particular (technical) domains such as access control, and thus require policies to be encoded programmatically, precluding benefits provided by more declarative definitions.

WebDSL supports separation of concerns by providing sub-languages catering for the different technical domains of web engineering. Linguistic integration of these sub-languages ensures seamless integration of the aspects comprising the definition of a web application.

In this chapter, we present the extension of WebDSL with abstractions for declarative definition of access control. The access control policy for an application is defined separately from the data model and user interface using *declarative rules*. While access control rules are defined as a separate concern, the extension is *linguistically integrated*. That is, access control rules use the same expression language, which refers to the same data models that are used in the rest of an application. Furthermore, access control checks are integrated

into the implementation, which allows rules to access the complete object graph, instead of requiring selected data to be sent to a separate engine. Rather than catering for a fixed policy, the extension provides the basic mechanisms for encoding *a wide range of access control policies* that can be adapted to the requirements of the application and integrated with its data model. Finally, the declarative and domain-specific nature of rules allows us not only to restrict access to pages and actions, but also to adapt the navigation options presented to users to prevent them from navigating to inaccessible pages.

The main contributions of this chapter are:

- The general approach of designing linguistically integrated domain-specific languages for different technical domains, realized by means of a transformational semantics that reduces separately defined aspects into an integrated implementation.
- The design of an access control sub-language for expressing a wide range of access control policies concisely and transparently as a separate concern.
- The use of access control semantics for reducing development effort. The developer can concentrate on the logical design of navigation, leaving the modality of navigation to the access control rules.

In Section 4.2 we introduce the access control extension of WebDSL by providing a group-based access control policy for a small wiki application. In Section 4.3 we discuss the expressivity of WebDSL access control by encoding standard policies such as mandatory, discretionary, and role-based access control. In Section 4.4 we present the transformational semantics for weaving access control rules into an application definition. In Section 4.5 related work is discussed. Section 4.6 contains an evaluation and possibilities for future work. Section 4.7 concludes this chapter.

## 4.2 Access Control

Web applications that give clients creation or modification access to its data, need to trust those clients to well-behave. Trusting the general public, as is done in sites such as Wikipedia, requires either faith in human nature or a large community of moderators checking modifications. Otherwise a site runs a high risk of corrupted data and/or (scripted) spam attacks. For most applications the only solution to prevent this, is to impose an access control policy allowing only known users to access and modify data. Implementing an access control policy requires checking the permissions of the user to open a page, to perform an action, or even to see part of a page, implying checks scattered across the definition of an application. Such embedded checks make it hard to understand the access control policy they implement. We have designed and implemented an extension of WebDSL to support *separate and declarative specification of access control* for an application. In this section we illustrate the language features for declarative access control with a small

```
1 entity Topic {
2   name      : String
3   authors   : {User} ( inverse = topics )
4   content   : WikiText
5 }
6 entity User {
7   username  : String ( id, name )
8   email     : Email
9   password  : Secret
10  topics    : {Topic}
11 }
```

---

Figure 4.1 Data model for wiki.

wiki example application, by extending it with a variation on discretionary access control. In the next section we give a more systematic account of the expressivity of the language for encoding standard access control policies such as mandatory, discretionary and role-based access control.

*WebDSL Running Example* Figure 4.1 and 4.2 show a simple wiki application that will be extended with access control. The simple wiki data model consists of a `Topic` entity and a `User` entity. Each `Topic` entity represents a wiki page. The `Topic.authors` property is a bidirectional association with `User.topics`.

The `topic` page views the contents of a `Topic` entity. The `editTopic` page contains a form to input the data of a wiki topic page. Making a change to a `Topic` adds the currently logged in user to the list of authors and the topic to `User.topics`. Additional features like versioning of topic entities and discussion pages of topics are not included in this example.

### 4.2.1 Authentication

The first step of any access control system is to establish the identity of the user (the *principal*) by means of an authentication procedure typically requiring the user to declare a (public) username and (secret) password. When a username and password combination corresponds to a registered user, that user is logged in, and his identity stored for the duration of the session.

Thus, the first step in setting up an access control policy is to declare the principal as a combination of an entity type to represent users, and their *credentials*, i.e. the properties of the user entity that play a role in authentication. Figure 4.3 shows the authentication definitions for the example wiki application. The `User` entity is declared as principal with `username` and `password` as credentials. Note that any entity can be used as principal; `User` is not a built-in notion. The credentials should consist of one or more properties with type `Secret` (passwords) and one or more properties that uniquely identify the user object.

From this declaration, a *session entity* `securityContext` is derived, which holds session data related to access control, in particular a reference to the principal. Session entities are attached to a client session and are accessible from any page definition. In Figure 4.2 we used the `principal` reference (shorthand for `securityContext.principal`) to obtain the principal and record that user

```

1 page topic( t: Topic ){
2   main
3   title { "Topic: ~t.name" }
4   template menubar { topicMenu( t ) signinMenu }
5   template body {
6     header { ~t.name }
7     par { ~t.content }
8     par {
9       "Contributions by "
10      for( u in t.authors order by t.name ){ "~u " }
11    }
12  }
13 }
14 page editTopic( t: Topic ){
15   main
16   title { "Edit topic: ~t.name" }
17   template menubar { topicMenu( t ) signinMenu }
18   template body {
19     header { "Edit Topic: ~t.name" }
20     form {
21       par { input( t.name ) }
22       par { input( t.content ) }
23       par { submit saveTopic { "Save Changes" } }
24     }
25     action saveTopic {
26       t.authors.add( principal );
27       return topic( t );
28     }
29   }
30 }
31 template main {
32   block( "page" ){
33     block( "menu" ){ menubar }
34     block( "body" ){ body }
35     block( "footer" ){ footer }
36   }
37 }
38 template topicMenu( t: Topic ){
39   menu {
40     menuheader { ~t }
41     menuitem { navigate editTopic( t ){ "Edit" } }
42   }
43 }
44 template signinMenu {
45   menu { menuheader { navigate signin { "Sign in" } } }
46 }

```



Figure 4.2 View and edit page for wiki topics with screenshots.

```

1 principal is User with credentials username, password
2 // generated from line above
3 session securityContext { principal : User }
4 function authenticate( name: String, pw: Secret): Bool {
5     var users := from User as u where (u.username = ~name);
6     if( users.length == 1 && users[ 0 ].password.check( pw ) ){
7         securityContext.principal := users[ 0 ];
8         return true;
9     }
10    else {
11        return false;
12    }
13 }
14 page signin {
15     main
16     template body {
17         var name: User
18         var pw: Secret
19         form {
20             "username: " input( name )
21             "password: " input( pw )
22             submit signin { "Sign in" }
23         }
24         action signin {
25             if( authenticate( name, pw ) ){
26                 return user( principal );
27             } else {
28                 return accessDenied();
29             }
30         }
31     }
32 }

```

---

Figure 4.3 Principal, security context, and authentication.

as author of the topic in `editTopic`. Furthermore, an authentication function is derived, which checks the credentials against the database (using an embedded query), and upon success initializes the `principal`. The authentication function can be used to implement a dialog such as the `signin` template to authenticate a user, given his credentials. The `signin` template and the generated `securityContext` session entity and `authenticate` function are shown in Figure 4.3. The `Secret.check` function checks the entered password with the hashed and salted password stored in the database.

#### 4.2.2 Restricting Access

Given the ability to authenticate a user against a database of known users, a very first basic access control policy is to distinguish anonymous visitors from authenticated users. For instance, in the wiki we could allow anyone to *view* topic pages, but only allow logged in users to *edit* the content of a topic. This policy is expressed by means of the *access control rules* in Figure 4.4.

The first rule states that the condition for accessing the `topic` page is `true`, thus access is always granted. The second rule states that the `editTopic` page can only be accessed if `loggedIn()` is `true`; the `loggedIn` predicate is generated by default when a principal is declared and consists of a simple `principal`

```
rule page topic( t: Topic ){ true }
rule page editTopic( t: Topic ){ loggedIn() }
```

Figure 4.4 Access control rules for the view and edit pages of the wiki.

`!= null` check.

In general, an access control rule has the form `rule r i( $\vec{x}$ ) {e}` with  $r$  the type of definition or resource (`page`, `template`, `function`, or `action`),  $i$  the identifier of the definition,  $\vec{x}$  its parameters (a list of identifier and type pairs, just like the parameters of a page definition), and  $e$  a Boolean expression over the parameters of the rule and the session entities of the application. When the expression evaluates to `true`, access is granted, when it evaluates to `false`, access is denied. We do not use a logic with more than two values to indicate that a rule is not applicable or that an error occurred in its evaluation. In those cases a rule should just evaluate to `false` and access be denied. When access control is enabled by declaring a principal, and no rule exists for a certain resource, access to that resource is denied by default.

*Representing Access Permissions* Most applications require a more sophisticated access control policy than just authenticating the user. Rather, some combination of the identity of the user and aspects of the state of the application are involved to make a decision. As an example, we develop a policy in which wiki users have view or edit access to topics based on group membership. Such a policy requires the administration of the access rights of users. Instead of using a specialized, built-in data type for this administration, WebDSL employs the same data models it uses for other data of the application, ensuring a seamless integration of access control with the rest of the application.

To model the group membership policy we introduce a `UserGroup` entity with a `name` and a set of users as `members` (Figure 4.5). Next we *extend* the declaration of the entity `User` with a property `groups`, as the inverse of the `members` property (if  $u$  in  $g.members$  then  $g$  in  $u.groups$ ). Entity extension is a modularity feature that allows properties to be declared together with the other definitions for the aspect they pertain to (akin to intertype declarations in AspectJ [Kiczales et al. 2001]). With this representation of groups, we can then define the notion of an access control list `ACL` as an entity with a `viewers` and `editors` property, which both refer to a *set* of groups, with the intention that members of these groups have the corresponding permission. Finally, the `Topic` entity is extended with an `acl` property to represent the access permissions for the object.

*Checking Access Permissions* Given the encoding of access permissions we can now define the rules that declare the access to specific pages and actions (Figure 4.6). The `memberOf` predicate tests whether the principal is member of one of a given set of groups  $gs$ , by checking if one of the groups of the principal occurs in  $gs$  (which is equivalent, through the inverse relationship between `UserGroup.members` and `User.groups`). This predicate is then used in the

```

1 entity UserGroup {
2   name   : String ( id, name )
3   members : {User}
4 }
5 extend entity User {
6   groups : {UserGroup} ( inverse = members )
7 }
8 entity ACL {
9   viewers : {UserGroup}
10  editors : {UserGroup}
11 }
12 extend entity Topic {
13   acl      : ACL
14 }

```

---

Figure 4.5 Representing permissions.

```

1 access control rules
2 predicate memberOf( gs: {UserGroup} ){
3   Or[ g in gs | g in principal.groups ]
4 }
5 rule page topic( topic: Topic ){
6   (t.acl.viewers.length == 0) || memberOf( topic.acl.viewers )
7 }
8 rule page editTopic( topic: Topic ){
9   memberOf( topic.acl.editors )
10 }

```

---

Figure 4.6 Checking permissions.

definition of the access control rules. The rule for the `editTopic` page requires that the principal is member of one of the editors groups. The rule for the `topic` page requires that the principal is member of one of the `viewers` groups, however, when no such group is registered, access is open for anyone.

*Restricting Navigation* The rules above forbid access to an `edit page` if the user is not a member of the editors groups. However, the menu of the view page for a topic contains a link to the edit page for that topic through the `navigate(editTopic(t)) {"Edit"}` element in template `topicMenu` in Figure 4.2. When a user without the proper permission follows this link, he will end up in the `accessDenied` page. Following these ‘dead’ links can be prevented by letting the visibility of the `navigate` link depend on the same check as specified for the target of the navigation. Thus, only relevant navigation options are presented to the user, improving the user experience. This behavior can be inferred because of the declarative (as opposed to programmatic) formulation of access control rules.

### 4.2.3 Administration

The administration of access rights also requires a user interface. Since WebDSL employs the same data models for access rights as for any other application data, the same user interface modeling can be used to implement a user interface for access administration. For example, an `editUserGroup` page can be defined to edit the collection of members of a group and an `editPermissions`

```

1 extend entity UserGroup {
2   moderators : {User}
3 }
4 extend entity ACL {
5   moderators : {UserGroup}
6 }
7 page editPermissions( t: Topic ){
8   main
9   template body {
10    header { "Edit Permissions of ~t" }
11    form {
12      table {
13        row { "Viewers: " input( t.acl.viewers ) }
14        row { "Editors: " input( t.acl.editors ) }
15      }
16      submit savePermissions { "Save" }
17    }
18    action savePermissions {
19      t.save();
20      return topic( t );
21    }
22  }
23 }
24 access control rules
25 rule page editUserGroup( g: UserGroup ){
26   principal in g.moderators
27 }
28 rule page editPermissions( t: Topic ){
29   memberOf( t.acl.moderators )
30 }

```

Figure 4.7 Administration of permissions.

page for editing a `Topic`'s ACL. Of course, the access to such administration pages should be controlled as well, lest the access control of regular pages becomes meaningless. If anyone can add members to a group, the rule for edit access of a topic has no value. Thus, to control the access to groups and access control lists themselves, we extend these with a `moderators` property indicating who can modify these objects (Figure 4.7). Then access control rules can be defined to restrict the access to the pages for editing permissions. A group may only be edited by the users in the `moderators` collection and an ACL may only be edited by members of the `acl.moderators` groups.

*Policy Combination* Often it is useful to equip an administrator with special rights, for instance to access any pages and perform all kinds of edits. With



```

1 access control rules admin
2   rule page * ( * ){
3     isAdministrator()
4   }
5   predicate isAdministrator {
6     adminGroup in principal.groups
7   }
8 access control policy
9   anonymous OR admin

```

Figure 4.8 Combination of rules.

the rules discussed so far, this would require adding to each rule a disjunct that checks whether the principal is an administrator, which would hardwire this policy and pollute many rules. Instead, WebDSL provides the possibility of combining sets of rules into a policy. For this purpose, a set of rules can be given a name, which is used to refer to it. Unnamed rule sets are combined into a single set with the name `anonymous`. Rule sets can be combined using the `OR` and `AND` operators. For example, to allow an administrator to access all pages, a *generic* rule can be used (Figure 4.8). A generic rule can use a wildcard for the name of the element and/or the parameters of the element. Such a rule applies to elements to which it matches. Thus a rule with signature `page *(*)` applies to all page definitions. Of course, a rule with a wildcard for the parameters cannot involve information from the parameter objects. Here, `adminGroup` is an application-global variable that identifies a particular object of type `UserGroup`. The generic rule is placed inside a rule set named `admin`. The `admin` rule set is then combined with all previously defined (`anonymous`) rules using the `OR` operator, which entails one has access to a page when either of the rules matching the same page or action succeed.

*Active Group* With the `admin` policy above, a user who is administrator can always apply all actions, similar to always being logged in as super user in a Unix system. Thus, an administrator is not protected against mistakes, nor does the administrator ever see the application as normal users see it. In order to enable the privileges of an administrator, or other group membership, only when needed, the policy can be refined by introducing the notion of *active groups* (Figure 4.9). The `activeGroups` property that is added to the `securityContext` represents the subset of the groups of the user in which he is active at the moment. By changing the `memberOf` and `isAdministrator` predicates to take into account only `activeGroups` instead of `principal.groups` this policy is implemented.

*User Proxy* A more generic solution for partially restricting access as an administrator, is to mimic login of a regular user. This way any issues that the user experiences can be easily reproduced, even on a live system. Figure 4.10 shows a `Person` entity used as principal. An administrator, indicated by the `isAdmin` property, can substitute another user by setting the current principal to that user. The original administrator principal is temporarily stored in `securityContext.realPrincipal`. This makes all access control checks equivalent to what the user would experience. When logging out, if there was

```

1 extend entity securityContext {
2   activeGroups : {UserGroup}
3 }
4 access control rules
5   predicate memberOf( gs: {UserGroup} ){
6     Or[ g in gs | g in activeGroups ]
7   }
8 access control rules admin
9   predicate isAdministrator { adminGroup in activeGroups }

```

---

Figure 4.9 Active groups.

```

1 principal is Person with credentials username, password
2 extend session securityContext{ realPrincipal : Person }
3 entity Person {
4   username : String
5   password : Secret
6   isAdmin : Bool
7   function proxy {
8     securityContext.realPrincipal := principal;
9     principal := this;
10  }
11 }
12 function signoff {
13   if( securityContext.realPrincipal != null ){
14     principal := securityContext.realPrincipal;
15     securityContext.realPrincipal := null;
16   } else {
17     principal := null;
18   }
19 }
20 template proxy( person: Person ){
21   submitlink action{ person.proxy(); }{ "Proxy" }
22 }
23 access control rules
24   rule template proxy( person: Person ){
25     principal.isAdmin && principal != person
26   }

```

---

Figure 4.10 User proxying.

a proxy active, the current user becomes the administrator `Person` entity again.

This is quite similar to running the `su` and `sudo` commands on Unix systems as root user. The `su` substitute user command changes the current user and the `sudo` command executes a single command as another user. That way a root user can give a command a more restricted environment than its own, or investigate why a command does not work for a particular user.

### 4.3 Access Control Policies

WebDSL provides high-level, policy neutral mechanisms for defining access control, that is, without making assumptions about the type of policy to be enforced. The flexibility of the mechanisms allows the adaptations of standard policies typically needed in practical settings, and enables the combination of elements from different policy models. In this section, we discuss the three major access control paradigms and we show how these policies can be

elegantly encoded in WebDSL.

### 4.3.1 Mandatory Access Control

Mandatory Access Control (MAC) [Sandhu 1993; Samarati and di Vimercati 2001] models are based on assigning labels (e.g. `TopSecret`, `Secret`, `Confidential`, `Unclassified`) to subjects and objects for determining access permissions. Subjects have a clearance label that indicates what type of resources the subject can access. Objects have a classification label which represents their level of protection. The relative importance of labels is determined by a partial order on labels. In MAC policies the distinction between the user (human interacting with the system) and the subject (process working on behalf of the user) is important, because a user can create a subject at any clearance label dominated by theirs. Domination of a clearance label means the label itself and all below it in the hierarchy.

MAC policies are mainly aimed at preserving confidentiality of information contained in objects. Protection of confidentiality deals with information flow, by preventing unsafe transfer of (the contents of) objects to other security labels in the system. To protect confidentiality two properties must hold. (1) The *simple security rule*, also known as the *read-down property*, states that a subject needs to have a security clearance higher than or equal to the security classification of an object to be able to read it. (2) The *liberal \*-property*, also known as the *write-up property*, states that a subject needs to have a security clearance lower than or equal to the security classification of an object to be able to write it. This is needed to prevent leaking confidential information to lower clearance labels. A stricter form can be used to prevent low subjects to overwrite high data (which is possible with the liberal \*-property). This form only allows writing where the clearance matches the classification, and is known as the strict \*-property.

The policy in Figure 4.11 is an encoding of the MAC policy in WebDSL. The policy considers reading as accessing the `topic` view page for a `Topic` object, and writing as saving a `Topic` with the `createTopic` page, which can only be used to create a page; editing existing topics is not possible in this example. The users in the model are simply the `User` entities. To provide a distinction between user and subject, the `securityContext` is extended to represent the current subject with an active clearance label.

Clearance and classification labels are represented by the entity `Label`. The partial order on `Labels` is represented by the `higher` and `lower` properties, which represent the direct parent and direct child `Labels` of a `Label`, respectively. The `dominates` predicate defines the transitive closure of the relation. The `User` and `Topic` entities are extended with a property representing the security clearance and classification, respectively. The `activeClearance` property of the `securityContext` session entity represents the clearance label of the subject.

The simple security rule is now implemented by the rule for `topic`, which states that topics may be viewed by subjects with a clearance label dominating

```

1  entity Label {
2    name : String
3    higher : {Label} ( inverse = lower )
4    lower : {Label} ( inverse = higher )
5    predicate dominates( l: Label ) {
6      l == this || Or[ l2.dominates( l ) | l2 in this.lower ]
7    }
8  }
9  extend entity User {
10   clearance : Label
11 }
12 extend entity Topic {
13   classification : Label
14 }
15 extend session securityContext {
16   activeClearance : Label
17 }
18 access control rules
19   rule page topic( t: Topic ){
20     activeClearance.dominates( t.classification )
21   }
22   rule page createTopic {
23     true
24     rule action save( t: Topic ){
25       t.classification.dominates( activeClearance )
26     }
27   }
28   rule action activateClearance( c: Label ){
29     principal.clearance.dominates( c )
30   }

```

---

Figure 4.11 Mandatory access control.

the classification of the topic. The liberal \*-property is implemented by the rule for the `save` action of the `createTopic` page: a topic can only be created if its classification (as indicated in the form of the `createTopic` page) dominates the subject's clearance label. The activation of the subject's label must also be protected to complete the implementation; only labels dominated by the principal's clearance label can be used as active clearance of the subject.

Note that administration of user-label assignments and editing of labels is not part of the MAC model and we followed this model by having predefined user-label assignments and no editing of labels.

### 4.3.2 Discretionary Access Control

Discretionary Access Control (DAC) models are based on listing permissions for users and objects [Samarati and di Vimercati 2001; Sandhu and Samarati 1994]. The user's identity and authorizations determine the permissions granted for each object. DAC policies are usually closed policies, only specifying the granting authorizations and denying by default. DAC policies often use the concept of ownership to determine permissions, the user that creates an object becomes the owner and has all the permissions for it. The owner can allow other users access to its owned objects (this decision is at the owner's discretion). This also puts the administration tasks in the hands of the owner.

```

1 extend entity Topic {
2   owner      : User
3   viewers    : {User}
4   editors    : {User}
5   moderators : {User}
6 }
7 access control rules
8   rule page topic( t: Topic ){
9     principal == t.owner || principal in t.viewers
10  }
11  rule page editTopic( t: Topic ){
12    principal == t.owner || principal in t.editors
13  }
14  rule page editTopicACL( t: Topic ){
15    principal == t.owner || principal in t.moderators
16  }
17  rule page changeTopicModerators( t: Topic ){
18    principal == t.owner
19  }

```

---

Figure 4.12 Discretionary access control.

These tasks include granting other users access to the object, revoking that access, allowing others to help with administration (delegation), or simply deleting the object. Policies vary greatly in administrative capabilities available, some of the variation possibilities are: allowing ownership transfer; the granting of administration tasks to others can be limited (for example, only one person can get these permissions besides the owner); the revocation of the permissions granted can be linked to the user that specified the permissions.

DAC policies are often described using the Access-Matrix Model [Sandhu and Samarati 1994], a generic model for describing access control policies. It is based on the idea that all resources controlled by a computer system can be represented as objects. By listing all the permissions for these objects, the entire access control system can be described. To use the access matrix model, one needs to identify *objects*, the resources that need to be protected, *subjects*, the users or processes created by the user that need to access objects, and *permissions*, the operations that apply to an object and which need to be protected in the system. These concepts are used to describe a policy matrix, with subjects as indices for rows and objects as indices for columns. The set of permissions, i.e. the operations a subject *s* may apply to object *o* is listed in the matrix at  $[s,o]$ . Since the Access-Matrix is usually sparse, it is rarely stored as an actual matrix in a system. Instead the matrix is represented by means of an *access control list* (ACL), with each object holding a list of the subjects and their permissions for that object, or as a *capabilities list*, with each subject holding a list of the objects and the permissions the subject has for those objects, or as an *authorization table*, storing permissions as triples of subject, object, and permissions.

The discussion of WebDSL access control in Section 4.2 already presented elements of a DAC policy. The example in Figure 4.12 is a variation using ownership of objects. The owners determine the configuration of the ACL for the objects they own, but can also promote other users to be able to configure

the ACL, implying a one level granting of administration rights. Ownership is represented by means of an `owner` property in the `Topic` entity. An ACL is used to hold the access rights, this is added to the object, in this case the `Topic` entity. The `moderators` set of users consists of the users with rights to change the `viewers` and `editors` sets of the corresponding `Topic`. The view and edit pages are protected by rules that verify that the principal is either the owner, or another user that is specifically allowed a certain type of access. The page for editing the ACL of a topic is accessible only to the owner of the topic or one of the designated moderators. Finally, the set of moderators of a `Topic` can only be changed by the owner (one level granting of administration rights). This implementation is an alternative (more pure) DAC policy, without the RBAC elements that are present in the example from Section 4.2.

### 4.3.3 Role-Based Access Control

Role-Based Access Control (RBAC) [Sandhu and Samarati 1994; Samarati and di Vimercati 2001; Sandhu 1998; Sandhu, Ferraiolo, and Kuhn 2000; Ferraiolo, Kuhn, and Chandramouli 2003] models have been the basis for access control research in the last decade and they are also widely applied in application frameworks. The observation leading to RBAC is that individual users are usually not that important in deciding on permissions (besides auditing purposes), rather it is the task they need to perform that determines the necessary permissions. A *role* corresponds to a group of activities needed to perform a job or a task. These activities form the permissions that are linked with the role. When users are assigned to roles, they gain the permissions assigned to those roles. This better reflects organizational structures, which make common operations easy. For instance, a change of function inside an organization only requires a change of role assignment in the access control system. This action would have been a lot more complicated in a DAC policy where the permissions are directly linked to the user.

The main benefits of RBAC are: *Access control administration*: user/role assignment and role/permission assignment are separated. The administrator is mostly concerned with user/role assignment, so the role/permission assignment can be hidden in an application. *Hierarchical roles*: many applications consist of a natural hierarchy of roles, where some roles subsume the permissions of others. *Least privilege*: a user can activate the minimal role able to perform a task, this can protect the user from malicious code or inadvertent errors (similar to MAC policies). *Separation of duties*: no user should have enough permissions to abuse the system on their own, this can be enforced by separating the steps in critical actions among roles. For example, a user should not be able to create fake payments and also accept them. *Constraint enforcement*: the roles can be extended with constraints on activation or assignments, this allows more specialized access control policies.

The formalisation of RBAC in [Sandhu 1998; Sandhu, Ferraiolo, and Kuhn 2000] proposes a family of models for RBAC. RBAC<sub>0</sub> is the basic model, which consists of users, roles, permissions as entities. Role assignment to users and

```

1  entity Role {
2    name      : String
3    juniors   : {Role}
4    predicate equalOrSenior( r: Role ){
5      r == this || Or[ r2.equalOrSenior( r ) | r2 in this.juniors ]
6    }
7  }
8  extend entity User {
9    roles : {Role}
10 }
11 extend session securityContext {
12   activeRoles : {Role}
13 }
14 var admin  := Role{ name := "Administrator" }
15 var viewer := Role{ name := "Viewer" }
16 var editor := Role{ name := "Editor"
17                   juniors := { viewer } }
18 access control rules
19   predicate isActive( r: Role ){
20     Or[ r2.equalOrSenior( r ) | r2 in activeRoles ]
21   }
22   rule page topic( t: Topic ){ isActive( viewer ) }
23   rule page editTopic( t: Topic ){ isActive( editor ) }
24   rule page editRoles( u: User ){
25     isActive( admin )
26     rule action save {
27       ! ( administrator in u.roles
28         && (viewer in u.roles || editor in u.roles) )
29     }
30 }

```

---

Figure 4.13 Role-based access control.

permission assignment to roles determine the configuration of RBAC<sub>0</sub>, which also provides the concept of a session, which is an activated subset of the user's roles. The permissions from the roles in the session are the ones that can be used in access control decisions. The concept of user controlled sessions creates a distinction between subject and user similar to MAC policies. RBAC<sub>1</sub> introduces role hierarchies to model lines of authority and responsibility. Senior roles inherit the permissions of junior roles, and junior roles inherit the user assignment of senior roles (other implementations of role hierarchies allow activation of junior roles to support hierarchies). RBAC<sub>2</sub> adds constraints to the RBAC model.

The notion of groups used in the example of Section 4.2 is similar to roles. They are activated by the user and carry permissions with them. The example presented in Figure 4.13 gives an RBAC implementation with hierarchical roles — editor as a senior role of viewer — and separation of duty constraints — administrators may not be viewers or editors. The permissions are encoded in the policy, and role/permission assignment is fixed during execution. This is a reasonable simplification that is used in many practical solutions such as Acegi [Alex 2008] and Seam [Yuan and Heute 2007]. The data model extension to implement RBAC consists of a `Role` entity, a reference to a set of `Roles` in the `User` entity, and a reference to a set of roles in the `securityContext` to represent the activated roles (roles in the session). The role hierarchy is constructed by specifying which roles are the direct junior roles in the `juniors`

property. The `equalOrSenior` predicate added to the `Role` entity verifies whether a role is equal or senior to another role. In this example application, the roles are defined statically as application variables. The access control rules, then, define that viewing a `Topic` requires the viewer role to be active or to be a junior of another activated role (editor), and editing a `Topic` requires the editor role to be active. Note that inside an access control rules section the members of the `securityContext` are directly accessible, which is why `activeRoles` can be used here instead of `securityContext.activeRoles`. For administration purposes the page `editRoles` is created that allows editing the `roles` property of a `User`. This page requires the administrator role to be active. Furthermore, implementing separation of duty, the save operation has an additional check to prevent illegal changes to the user/role assignments.

## 4.4 Transformational Semantics

In the previous sections we have seen that access control is defined separately from the rest of a WebDSL definition. In order to enforce the rules specified in a policy, local checks are introduced into pages, templates, and actions. In this section, we give a high-level description of the weaving transformations realizing this. The resources of a WebDSL application that can be protected are pages, templates, and actions. Furthermore, it is possible to restrict access more specifically to resources relative to other resources, e.g., actions within pages and actions within templates, or to resources that use another resource such as actions that use functions. The semantics of protecting a resource differs for these resources and the combinations, which indicates the need for the semantics description given in this section. In the transformations we use  $\vec{x}$  to denote a list formal parameters  $x_1 : S_1, \dots, x_n : S_n$ .

### 4.4.1 Policy Normalization

The first step in the implementation is the normalization of rule sets and the policy definition to a single set of non-overlapping rules, as defined in Figure 4.14. (Note that only the rules for `OR` are shown; the rules for `AND` are dual, with `&&` instead of `||`.) We say that two rules *match* if they have the same signature, i.e., resource type, name, and parameters are the same. If a rule set contains two matching rules, they can be merged into a single rule with the conjunctions of the two expressions as expression (*Combining Rules*). The `OR` operator applied to a pair of matching rules turns into a single rule with the disjunction of the expressions. The `OR` operator applied to two sets of rules produces the pairwise disjunction of matching rules. That is, assuming that the argument sets are normalized and thus contain for each resource at least one rule, if a matching rule exists in each set they are combined with the `OR` operator. Rules for which no matching counterpart exists are taken as is.



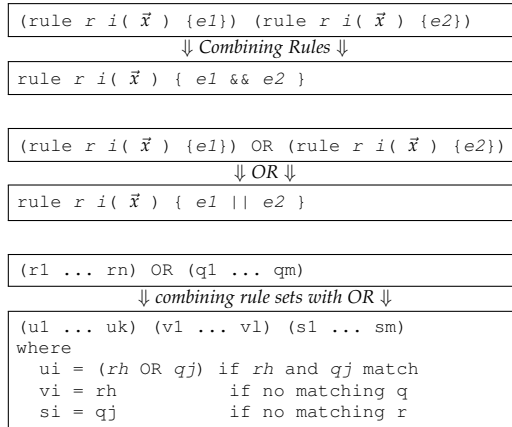
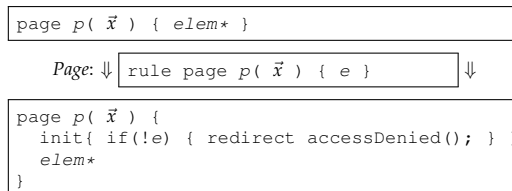


Figure 4.14 Policy normalization.

## 4.4.2 Rule Weaving

Figure 4.15 defines the instrumentation rules that insert checks into pages and actions. As an example we consider the following simplified page rule transformation:



The rule states that an access control rule with signature `page p( $\vec{x}$ )` inserts an `init` block in the page definition with the signature `page p( $\vec{x}$ )` containing a redirect to the `accessDenied` page in case the condition of the rule evaluates to false. The notation is slightly simplified here for clarity, in the actual implementation the  $\vec{x}$  also matches with different names for the arguments and the inserted `e` has the correct names substituted accordingly.

The weaving of pages is more interesting if the page has other initialization statements, which is shown in the *Page* transformation in Figure 4.15. The other transformations shown are the following: *Action* protects the execution of an action, *Template* controls the view of the template's elements, *PageAction* shows what a nested action check in a page means, *TemplateAction* shows a nested action check in a template, *Navigation* is the semantics of inferring navigation link visibility from page rules ( $e'$  is  $e$  with formal arguments  $\vec{x}$  substituted by actual arguments  $\vec{y}$ ).

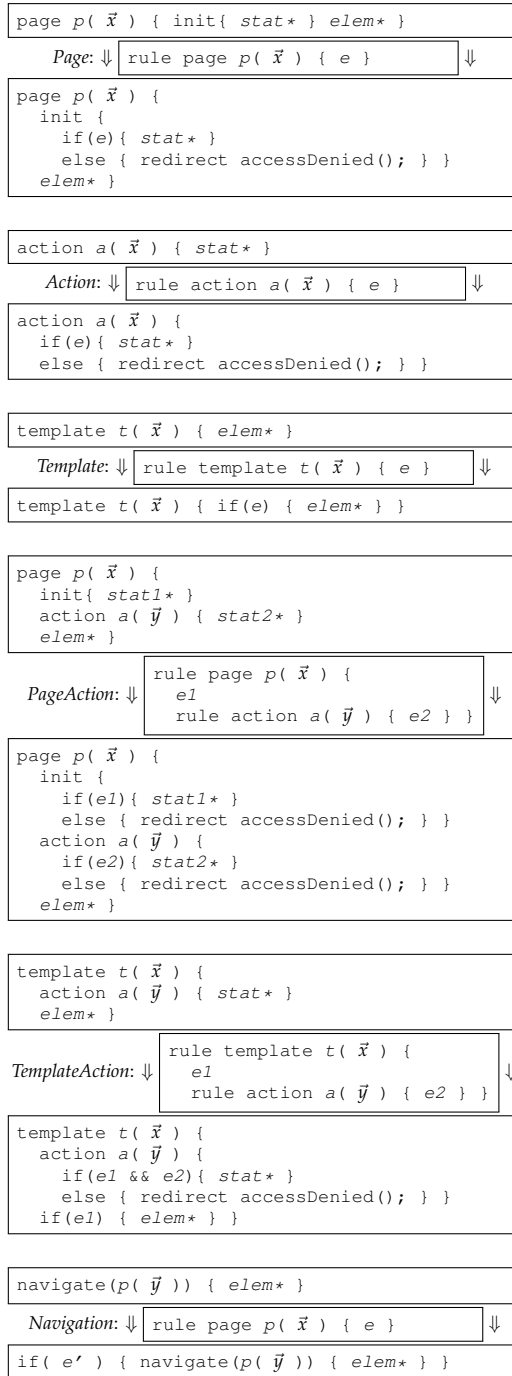


Figure 4.15 Weaving transformation rules.

## 4.5 Related Work

In previous work our research group has developed MetaBorg [Bravenboer and Visser 2004], an approach for embedding domain-specific languages in general-purpose languages, and StringBorg [Bravenboer, Dolstra, and Visser 2010], an approach for syntactically embedding query and shell languages to prevent injection attacks. This work extends the repertoire to the embedding of DSLs in DSLs with a global-to-local weaving transformation as *assimilation*. We have implemented WebDSL, in general, and the weaving transformation, in particular, with the transformation language Stratego [Visser 2004].

### 4.5.1 Language Design

Mikkonen and Taivalsaari [2007] argue that software engineering principles have degraded with the recent paradigm shift to web applications. The incremental growth from static HTML pages to desktop application replacements has left a trail of languages which require tool support to cope with. Developers need to learn these technologies which prevents them from focussing on learning actual web application design principles, such as access control. We review the software engineering principles of Mikkonen and Taivalsaari [2007] for WebDSL access control: *Separation of concerns*: One of the main goals of WebDSL access control is achieving separation of concerns while still having an integrated language with static verification. WebDSL applications can be easily adapted to support a different access control policy. *Information hiding*: The details of how and where access control is applied is hidden in the semantics of access control rules. *Consistency*: Access control checks are specified in the same expression language that is used in other parts of WebDSL applications. *Simplicity*: one mechanism can be used to define a wide range of policies. *Reusability*: Policy specifications can be reused for other pages and in other WebDSL applications. *Portability*: WebDSL access control is translated to normal WebDSL which is a model that abstracts from platform specific details.

Another benchmark is the requirements formulated by Evered and Bögeholz [2004] for an ideal access control mechanism based on a case study of a health information system. *Concise*: Access control checks in rules simply become the boolean WebDSL expression that is executed for determining access. Reuse through predicates limits the amount of mechanical repetition. *Clear*: Matching of access control rules with resources is based on clear semantics and can easily be verified to be correct. *Aspect-oriented*: Access control can be specified separately from the rest of a WebDSL application, weaving takes care of integration. *Fundamental*: All resources that can be protected by WebDSL access control are denied access by default, this forces the application developer to explicitly specify conditions for access. *Positive*: The access control rules determine conditions for allowing access, they are positive authorizations. *Need-to-know*: Access control can be used to hide information on pages, more specifically template contents and navigation links. *Efficient*: Because access control is integrated with the rest of the WebDSL application it can use the

same database interface and caching mechanisms.

Tschantz and Krishnamurthi [2006] present a set of properties for examining the reasonability of access control policies under enlarged requests, policy growth, and policy decomposition. We discuss their properties for WebDSL access control. *Deterministic*: If the application's data is considered part of the policy, then identical requests always result in the same access control decision. *Totality*: A decision to allow or deny is always made. The default to deny can be caused by a condition evaluating to false, no rule matching the resource, or an error occurring during checking. *Safe*: Since WebDSL access control is integrated with the application it is not possible to make incomplete access control requests. *Independent composition*: It is possible to reason about rules in isolation, combining them will not change the result of an *individual* rule. *Not monotonic*: Decisions can change from granting to non-granting by adding another rule, caused by the single rule combination strategy of taking the conjunction of matching rules. We believe that the standard way of combining rules by conjunction is easier to comprehend than having multiple rule combination strategies, and it results in a deny overrides strategy that we consider a safe combination of rules.

#### 4.5.2 Policy Languages

The Ponder policy specification language [Damianou et al. 2001] is a policy language aimed at specifying access control for firewalls, operating systems, databases, and Java programs. The language has many features such as built-in notions of groups and roles, delegation, obligations (which specify what a user must do), meta-policies for defining constraints on the policies themselves (for example Separation of Duties). The paper presents examples of policies to cover the different elements available in the language. However, there is no information on how the policies are enforced in different contexts. This makes it hard to compare the results of using Ponder to specify policies and our own approach. Several of the elements available in the Ponder language, such as delegation, obligations, and policy reuse, are still open for exploration in WebDSL and provide options for future work.

XACML (eXtensible Access Control Markup Language) [Moses et al. 2005] is a standard that describes both a policy language and an access control decision request/response language (both written in XML). A policy is described using rules that specify conditions for being applicable to a request. The requests in the request/response language are access control queries and the responses can be permit, deny, indeterminate (an error occurred) or not applicable (cannot answer this request). In WebDSL we opted for 'allow' and 'deny' as the only results for access requests. XACML rules are combined in policy sets, and policy sets in a policy, both of these operations are controlled by selecting a combination algorithm. In WebDSL access control rules are combined by using the conjunction of the expressions, rule sets can be combined in a policy with a boolean expression over the sets. For finer grained authorization in XACML attributes are used, these are characteristics of the subject, resource, action, or

environment in which the access request is made. Most of these attributes have to be specified in the XML request message when checks are needed. WebDSL access control is integrated with the WebDSL application, there is no need for creating requests and explicitly transmitting all the required data, this avoids any inconsistencies.

### 4.5.3 Frameworks

Acegi [Alex 2008] is the security component of the Spring Java web application framework. In keeping with the Spring approach, Acegi is based on XML configuration of the framework components. Enforcing access control is done by an aspect mechanism, these aspects are the components that need to be configured to protect either URLs or specific methods. The weaving occurs when the protected resource is invoked for the first time. In WebDSL access control weaving is done statically which allows better error reporting and compile-time guarantees. In Acegi a security context is available where the authenticated user can be stored together with a collection of granted authorization objects, for instance roles. This can be used to specify RBAC checks in the XML configuration. These checks are application wide and cannot be further customized with conditions. An active roles collection as used in the RBAC example in Section 4.3 for enforcing the concept of least privilege has to be managed in the application to work in this framework. When fine-grained access control is needed, Acegi offers a DAC policy implementation which protects generic objects by listing the permissions available for each user. This is stored in different database tables than the actual application. The combination of RBAC and DAC has to be specified in the XML configuration. Fine-grained access control must be encoded in the generic DAC policy, which might involve duplicating information available in the application such as ownership of an object. WebDSL provides an integrated way to specify access control, where policies are combined in the rules, data is integrated with the application, and accessible for administration.

The Seam Java web application framework [Yuan and Heute 2007] offers a security API for access control. The basic mode is controlled by including restrict annotations in the application code which verify whether a principal has a certain role when accessing the annotated Java method or page URL. A DAC policy can be implemented in a reserved function that takes the type of the object, the action, and a reference to the specific object of the access request. This is similar to the options Acegi has to offer and suffers the same drawbacks. The advanced mode consists of using JBoss Rules [Yuan and Heute 2007] for determining permissions. This is a logic language with an inference engine that can deduce permissions from facts available in the engine. These facts specify the access control policy. Although JBoss Rules allows separation of concerns for access control, it is a separate engine which must be invoked correctly. Any check based on information related to data in the application needs to have this data supplied in the requests. WebDSL access control is integrated with the application and does not have the risk of inconsistencies between known

data during access control checks and actual data of the application. Besides the integration issue, the semantics of the rules is not specific to access control but simply generic JBoss Rules, which prevents any assumptions to be made about the access control policy (such as the visibility inference in WebDSL).

## 4.6 Discussion

While WebDSL access control elements are specified in separate language constructs, the effect requires integrating with the WebDSL application. The first type of integration is the use of WebDSL expressions for specifying access control conditions. Besides the expressions, the data model is also completely accessible for use in checks. Matching of arguments allows access to all the data relevant for the type of resource that a rule protects. This data model integration works the other way around as well, the data from WebDSL access control is available in the WebDSL application which provides a simple way to produce administration support. The semantics of the access control rules illustrate that it is a pure WebDSL to WebDSL transformation, which lets WebDSL access control reuse all the code generation.

WebDSL access control encodes policies and can be seen as a high-level mechanism for access control. The default access control decision of denying access provides a safe default and allows an incremental approach when specifying the policy. Section 4.3 shows that WebDSL access control is transparent and concise in expressing Mandatory, Discretionary, and Role-Based Access Control and provides good support for the management of such policies. Besides these paradigms there is a strong connection with the application allowing application-specific customizations to be expressed easily.

By viewing access control as a proper language element, it becomes possible to infer related elements from the policy specification. Assumptions can be made about navigation to pages and visibility of page elements. This greatly increases the productivity of application developers and also helps to make sure the applications produced are consistent. This is a distinction from using generic aspect mechanisms for access control, because there the semantic value of access control rules is lost.

The applications we have build in WebDSL so far are isolated applications. If the need arises to incorporate a WebDSL application into a larger system and access control is going to be handled externally, this would only require changing the checks in the rules to poll the external system instead of performing a local check. The separation of access control rules will help this conversion, and the semantics of access control rules as discussed in this paper are still applicable. A similar argument can be made for authentication, the user representation and authentication function can be customized to use an external authentication system.

### 4.6.1 Future Work

Although we have created a flexible language for access control, abstractions for common policies are still possible. Mainly for RBAC, built-in abstractions can provide a more concise mechanism to support a role based policy. Separation of duty checks as presented in the RBAC example in Section 4.3 could be expressed declaratively instead of explicitly specifying where the check should be enforced.

The access control rules specified in WebDSL are coupled with the presentation of the application, this allows inconsistencies where similar pages have different rules. Better abstraction and encapsulation of entities and their operations is needed in WebDSL which will allow access control to protect entities and their specified interface directly and infer the current type of checks. This would require tracing the use of these interfaces in pages, but will provide better consistency verification of the access control policy.

Another approach is to declare access control directly on data operations. The access control rules state which entities and entity properties are allowed to be retrieved or edited. A downside of such an approach is that it must hook into all data retrieval implementation aspects. Besides potential overhead, there are also more conceptual problems, in particular regarding collections. For example, what if an ordered list of entities is retrieved, of which some of the entities are not allowed to be read. It is not clear whether there should still be partial access to this collection, since part of the entities are accessible. List operations would have to take into account the inaccessible entities, which could be considered an information leak. If partial access has the effect of making the entire collection fail to load, it would likely lead to confusing errors in the application operation. A situation where data access control seemingly works well is the search page. Entity objects are retrieved separately as search results which may need to be filtered. However, even there it cannot be implemented in a generic way. Besides hiding results, also the statistics such as the number of hits need to be updated. Faceted search shows categories derived from the search results, and would also be affected. These consequences would have to be addressed in the search implementation itself.

Hiding navigation links to inaccessible pages is not always the best solution to indicate an access control issue. A more generic mechanism of handling inaccessible links could be added. For example, to display the link as regular text or greyed out. Or to inform the user why access is prevented and whether and how access can be obtained.

Logging of access control requests/decisions (not just writing to a text file but integrated logging, persisted like other application data) is required for doing access control audits and intruder detection. Adding this to WebDSL would be a major improvement. An important point here is that the amount of logging data can easily become unwieldy; a specification is needed that determines what information is stored, how detailed the entries are, and when they may be deleted or archived.

The DAC policy example presented in Section 4.3 included some facilities for delegation of access control, this could be specified in a more general way. Several models have been proposed for delegation [Zhang, Ahn, and Chu 2003; Zhang, Oh, and Sandhu 2003], which also shows there is a need for abstracting delegation details from access control policies. Supporting high level definitions of access control delegation would be an interesting addition to WebDSL.

Temporal policies have been modeled in the literature [Latif 2005] and the application of such models would provide an interesting case for WebDSL access control. We have investigated an extension of WebDSL with workflow abstractions [Hemel, Verhaaf, and Visser 2008], which also provides options for exploring access control policies in that area. This can be compared to other studies regarding workflow and access control [Bertino, Ferrari, and Atluri 1999].

Digital rights management has been connected to access control in [Park and Sandhu 2004], where a conceptual model for usage control is presented. Adding this functionality to WebDSL would provide useful insights in the applicability of such an approach.

## 4.7 Conclusion

The extension of WebDSL with a declarative access control language provides us with insight in how enforcing access control could be done better in general web applications. Firstly, the use of integrated, access control specific, aspect-oriented language elements result in a clear extension of the base language. Secondly, WebDSL access control shows that various policies can be expressed with simple constraints, allowing concise and transparent mechanisms to be constructed. Finally, the advantage of having a language element for access control, allowing assumptions to be made about the related parts in the application. Practical solutions for access control often consist of libraries or generic aspect-oriented implementations of fixed policies. These rarely have clear interfacing capabilities and require manual extension and integration with the application code. The extensions and integrations provide room for errors that possibly invalidate the whole access control policy. The realization that access control as a language element is necessary will provide the means to defeat the errors caused by encoding policies in application code. Integration of the language means that language extensions influence the semantics of access control elements. New basic elements and new abstractions require new rule types. Using transformational semantics, access control rules can be defined clearly and concisely.

With this chapter concluded, the major WebDSL language features have been described. In Chapter 5 we will investigate WebDSL language design iterations, using a generic pattern used to guide feature evolution.





# The WebDSL Language Evolution Pattern

---

# 5

## 5.1 Introduction

A healthy programming language is continuously developed. Practical applications provide invaluable feedback on further directions for language design. New abstractions are introduced when additional concerns are discovered. Existing abstractions may require reimplementations with better runtime components. The evolution of a programming language can be described as a continuous process with an iterative pattern of creating libraries, implementing linguistic abstraction, and expanding core language features. This chapter explains the WebDSL language evolution pattern, and shows its application to the major language features.

Besides the main abstractions for data persistence, user interfaces, and functions, a web application can require many additional reusable features. For example, authentication through an external single sign-on server, generating PDF reports, sending and receiving emails, providing and invoking a web service API, file upload and download, and customized internal site search. The WebDSL language provides hooks into the underlying platform to incorporate external libraries. In particular, Java libraries can be included and imported within the WebDSL application to make them available. JavaScript widgets and user interface components can be imported into templates. JavaScript fragments can be embedded in WebDSL templates, with the option to insert dynamic values such as a DOM element id value. By using these generic extension hooks, an application can lose some of its analyzability. However, a completely closed system that does not provide any escape mechanism is not very practical. To regain analyzability and increase coverage, a commonly required feature can be integrated into the language. This allows for a properly designed abstraction with custom syntax, semantics, and static verification.

Voelter et al. [2013] describe 7 design dimensions for Domain-Specific Language (DSL) design. This relates directly to the WebDSL language evolution pattern, because language evolutions aim to improve the language in one or more of these design dimensions.

1. *Expressivity* A language  $L_1$  is more expressive in a domain than a language  $L_2$ , if for each program in the domain, the size of the program encoded in  $L_1$  is smaller than the encoding in  $L_2$ . Programs for a particular domain typically use a set of characteristic idioms and patterns. A language created for this domain can provide linguistic abstractions for those idioms or patterns, which makes their expression more concise and simplifies their analysis and translation. Abstractions can be built into the language (linguistic abstractions), or they can be expressed by concepts available in the language (in-language abstractions). A language

with support for in-language abstraction, can be used by the language developer to provide collections of domain-specific abstractions in a standard library to language users.

WebDSL input handling templates were initially using high-level linguistic abstraction with direct code generation, lacking flexibility for customization. Subsequent implementation iterations evolved to a standard library of components using smaller core linguistic abstractions to enable in-language abstraction. The resulting solution retains the original high-level abstractions, while also having flexibility to customize and make variants.

2. *Coverage* A language fully covers a domain if each program relevant to the domain can be written in the language. The coverage of a domain by a language is the percentage of programs in the domain that can be expressed in the language.

In the WebDSL language design, choices cater for typical web information systems. One example is the automatic handling of page URLs, which are derived from the page name and arguments separated by slashes. This was a limitation in coverage as some applications require free-form URL construction. We extended the WebDSL language with a feature for URL customization, to increase coverage.

3. *Semantics*

Semantics can be partitioned into static semantics and execution (or dynamic) semantics. Static semantics are implemented by the constraints and type system rules. Execution semantics denote the observable behavior of a program and can be defined using a function that maps a program from one language into a more general language that has the same observable behavior. The technical implementation of the mapping can be a literal transformation to another program, or an interpreter that reads and executes the program.

For WebDSL, platform conformance through code generation was the initial strategy for execution semantics. Generated code can be tailored to any target platform. The code can look exactly as manually written code, without additional support libraries. The initial exploration of the web programming domain relied on observing and capturing patterns in solutions meant for regular Java programming (without code generators). When the requirements for WebDSL and limitations of this initial target became more clear, the generated code moved to more WebDSL specific code, such as a custom framework for user interfaces. Chapter 6 provides an overview of defining static semantics and execution semantics for WebDSL.

4. *Separation of Concerns*

A domain may be composed from different concerns that each cover a different aspect. Either a single integrated language can be designed or

separate concern-specific DSLs. There may also be cross-cutting concerns that could be handled by the execution engine, or modularized in the DSL, or remain cross-cutting in the DSL.

WebDSL provides specific languages for web programming concerns, but linguistically integrates them into a single language. WebDSL developers have freedom in factoring declarations into modules, such as grouping data models together, or combining data model and user interface for a specific application feature. Access control is a cross-cutting concern that has been modularized in the DSL, the generator automatically injects permission checks into the generated code. Access control rules can be defined separately from the components being protected, such as page definitions.

#### 5. *Completeness*

Completeness refers to the degree to which a language can express programs that contain all necessary information for execution. Integrating additional code or specifications in the case of an incomplete language can be done in several ways: calling code written in another language by declaring a foreign function interface, directly embedding another language, composing manually written code without modifying generated files, and inserting manually written code in protected regions of generated files.

In WebDSL, a native interface provides access to construct and call Java classes with static and non-static methods and fields (foreign function interface). JavaScript can be added in the user interface template declarations (language embedding). WebDSL does not analyze this JavaScript, however, this feature enables invoking many useful JavaScript functions and libraries. Native class interfaces and JavaScript embedding can be used as a first step in identifying a new language feature for WebDSL. Library integration can be explored without immediately having to design syntax and update the compiler for full linguistic integration. For example, the search sublanguage was iteratively developed, starting with native class wrappers around the Hibernate Search [2023] and Lucene [2023] libraries. Next, it was improved with custom syntax and checks for declaring search mappings and querying, providing full linguistic integration.

#### 6. *Language Modularity*

Reuse of modularized DSL parts makes designing DSLs more efficient.

One example of language embedding in WebDSL is the integration of the Hibernate Query Language (HQL) syntax as expressions. Queries can refer to WebDSL entity types and to variables in the scope of the query expression. Although not used in WebDSL syntax itself, the implementation of WebDSL reuses the JavaFront Stratego library for generating syntactically valid Java code, and enabling term rewriting using concrete Java syntax fragments.

## 7. *Syntax*

Linguistic integration of domain concerns requires syntax design. A good notation makes expressing common concerns simple and concise, and provides sensible defaults. Syntax can be evaluated based on several criteria: writability, readability, learnability for new users, and effectiveness for experienced users.

WebDSL contains sublanguages with syntax designed for a particular technical concern in web programming, e.g. entity declarations concisely describe persisted data. Most of the syntax design happened in the early stages of sublanguage implementation. However, there have been many small incremental adjustments to the WebDSL syntax, such as removing unnecessary keywords and brackets, and providing shorter notations for commonly occurring fragments. Capturing common patterns in the code and linguistically integrating them as new abstractions provides opportunities for improving syntax in the areas of writability, readability, learnability, and effectiveness.

Section 5.2 describes the generic evolution pattern used in the development of WebDSL. The pattern is then explained in more detail with applications to the evolution of core WebDSL language features: user interface templates (Section 5.2.1), persisted data models (Section 5.2.2), and static code template expansion (Section 5.2.3). Section 5.3 analyzes WebDSL language features that are not required in all applications, however, are common enough to be incorporated in the language. The language features are: email notifications (Section 5.3.1), file and image support (Section 5.3.2), and internal site search (Section 5.3.3). Section 5.4 discusses considerations for the effort of creating compiler extensions and code library management, after which Section 5.5 concludes.

## 5.2 A Generic Language Evolution Pattern

Reflecting on the development of WebDSL, a recurring pattern can be identified for the development of features in the language. This pattern is shown in Figure 5.1. It consists of three phases:

- discover new abstractions,
- domain-specific language integration,
- reimplement using new core abstractions tailored to the language.

These phases can be repeatedly applied, when current features can be implemented in a better way, or as additional abstraction opportunities become clear. To illustrate the pattern we walk through the evolution of user interface templates in WebDSL shown in Figure 5.2.

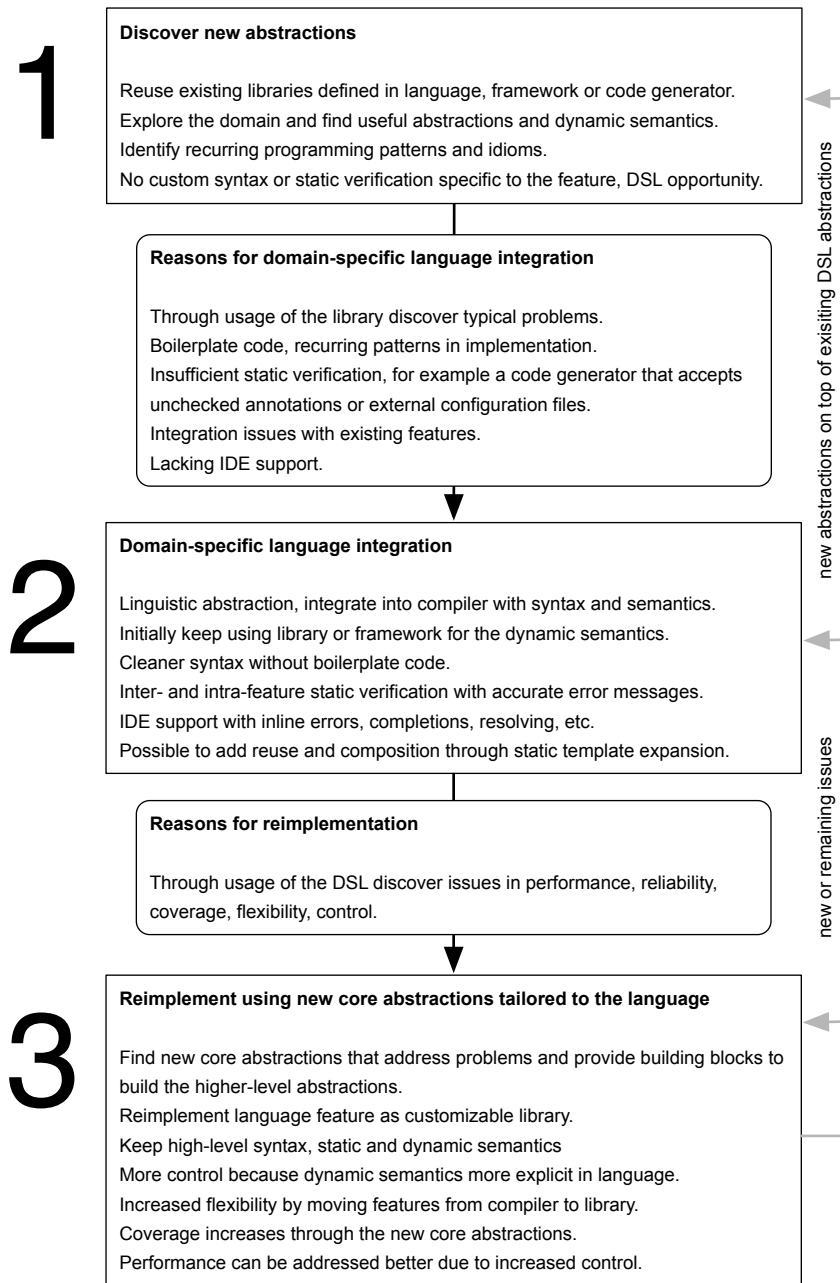


Figure 5.1 Generic language evolution pattern.

### 5.2.1 Evolution of User Interface Templates

The initial features of WebDSL were based on common aspects of web frameworks. To get an idea of the requirements, several frameworks were experimented with. This is part of the first phase. There is no actual DSL abstraction yet, because the domain and scope are not clear yet. These are discovered by seeing what is commonly being used. A more thorough analysis was done on the Seam Java web framework. Application code consisted of Hibernate entity classes for persistence, JSF (Java Server Faces) XML files for rendering pages and supporting Java classes to handle forms and submit actions. The language in this case is the general purpose Java language, together with the supporting languages like XML for templates. There were several opportunities for reducing boilerplate code, by combining templates and backing Java code into a single definition, by abstracting over the persistence annotations required by Hibernate (e.g. different annotations are required for primitive fields, references, and collections), and by automatically generating boilerplate project files required for compilation and deployment. These motivated a move to phase 2, creating a DSL where syntax and semantics for persistence and user interfaces are integrated as language abstractions.

Phase 2 is creating a domain-specific language that generates the application code that was manually written in phase 1. The WebDSL syntax was created for entities, templates, and functions. The compiler provided static analysis on these language features and generated a Seam Java web application. This version of WebDSL was quite useful for prototyping simple web applications. Applications were very concise, due to generating the boilerplate code. Application faults were caught early with static analysis. However, the applications had several problems. Performance was bad, even for simple pages the load time was close to a second. This was likely both due to general aspects of the framework and a specific usage pattern to try to get reliable and clear semantics (which sometimes required workarounds in the generated code). The abstractions were very inflexible, all input and output templates were part of the generator, changing a simple CSS class required changing the generator. Some pages would give large stacktraces caused by an error deep in the framework. The framework relied on in-memory server-side user sessions for handling forms and submit, which polluted the URL with a meaningless identifier, and caused annoying timeout behavior. AJAX support in JSF was not integrated in the design and experiments made this clear as well. Data validation was closely tied to the presentation layer, meaning that it was hard to create a data validation abstraction that covered not just form checks, but also data invariants. The high-level semantics of templates with type-derived inputs and outputs were appealing, but the underlying foundation was shaky. This was motivation for looking at a replacement implementation for the dynamic semantics of templates, and going to phase 3.

Phase 3 is finding core abstractions and reimplementing features as library components. Most of the problems we had were related to the solution for user interfaces in the Seam web framework. By building applications

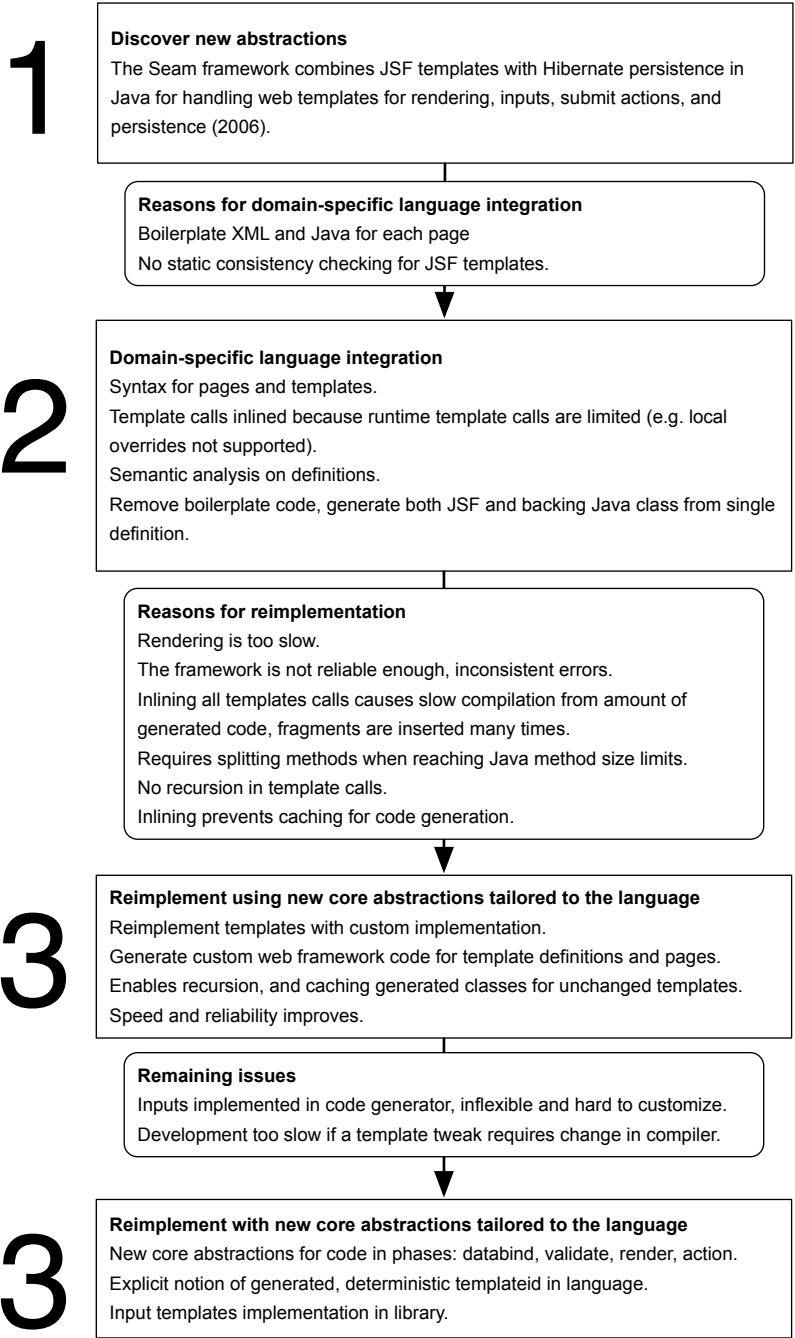


Figure 5.2 Generic evolution pattern applied to user interface templates.



```

1 override attributes inputInt { class = "int-1" } // override class attribute of built-in
2
3 page root {
4   var i := 0
5   form {
6     input( i )[ class = "int-2" ] // class attributes are merged, result: "int-1 int-2"

```

Figure 5.3 Code example of attribute override.

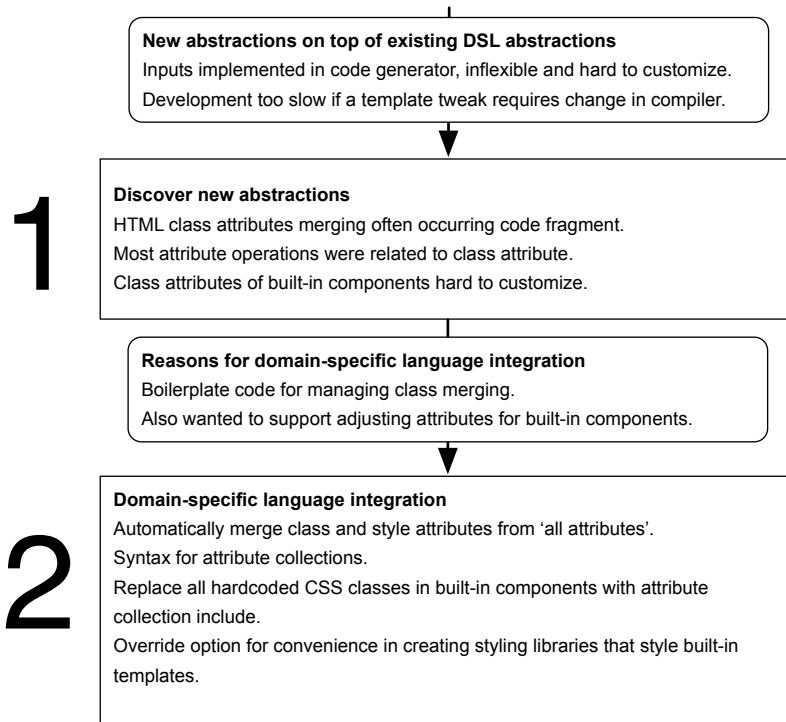


Figure 5.4 Generic evolution pattern applied to attribute collections. Continuation of refining user interface template abstraction.

we discovered the abstractions that were working and those that were not. The main primitive required for building our own template solution was a deterministic and unique template identity generator. This allowed the creation of forms with inputs and submits without the application developer having to specify custom ids. Furthermore, code needed to be executed in phases, to allow data binding and validation before handling actions. The primitives required were the `databind`, `validate`, `action`, and `render` blocks with function code for that specific phase. Input and output templates could now be moved to the standard library, while retaining the high-level syntax and semantics that were introduced in the DSL in phase 2.

The process repeats itself, in the template abstractions we still see opportunity for reducing boilerplate code. This means we go back to phase 1 (although for a smaller part of the language), in this case the code we want to improve is already written as a library in the DSL. We want to improve the library code through introducing new high-level language constructs. For example, a form of generics or code templating in user interface templates would help in reducing duplication of similar templates that work on different entity types with different sets of properties. To address this issue, we added a static code template expansion with IDE support which enables creation of better libraries for input and output templates for all types. This feature is discussed in Section 5.2.3.

Another refinement of the template definitions is related to attribute handling. The most commonly used attribute is the class attribute to set CSS styling for HTML elements. In HTML, multiple classes can be attached as a list separated by spaces. This merging of class attributes was made implicit to avoid having to describe class attribute concatenation in many templates. Similarly, style attributes are automatically merged. Additionally, some classes were part of the built-in library input and output components, these were harder to customize or remove. These required adding the custom class to each call or wrapping the template for just adding a class. To solve this problem, we added an abstraction for attribute collections to the language. This enables referring to a group of attributes belonging to a built-in or library component, and allow overriding them more easily. For example, the HTML style and class attributes for all the `submit` and `submitlink` components in the application can be overridden through the `submit` `attributes`. An example code fragment of attribute overriding is shown in Figure 5.3. This continuation of refining the template language with better attribute handling is shown in Figure 5.4.

Each phase improvement requires design and implementation effort. These are not trivial, especially coming up with new core abstractions to move to phase 3 can be hard. However, this is also where the biggest improvement is for the language. Enabling further abstractions to be built on top of those primitive building blocks. The DSL can then move beyond the original framework it was based on.

### 5.2.2 Evolution of Persisted Data Models

Not all WebDSL language features have received multiple iterations like the user interface templates. In particular, persistence and search heavily rely on Hibernate ORM [2023], Hibernate Search [2023], and Lucene [2023]. These language features have not yet been reimplemented tailored to the language, and can be considered at phase 2. They are wrapping an existing framework with concise syntax and static checks, and generating the boilerplate code. The evolution of persistence is shown in Figure 5.5. In practice, for our applications the persistence abstraction has been largely sufficient, however, there are limitations. For example, performance can become problematic for certain cases, like when editing very large collection type properties.

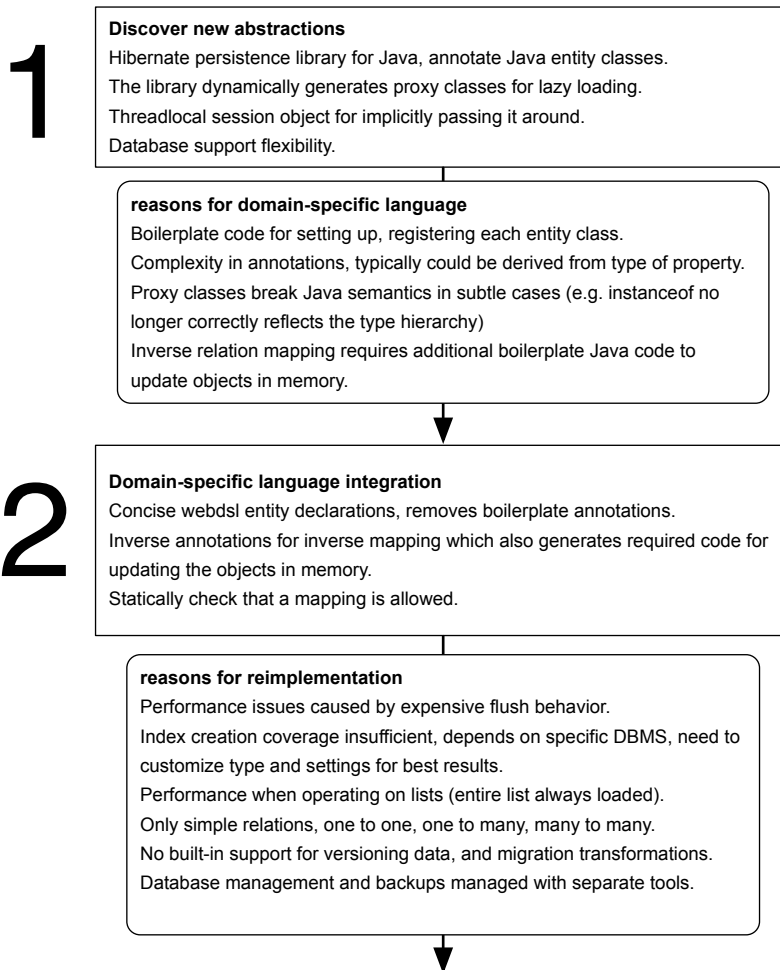


Figure 5.5 Generic evolution pattern applied to persisted data models.

### 5.2.3 Evolution of Static Code Template Expansion

A convenient feature for web application frameworks is to generate code for an administration interface to view and edit entities in the database. An example of this is the scaffolding feature in Ruby on Rails [2023], which generates an admin interface based on the data model. A downside of the scaffolding approach is that as soon as edits are made to the generated code, it can no longer be updated after data model changes without manual intervention. In the initial version of WebDSL we included derive statements that would generate CRUD pages, and view and edit rows for entities. The generated code for these is not exposed to the client, so they can always be kept up-to-date

```

1 expand
2 Int Float Long Bool String Text WikiText Email Secret Date
3 to Type {
4   template input( label: String, s: ref Type ){
5     controlGroup( label )[ all attributes ]{
6       input( s )
7       elements
8     }
9   }
10  template output( label: String, s: ref Type ){
11    controlGroup( label )[ all attributes ]{
12      output( s )
13      elements
14    }
15  }
16 }

```

Figure 5.6 Code example of static code template expansion.

```

expand
Int
Float
to Typename {
  template showTypename( s: Typename ){
    display( s )
  }
}
template display( i: Int ){
  "value: " ~i
}
page test {
  showInt( 42 )
}

```

Figure 5.7 Editor screenshot of static code template expansion. The `Typename` parameter can be used in any AST location where an identifier is expected, and can also be part of an identifier and still get replaced.

whenever the set of properties of the entity is changed. Although it is a useful feature for prototyping, it quickly became clear that more customization was needed to keep these derived definitions useful for real applications. Besides admin pages, and input/output rows, there are other repetitive fragments that can appear in several other syntactic locations, e.g., entity definitions, statements, template elements. To generalize these features, we have designed a static code template expansion mechanism. The defined pattern is syntactically checked, and semantically checked for each instantiation occurrence. Because WebDSL has static verification checks for many cases with descriptive errors, any resulting semantic error can be easily traced to the cause. A code example of this feature is shown in Figure 5.6. The IDE screenshot in Figure 5.7 demonstrates the editor integration of this feature. The evolution of this feature is described in Figure 5.8.

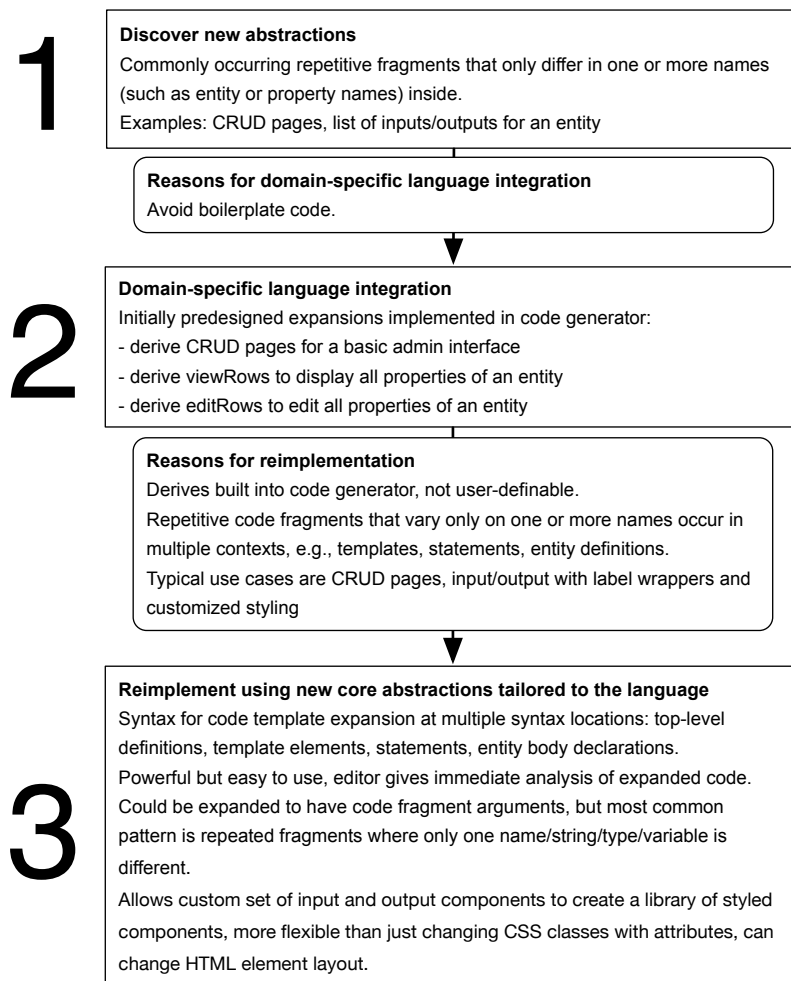


Figure 5.8 Generic evolution pattern applied to static code template expansion.

## 5.3 Evolving Compiler Extensions

Web programming features that are required frequently can be implemented as language extensions in the compiler. This allows specification of a syntax specific to the feature, as well as static analysis and code generation. Because this requires implementation effort from the language designer, the features should be common enough to occur in multiple web applications. In this section, we discuss sending email notifications, file upload and download, and internal site search. These are examples where a feature has been added to the language itself, instead of only building a library.

```

1 email confirmRegistration( u: User ){
2   to( u.email )
3   from( "admin@webdsl.org" )
4   subject( "webdsl.org registration confirmation" )
5   par { "Dear ~u.fullname," }
6   par {
7     "Welcome to webdsl.org. Your registration has been confirmed. "
8     "You can find your profile at "
9     navigate user( u ){ output( navigate( user( u ) ) ) }
10  }
11 }
12 function confirm( u: User ){
13   email confirmRegistration( u );
14 }

```

Figure 5.9 Example usage of email definition and send function.

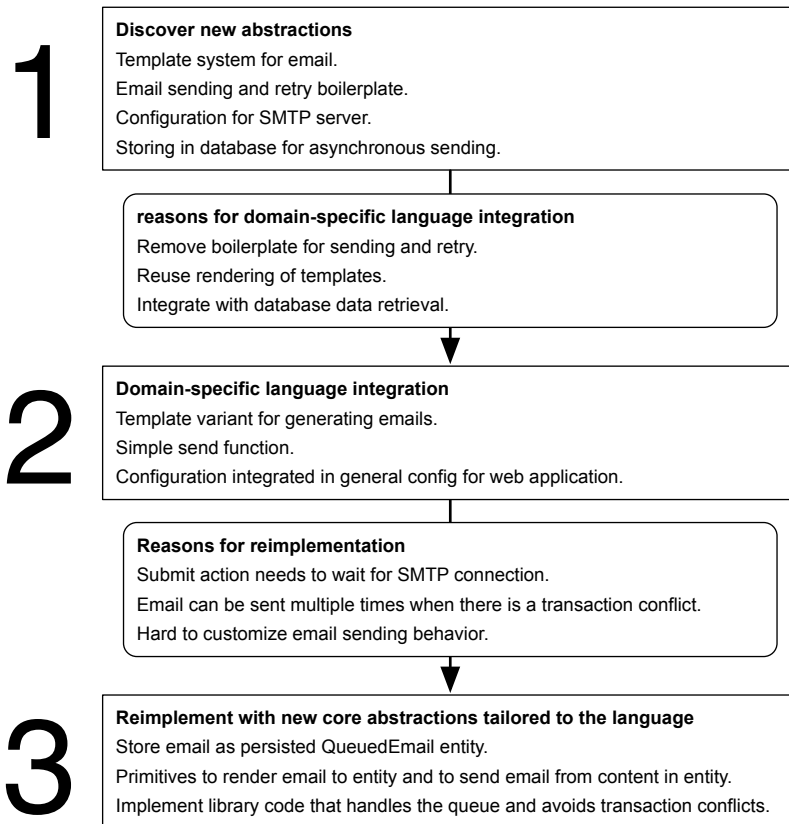


Figure 5.10 Generic evolution pattern applied to email.

### 5.3.1 Evolution of Email Notifications

Web applications often use email notifications to inform the user of important events happening in the system. Examples of such events are the arrival of a new message, moderation needed for a user account, a password reset request, or a change is made by another user in important data. The main function of an application can also be sending email notifications, e.g. in a monitoring application.

Constructing an email is similar to constructing a page. An email is also a template that must be filled in with entity data. Email templates are similar to regular templates, but must be called with the `email` command from functions. An example email template for confirming a user registration is shown in Figure 5.9. The difference with regular templates is that additional definitions are available, namely `subject`, `to`, `cc`, and `bcc` templates. These extra template calls are used to provide the meta information for sending an email. The `send email` command consists of the `email` keyword followed by a call to the email template. Sending email requires configuration settings for the SMTP server. These are configured in the `application.ini` configuration file. The provided abstraction avoids having to specify boilerplate code for rendering an email and connecting with the SMTP server. Email handling code is error-prone, because a small mistake in such code can easily lead to an accidental spambot.

The evolution of the email feature is shown in Figure 5.10. In the initial implementation of this feature, connecting to the local SMTP server was done synchronously, which meant that any email notification triggered from a `submit` function would cause the `submit` action to take several seconds. Another issue was that if the action failed for some other reason, e.g. a database transaction conflict, an email could be sent again. The feature was improved by making the sending asynchronous, the email is first stored as a `QueuedEmail` entity in the database. This avoids the problem of sending multiple times, because if the actions fails, the email is not added to the queue. A background job takes care of sending the emails in the queue, which is implemented in the standard library of WebDSL. It has no other tasks and is the only thread that updates or removes `QueuedEmail` entities from the queue, which avoids transaction conflicts. If there is an issue while sending the email, this function will retry after some time has elapsed.

### 5.3.2 Evolution of Files and Images

Storage of binary data files such as images and PDF files is another common feature in web applications. The `File` and `Image` property types handle storage of binary data in WebDSL. This feature uses the database support for storing binary data files. The `output` template for `Image` will display the image, whereas the `output` template for `File` type will display a download link. The `input` templates for both types use the HTML file input tag to display an upload file widget.

```

1  template addPhoto {
2    var photo := Photo{}
3    action add {
4      photo.save();
5      photo.owner := principal;
6      return showPhoto( photo );
7    }
8    form {
9      label( "Photo" ){ input( photo.data ) }
10     submit add { "Save" }
11   }
12 }
13 page showPhoto( p: Photo ){
14   output( p.data )
15 }
16 entity Photo {
17   data : Image
18   owner : User
19 }
20 entity User {
21   photos : {Photo} ( inverse = owner )
22 }

```

Figure 5.11 Image upload and output.

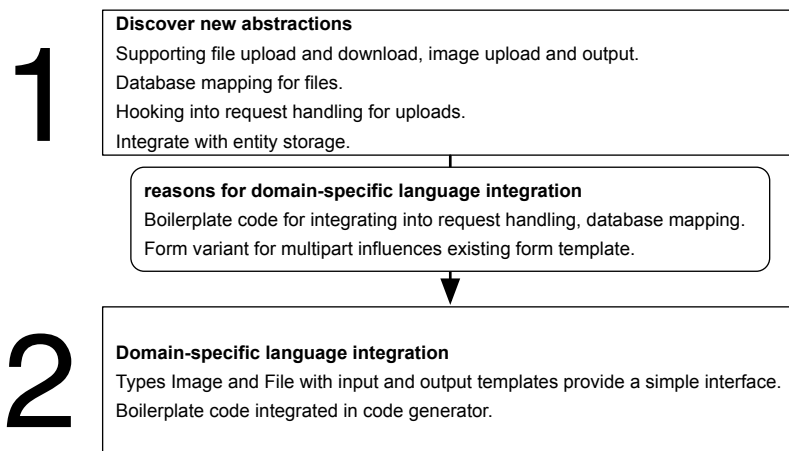


Figure 5.12 Generic evolution pattern applied to files and images.

The example in Figure 5.11 shows a `Photo` entity used for storing an image. The `addPhoto` template displays the upload image interface and stores uploaded images as a new instance of `Photo` belonging to the logged in user. The `showPhoto` page shows an uploaded photo using the `output` template for type `Image`.

The implementation of file and image handling builds upon the features available in the platform. The evolution of file handling followed a straightforward path to integration in the DSL, and is shown in Figure 5.12. In the



generated Java code, file and image content are handled with binary streams and persisted as blob columns. Additional functionality, such as transforming images can be implemented using standard image manipulation functions, either by a custom wrapping function or direct invocation of Java library classes. Handling upload on the server uses standard servlet code for this purpose. The database is required to have support for binary files, and serialization is already covered by the Hibernate framework. Storing files can be handled in several other ways, storage can be done in a separate database, or even in the file system directly. These variants could still be implemented for optimization or scalability by creating a Java library for it. The built-in file handling provides easy management of application file data: the data is stored together with all the other data in the database, and file data can be handled just like other entity properties. Storing files typically requires some configuration of the database engine, e.g. changing the maximum file size limit in MySQL. It can also require changing the maximum file upload size in the web proxy or application server.

### 5.3.3 Evolution of Internal Site Search

Internal site search enables the content of a web application to be indexed, searched, and displayed. This is different from external web search engines like Google and Bing, which index only the externally visible rendered pages. Internal site search can provide a much richer search mechanism by utilizing the data model directly. This enables features like faceted search. Faceted search is a dynamic way of representing search filter options when browsing a collection of items. For example, in a webshop the search query results can be refined by selecting a specific category of items, a price range, and a particular brand name.

The action of searching consist of four phases: query formulation, action, review of results, and refinement [Shneiderman, Byrd, and Croft 1997]. A search engine can provide spell corrections and autocomplete suggestions. A search action can be triggered after a button is pressed or automatically when the search query is changed. Completion is based on terms that occur in the indexed data and thus guarantee to produce a result. Shown search results need to represent the found entities in a concise manner, highlighting the elements relevant to the search. For example, in a long text fragment only the fragment around the searched word is shown.

The example in Figure 5.13 shows a search mapping for a `Publication` entity. The + indicates fields that are searched by default. The title field without any analyzers is used for autocomplete suggestions. The search function uses the searcher language to build up a search query, in this case the default settings are used for the given query. Limiting results and pagination is achieved by invoking functions on the searcher type that results from a searcher language invocation.

The specification of internal site search supports definitions for configuration, search mapping and constraints, retrieval of data, and user interface. The evolution of this feature (shown in Figure 5.14) is similar to persistence,

there is custom syntax that ties into configuration and operations in Hibernate, Hibernate Search, and Lucene. Configuration is needed to manage the external index, which is stored separately from the application database. Search constraints determine what entities and entity properties are searched and how the values are analyzed. Analysis consists of character filters, tokenization, and token filters. Common analysis operations are lowercasing and stemming. Retrieval of data is governed by a `Searcher` type that is available for each entity with a search constraint specification. The user interface can be expressed with WebDSL templates. A more detailed description of the search features in WebDSL can be found in Van Chastelet [2013].

## 5.4 Discussion

*Compiler extension implementation effort* Compiler extensions take more effort to create than libraries. WebDSL is implemented using SDF for syntax definition, Stratego for compilation, and Spoofox for the IDE. It is not a trivial step to go from a library implemented in WebDSL to a language feature, because it requires knowledge of these tools and the WebDSL implementation itself. These tools are still being improved to provide better abstractions for the creation of programming languages. Another way to address the problem is to provide a new sublanguage in WebDSL that handles common implementation patterns for language features so these can be defined entirely in libraries.

In SugarJ [Erdweg 2013] language extensions are implemented as library components. The syntax and semantics are specified with embedded SDF and Stratego. The method has been applied to create Java extensions as libraries that introduce additional language syntax and semantics into applications.

*Common mechanisms for external tools* Extensions that invoke external tools often require similar features, such as asynchronous scheduling, throttling, retry behavior, managing temporary files, and running tools in a sandbox. Typical examples of such extensions are email and PDF generation. Sending an email might temporarily fail, and needs to be retried. After a few failures there might be some fallback mechanism required that informs a user about the failure. The PDF generation tool is an external process that generates a file, this file needs to be imported into the database and can then be removed. All these features end up having to be reimplemented in many cases. These could benefit from a better abstraction for invoking external tools, and a library of utility functions.

*Library repository* In order to make libraries visible to other developers, a repository has to be maintained. There are many sources of inspiration from general-purpose languages. Each language has some form of library repository, such as CPAN for Perl, Cabal for Haskell, and the Maven repository for Java. IDEs can provide seamless integration with library repositories, such as DrRacket for Racket [2023] that automatically downloads the library when importing it in a file.

```

1 entity Publication {
2   title      : String ( id, name )
3   authors   : {Author}
4   description : WikiText
5   citations  : Int
6   search mapping {
7     title using none as suggest (autocomplete)
8     // for autocomplete suggestions use original title name
9     +title^1000.0 using titleAnalyzer (spellcheck)
10    // ^ boosts ranking for this property
11    +authors
12    +description using snowball (spellcheck)
13    // spellcheck enables analysis for 'did you mean' suggestions
14    citations
15  }
16 }
17 default analyzer titleAnalyzer {
18   tokenizer = StandardTokenizer
19   token filter = StandardFilter
20   token filter = LowerCaseFilter
21 }
22 function search( query: String ): [Publication] {
23   return (search Publication matching query).results();
24 }

```

Figure 5.13 Search analyzer specification.

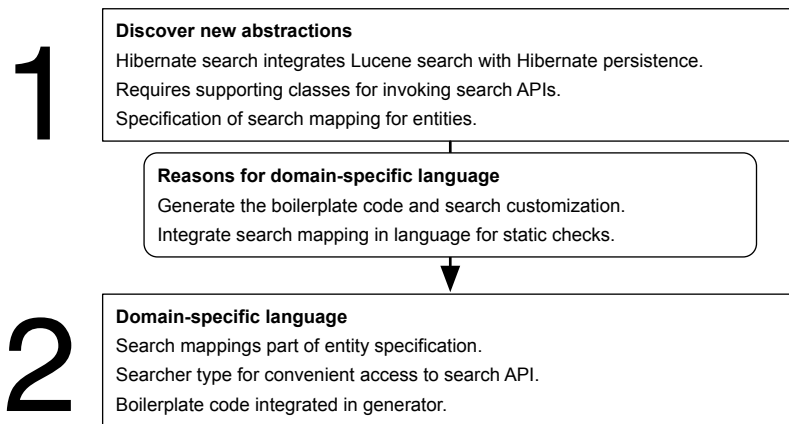


Figure 5.14 Generic evolution pattern applied to search.

*Client-side code abstraction* To avoid tampering, all WebDSL function code runs on the server. However, for integrating a JavaScript input widget, there is often some glue JavaScript code required. A typical approach is to put a hidden input field next to a JavaScript widget, and implement a hook that updates the state of the hidden input with the state of the widget. When submitting such a form, the value of the hidden input represents the selection in the widget. Besides input widgets, some applications require more client-side behavior such as animated page elements or transitions. The current way to

handle client-side code is to include JavaScript in a WebDSL template or return JavaScript to be executed as a response to an action. There is no checking or integration for this JavaScript embedding. There is room for improvement by incorporating client-side functionality as a sublanguage that can be checked for consistency with respect to the server-side components.

## 5.5 Conclusion

This chapter reflected on the evolution of the WebDSL language. The generic evolution pattern used in the development of WebDSL consists of first discovering new abstractions, then implementing domain-specific language abstraction, and finally reimplementing features using new core abstractions tailored to the language. The pattern repeats when additional problems or improvements are discovered. The WebDSL language design went through several iterations. In particular, the user interface language improved significantly from the initial implementation with rigid input components defined in the code generator. The current implementation provides customizable library-defined components for input and output templates using a small set of language primitives. Several examples of the pattern applied to evolution of language features have been discussed, including persistence, static code templates, email generation, files, and internal site search. For every feature in the language, a trade-off has been made to determine whether it is worth designing and implementing language support, or to keep it as a library implementation using custom Java or JavaScript code.

This chapter concludes the discussion of WebDSL language features. In Chapter 6 we analyze the WebDSL compiler, IDE and runtime implementation.



# The WebDSL Compiler, IDE, and Runtime

---

# 6

## 6.1 Introduction

Web application concerns are represented as first-class citizens in the WebDSL language. This differs from common web programming solutions where a general-purpose language only provides a low-level programming layer and a large framework is built on top. The implementation of a large language imposes its own set of software problems. For example, how to organize and maintain such a language, how to get sufficient performance, how to provide IDE support such as code completion and reference resolving, and how to make building and deploying seamless. The implementation of such a language benefits from reusable patterns for language analysis and code generation.

The design and implementation of the WebDSL compiler explores the interaction between analysis and transformations, in a high-level transformation language based on the paradigm of rewrite rules with programmable strategies (Stratego). The implementation uses transformations that gradually transform a high-level input model to an implementation. Language constructs are desugared down to a core language, and subsequently the core language is transformed to fragments in the target language (Java). The use of concrete object syntax guarantees syntactic correctness of code patterns, and enables the subsequent transformation of generated code. This approach is called code generation by model transformation [Hemel et al. 2010]. WebDSL is the largest programming language created with the Stratego program transformation language and the Spoofox language workbench, in which the compiler and IDE have been iteratively developed. In this chapter we will provide a high-level overview of the WebDSL compiler, IDE and runtime implementation.

Section 6.2 provides an overview of the WebDSL compiler pipeline, which consists of analysis, transformations, code generation, and Java compilation. Section 6.3 explains the analysis that happens in WebDSL, which is divided into the following steps: parsing and imports, declare global definitions, name resolution, and check constraints and report errors. The compiler continues with transformations, illustrated with several examples in Section 6.4, which provides reuse in the WebDSL language semantics by encoding language features into other language features, and reducing an application definition to a smaller core language. Section 6.5 shows which part of the WebDSL compiler is reused for the IDE, and explains the type of caching that has been implemented in the IDE for reducing error feedback time. After the front-end is done, the back-end generates the application, this is covered in Section 6.6. The resulting application code is based on a standard runtime library shared among WebDSL applications, and application-specific generated

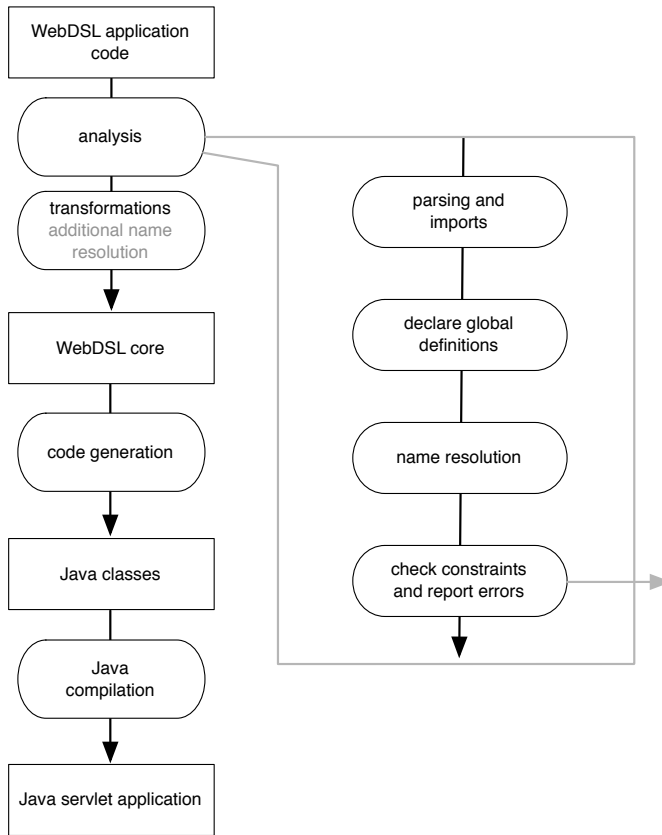


Figure 6.1 WebDSL compiler pipeline.

classes. We have designed multiple caching strategies in the compiler to reduce recompilation times, which are discussed in Section 6.7. Section 6.8 shows a typical deployment scenario for WebDSL applications. Section 6.9 discusses design considerations and opportunities for improvement related to the implementation of the WebDSL compiler, and Section 6.10 concludes this chapter.

## 6.2 Compiler Pipeline

Figure 6.1 shows an overview of the WebDSL compiler pipeline. The front-end of the language first parses the application files to create an abstract syntax tree (AST). Program analysis is performed on the AST. This means resolving definitions from use-site to declaration-site, calculating the type of each expression, and checking constraints to report errors. If there are errors, they are reported and the compiler is done. If there are no errors, the next step is program transformation. Some language features are expressed in terms

of other simpler language features. These are desugared during program transformation. This means there is a core WebDSL, a subset of the language, which reduces the number of language features to support in the back-end. Through desugaring, new WebDSL fragments are generated. These can require additional binding and type analysis to construct the fully analyzed AST. After the application is normalized to the core language, the back-end generates code. Bindings, types and all other information required for code generation of a top-level definition is merged into their AST node. These are the language elements like entities, pages, templates, and functions. With all required information merged into the AST, code generation is deterministically derived from the top-level AST nodes. This step can be cached easily by using the AST node as a cache key. The Java code that is generated is a complete Java Servlet application. Part of the dynamic semantics is defined in a precompiled runtime library written in Java. This runtime library is automatically included in the generated project. A supporting Ant build script is used to coordinate build activities such as copying template files, invoking the WebDSL compiler, invoking the Java compiler, and deploying generated applications. The generated Java servlet application runs in any Servlet container such as Tomcat or Jetty. The generated code uses the Hibernate ORM [2023] library for database persistence, which has support for various databases. We use MySQL on the production server and the H2 in-file or in-memory database engine for tests. Search is powered by the Lucene library and Hibernate Search.

The compiler is a standalone command-line compiler created using the Stratego/XT language and toolset for program transformation [Bravenboer et al. 2008]. The syntax definition is specified in SDF, and the compiler itself is written with Stratego transformation rules and strategies. The IDE, an Eclipse plugin, is created with the Spoofox language workbench [Kats and Visser 2010]. Spoofox provides an interface to language editor services for Stratego projects. The plugin includes the compiler and adds IDE features such as inline error markers, content completion, and reference resolving. Additionally, the WebDSL plugin integrates the build and deployment actions into a convenient project build action.

## 6.3 Front-End Analysis

The WebDSL language performs static verification on application code. These checks are executed before the compiler continues further processing of transformations and code generation. If there is a fault in the application code, the application developer must be notified. The fault must be corrected before being able to continue the build of the application.

*parsing and imports* The compiler first parses the application code to create an AST. If the application is not valid WebDSL syntax, the AST cannot be constructed and no semantic analysis can be performed. WebDSL has a simple import system, where any file imported is included once in the application code, and there are no language elements to restrict visibility. This means that



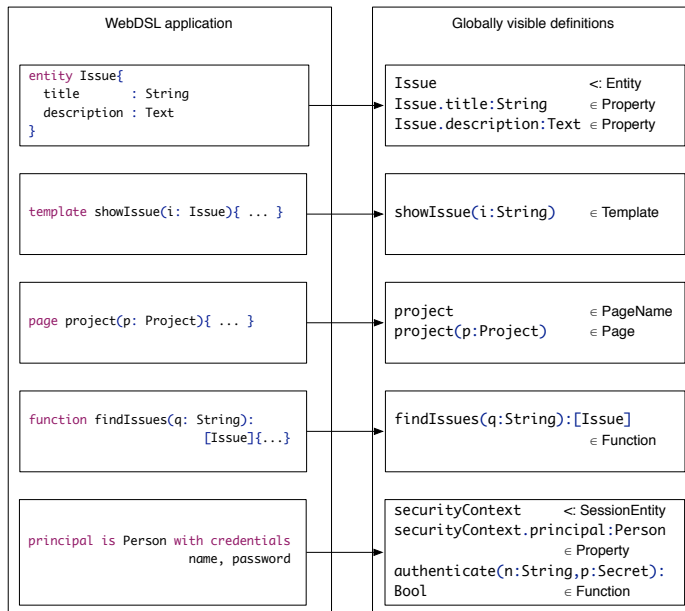


Figure 6.2 Declare globally visible definitions. The `Issue` entity declares an `Issue` type, and for each property a relation between entity and a property with type. The `showIssue` template declares a signature with name and parameter types. Function signatures are also a name with list of types. Page names are declared with just the name, because these need to be unique, but also with the signature, which allows checking `navigate` elements. Each language construct can declare several facts. If the semantics of a language construct implies that other WebDSL definitions are generated and can be called, these are declared in this phase as well. For example, a `principal is Person` declaration implicitly generates the `securityContext { principal: Person }` session entity, so `securityContext` is declared when analyzing `principal is Person`.

imports are transitive, any file that can be reached through imports starting at the main application file is included into the application AST.

*declare global definitions* When a valid AST has been loaded for the whole application, the first analysis traversal can be performed. This traversal records globally visible definitions. These are the entity names, its properties with types, page and template signatures, function signatures, and any other information required for resolving bindings and types. An example analysis phase for declaring globally visible definitions is explained in Figure 6.2.

*name resolution* After declaring the globally visible definitions, it becomes possible to resolve global and local bindings. Variables and calls are connected to their referred definitions. The type of each expression can be determined. This phase is illustrated in Figure 6.3.

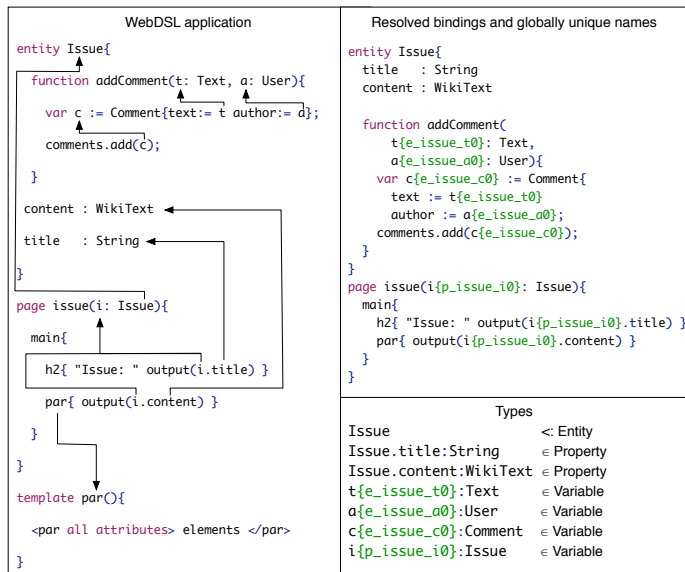


Figure 6.3 Resolve bindings and types. The entity function `addComment` has local references to the `t` and `a` arguments, and the variable `c`. The `issue` page refers to the globally defined `Issue` entity type, and its properties `title` and `content`. Other global references are to templates `main`, `h2`, `par`, `output` (`String`), and `output` (`WikiText`). Local references are to the page argument `i`.

Template and function calls have support for overloading, allowing definitions with the same name but different argument types. Overloading is resolved in this step by determining for each call a compatible definition with the closest matching types. If a closest match cannot be determined, the call is not resolved and an error will be reported in the constraints phase. This can happen if a template call with arguments `A` and `B` could refer to a template definition where argument `A` is more specific in type and another template definition where argument `B` is more specific.

To handle storing of resolve and type information, each local variable is given a globally unique name. This way an index can be created that holds type information for the whole application. Besides type information, any additional information related to bindings can be added in this phase. For example, a closure of the local variables can be added to elements that will be lifted out of their context. This is illustrated in Figure 6.4. Globally unique names are derived deterministically from the context to allow caching in later phases.

*check constraints and report errors* At this point in the compiler pipeline all information required for determining application faults is available. Global definitions have been stored and local bindings are resolved. The ‘check constraints and report errors’ phase takes care of checking static analysis

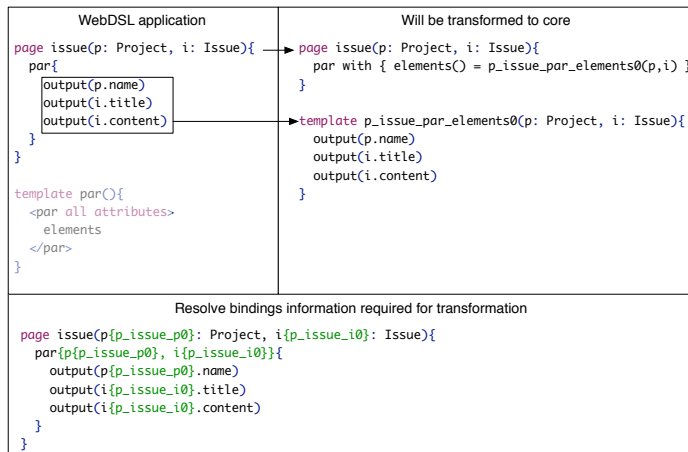


Figure 6.4 Insert binding information in AST where required for transformation. The `elements` template passed to `par` will be lifted to a separate template with a generated name `p_issue_par_elements0` in the transformation phase. Performing that transformation requires the closure of local variables that are in scope and used. In the resolve bindings phase shown at the bottom this information is available and can be inserted at the `par` template call.

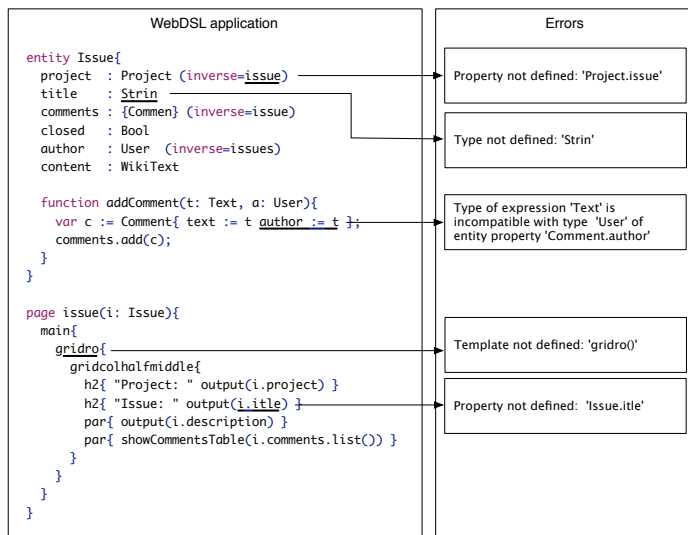


Figure 6.5 Check constraints and report errors. The properties `Project.issue` and `Issue.title` do not exist. The type `Strin` does not exist. The template `gridro()` does not exist. `Issue.author` is of type `User` and cannot be assigned a value `t` of type `Text`.

Application Code	<code>var u := User{}</code>	
Resolved bindings and types	<code>var u{u_p_user0} := User{}</code>	<code>u{u_p_user0}:User ∈ Variable</code>
Transformation	<code>var u{u_p_user0}: User := User{}</code>	

Figure 6.6 Analysis and transformation of variable declaration with inferred type. The application code contains the declaration of a variable `u`. In the resolve bindings and types phase, the variable is given a unique name, and requires a type to be specified for that name. At this point the type of the initializer expression can be determined, which is given to the variable `u{u_p_user0}`. In the transformation phase the actual AST node is changed to a variable declaration with explicit type, so the back-end code generator can retrieve the type from the node itself.

rules and collecting the resulting errors for feedback to the developer. In a command-line compiler these errors will be shown on the standard output with file location. In an IDE the errors will be marked at the faulty application element in the editor. Additionally, the file with the error is marked in project navigation views and the error is listed in the project errors overview.

Some checks can be based entirely on the globally declared information. For example, checking for duplicate entity or page names. Most checks state a requirement of a node in the AST. For example, does a referenced type exist, is a called template defined, are template argument types valid, does a variable reference point to an existing variable or argument. A check of a variable reference in this phase is trivial, because the resolved binding information indicates whether it refers to an existing name. With the binding and type information of variables and calls, all expression should produce a type. If the type of an expression cannot be determined there is an error that needs to be reported to the developer. Figure 6.5 illustrates constraint checking and error reporting. It has several incorrect property name references, template call references, and type incompatibilities.

## 6.4 Front-End Transformations

The compiler continues with transformations after analysis has been performed and the application does not contain any faults. The goal of the transformations is to reuse WebDSL language semantics to encode other language features, and thereby reducing the implementation effort of the back-end. The application is transformed to a core of the WebDSL language. Entire sublanguages can be created based on existing language features. A good example is the access control language, for which the semantics is defined as transformations on WebDSL definitions. Transformations can also be very small, such as the transformation for adding an inferred type in a variable declaration or `for` loop.

Transformations that introduce WebDSL application fragments may require an additional ‘resolve bindings and types’ pass on the new fragment. By making sure that each transformation leaves the application AST in a valid and analyzed state, the transformations become independent. New transfor-

Application Code	<pre> var cs := getComments(); var u := principal; var t := [ c.text   c in cs where c.author == u ]; </pre>
Resolved bindings and types	<pre> var cs{cs0} := getComments{g0}(); var u{u0} := principal{p0}; var t{t0} := [ c{c0}.text   c{c0} in cs{cs0}               where c{c0}.author == u{u0} ]; </pre> <p> cs{cs0}: [Comment] ∈ Variable  u{u0}: User ∈ Variable  c{c0}: Comment ∈ Immutable Reference  t{t0}: [Text] ∈ Variable  principal{p0}: User ∈ Variable  getComments{g0}: [Comment] ∈ Function </p>
Transformation	<pre> var cs{cs0} := getComments{g0}(); var u{u0} := principal{p0}; var t{t0} := listcompr0{l0}( cs{cs0}, u{u0} ); ... function listcompr0{l0}( cs{cs1}: [Comment], u{u1}: User): [Text] {   var list{l0}: [Text];   for( c{c1} in cs{cs1} where c{c1}.author == u{u1} ){     list{l0}.add( c{c1}.text );   }   return list{l0}; } </pre>

Figure 6.7 Analysis and transformation of list comprehension. The WebDSL expression language has a list comprehension construct as a convenience for performing `filter` and `map` operations. This example shows a variable `texts` being initialized with a list comprehension expression. When bindings from variable references to variable declarations are resolved, the type of the list comprehension can be determined. In this case a list of `Comment` entities is transformed to a list of `Text` values, which correspond to the `text` property of each `Comment`. List comprehensions are desugared entirely to WebDSL core. A function is generated that iterates over the collection, filtering the elements, and applying the expression to each element. The location of the list comprehension expression is replaced with a call to the generated function `listcompr0`. Newly generated code fragments are analyzed to resolve bindings and types. Transformation rules are applied exhaustively, definitions fragments can be transformed code fragments that need further transformations.

mations can be added without interfering with existing ones. Transformation rules are exhaustively applied. When no further changes can be made, the transformation phase is done. The following figures illustrate the implementation of several language features and their implementation as transformations. Variable declaration with inferred type in Figure 6.6, list comprehension in Figure 6.7, entity extension in Figure 6.8, and template inlining in Figure 6.9.

## 6.5 IDE support

To assist programmers, an Integrated Development Environment (IDE) provides more direct feedback. First of all, errors are marked in the file itself, which saves the programmer from matching an error report to its location. Syntax highlighting provides visual hints of the program structure. Reference resolving allows convenient navigation from use sites to declaration sites. Content completion suggests what the available options are for completing

Application Code	<pre>entity User {   name : String }  extend entity User {   isAdmin : Bool   function mayEdit( c: Comment ): Bool {     return c.author == this    isAdmin;   } }</pre>
Declare	<pre>User &lt;- Entity User.name:String ∈ Property User.isAdmin:Bool ∈ Property User.mayEdit(c:Comment):Bool ∈ Function</pre>
Transformation	<pre>entity User {   name : String   isAdmin : Bool   function mayEdit( c: Comment ): Bool {     return c.author == this    isAdmin;   } }</pre>

Figure 6.8 Analysis and transformation of extend entity. Entities definitions can be spread over multiple files. The `extend entity` declaration adds properties and functions to an already defined entity. The added properties and functions become part of the entity definition as if they were all in one declaration. The `declare` phase registers the properties and functions of the entity. The entity extension elements are handled the same as the regular entity elements. It does not matter whether a property is declared in the entity definition or one of its extensions. It will be accessible from each fragment of the entity definition. The transformation phase actually merges the entity with extensions into one entity definition. The back-end is now a simple one-to-one mapping to the target platform representation of entities. In the current compiler this is a Java class with Hibernate persistence annotations.

an expression or other definition. Outline views provide an overview of the definitions in a file. Besides editor support, the IDE provides an integrated solution for building and deploying applications.

The WebDSL IDE has been created with the Spoofox language workbench [Kats and Visser 2010; Kats 2011]. Figure 6.10 shows a screenshot of the WebDSL editor. Spoofox provides IDE creation for projects based on the Stratego transformation toolset. The WebDSL editor invokes the analysis from the compiler to provide editor support. This is illustrated in Figure 6.11. The parsing phase is needed for the editor to provide syntax highlighting. Global definition information and name binding analysis is required for reference resolving and content completion. Constraints need to be checked and reported in the editor. The IDE analysis is done at this point, because it can already provide all the editor features. It keeps the analysis information in memory until there is a change and reanalysis.

The parser in the editor is not entirely the same as in a command-line compiler. A variant of the parser is required that provides error recovery, making sure that an invalid syntax still produces a valid AST for analysis. For example, when a user types an entity variable name followed by a dot, the editor should be able to come up with valid completions of property

Application Code	<pre> page root {   main {     gridrow {       showProjects     }   } } template gridRow {   div[ class = "row", all attributes ]{ elements } } template div {   &lt;div all attributes&gt; elements &lt;/div&gt; } </pre>
Transformation	<pre> page root {   main {     &lt;div class = "row"&gt;       showProjects     &lt;/div&gt;   } } </pre>

Figure 6.9 Inline templates optimization. Wrapper templates are quite common, because these can provide an abstraction for CSS styling classes. For example, the `gridRow` template wraps an HTML `div` element around the template elements passed to it. A CSS styling class is applied to it, in this case a `row` for a flexible table-like layout. Such simple templates can be easily inlined, which saves some `templatecall` overhead at run-time. The transformation for inlining determines whether a template is safe for inlining, in particular this means there are no uses of the unique template identifier inside it. Then, the template elements are inserted at the call site with the original `elements` applied at the `elements` call. An inlined template definition is removed from the application AST.

```

entity User {
  username : String
  password : Secret
  allowEdit : {User} (allowed=from User as u where
    u.username <= ~this.username) }
entity Note {
  author : User (inverse=notes)
}
template note( n: Note ){
  navigate editNote( n ){ "Edit note" }
  "-n.authr.username: There is no page with signature editNote(Note)"
  output( n.content
}
principal is User with credentials username, password
access control rules
rule page root { "true" }
rule page editNote( owner: User ){
  principal == owner || owner in principal.canEdit
}

```

Figure 6.10 Editor screenshot, example intentionally seeded with faults to show static checks in IDE

names. Even though the application is not a valid syntax as described in the language definition, because a variable followed by just a dot is not a valid expression. Spofax handles this automatically, through generating error recovery productions in SDF [De Jonge et al. 2012; De Jonge 2014].

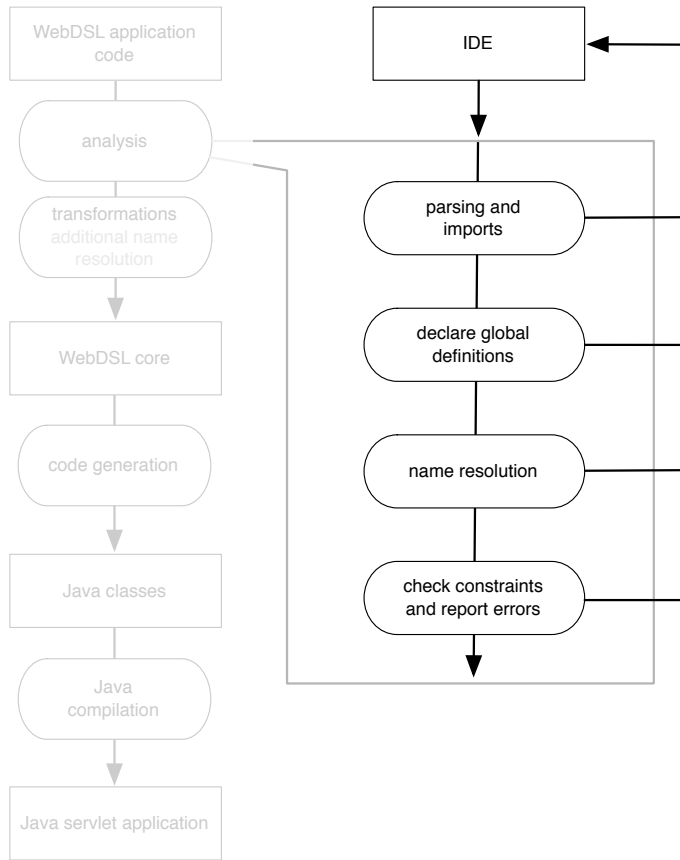


Figure 6.11 The IDE reuses the analysis steps, using the parsing for syntax highlighting, resolved names for reference resolution, registered definitions for content completion, and reporting errors with location info inside the editor with error markers. Parser requires error recovery to be able to parse incomplete programs and still provide editor support.

### 6.5.1 IDE Caching

The WebDSL compiler is implemented as a whole-program compiler. Such a compilation strategy is not optimal for editor usage. WebDSL has transitive imports, meaning that a file imported anywhere is visible everywhere. An editor window shows only a single file. While editing the file, analysis would by default analyze the whole program on each change. The analysis on each small change quickly becomes too slow for interactive usage. To address this problem a caching mechanism is implemented to optimize IDE usage.

The caching for IDE usage works as follows. In the background, all files are analyzed for globally visible declarations. This information is stored. When a



file is opened in an editor, the cached declare information of all the other files is loaded. Then the complete analysis is run for the changed file. This means that all the constraints can be evaluated for the file. Any file that depends on definition signatures in the changed file is scheduled for re-analysis in the background. If the user tries to resolve to a definition outside the current file, the referred file and position are read from the cache. The referred file is opened in a new editor and the cursor is navigated to the referred position. This also handles aspect-oriented features like `extend entity` correctly, because properties and function signatures are part of the cached declare information.

## 6.6 Back-End

WebDSL core gets translated to Java servlet code in the back-end of the compiler. The Java code is compiled and deployed to an application server. The back-end consists of two components:

- the runtime library that is the same for each application;
- and application-specific generated classes.

The code generation approach and runtime library are described in this section.

### 6.6.1 Code Generation

The WebDSL back-end compiles core application code to Java. There are several approaches to generating target application code. The most straightforward way is to generate a String that is written to a file. This is the approach most code generation template engines use. This String concatenation method does not provide any static checks to ensure that correct syntax is generated. Another method is to build up an AST using an API, where the API enforces the construction of correct syntax. The API provides a method to prettyprint the contents of an AST to a file. This method guarantees that the syntax of files is correct. However, it is also harder to use than String concatenation, because the source code fragments of the target language are no longer literally in the templates. The abstract syntax represented with the API is also less concise than the concrete syntax of the language. There is a method that combines the benefits of these two methods, namely concrete object syntax [Visser 2002; Bravenboer and Visser 2004]. Textual patterns in the syntax of the language are compiled to the corresponding abstract syntax pattern. This provides the same level of reusability as template engines, while also guaranteeing syntactic correctness of code patterns. Another benefit of the concrete syntax approach is that it allows subsequent transformations on the template fragments. Because a structured representation has been constructed, it can be augmented to include convenient features that are not in the target language. There are two additions used in WebDSL's target Java that help code generation, namely identifier composition and partial classes and methods.

*Identifier Composition* When generating a class field and the getter and setter method, several String concatenations are required to construct the names. The name of the field is changed to avoid collisions with reserved names such as Java keywords ‘class’ and ‘public’. The getter and setter names capitalize the field name and are prepended with `get` and `set`. While this is straightforward code, it is quite repetitive and distracting when trying to understand a template fragment. To avoid this problem we extended the target Java language with the `#` operator, which composes its two operand identifiers into a single identifier following Java’s naming conventions. Thus, `get#name` becomes `getName` and `_#name` becomes `_name`. The `#` operator is implemented by a transformation that replaces composite identifiers by regular Java identifiers. The Java extension and the transformation is reusable for other code generators that produce Java code.

*Partial Classes and Methods* In the initial WebDSL implementation, the generator was constructed in a centralized fashion. A single “God rule” was associated with each generated artifact, such as a template or an entity. Much like a “God class”, an anti-pattern in object-oriented programming, such a God rule dispatches a large number of smaller transformation rules to generate a monolithic target artifact like a Java class. These rules would grow to unwieldy size as new language features were added. This pattern is a code smell that hinders the extensibility and maintainability of the generator. The reason for the God rule is caused by the structure the target language: Java does not support composition of classes. Other platforms, such as C#, provide partial classes, which allow subdividing classes into smaller units. Our extension uses Java’s annotation syntax to identify partial classes and methods with the annotation `@Partial`. In the compiler, all the generated class fragments are collected and merged. Partial classes with the same name and package are merged into a single Java class. Similarly, the bodies of partial methods with the same signature in the same class are merged into a single method. The rules should not make assumptions about the order in which they are merged, as this is not defined. If an ordering is required, the method should be refactored into a regular method that calls partial methods in the required order. These partial methods are then the extension points for the entire method.

*Runtime* While all Java code could be placed in code generation templates, these would explode the amount of code being generated, killing performance of application compilation. A precompiled runtime library for common components is convenient to develop and unit test, just like a regular Java library. Having the constant code in a precompiled library saves compilation time in both the code generation and Java compilation steps.

*Code Generation Patterns* What remains are classes that implement the specific behavior defined in the WebDSL definitions. Figure 6.12 gives an example of generated classes for a WebDSL application. By generating specific code for each WebDSL feature, security issues related to flexible interpreter implementations are avoided. Furthermore, there is no need to dynamically store view states or have in-memory session data to manage page views and navigation.

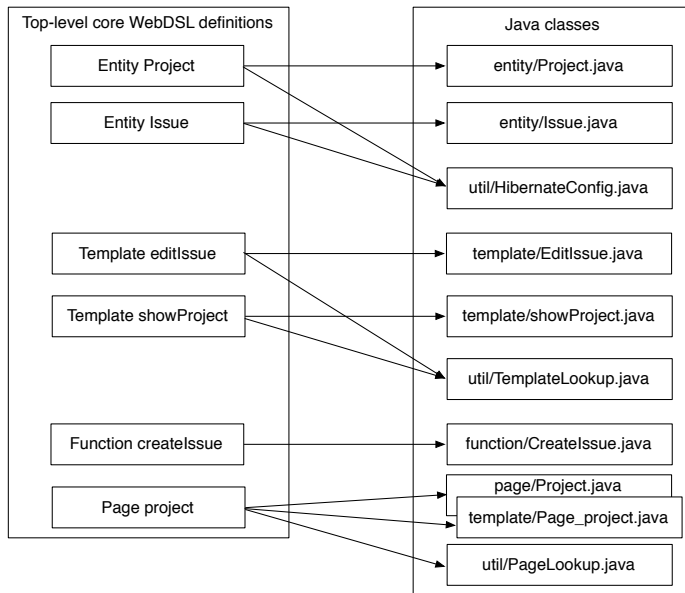


Figure 6.12 Code generation from core WebDSL to Java classes. Simple dependencies mean that the top-level definition can function as key for caching and avoid recompilation of unchanged classes. Entities are transformed to Hibernate ORM entity classes. Entity methods in WebDSL become methods in these entity classes. Global functions each get their own class with a static method containing the implementation. Pages and templates become classes with methods to handle each phase of request handling. Template variables are fields in the template class. There are some managing classes that incorporate information from multiple WebDSL definitions, such as the template and page lookup classes.

Any data that is dynamic in the application is stored in entities. The generated class for a template takes care of reconstructing the right state for rendering and handling form submits. Submit functions are part of the template context and do not require separate controller declarations and access control checks. The reachability of submits and inputs is determined by the template structure entirely. This relieves the application programmer from managing separate controller actions and data passing from rendered forms to submit action handlers. It also avoids the security problems that can arise from separate components for form and action that make it tempting to share data through the client for programming convenience.

### 6.6.2 WebDSL Request Lifecycle

The main runtime behavior of WebDSL applications is handling browser page requests for retrieving a page (GET) and requests for posting form data (POST). The request processing lifecycle is shown in Figure 6.13. The dispatch handler

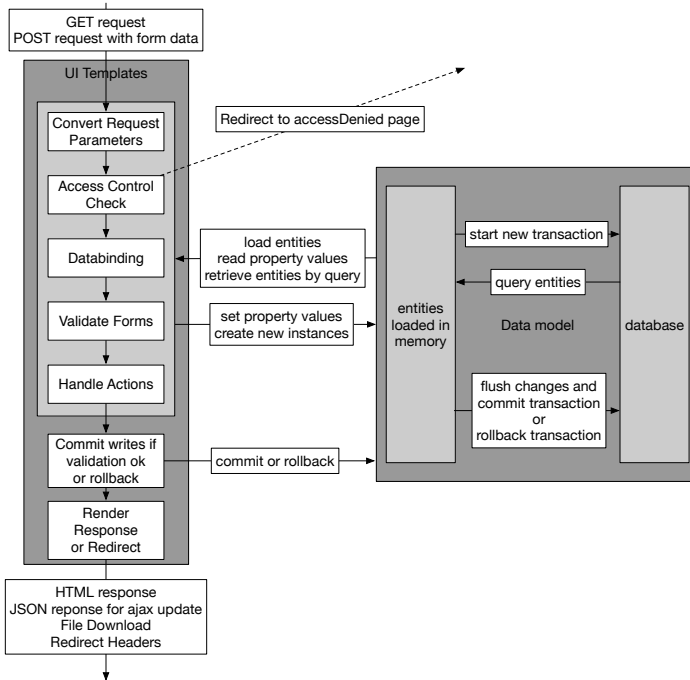


Figure 6.13 WebDSL request lifecycle

starts by analyzing request parameters to determine the page that was accessed and load the arguments to the page. Session data is also loaded to determine whether a user is logged in. This is followed by an access control check, which can deny access to the rest of processing and redirect. In case of a GET request that only reads data, the templates are rendered, and no database transaction commit is required. In case of a form submit the templates are evaluated in multiple phases: databinding processes inputs and transforms request data to updates in the loaded entities, validation then checks for failing validation rules before deciding to execute the requested action, and finally the requested action is executed. In case of validation failure, the transaction is rolled back, and the page is rendered with errors. When validation succeeds, the transaction is committed and the response is rendered or a redirect is triggered. An AJAX update request is a variation of this process, where only part of a page is rendered and returned. Throughout the request phases, the persistent data model is accessed to load entity data by id or through queries. Entity updates are tracked, and flushed back to the database when committing. Database transaction semantics decide how to resolve conflicts in updating persisted data.

### 6.6.3 Runtime System

The runtime library of WebDSL contains the code that is constant for each application. There are many components in the runtime library:

- dispatch logic;
- implementations of standard library functions;
- superclasses with common code for generated definitions such as templates and pages;
- servlet deployment logic with database initialization;
- support code for handling dynamic environments such as local template overrides;
- utility classes to interface with the persistent storage;
- and interface and wrappers for search classes.

The primary component is the dispatch logic. This governs the way a browser request is handled at the server. Figure 6.14 provides a flow diagram of the dispatch logic.

An example runtime class structure is shown at a high level in Figure 6.15. The Dispatch servlet looks up the page class that is being requested. The Page superclass contains further links to Template lookup, persistence session, and function code. Common code for Pages and Templates are placed in their superclasses. Besides these superclasses, the precompiled Java library contains utility functions for built-in components such as sending email, filtering HTML, filtering JavaScript, filtering URLs, markdown rendering, etc.

An example configuration of objects at runtime in a page request is shown in Figure 6.16. The dispatch servlet, page class, and persistence session are registered as Java Threadlocal objects, which can be retrieved anywhere in the thread without having to pass references to these objects around. In case of a form submit, the template instances with state are kept in memory while handling the phases. That way the variables that have data binding and validation will still be available in subsequent action or render with validation error phases. If there is only a render phase, the template instance is discarded after it has rendered its contents. The persistence session keeps references to all loaded entities. It can perform a flush to check for entity changes and send them to the database, this happens automatically when a query is performed or the transaction is about to be committed. An environment is used to keep track of local overrides for templates.

## 6.7 Compiler Caching Strategies

In this section we discuss compiler caching strategies that have been applied to decrease compilation time when developing WebDSL applications.



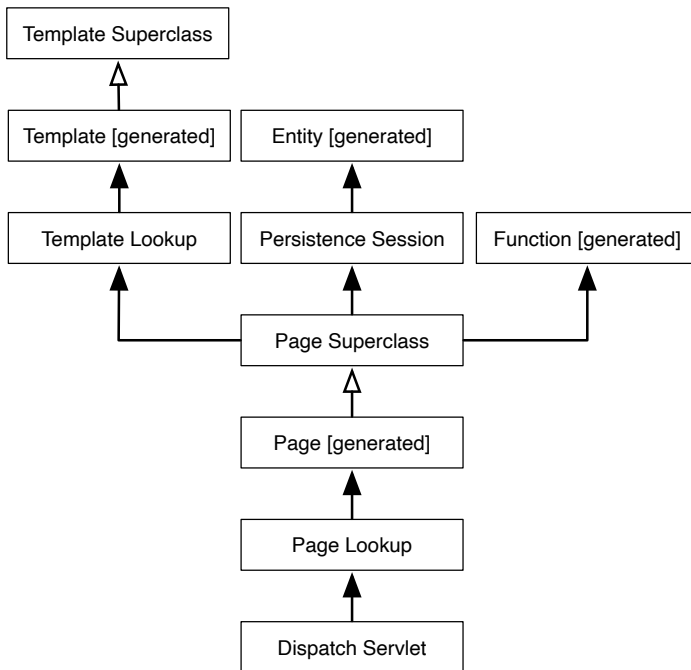


Figure 6.15 High-level class structure of WebDSL runtime code.

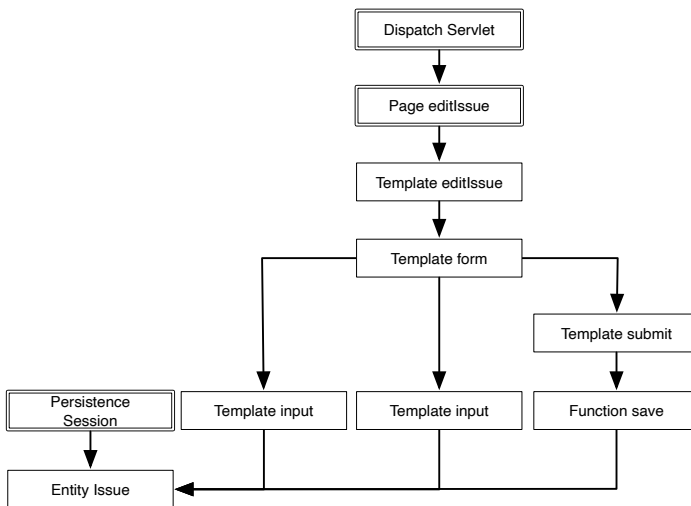


Figure 6.16 Example object instantiation at runtime. Double border objects are registered as ThreadLocal and accessible from anywhere in the thread, this avoids having to pass references around.

### 6.7.1 Code Generation Cache

The back-end Java code generator is cached for top-level WebDSL components, such as entities, functions, templates and pages. The AST fragment containing WebDSL core that comes from the front-end can be used as a key for the cache. This is due to the design decision that any information required for code generation is inserted into the AST, and the back-end rules are implemented as a pure function that takes only this AST fragment as input. Examples of this information are the types of entity properties and expressions, and the static components of unique names for templates that are invoked. Some generated Java classes depend on many components and are often regenerated, such as the lookup classes for pages and templates. Other classes are quite stable, such as template classes for simple CSS styling templates. The templates that are in the standard library are precompiled when building the compiler itself, and shipped with the compiler to optimize clean first builds.

### 6.7.2 Compile Unit Cache

The back-end caching strategy does not reduce the time spent in the front-end, for analyzing and desugaring all the definitions that have not changed. If a definition has not changed and will generate the same classes, e.g. a template together with lifted templates and other derived classes, then it does not have to be analyzed completely either. A compile unit cache can throw away AST top-level elements for unchanged elements after analysis and checking. Transformations that would eventually lead to the generated classes are not relevant if there are no changes that influence the unit. Even the declare globals phase can be cached for files that have not changed, as is done for IDE caching. The dependencies determine when a template or entity top-level AST node can be discarded. In case of templates these are often quite simple, they just depend on the signature of a called template, unless that template is inlined. For entities, however, there are many places in the application code that refer to entity types and their properties. The signature of an entity is in a sense, the whole entity definition with all its properties and annotations, and function signatures. With a simple comparison of new and old top-level AST nodes, it can be determined which templates have been changed. These are the most commonly occurring changes, small incremental template tweaks for layout and styling. The only features that have to be taken account are template inlining and access control rules. Access control matching rules need to be preserved in the AST to correctly generate the template. Changes to inlinable templates require recompilation of all templates that have it inlined in the generated code. Lookup classes depend entirely on the declared information, so these are correctly constructed.

The optimization of compile units caching saves time in the complete chain of analysis and code generation. Figure 6.17 shows timing results of performing compilation of several applications, and the speed up by the code generation cache, and the compile unit cache. The difficulty here is that the compiler has



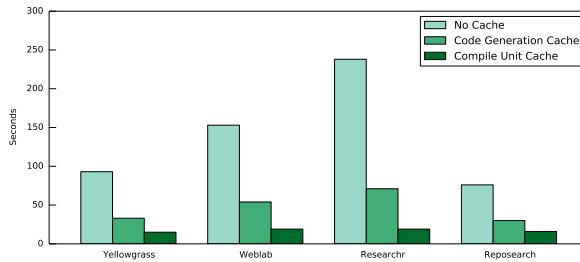


Figure 6.17 Compile times for clean build, with code generation cache, and with compile unit cache. Yellowgrass issue tracker, WebLab learning environment, Researchr conference website hosting, and Reposearch code search engine.

been designed as a whole-program compiler. The compile unit cache has been implemented to explore the possibility of caching by completely removing application fragments before going through subsequent compiler steps. The dependency analysis is currently manually implemented by reasoning about coarse-grained language feature dependencies, which means there is room for improvement. If dependency analysis can be automatically derived from the analysis rules additional unnecessary recompilation can be prevented.

## 6.8 Application Deployment

The WebDSL compiler creates a war file with a complete Java web application. A typical deployment scenario is illustrated in Figure 6.18. Here, the war file is copied into the webapps folder of a Tomcat application server. On initialization of the application, MySQL database table schemas are created if they do not exist, and updated if new columns are added. Nginx receives incoming requests first and decides based on the domain which application and application server are requested. Using a reverse proxy server to handle outside requests is more secure than directly exposing an application server to the web. The Apache Httpd and Nginx projects get a lot more scrutiny because these are used everywhere. Additionally, they can be set up to connect to multiple Tomcat instances, and take care of common web deployment configuration, like HTTPS encryption. A Tomcat application server can host one or more web applications, and a MySQL instance can host databases for multiple web applications.

## 6.9 Discussion

*Build speed and development* An important feature of scripting languages such as PHP and Ruby is that there is no compilation step that takes time. The effect is that as soon as the file is saved, the code can be tried by refreshing the page

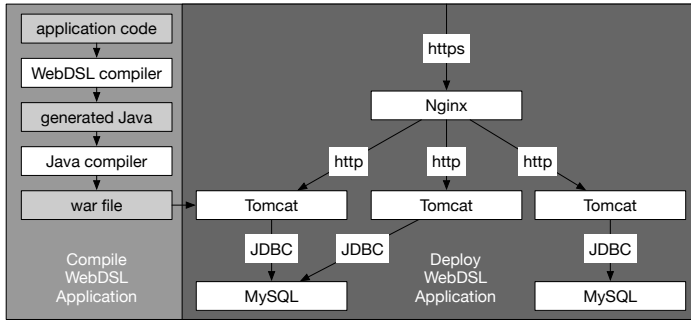


Figure 6.18 Deployment scenario, each box can be a separate server or the same depending on application requirements

in the browser. This immediate feedback contributes to a better development experience. Unfortunately, the current WebDSL compiler does not provide such immediate feedback. There is time lost with building and deploying the application. Building an application for the first time can take between seconds for a small application and minutes for very large applications. When rebuilding the application, the cached compile units and code generation save much time, however, these could be improved to become more fine-grained.

After Stratego compilation, there is a Java compilation step, and a deploy servlet application step. Incremental compilation of Java code is provided by the Eclipse Compiler for Java (ECJ). The standard deploy behavior in a servlet container like Tomcat is to reload all classes, which becomes slower as the application grows. Possible improvements for servlet deploy-time are based on custom classloaders that reload smarter. An example system that provides improved classloading is JRebel [Perforce 2023].

Besides improving the compilation and deployment of the regular Java back-end, another option is to implement a different WebDSL back-end specifically for development. An interpreter could be created to avoid compilation and deploy time. However, maintaining multiple back-ends is not easy, they are likely to diverge in subtle ways. This can lead to the unfortunate situation where the application works fine in development, but triggers a bug in the production environment compiled with the regular back-end.

While this chapter focuses on the implementation aspects of WebDSL itself, it also constitutes one of the most complex applications written in the Stratego language and Spoofox language workbench. As such, it also provided a case study for developing real-world programming languages using Spoofox. IDE analyses in a large language requires caching strategies, which we explored to increase performance of the WebDSL editor [Bruning 2013]. The performance requirements, together with the patterns for implementing analysis have been important for guiding further Spoofox development. New domain-specific languages for language definition were developed that provide increased support for such requirements. In particular, the NaBL name binding language provides abstractions for describing name resolution rules with caching provided

by a task engine [Wachsmuth et al. 2013]. Another direction is the specification of type systems using scope graphs in the Statix language [Van Antwerpen et al. 2018]. Zwaan, Van Antwerpen, and Visser [2022] present a technique to automatically derive incremental typecheckers from type system specifications written in Statix, using the WebDSL specification developed in De Krieger [2022] as a benchmark.

*Retargetability* Having multiple back-ends is possible, however, it is quite hard to get consistent behavior when targeting different languages with source-to-source transformations. Typically part of the underlying language semantics becomes part of the DSL semantics, such as primitive type value ranges. Additionally, libraries like Hibernate have complex semantics and not trivial to reimplement for another platform. Another issue is that the monitoring tools will be different for each platform, e.g. a Java process can be inspected with JVisualVM which provides information on threads, heap, and CPU activity. Improving retargetability would require migrating to a virtual machine or interpreter for the DSL core, without relying on the target language features through code generation. Although retargetability sounds nice in principle, for web application code running on the server it does not matter that much. The end-user of the application will not notice what kind of application code is running as long as it works correctly and is responsive enough.

## 6.10 Conclusion

WebDSL is the largest programming language created with the Stratego program transformation language and the Spoofox language workbench, in which the DSL compiler and IDE have been iteratively developed. This chapter gave a high-level overview of the WebDSL compiler, IDE and runtime implementation. Our approach of designing and implementing a new language like WebDSL is indeed feasible. We have used the WebDSL compiler for several real-world applications with thousands of users. Our experiences in using WebDSL to design, implement, and operate these applications are further described in Chapter 7, in which we evaluate practical applicability and reliability of WebDSL.

# WebDSL in Practice

---

# 7

## 7.1 Introduction

In previous chapters, we explained the WebDSL language (Chapter 2), with specific focus on user interface templates (Chapter 3) and access control (Chapter 4). We then discussed evolution of the language (Chapter 5), and compiler implementation (Chapter 6). In this chapter, we evaluate the practical applicability and reliability of WebDSL. We answer the following questions, by reflecting on our experiences:

1. How well did the focus on web programming abstractions, removing boilerplate code, and providing timely and accurate feedback on problems in application source code work in practice?
2. We set out to build a system that ensures reliability (robustness, performance, scalability, and security) of applications, did we indeed reach these goals?

We have been developing and using WebDSL for over 10 years to create information systems for academic workflows. The initial WebDSL research was focused on DSL compiler design [Visser 2007; Hemel et al. 2010], language design for access control and data validation concerns [Groenewegen and Visser 2008; Groenewegen and Visser 2013], and static consistency checking [Hemel et al. 2011]. The applications we created at that point were prototypes and case studies, with few external users. We continued working on WebDSL: adding language features, improving runtime efficiency, reducing compile times, and fixing bugs. Because WebDSL became more reliable, the applications became more ambitious. By now we have developed several applications with thousands of users:

- **EvaTool [2012]**: a course evaluation application that supports processes for analyzing student feedback by lecturers and other staff.
- **WebLab [2012]**: an online learning management system with a focus on programming education (students complete programming assignments in the browser), with support for lab work and digital exams, used by over 40 courses at TU Delft.
- **Conf Researchr [2014]**: a domain-specific content management system for creating and hosting integrated websites for conferences with multiple co-located events, used by all ACM SIGPLAN and SIGSOFT conferences.
- **MyStudyPlanning [2016]**: an application for composition of individual study plans by students and verification of those plans by the exam board, used by multiple faculties at TU Delft.

*Codebase Size* To get an idea of the codebase size, Figure 7.1 shows the WebDSL lines of code per language element for each of these applications. We learned many lessons while developing these applications, which we used to improve the reliability of the WebDSL language and its runtime. The abstraction layer that the WebDSL language provides between application specification and implementation, entails that the time invested in fine tuning reliability, robustness, performance, scalability, and security of the language and its runtime benefits all applications. Engineering a reliable runtime requires coordination between all the heterogeneous components of a web application, and takes a lot of experimentation to improve.

*Team Composition* For the majority of the applications operation time our AWE (Academic Workflow Engineering) team consisted of one team leader (Eelco Visser) and only two programmers (me and Elmer van Chastelet). WebLab and EvaTool started out being developed by additional programmers for the first one or two years, and after that became part of our portfolio of maintained applications. Besides working on requirements engineering, feature implementation, and bug fixing in the web application software, we maintained the server hardware and software configuration for deployment as well. We also worked on improving the language runtime and adding additional language features based on the typical usage patterns we observed. More recently, our team has expanded with one part-time programmer maintaining and supporting LabbackDocker (Daniel Pelsmaeker) and another programmer working on new application features and providing support (Max de Krieger).

The first sections of this chapter provide details of the most prominent WebDSL applications, namely EvaTool (Section 7.2), WebLab (Section 7.3), Conf Researchr (Section 7.4), and MyStudyPlanning (Section 7.5). In the next sections, we highlight interesting events and improvements made to WebDSL in the areas of robustness (Section 7.6), performance (Section 7.7), and security (Section 7.8). Section 7.9 is a collection of reflections on our experiences from applying WebDSL in practice to answer the questions stated at the beginning of this chapter. Section 7.10 discusses threats to validity, and Section 7.11 concludes this chapter.

## 7.2 EvaTool

The EvaTool platform is a web application designed to facilitate in gathering course evaluation data and communicating this data to various employees involved with the quality of education. The platform guides the feedback process applied at various faculties within TU Delft. Figure 7.2 shows a screenshot of the EvaTool application. Table 7.1 shows usage statistics of EvaTool.

*Evaluation Workflow* An evaluation is designed to follow the workflow of the faculty education feedback process:

1. create evaluation for a course or education;

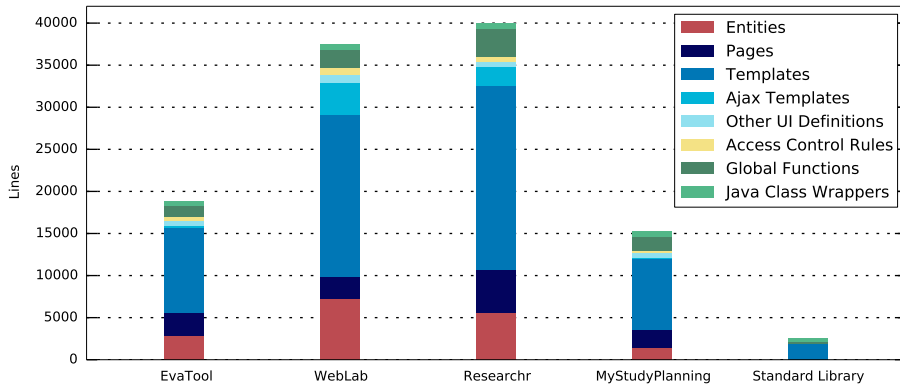


Figure 7.1 Lines of code per language element. These statistics were extracted by parsing the application files, and pretty printing the AST with a standardized layout. This process avoids differences from code layout styles. The code base versions are from October 2021. **Entities** include entity definitions containing properties and functions, search specifications, entity validation rules, and data initialization such as enums. The division between **Pages**, **Templates**, and **AJAX Templates** still depends on coding style, a page or AJAX template can call one template to handle all the content. **Other UI definitions** include HTML wrappers, attribute customization, statically expanded code templates (e.g. custom input and output definitions for different types), and routing configuration. **Global functions** includes definitions for background execution and web services. The statistics for applications include the standard library, which is also shown separately in the figure.

2. gather evaluation data;
3. ask the responsible course instructor to respond;
4. and review by Director of Studies.

*Features* EvaTool can send email notifications in order to ask for action, send reminders, inform about status, and invite people to log in. EvaTool provides various ways to project evaluation data. A user who is privileged to access one or more evaluations will have a filterable dashboard. From this they can directly view the evaluation or create a printable version. A custom visual representation can be created using a fact-sheet designer tool built into the application.

*Access Control Model* EvaTool stores, organizes and presents sensitive data. Various roles give users privilege to access (and enter) evaluation data. The user roles in EvaTool are designed to reflect the relevant employee roles from the educational organization with their corresponding privileges and at the corresponding level (course, education programme and faculty level). An overview of the roles and privileges of the users with access is available in

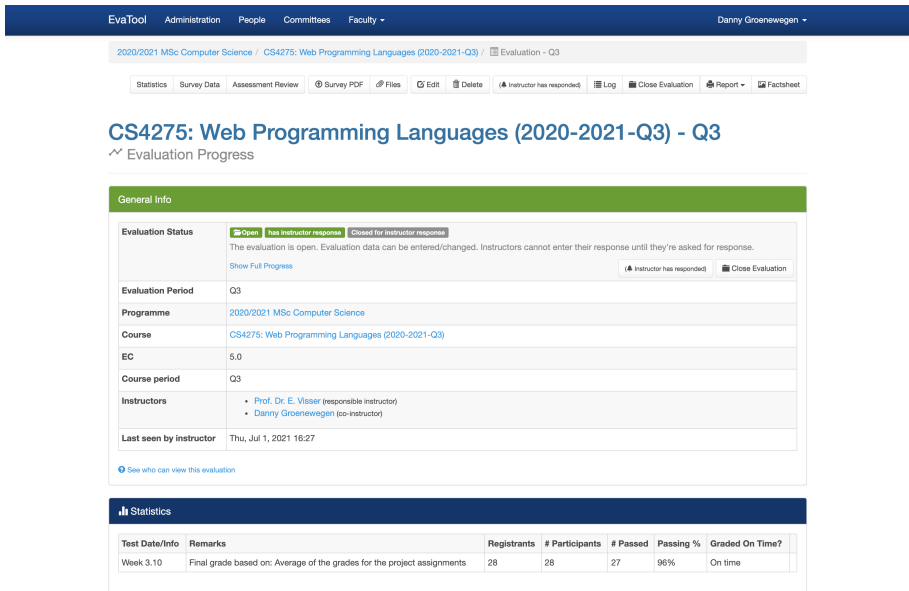


Figure 7.2 Screenshot of EvaTool application. This is the course evaluation status page which indicates the current status and next step for completing the evaluation.

Table 7.1 EvaTool Application Usage Statistics in July 2023. An education cohort is one study year of an education program, e.g. Bachelor Computer Science 20/21.

Description	Number
In Operation Since Year	2012
Education Committee Members	280
Instructors	1,220
Education Cohorts	376
Course Evaluations	8,358

each evaluation. A designated super user of a faculty is able to assign the roles to other people within the faculty, such as members of Education Committees, Programme Coordinators and Course Instructors. An overview page lists for each user exactly the capabilities awarded based on their combined roles. On evaluation forms the super user can also directly see “who has access to this”, which helps them in self diagnosing access configuration problems. The application uses the university Single-Sign On so that employees automatically have an account with their details.





Table 7.2 WebLab Application Usage Statistics in July 2023. A course edition is one run of a course in a study year, e.g. Concepts of Programming Languages 21/22. Programming, essay (open) questions, and multiple choice questions are the most used question types in WebLab. An assignment submission is an attempted solution by a student, and in the case of programming assignments typically involving multiple program runs to evaluate the entered solution.

Description	Number
In Operation Since Year	2012
Staff Accounts	1,203
Student Accounts	11,553
External Accounts	1,504
Course Editions	380
Programming Questions	15,143
Essay Questions	14,502
Multiple Choice Questions	21,296
Assignment Submissions	5,677,793

nized in a folder structure. Grading of assignments can be configured with simple schemes or custom formulas. Assignment data can be imported and exported to provide a way to manage them in a version control system such as Git. There is an exam dashboard which contains all the relevant action for running an exam. This includes actions for creating exam tickets, configuring deadlines and personal deadline extensions, and inspecting student progress. A grading interface streamlines the manual grading process by providing a convenient way to traverse solutions.

*LabBack* Running student program jobs in a reliable and safe way turns out to be pretty difficult. The guest code can contain malicious fragments, whether intentionally or not. Potential issues are, for example, allocating too much memory, infinite looping, and thread bombing. The original LabBack [Vergu 2012] is a Java process that compiles Java or Scala files in memory, loads them in the classloader, and runs them with restrictions from the Java security manager. One particular problem with this approach is that although heap usage can be constrained easily in the JVM, CPU usage is hard to constrain. This requires additional setup on the host machine, using an additional tool like `cpulimit` or `cgroups`, in order to avoid one process from occupying too many cores. Another limitation is that the programming assignments have to run in the JVM. We used a creative solution for C using Emscripten [2023] and a JavaScript interpreter in Java, however, this was not ideal because error messages were completely different from a real environment. Also Python could be run on the JVM with Jython, but would miss all the essential libraries, in particular NumPy and Matplotlib.

*LabBackDocker* As Docker became more popular, and provided a natural fit for implementing job runners, we later developed LabBackDocker [Crielaard,

Bruin, and Aerts 2017]. This is a Java application that schedules jobs on a Docker server. LabBackDocker enables courses to run assignments with any programming environment that can be configured as a Docker image. It allows setups with custom sets of libraries and evaluation commands. There are now courses that use Python with native libraries, Java configurations with custom libraries and test coverage analysis tools, and many other languages like C, Haskell, Agda, mCRL2, and Stratego. One downside compared to the legacy back-end is that a Java process is slow when restarted every time, because of dynamic class loading and JIT compiler optimization warmup. The docker images are newly instantiated on every run in order to avoid any interference from old student code. In particular, the Scala compiler benefits a lot from staying loaded and optimized in the JVM in the legacy back-end.

*Exams* Exams functioned as excellent stress tests for WebDSL, WebLab and LabBack(Docker), by now we have run hundreds of live exams. Part of the exam setup was creating an environment with the IT support team to boot up a system with a browser, that can only access WebLab. This required many test sessions, because there would easily be some shared drive or other mechanism through which students could potentially communicate in the exam. WebLab cannot use any CDNs for typical libraries such as JQuery or Bootstrap, because that would require opening a route to Google or other servers in the firewall, and can unintentionally give access to a search engine that has a huge cache of internet pages. Other problems we have encountered were network disruptions, Nginx configured with too strict limits, and an unavailable single sign-on server. Also computers simply having random hardware problems, such as a broken mouse or disconnected cables from students plugging the connectors into their laptops during regular lab hours. WebLab itself initially had some features that turned out to be problematic for the exam situation, such as a user profile page with editable text which would be visible to other students. The LabBack problem of not sufficiently limiting CPU became apparent with a Scala exam, where a relatively simple parallel list operation would put the system on full load. The JVM default settings for maximum heap space and garbage collector threads look at the machine specifications, and for a server would allocate gigantic heapspaces and over 50 garbage collector threads, which cause huge overhead for small programs. An important issue we resolved in WebDSL itself was a deadlock problem in the database handling, caused by an outdated configuration of the connection pool library. We have now had very few issues in the last 5 years of running WebLab exams, although WebLab itself is continuously being tweaked to become better for this purpose. For example, an announcement feature was introduced more recently to inform every student in the exam of important updates to exam questions. Student numbers are also growing, in 2022 we had exams with over 600 students concurrently working with the system and executing code on the back-ends without any problems.

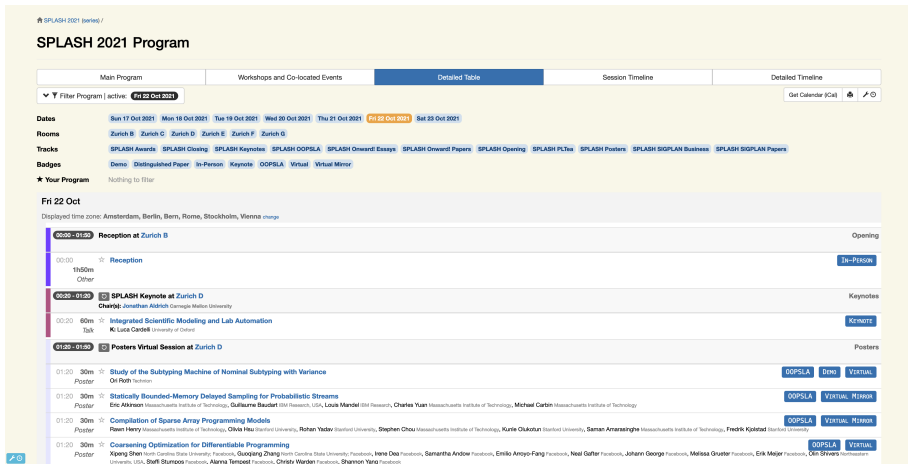


Figure 7.4 Screenshot of Researchr conference application. This is the detailed program view that shows session blocks and scheduled presentation events. Filters can be selected at the top to narrow the visible program, e.g. only showing a certain track.

## 7.4 Conf Researchr

Conf Researchr is a domain-specific content management system for creating and hosting integrated websites for conferences with multiple co-located events, used by all ACM SIGPLAN and SIGSOFT conferences. Figure 7.4 shows a screenshot of the Researchr application. Figure 7.5 shows the number of monthly users reported by Google Analytics. Figure 7.6 shows the progression of Conf Researchr in terms of number of conference website instances and conference days since its inception. Table 7.3 shows application usage statistics of Researchr. While conference websites for a single presentation track can be quite simple, in the case of multi-level conferences with co-located workshops and conferences, the website can grow in complexity quickly. For computer science conferences we observed that these websites were often created from scratch every year, because every time a different group of people is responsible for setting it up. The idea for the Researchr conference application is to provide a unified model for entering conference data and standardizing the process of creating conference websites.

*Features* The application contains built-in notions of publications, authors, presentation events, sessions, rooms, time slots, and call for papers. It automates many of the steps involved in setting up a conference website. Data from accepted publications can be imported from EasyChair or HotCRP. These papers become events to schedule in rooms for presentations. Room scheduling detects conflicts such as a speaker in multiple tracks at the same time. Main conference, subconference and track management can be delegated using the

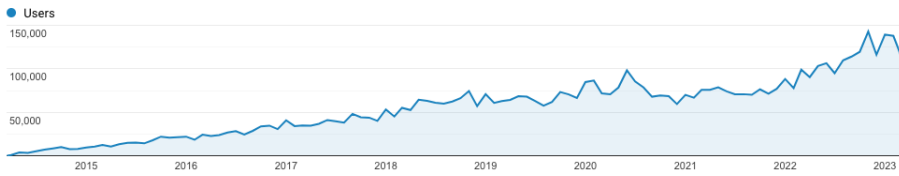


Figure 7.5 Monthly users of Conf Researchr reported by Google Analytics.

Table 7.3 Researchr Application Usage Statistics in July 2023. A Conference Edition is one year’s edition of a particular conference. Events are scheduled program items such as paper presentations or demo sessions.

Description	Number
In Operation Since Year	2014
User Accounts	48,541
Conference Editions	1,128
Conference Tracks	3,082
Events	28,558

access control configuration options. Additional features provide convenience for online conferences, such as schedules with localized times. Furthermore, it has generic Content Management System features for flexibility in creating information pages. The data entry is managed by designated conference managers, which involves a multi-level access control model for allowing content editors of subsections like tracks or workshops.

*Multitenancy* Researchr is a multi-tenant application, it hosts many conference edition websites from a single application instance and a single database. This makes the overhead for adding additional conference editions minimal. A single conference edition might not get many requests, especially for past editions. However, because the overhead of hosting another edition is so low, we can easily host many past years of a conference. Clients of the system have been using this effectively by importing data from older versions of conference websites as well.

Adding another conference edition does not require any additional deployment setup for Tomcat or MySQL. The only additional server script we use is for setting up HTTPS certificates in Nginx with Let’s Encrypt [Let’s Encrypt 2023]. We extended WebDSL with URL rewriting capabilities to provide better support for multi-tenancy with clean URLs. This allows the application to select a conference edition context based on the domain name in the URL.

The instances are not entirely separated, because users can have a general user account that can be applied to multiple conference editions. This is a convenient feature for allowing users to stay logged in when going to new conference edition websites. They only have to create one account for all the

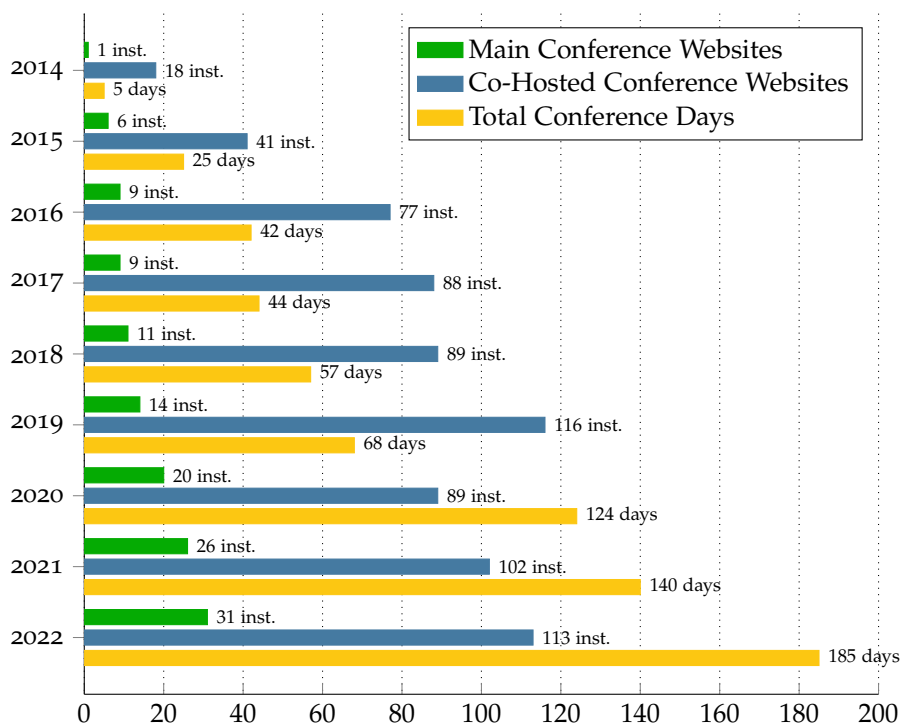


Figure 7.6 A breakdown showing the number of Conf Research website instances for main conferences and co-hosted conferences, and the total number of conference days for each year since the application's inception.

conference editions hosted on the platform.

*Support Tasks* Operation of the researchr application involves a few maintenance tasks. Although publication data can be automatically matched to authors based on email address, occasionally there is a missing connection due to an email address change.

When setting up a new edition of a conference, many settings can be copied from the previous edition as a starting point. This has been made into a workflow in the application, a getting started page guides the administrator of the conference through the options of copying components from previous year. Workshop proposals, to be colocated with a conference, have also been integrated as a workflow in the application.

*Supporting Online Conferences* During the Covid pandemic many conferences switched to a fully online edition. To support this style of conference, a number of features were added. The main change was the introduction of timezone settings in the schedule page, where previously only the local timezone of the conference was relevant. Some conferences also ran duplicate sessions of talks to suit different time zones. Besides timezone settings, other features that were requested was the option for adding participation info. This can contain for

---

Table 7.4 MyStudyPlanning Application Usage Statistics in July 2023.

Description	Number
In Operation Since Year	2016
Staff Accounts	676
Student Accounts	11,145
Master Cohorts	70
Track Configurations	551
Imported Courses	8,757
Student Submissions	19,617

example Zoom or Youtube links for the presentations. Features we did not add were direct chat or adhoc voice communication, although these were typically covered by tools like Slack, Discord, and Zoom.

## 7.5 MyStudyPlanning

MyStudyPlanning is a web application that supports the individual study program selection by students and approval of these submissions by staff members. Figure 7.7 shows a screenshot of the MyStudyPlanning application. Table 7.4 shows usage statistics of MyStudyPlanning.

The individual study program selection supported by this application consists of two phases: setting up the track options, and the approval workflow. The track options setup phase contains the following steps:

1. import new course data;
2. configure selection options and constraints for tracks;
3. review track configuration by track coordinator;
4. and finally open it to students.

The approval workflow is as follows:

1. student selects master track and courses;
2. coordinator judges submission;
3. board of examiners judges submission;
4. support staff enters approved program in central administration.

Staff members receive notification emails when there is an action expected, and students receive notifications of status updates. Coordinating staff members can select several overviews of students, to inspect the progress in each education track and cohort.

## Course Selection

Please choose your preferred research groups.

If you have any questions with regards to choosing courses you can reach out to the master coordinators of your track for more information:  
Data Science & Technology: [cs-dst@tudelft.nl](mailto:cs-dst@tudelft.nl)  
Software Technology: [cs-st@tudelft.nl](mailto:cs-st@tudelft.nl)  
Artificial Intelligence Technology: [cs-ait@tudelft.nl](mailto:cs-ait@tudelft.nl)

### First Choice

#### Programming Languages

CS - Software Technology 2020/2021  
<https://ijp.eew.tudelft.nl>

**highly recommended** [add all courses](#) [remove all courses](#)

- CS4206 A Computer Construction A [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#)
- CS4130 Seminar Programming Languages [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#)
- CS4280 Language Based Software Security [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#)
- CS4209 B Computer Construction B [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#)

**recommended** [add all courses](#) [remove all courses](#)

- CS4275 Web Programming Languages [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#)
- CS4326 Software Verification [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#)
- IN4333 Language Engineering Project [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#)
- IN4287 System Installation [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#)

**broaden your horizon** [add all courses](#) [remove all courses](#)

### Second Choice

#### Software Engineering

CS - Software Technology 2020/2021  
<http://www.cs.eew.tudelft.nl>

**highly recommended** [add all courses](#) [remove all courses](#)

- CS4235 Release Engineering for Machine Learning Applications [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#) (add to year 1 (add to year 2))
- IN4315 Software Architecture [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#)
- CS4110 Artificial Intelligence for Software Testing and Reverse Engineering [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#) (add to year 1 (add to year 2))
- IN4334 Analytics and Machine Learning for Software Engineering [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#) (add to year 1 (add to year 2))

**recommended** [add all courses](#) [remove all courses](#)

- IN4265 Gradually Distributed Software Engineering [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#) (add to year 1 (add to year 2))
- CS4300 Seminar Software Refactoring [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#) (add to year 1 (add to year 2))
- CS4156 Dynamic and Static Program Analysis for Software Security [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#) (add to year 1 (add to year 2))
- CS4220 Machine Learning 1 [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#)

### Third Choice

#### Web Information Systems

CS - Software Technology 2020/2021  
<https://www.wis.eew.tudelft.nl>

**highly recommended** [add all courses](#) [remove all courses](#)

- IN4331 Web-scale Data Management [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#) (add to year 1 (add to year 2))
- IN4325 Information Retrieval [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#) (add to year 1 (add to year 2))
- IN4326 Seminar Web Information Systems [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#) (add to year 1 (add to year 2))
- IN4252 Web Science & Engineering [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#) (add to year 1 (add to year 2))

**recommended** [add all courses](#) [remove all courses](#)

- CS4145 Cloud Computing [add](#) [Q1](#) [Q2](#) [Q3](#) [Q4](#) [remove](#) (add to year 1 (add to year 2))

**broaden your horizon** [add all courses](#) [remove all courses](#)

requires 25 EC more for the requirement of 120 EC [Review for Submit](#) [view course planning](#)

year 1	<a href="#">Q1</a>	<a href="#">Q2</a>	<a href="#">Q3</a>	<a href="#">Q4</a>
year 2	Q1 0.0	<a href="#">Q2</a>	<a href="#">Q3</a>	<a href="#">Q4</a>

Figure 7.7 Screenshot of MyStudyPlanning application. This shows a section of the course selection interface, where courses are presented for each selected research group. From this section students can easily choose elective courses based on their preferred topics.

*Challenge* Before we started this project, the process was based on a paper form that needed to be filled in by the student. Then they would have to get a signature from a master coordinator, and finally hand it in at the service desk so that it could be approved by the board of examiners. Since the documents were often handwritten instead of printed, there was tedious work and opportunity for errors in deciphering student handwriting and manually adding up all the study points.

The difficulty in automating this process and making it fully online was in the variations that each faculty used, the departments for handling student administration in each faculty are separated and do not follow the same workflows exactly. Similar projects have been attempted and usually fail. An external software consultancy company works closely with a few pilot users, however, they are likely to specialize too much to the workflow of these particular users. For adoption of the software it needs to be approved by a larger committee where it often turns out the software is not flexible enough to handle the workflows of each individual faculty.

*Interesting Implementation Aspects* Single sign-on is an important part of this application, this was already implemented for other applications EvaTool and WebLab and could be reused.

The course selection page uses the inline AJAX refresh (see Section 3.7.2) to make the entire page refresh immediately after an action. This improves the user experience by avoiding page reloads and unintentional change of scroll state (part of the page being viewed does not change after clicking on an action). There are several constraints connecting components on the course

selection page. Selecting a course in one category prevents selecting it in other categories. Other categories listing the same course show a message that the course is already selected. Updating these connected components in the user interface is handled automatically by refreshing the course selection page component.

Another example of effective use of inline page refresh is the track constraint configuration page. The categories, including mandatory and optional courses to be selected, are constructed on a single page. Course details are imported by looking up the entered course codes in the imported course data.

## 7.6 Robustness Engineering Experiences

In this section we list some of the improvements we made and experiences we had regarding robustness. We define robustness as: applications should not crash and should not show glitches in availability.

*Single Application per JVM* In an ideal scenario for deploying Java web applications, it is enough to have one application server instance hosting all the applications, and one database server instance hosting all the databases. What we experienced in practice is that there are many reasons why the JVM can crash:

- Hanging Tomcat JVM due to expensive page request
- Tomcat crashing automatically after 50 days [Lopes 2014]
- Crashed JVM due to bug in JNI code of a library
- Maximum open file handles reached for process

These problems can all be solved, some require changing the OS environment, or JVM parameters, or the bug has been fixed in a newer JVM/Tomcat/library. Even though they can be solved, having all applications in one application server means if one crashes the JVM, all applications are down. For robustness in our application deployment we switched to one application per Tomcat instance. With MySQL we experienced few robustness issues. However, for performance tuning it can be useful to have one application database per MySQL instance to have more control over settings.

*Fluent Redeploy* When redeploying an application there is a small delay between the war file being deployed and the first request being accepted. This delay can be reduced by using tomcat war file versioning e.g. by copying a war file with version number appended, e.g. ROOT##42.war. The new war file is deployed next to the old one, and requests are directed to the new application instance as soon as it is finished deploying. A WebDSL application starts with checking the database schema for updates. In the case of adding a new property to an entity with many saved instances, this can be slow. The schema update can also be done in advance to avoid the delay in deployment. Another issue was related to template identity. The identifiers are partially based on



a static id assigned to template calls at compile-time. If this id is not stable between recompilations, input and action ids can change. This means that if a user is looking at a loaded page, then a redeploy is performed on the application server, the forms on the loaded page are no longer valid. A page refresh is needed to get back to a working page. We improved this behavior by making the ids more deterministic, using the AST location as unique identifier, which resulted in fewer failed actions after a redeploy.

*Data Migration* In the most commonly occurring data migration scenario, adding a new property or entity, the database schema is updated when the application is deployed. This does not support every type of change, in particular changing the type of an existing property and potentially removing data is not allowed. However, a new property can be created with the new type. Data migration can then be expressed on an administrators page, by implementing an action button in WebDSL to perform the migration and initialize the new property values, optionally based on existing properties or other functions. To avoid a delay when deploying, database schema updates can also be done in advance on the live database. WebDSL shows the updates to the database schema in the log, which can be applied to the existing database. A comparison check is performed when deploying an application, only the missing schema definitions are added. For experimental or rarely used features, it is also an option to handle the connection with existing entities through dynamic composition. This can become relevant for entities with millions of rows. When the particular feature is required, a query retrieves the corresponding entity that holds the data for the new feature, without encoding it directly in the entity relationships. From an abstraction perspective this is not ideal, conceptually a regular reference property would be expected. The boilerplate code for retrieving the dynamic composition could be transformed into a small language feature. The Acoda tool [Vermolen 2012] involved an experiment into applying data migration on WebDSL applications. Besides the experimental status of the tool, a conceptual issue was that it introduced a separate DSL for describing migrations. From a WebDSL programmer perspective, it is much easier to be able to express data migrations in the WebDSL language itself. In practice, creating a migration action has been sufficient for migrating data in our applications.

*Transaction Retry* In the majority of requests there is no issue with concurrent edits of the same data. Since we rely on the transaction behavior of the database to handle conflicts, there are specific scenarios where a request fails because another transaction committed changes at the same time, e.g. by code in a background task. This situation was observable as a page sometimes not loading, or an action failing to complete. We added a retry mechanism to handling requests in the specific scenario of a concurrent change. By default, requests are tried upto 3 times before giving up. In most situations this is enough to let the update be processed. The request is handled as if it came after the commit that caused the conflict. Transaction semantics can not be hidden entirely from the WebDSL application developer, as we experienced in

a WebLab scenario. A new feature was added to calculate an average grade for all exam assignments, updated every time a change was made by any student. This led to all student transactions being in conflict, because they were trying to update the same row storing the average grade in the database. The problem of describing derived values concisely and deriving a robust evaluation was inspiration for the IceDust language [Harkes, Groenewegen, and Visser 2016; Harkes, Van Chastelet, and Visser 2018; Harkes 2019].

*Submit Failure Feedback* Another improvement was made in the handling of failed actions. When data is updated in the database, it might happen that a form is no longer available, meaning the unique ids of the inputs and actions will be ignored. If the user then submits the form, it is not recognized as a valid action. The initial implementation would cause the button not to get a response, which was confusing for users. We improved this behavior by explicitly notifying the user with a customizable message. For example, in WebLab, the submit button for an assignment becomes unavailable after the deadline passes. The student now gets a message explaining that the deadline has passed, when they press the submit button.

## 7.7 Performance Engineering Experiences

In this section we list some of the improvements we made and experiences we had regarding performance. We define performance as: applications should have no noticeable delay in response times, and this should hold also when the amount of data increases (scalability).

*In-Memory Page Cache* In many applications there are more users reading data than writing data. In that case, a page cache is very beneficial for performance, e.g. in CMS-style applications such as Researchr. By building a page cache into the runtime we can automatically handle cache invalidation. After checking access control, the rendered page is retrieved from cache if available. The cache is filled automatically and keeps the most recently used pages in cache. Cache space is allocated for anonymous users, and for logged in users, which helps improve the browsing speed for a user session on the website. All caches are invalidated when an entity change happens. If it is a session entity change, only the page cache for logged in users is invalidated.

*Query Prefetching* In the data model of the WebDSL runtime, we use Hibernate ORM [2023] to implement objects with persistence. In the runtime, we made the decision to have a default configuration that retrieves reference properties lazily. In many cases this is a good default. If the reference is not used, it does not have to be loaded from the database. The effect is that the queries that get generated are small and fast. However, the number of executed queries is high. When there is an iteration over a large collection, and for each iteration a query is executed, it is often faster to do one query with a join for the extra needed data (eager fetching). This is referred to as the N+1 problem in ORM terminology. We have experimented with deriving automatic prefetching [Gersen 2013], which showed us that the decision for eager fetching can be partly static, by

analyzing access patterns in the application code. However, there is also a dynamic component. The actual speed improvement depends on the table sizes and several database settings. To have more control, we also added prefetching syntax to the language to force eager fetching. This turned out to be very convenient in practical situations where a single page was getting too slow.

## 7.8 Security Engineering Experiences

In this section we list some of the improvements we made and experiences we had regarding security. We define security as: applications should prevent attacks from malicious sources, where vulnerabilities in the web technology stack are abused.

*Improving CSRF Protection* Easy to guess id attributes in form inputs are vulnerable to Cross-Site Request Forgery (CSRF) attacks. A malicious website can create a link or image that is a forged request to execute an action on the targeted application. If the victim is logged in to the targeted application, the browser will perform the request using the victim's credentials. Template id generation in WebDSL depends on the data and is hashed to make them hard to guess. We further improved this protection by including the principal user entity id in all template id attributes. This can be done transparently because the compiler controls id generation, and the entity used as principal is explicitly identified for access control. A common way to do CSRF protection in frameworks like Django [2023] is to rely on adding an additional CSRF token to all the forms. The token is a random secret value associated with a user session that needs to be submitted with the request parameters to perform the action. Although it makes protection convenient, it is still something that a developer can forget to include, or cause confusion if it is used incorrectly and blocks a submit unintentionally.

*Server-side and Client-side Components with Rendering* Markdown syntax is used in many WebDSL applications for creating texts with markup and links. The `WikiText` type in WebDSL automatically renders the content as Markdown in an output. Because HTML markup and hyperlinks can be created based on links in the text, the regular HTML escaping is too rigid for the output of Markdown. A separate whitelist filter is used for the output of Markdown, to allow a safe subset of HTML to remain in the resulting page source. In some cases, however, such as a CMS where a trusted user is editing the page source, full control over HTML might be required. For these cases it is possible to turn the filter off with `rawoutput`, which requires more responsibility from the application developer to keep the application secure. Besides components that perform server-side rendering, there are also client-side rendering components. One example is the MathJax [MathJax 2023] library that we use in WebLab to allow LaTeX when editing text. A problem that we initially did not catch was that this library can also generate hyperlinks based on different syntax. This allowed potential XSS to be performed, by including a hyperlink with

JavaScript, encoded as a link in MathJax. The MathJax library includes a safe mode that limits the type of urls that can be constructed, which must be enabled to avoid XSS.

*Force HTTPS* A feature that is best solved before requests go to the application server at all, is forcing request to go over HTTPS. This makes sure all sensitive form data and cookies gets sent encrypted. This is simple to configure in Apache httpd or Nginx and can be configured with HSTS headers so that browsers cache the decision to access the site over HTTPS.

*Single Sign-On* One of the larger security issue we experienced was a bug in the A-Select single sign-on Apache module provided by the university. The module was vulnerable to a directory traversal attack, which would circumvent the filter that blocked access. The lesson learned here is to be very careful with external authentication integration, the application code might have a perfect access control model, but if you cannot trust the signin procedure it is useless.

*Deployment Isolation* Since we were running the servers, we also managed a Jenkins instance for our research group. This became a problem when the jenkins user started executing suspicious commands on the server. It turned out that a vulnerability in Jenkins was abused to run arbitrary scripts. The lesson we learned was to not trust that other developers of other web applications get security right, and deploy applications (especially those with scripting components) in as much virtualization and isolation as possible.

*Heartbleed* Heartbleed [Synopsys 2020] was a serious vulnerability in the widely used OpenSSL library, and affected our servers as well. It was publicly disclosed in 2014. Due to a missing bounds check in the TLS heartbeat extension in the OpenSSL implementation, more data could read than should be allowed. This bug allows reading the memory of affected systems, which compromises secret keys to identify services and encrypt traffic, names and passwords of users, and communication content.

*Log4Shell vulnerability* The Log4Shell vulnerability [National Vulnerability Database 2021], publicly disclosed in 2021, affected many systems including WebDSL applications. This is the largest security vulnerability we have encountered so far. Log4Shell was a zero-day vulnerability in Log4j, a popular open-source Java logging framework. Log4j had a lesser known feature that allowed requests to arbitrary LDAP (Lightweight Directory Access Protocol) and JNDI (Java Naming and Directory Interface) servers to retrieve Java class files and execute them, which enabled an attacker to execute arbitrary code. The attack was simple to exploit, anywhere the attacker could cause a string to be logged, e.g. by passing it in a HTTP header like User-Agent, a JNDI lookup command could be inserted. Using a command such as `$jndi:ldap://example.com/file`, the application server JVM would query that URL, load the Java object, and execute it.

## 7.9 Reflections on Experiences

In this section, we evaluate practical applicability of WebDSL by reflecting on our experiences of the whole design and implementation process of these applications: requirements gathering, development, testing, deployment and server maintenance, application support, version migrations, bug fixes, security fixes, and feature additions.

*Abstraction* The persistence abstraction has been very stable, sticking close to its original design. One problem that plagued us for a while in WebLab operations was a subtle bug in the library for handling connections to the database. This caused a frustrating issue where in some cases no further connections could be created, usually during an exam with peak load. After this bug was found and fixed in WebDSL, we did not encounter such large problems anymore. Besides this problem we have had very few bugs related to saving and loading entity data.

The language features for function code have also been pretty stable, although we did add several small improvements, such as string interpolation and typecase. Extensions for calling into Java libraries are used in several applications, e.g. to invoke the LabBack component to run student code in WebLab.

The user interface abstraction received several iterations, and is the most interesting aspect of the WebDSL language. We have explored abstractions for AJAX initially in the nested page template abstraction (Section 3.7.1). While this enabled creating single page interfaces, for smaller dynamic behavior in forms it was not suitable. The inline refresh (Section 3.7.2) abstraction was created later to address this shortcoming. This feature is heavily used in MyStudyPlanning, where the course selection page handles all the selection events by updating the page contents directly with a placeholder refresh. While this feature enables dynamic forms, there are some limitations because it needs to work together with the declarative nature of regular WebDSL input forms. The benefit of both abstractions is that because they run on the server, the code is safe from any client-side JavaScript tampering. This aspect is where the most recent feature additions in WebDSL have been. It will be worth investigating client-side rendering abstractions to have complete flexibility, however, that would require extra care for handling data securely. The extensibility option to include JavaScript is used in several applications, e.g. to include a component for client-side sortable tables, and a component for improved dropdowns with search. Also in cases where custom client-side behavior was required, the solution was to write this in JavaScript and include it. For example, the client-side filtering in the Conf Researchr program is written as a small reusable JavaScript library.

The access control abstraction is used in all applications. Every application has multiple types of roles that are connected to specific data, such as course managers in WebLab, or web chairs in Conf Researchr.

Reflecting on our experiences, the abstractions in the WebDSL language are effective in creating real-world web information systems. There is room for

improvement when it comes to client-side code and rendering, however, in our web information system applications there were only a few cases where this was really needed.

*Boilerplate Code* Boilerplate code is generated for many low-level aspects of web programming, such as setting up entity persistence mapping to database tables, generating form input identifiers, and importing form data in action handlers. It is hard to measure the impact of code that you did not have to write. This only becomes apparent when trying to implement the same application in a different framework. A comparison of small applications implemented in other solutions is made in Chapter 8. It makes clear that there is a significant reduction in boilerplate code in persistence and user interface handling, avoiding bookkeeping code and removing the potential for errors in lookups (e.g. retrieving inputs in action handlers).

We can look at the amount of generated code to provide a rough indicator of generated boilerplate code. The WebDSL compiler generates Java code from WebDSL application source code. One application with 30,000 lines of WebDSL turns into 720,000 lines of Java code (factor 24), and another application of 18,000 turns into 480,000 lines (factor 26.7). This size is only a rough indication of the amount of boilerplate code that did not have to be written, it is likely that an equivalent application written in Java directly would be fewer lines of code. Compiler techniques used in WebDSL, such as desugaring definitions iteratively down to a core language, can cause quickly expanding size of resulting Java code. Some language features (e.g. entities) generate more code than others (e.g. functions), the distribution of language features determines the increase factor. It is actually better for the WebDSL compiler to try to reduce the amount of generated code, because it will improve build and deploy performance. Some improvements were also specifically made in this area, previous versions would have a larger factor of generated code.

When specific repeated patterns emerge in the applications, we can initially use static code templates to generate the code fragment, or look for potential language additions that cover the requirements. The topic of choosing how to improve the language iteratively is covered in Chapter 5.

*Fast Prototyping* Beyond an initial data model sketch on a whiteboard, we usually did not spend much time designing systems on paper. With only a rough idea for an application, it is very easy to create a working prototype with entity persistence and user interfaces in WebDSL without a large time investment. Especially when there is no live application and real data to preserve yet, the entire data model can be redesigned, and user interface code adapted over and over. When changing large fragments of the data model or user interface, strong consistency checking in WebDSL is the most visible. It prevents application faults immediately when changes are made, the IDE reports issues such as broken links and references, wrong types, incorrect template calls.

With a live database, it requires some more thought when redesigning entity data, as migration code will have to be written when changing existing data

properties. New entities and properties are still easy to add, tables and columns are created automatically when deploying a new version of the application. User interfaces and other definitions remain easy to change completely, even with live data.

Our typical way of working is understanding the workflow and intended use cases, implementing it as a prototype, and discuss with users to see what is working and not working, and what is missing. Then, adjust the prototype and repeat. A tangible prototype avoids misconceptions between application designer and client. It also makes the client enthusiastic by seeing the progress every time.

*Maintainability* As the applications become larger, the focus shifts away from only fast prototyping. Instead of writing large code fragments, it becomes more common to read old code and make adjustments. In this situation, the abstractions are really beneficial, as they reduce the effort to grasp the code. Entity definitions always deal with persistence, user interface templates with output and input of data, and access control rules always check the access policy. We are continuously evolving the applications, adding new features and changing old behavior, and have not run into blocking problems. Code written many years ago remains understandable, by the writer and by other programmers.

*Integrated Development Environment* The WebDSL IDE in Eclipse is used daily in our web application development work. Compared to a generic code editor, especially having syntax coloring and inline error markers makes it much easier to work with WebDSL code. Besides those features, also reference resolving and semantic code completion are convenient to have (e.g. to complete template names or entity property names). Inline error markers update fast enough due to the caching of file analysis. The code for detecting errors is shared with the compiler, so there are no differences between editor feedback and compiler feedback. Introducing a variant of the IDE suitable for dark mode also increased coding comfort quite a bit, this is used by all developers.

Initially, we fully integrated the build and deploy steps into the IDE. Unfortunately, in our experience, the Web Tools Platform for Eclipse, which provides support for interacting with a Tomcat process, would often lose track of running processes. This made redeployment very flaky, where often you would get an old version of the application after deploying. We decided to switch back to regular command-line deployment, because it did not suffer from such problems. The `webdsl run` command builds the application, unpacks Tomcat, starts Tomcat, and deploys the application. The output log is right in the terminal, and a simple `ctrl+c` exits the process reliably.

The IDE is not perfect, because there are some cases where updating errors is delayed. The cause of these problems is likely a situation where the cache is not cleared entirely. This has not been a high priority problem, so it was not addressed. Another issue is that it has been difficult to keep the WebDSL IDE updated with latest Spoofox developments, in particular the switch to new languages for static semantics involves quite some migration effort. This

has become clear in Max de Krieger's thesis work on migrating to SDF<sub>3</sub> and Statix [De Krieger 2022]. A negative side effect is that the latest Eclipse versions are having compatibility problems loading the WebDSL IDE plugin. We might explore other IDE platforms than Eclipse. The IDE options have improved over the years, with the availability of more easily pluggable IDEs such as Visual Studio Code [Microsoft 2023b], and standardization of IDE plugins for language support through the Language Server Protocol [Microsoft 2023a].

*Security* In Section 7.8 we listed some of our experiences we had regarding security. The main conclusion here is that web application security cannot be guaranteed by a programming language itself. A large part of the risk lies in the underlying software and hardware stack. Security is a continuous process for live web applications, it requires regular updates and staying up to date with news about discovered vulnerabilities.

When looking at web application vulnerabilities specifically, the cases where we had problems were often related to including an external library. Examples are the issues with an authentication library for Single Sign-On, and the integration of components for Markdown and LaTeX rendering as discussed in Section 7.8. We have not found evidence of data tampering, CSRF, or XSS issues arising from the use of standard library WebDSL components. The default input templates are well tested, however, when creating new input templates for WebDSL it takes some attention to avoid potential issues with tampering. The databind section in the template needs to make sure that only valid options can be accepted by the component. Another potential problem is incorrect use of `rawoutput`, this feature provides flexibility to skip the HTML escaping, which means the fragment should be safe (e.g. through custom filtering) or has been created by a trusted user. The reason for having a feature like `rawoutput` is extensibility. For example, it is required for the Markdown output rendering of WikiText type because it can produce HTML.

The access control policy in the applications is another security aspect. This feature has worked really well in the deployed applications, we have not observed issues where URL or form data tampering provides more access than the policy allows. While we did not have issues enforcing the access control policy, the specification of the policy rules can have mistakes. For example, in WebLab there was a case where too much information was visible. A newly added feature for student dossiers, an overview of all the questions in the course with the grades, initially did not take into account visibility of the assignments. This caused a problem where students could observe exam question titles before the release of the exam. In larger applications it can become harder to make sure the same capabilities (e.g. viewing assignment titles), is enforced with the equivalent access control restrictions in every context. Support for specifying access control rules on data could be interesting for these cases, however, there are some open questions for the design of such a feature in WebDSL, as discussed in Section 4.6.1.

*WebDSL Application Performance* Flexibility and expressivity has been chosen over raw performance in the design of the WebDSL language. In practice,



we see that applications need to grow the database quite a bit before a page might become too slow, and the cause is often inefficient data retrieval (see Section 2.4.7 and Section 7.7). Manual prefetch hints that optimize data retrieval can help in many of these cases.

The deployed applications are using MySQL, which requires configuration tweaks for larger applications to run smoothly, e.g. increasing memory limits and cache sizes. There are several supporting tools to identify such setting tweaks.

In the user interface part of the language, one problem we sometimes encounter is that the amount of HTML can become quite large, which usually also indicates a usability problem. For example, a page might have too many buttons (thousands), in this case there might be a better user interaction possible by splitting activities across different pages. Or a page loads too much data, and needs pagination or filtering. Sometimes there might also be an image or other resource that is too large, for these cases running a website scanning tool like GTmetrix [2023] quickly uncovers the problems. Page caching speeds up stable pages like the conference pages in Conf Researchr. Overall, the performance is suitable for the applications we operate.

*WebDSL Compiler Performance* The performance of the WebDSL compiler could be further improved. While the caching features in the compiler help a great deal, every round-trip from save to deploy still takes at least 20 seconds. Comparing to other tools this is very slow. For example, when developing user interfaces with React [2023], there is a process that detects file changes and can immediately process an update and deploy changes without noticeable waiting time. The discussion in Section 6.9 reflects on the problem of build speed and identifies potential improvements.

We notice the delay especially when finetuning user interfaces, only making small changes for which you want to inspect the result. On the other hand, for an experienced developer implementing larger fragments, it might take a while before a rebuild and redeploy is necessary. The abstractions help to make fewer mistakes, and there is immediate feedback in the editor when inconsistencies arise. For tweaking styling in CSS, we can also use the browser support for live editing, and afterwards copy the fixes into the application.

*Testing* A comprehensive regression test suite for the WebDSL compiler employs browser testing to perform complete integration tests. Browser tests are specified through an interface that wraps the Selenium [2023] library in WebDSL. This allows starting a browser, clicking buttons, entering input data, and inspecting the resulting page content. The test suite involves hundreds of small WebDSL applications where specific language features are tested. We created a separate testsuite specifically for verifying the consistency checks and the errors that the compiler and IDE should produce. The tests in this testsuite run the WebDSL compilation, and check expected errors in the output messages of the compiler.

Although the WebDSL language itself is tested well, for specific applications we have not created automated regression tests often. Testing is done manually

while developing a change or new feature. Common behavior is already tested in the compiler. It is rare to encounter a confusing bug, e.g. in persistence or in rendering. When we do observe an unexpected error, it is more often an issue in the compiler, and not in the application. In this case, a fix is created for the compiler, together with a new regression test. Unfortunately, we do not have time to fix every issue, as application improvements often take precedence. Especially when there is a trivial workaround, an issue might remain open longer.

*Wearing Many Hats* An issue that we experienced is that it is hard to separate the roles of language designer, application developer, operations support, consultant, all at the same time. Often the most practical issue has precedence, which is typically some support question asked over email. Especially support requests can be costly for focus, because they require context switching even though the individual task might be small. Some issues simply take precedence over everything, such as hardware issues like a failing harddisk or crashed server, or patching security vulnerabilities. Working on the language is the least urgent, when application features are requested as well, even though language improvements benefit all the applications and have the most impact.

*Flexible Access Control* To reduce support requests it is important to think about which actions the users can help with, this is where the access control models get interesting. In all our applications there are multiple levels of roles connected to a section of the application. For example, courses in WebLab have instructors, teaching assistants, and observers. Some roles allow managing the roles with lower permissions in their own section of the web application. It is important to start early with adding these features to enable users to do more and support each other, in order to reduce support requests to our development team. The access control model should handle people with multiple roles in different contexts, people leaving and joining, and transferring tasks from one person to the next.

*Being Selective With Time* Another observation we have made is that because the language enables fast prototyping and quick results, the amount of requested features can also increase quickly. Care has to be taken that these features are broad enough to be useful to multiple users and not just serve one person's workflow.

*Not Targeting Other Platforms* We have experimented with targeting other platforms with WebDSL, namely Python, but we experienced from the start that it was very hard to keep a consistent semantics among those back-ends. In particular, the used libraries can have a large impact on the behavior and might not be easy to reproduce in other platforms. Running a web application also gives some freedom when it comes to the platform. As long as the application is working as intended and performing well, the user does not have to care which implementation platform is being used on the server. A similar issue occurs when code is being moved to the DBMS, or to the client, small details in the semantics might be tricky to get aligned, e.g. integer overflow behavior or

string equality checking while taking database collation settings into account.

*Language Maintenance* Maintenance on the language, in particular the libraries in the runtime system, has been slow due to giving priority to supporting and operating the applications. New features, in particular client-side operations, websockets, have not been thoroughly investigated as abstractions. In principle there is no limitation that would prevent the inclusion of abstractions for these concerns.

*Opportunities* We see many opportunities for new software to address workflow problems, however, having an effective language does not save you from the effort to dive into the heads of the clients to understand what they are really asking for. Continuous prototyping can help a lot in requirements gathering, and this is the process we have used in all our applications. Emphasizing our unique approach with a custom programming language can generate some resistance and worry about future support and development. What we see in practice, however, is that applications tied to a particular framework developed by external parties, are often abandoned when the contract ends or getting entirely redone in the next popular framework.

## 7.10 Threats to Validity

The first threat to validity is discussed in Section 1.8: in the Technical Action Research method, a major threat to validity is that the researcher, who developed the artifact, is able to use it in a way that no one else can. Most members of our academic workflow engineering team joined in later phases of the project, and were not involved in the entire artifact design and implementation process. We have also had students contribute to our application development, e.g. in the case of EvaTool and WebLab. On the other hand, we do know the limits of the system quite well, because we are a team that also maintains the language. WebDSL did not attract other users beyond our group, although it does not help that documentation and smoothing the new user experience were not a high priority.

The experiment we performed is very open. For evaluating practical applicability, the criteria that matters most to us, it does not make sense to have a very controlled and repeatable experiment. Running it in the wild is the only way to get answers about practical applicability. A threat to validity in this case is that the discussed application case studies are not open source software. The reason for this is that funding is acquired for developing these applications, which requires protecting the value that these applications provide and prohibit sharing it freely. The funding for these applications also sustains maintenance and further development of WebDSL itself.

Another threat to validity is that we do not create fully equivalent applications in other technical solutions to compare against. Unfortunately, this would be too much effort, for very little practical value to our clients. We did make smaller comparisons such as the analysis performed in Chapter 8.

The WebDSL language helps us create meaningful applications that tackle real automation problems. In our experiences at the university, external software development companies often fail to fully understand the requirements and think along with the client. There are many failed projects, contract conflicts, and applications that are in production but not further developed. Our experience with both the software and the domain of university education provides an edge to making these applications. This edge was important for getting real users, however, for the experiment it could be considered a threat to validity, is that edge a necessity? In our experience it was very productive to have a close connection to the real clients, otherwise you cannot prototype very fast, even if the chosen software system supports it. In an earlier stage of our application development activities we tried developing a system for an external client in an unfamiliar domain. The progress was very slow, largely because communication was slow, and we did not have much opportunity to talk with actual users. Our focus shifted to the academic domain, where such problems were not forming bottlenecks, and we could focus on the technical implementation instead.

## 7.11 Conclusion

In this chapter, we evaluated the practical applicability and reliability of WebDSL based on our experiences designing, implementing, and operating several real-world applications. We summarize the answers to the questions stated in the introduction of this chapter:

1. *How well did the focus on web programming abstractions, removing boilerplate code, and providing timely and accurate feedback on problems in application source code work in practice?*

The web programming abstractions and reduction of boilerplate code enabled fast prototyping in the initial phase of development, as well as maintaining source code over a longer period of time. Abstractions keep the code readable, the purpose of a code fragment is immediately clear from the language element used. Boilerplate code is generated for many low-level aspects of web programming, such as setting up entity persistence mapping to database tables, generating form input identifiers, and importing form data in action handlers. The compiler and IDE result in a productive development environment, with immediate feedback on problems in application code. These features enable making changes while minimizing risk of breaking old code. However, the build and deploy speed is an annoying bottleneck in application development.

2. *We set out to build a system that ensures reliability (robustness, performance, scalability, and security) of applications, did we indeed reach these goals?*

Our applications have been running stable for a long time, with usage numbers steadily increasing. Some smaller improvements have been made to improve robustness, such as a transaction retry mechanism,

and feedback on submit failure. The most critical issue we encountered in the generated application code was a problem in database connection handling, which was resolved. Application performance is good enough out of the box for medium-sized web information systems in the university and academia domain, in some cases it required manual tweaks for optimizing database access when loading many entities. A programming language can do a lot to avoid security vulnerabilities in web programming, however, security depends on the whole software and hardware stack, not just the application code.

In Chapter 8 we will compare WebDSL to other solutions found in practice and in related work.

# Related Work

---

# 8

## 8.1 Introduction

WebDSL has been designed to integrate web application programming concerns into a single language. Comparing with existing languages and frameworks, it functions as a full-stack web programming solution like the Spring [2023] framework for Java, Django [2023] framework for Python, Ruby on Rails [2023], Laravel [2023] for PHP, and Sails [2023] for Node.js. React [2023], Angular [2023], and Elm [2023] are purely client-side rendering frameworks. These are not full-stack solutions, and require a separate server-side application to handle concerns like persistence and access control.

Integrating concerns as language features requires making design decisions up front. This means WebDSL is particularly suited for information systems. However, it might not be a good choice for other styles of web applications. For example, if you need raw performance, or a client-side rendered user interface with mostly web socket communication, then the current language features in WebDSL are not sufficient.

In this chapter, we first discuss conventional full-stack web frameworks (Section 8.2), highlighting points from the initial motivation described in Chapter 1. Then, we compare WebDSL with other multi-tier programming languages for web programming proposed by other researchers (Section 8.3). Because these are also language-integrated solutions, they are the most similar to WebDSL and give opportunity for a detailed comparison. In particular, case studies and practical experience reports are discussed, because they provide information on the practical usability of the tools and approaches. Next, a general comparison with modeling and low-code tools is made (Section 8.4), although the gap between these graphical model editors and WebDSL is larger than other discussed solutions based on programming languages. Section 8.5 concludes this chapter on related work.

## 8.2 Conventional Full-Stack Web Frameworks

The motivation for WebDSL is based on addressing shortcomings in conventional web frameworks. Examples of popular full-stack web programming solutions are the Spring [2023] framework for Java, Django [2023] framework for Python, Ruby on Rails [2023], Laravel [2023] for PHP, and Sails [2023] for Node.js. These frameworks are popular for a reason, they get the job of building a web application done. These web frameworks provide abstraction over web programming details and allow reuse of existing knowledge for building web applications. Due to the large number of people using these frameworks they are well-tested and it is easy to find many tutorials, examples,

and answered questions about particular problems. As explained in Chapter 1, there is still room for improvement in several areas, which can be achieved by a programming language dedicated to web programming. The areas of improvement are:

1. *Abstraction* Web frameworks are designed within the constraints of a general-purpose programming language. Design choices in the programming language can cause notational overhead and potential confusion when used in a web programming context. For example, object identity has different requirements when objects are being persisted in database tables with an ORM (Section 2.4.4). Cross-cutting concerns such as access control are not easily specified in abstraction features provided by most general-purpose programming languages.
2. *Static Verification* The general-purpose programming language's compiler and IDE sees the web framework as a regular library, and is limited to the typechecking strength in the language to enforce correct usage. Third party linting tools can provide additional framework usage checks, but require the programmer to make the right selection of tools for the combination of frameworks and features used.
3. *Security* General-purpose programming languages are not designed with web security concerns in mind, even simple features like string interpolation can encourage wrong programming patterns that introduce injection vulnerabilities (Section 1.5.2). More complex features like `eval` in Ruby, can lead to arbitrary code execution vulnerabilities (Section 1.5.5). Creating user interfaces in the templating library of a web framework may require the programmer to explicitly enable security features, such as managing a CSRF-token.

To illustrate these issues with a concrete example, we investigate code fragments for the Django web framework in more detail, and make a comparison with WebDSL code.

### 8.2.1 Django

Django is a high-level Python web framework for rapid development of web applications. It takes care of common tasks in web programming, so the programmer should be able to focus on writing the code that is specific to the application. To compare Django to WebDSL, we will look at concrete code fragments based on the getting started Django Tutorial [2023]. In this tutorial, the construction of a simple questionnaire application is described.

Figure 8.1 shows the data model definition in Django for persistence to a database using Object-Relation Mapping (ORM). This fragment creates a `Question` and a `Choice` class for storing and retrieving questionnaires. The `__str__` method returns the string representation of the object in Python. The equivalent for `__str__` in WebDSL is the `name` property, which can be declared as property or annotation (Section 2.4.2). The first thing to note here, is that

```

from django.db import models

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
    def __str__(self):
        return self.question_text

class Choice(models.Model):
    question = models.ForeignKey(Question)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
    def __str__(self):
        return self.choice_text

```

---

Figure 8.1 Django tutorial data model

```

1  entity Poll {
2    question : Text (name) // can also be WikiText to get Markdown input/output
3    choices  : [Choice]
4    published : Date
5  }
6
7  entity Choice {
8    answer : Text (name)
9    poll   : Poll (inverse=choices) // bidirectional association for easy navigation
10   votes  : {Vote} // store votes and derive total
11   total  : Int := votes.length
12 }
13
14 entity Vote {
15   // could also reference user making the vote and other relevant info
16   chosenOption : Choice (inverse=votes)
17 }

```

---

Figure 8.2 WebDSL version of Django tutorial data model

the data persistence class is a regular Python class. Programmers need to keep track of distinguishing between Python classes used for persistence, and regular Python classes without ORM behavior attached. The language itself has no notion of ORM built-in, so it will not report issues when e.g. trying to store data in a class that has no persistence behavior. The second thing to note is the amount of boilerplate code, such as the word ‘model’ being used 10 times, in order to distinguish a persistence class from a regular Python class. Database table creation commands can be derived from the declared model, and can be installed to the database using a few manual commands. The commands for updating the database have to be repeated after every data model change. In WebDSL, we have automated the database table creation steps, the database is initialized and updated automatically when running the application. Figure 8.2 shows the related fragment in WebDSL. This example has been expanded compared to the Django fragment, to show WebDSL coding style and create a complete application (together with the code in Figure 8.4). The `Poll` entity holds the question text and references the choices belonging to it. The `Choice` entity holds choices and references the votes. Bidirectional associations (declared using `inverse` annotation) between `Poll.choices` and `Choice.poll`, and `Choice.votes` and `Vote.chosenOption`, allow easy navigation between related entities without writing any explicit join queries.



```

Polls/view.py

# ...
def detail(request, question_id):
    1 question = get_object_or_404(Question, pk=question_id)
    2 return render(request, 'polls/detail.html', {'question': question})

def vote(request, question_id):
    1 question = get_object_or_404(Question, pk=question_id)
    3 try:
        selected_choice = question.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the question voting form.
        return render(request, 'polls/detail.html', {
            4 'question': question,
              'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        # Always return an HttpResponseRedirect after successfully dealing with POST data.
        # This prevents data from being posted twice if a user hits the Back button.
        5 return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))

def results(request, question_id):
    1 question = get_object_or_404(Question, pk=question_id)
    2 return render(request, 'polls/results.html', {'question': question})

Polls/templates/polls/detail.html

6 <form action="{% url 'polls:vote' question.id %}" method="post">
  {% csrf_token %}
  <fieldset>
    <legend><h1>{{ question.question_text }}</h1></legend>
    4 {% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}
    {% for choice in question.choice_set.all %}
    3 <input type="radio"
      name="choice"
      id="choice{{ forloop.counter }}"
      value="{{ choice.id }}">
      <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br>
    {% endfor %}
  </fieldset>
  <input type="submit" value="Vote">
</form>

Polls/templates/polls/results.html

<h1>{{ question.question_text }}</h1>
<ul>
  {% for choice in question.choice_set.all %}
  <li>{{ choice.choice_text }} -- {{ choice.votes }}</li>
  {% endfor %}
</ul>
<a href="{% url 'polls:detail' question.id %}">Vote again?</a>

```

Figure 8.3 Django tutorial user interface

Figure 8.3 shows fragments of the Django tutorial that together form a user interface component with submit action for the questionnaire application. We will highlight the points of boilerplate code and potential errors, these numbers correspond to the arrows in the figure:

1. The programmer must explicitly load arguments from URL parameters, e.g. `question_id`, this is boilerplate code that results from having separate view templates and controller classes.
2. View templates must be explicitly seeded with data by the controller class, in this case it is only the single occurrence of `question`, however

the number of passed arguments increases with more complex view templates. Additionally, the page reference `polls/detail.html` is a dynamic string value, which will not be checked by a compiler to be a correct page reference.

3. The names used for input elements in the view and unpacking this data in the controller must match. In this case, the name `choice` connects the view with the controller. This name is not checked by the compiler. Additionally, there is boilerplate code required for unpacking all the submit parameters from `POST` data in the controller, which also increases for more complex view templates.
4. Enforcing data validation is spread over the controller and the view template. The check and message are specified in the controller, and the view template must check for the error and insert this error at a suitable location based on a matching argument value. This style of data validation causes boilerplate code from passing additional arguments to the view template, and introduces potential mistakes from incorrectly or omitting checking for data validation errors produced in the controller.
5. In general, a redirect is required after a form submit to avoid accidental form resubmission. This is not enforced automatically, but requires boilerplate code from the programmer to declare the redirect.
6. The programmer must remember to include the CSRF token to enable Cross-Site Request Forgery protection. Forgetting to include this fragment makes the form vulnerable to attacks, which will not be reported by the compiler as an error.

Figure 8.4 shows the fragment expressed in WebDSL code. The shortcomings of boilerplate code and potential errors observed in the Django fragment are addressed in the following ways in WebDSL:

1. Page argument retrieval from the database is automatic, in this case the `p` argument with type `Poll`.
2. Data can be referenced in user interface templates directly, e.g. `p.question` displays the question from the `Poll` argument.
3. Input element names and data retrieval from submit parameters are part of the abstraction provided by input templates. The `radio` input template generates parameter names using deterministic template identifiers, and updates the `r.chosenOption` property.
4. Data validation is integrated in the language, and can be enforced by a single `validate` declaration indicating the check, error message, and location to show the error.
5. Redirect after submit is automatic, either the user specifies a `return` page to redirect to, or a redirect to the same page happens.

```

1 page poll( p: Poll ){
2   var r := Vote{}
3   div{ ~p.question }
4   form {
5     radio( r.chosenOption, p.choices )
6     validate( r.chosenOption != null, "Pick one option" )
7     submit action {
8       r.save();
9       return viewAnswers( p );
10    }{ "Confirm" }
11  }
12 }
13
14 page viewAnswers( p: Poll ){
15   div{ ~p.question }
16   for( o in p.choices order by o.total desc ) {
17     div{ ~o.total ": " ~o.answer }
18   }
19 }
20
21 // root page with demo data setup to complete application code
22 page root {
23   submit action {
24     var p := Poll{ question := "Tabs or spaces?" }.save();
25     p.choices := [
26       Choice{ answer := "tabs" }
27     , Choice{ answer := "spaces" }
28     ];
29     return poll( p );
30   }{ "go" }
31 }

```

---

Figure 8.4 WebDSL version of Django tutorial user interface

6. Cross-Site Request Forgery protection is automatically enforced in generated template identifiers by seeding these with the unique data being accessed and the identity of the logged-in user.

In the Django example, the separation of the controller from the view template is at the root of much boilerplate code. The view and controller definitions need to be kept in sync, which causes mental overhead for the programmer. For example, if an additional form input is added in the view, an additional request parameter needs to be decoded in the controller, possibly requiring additional error handling in both the view and controller as well. Besides additional inputs, other constraints also require checks in both locations. For example, a time limit feature would require a condition to be checked in the view to decide whether to render the form, and the same condition check in the controller to decide whether to handle submits from this form.

In the Django tutorial, a race condition is pointed out, because adding a vote is done by updating the same value in the database from different transactions (`selected_choice.votes += 1`). Database transaction semantics might lose votes when multiple votes happen at the same time, because when two transactions both increment a number by 1, the end result would be the same number and not considered a conflict, even though a vote is lost. The solution suggested in the Django tutorial is to use a feature for executing direct

SQL expressions to increment transactionally. This solution breaks down the abstraction level provided by the framework, programmers will need to decide when it is required to escape to direct database manipulation. In WebDSL, this transaction conflict could also occur if votes were counted with an integer property. Instead, the example shows a more extensible way of expressing this data and avoiding transaction conflicts, using a separate `Vote` entity. This entity could also be used to store more information about the vote action, e.g. to avoid double votes. Each transaction will only add data, namely additional `Vote` entities, and will not try to update a value that has high potential for transaction conflicts.

### 8.3 Multi-tier Web Programming Languages

Several multi-tier web programming languages have been proposed that generate code for client, server, and database from one application definition. In general, they use the same technique as WebDSL to achieve abstraction, namely a language-integrated solution. In particular, we will be looking at the `Ur/Web` and `Links` languages, which are also full-stack languages used to build the entire web application. These are complete languages with a large number of features that could be examined in detail. Just looking at the persistence solution already shows different approaches being used, where WebDSL uses ORM abstraction, `Links` generates SQL from list comprehension syntax, and `Ur/Web` integrates SQL as a sublanguage. We recognize that the research focus and design philosophy is also different for these languages, e.g. `Links` explores advanced features for translating generic code to queries, and `Ur/Web` explores typechecked metaprogramming abstractions. However, the authors provide working demo applications which can function as concrete comparison cases. For the code comparisons we examine essential web application features, namely data retrieval and form input handling for server-side storage of data. We compare equivalent WebDSL code to the examples. Although a larger case study would be insightful, it is time-consuming to understand the language idioms and create these applications in all the languages to get a representative comparison. That is why we compare example code provided by the tool authors themselves. The comparison is also based on available published material, of which some parts might be outdated, or do not reflect the latest status of the tools.

Although a small application comparison is useful, the benefits provided by WebDSL's language features in terms of abstraction, static verification, and security become more pronounced when application codebases get larger (see Chapter 7). For example, the compared languages mix client- and server-side code. This has implications for security, because the client-side code and data in the JavaScript engine of the browser is vulnerable to tampering by the user. Controlling access and ensuring valid data enters the system cannot be done purely on the client. In WebDSL, all pages and actions that can be requested are forced to be protected by an access control rule, and action code only runs on the server.

### 8.3.1 Ur/Web

Chlipala [Chlipala 2010] develops the Ur functional programming language which aims to provide verification of rich program properties through type-checking. Ur/Web [Chlipala 2015] is Ur with a special standard library that provides web programming features on top of the Ur base language. Ur/Web supports construction of dynamic web applications backed by SQL databases. It enables checking dynamic web application code that generates HTML, JavaScript and SQL.

*Language Design Goals* There are many similarities with the WebDSL design goals, in particular the focus on static analysis, and using a custom programming language to abstract over web programming concepts. The Ur/Web manual [Chlipala 2020] states that a well-typed Ur/Web application should not crash during page generation, but may also not:

- Suffer from any kinds of code-injection attacks
- Return invalid HTML
- Contain dead intra-application links
- Have mismatches between HTML forms and the fields expected by their handlers
- Include client-side code that makes incorrect assumptions about the “AJAX”-style services that the remote web server provides
- Attempt invalid SQL queries
- Use improper marshaling or unmarshaling in communication with SQL databases or between browsers and web servers

While WebDSL shares many aspects in the list above, there is a difference in how some of these are achieved. In particular the way mismatch between forms and action handlers is handled in WebDSL, is through a combination of deterministic id generation by the runtime, and standard library definitions of input components. The application developer does not need to think about form field names. Additionally, it enables additional protection against CSRF by avoiding easily guessable form parameter names. One feature that WebDSL currently does not provide is a way to write client-side code as part of the language (JavaScript can be included but is not checked against other application definitions). The “AJAX”-style in WebDSL is not client-side code but server-side calculated updates to the DOM. The remainder of this section will discuss features of the Ur/Web language using examples of these features, and compare them to WebDSL.

```

table t : { A : int, B : string }
PRIMARY KEY A
fun list () =
  rows <- queryX (SELECT * FROM t)
    (fn row => <xml><tr>
      <td>{[row.T.A]}</td>
      <td>{[row.T.B]}</td>
    </tr></xml>)
  return <xml><table>{rows}</table></xml>
fun main () =
  xml <- list ();
  return <xml><body>{xml}</body></xml>

```

---

Figure 8.5 Table Ur/Web demo

```

1 entity T {
2   b : String
3 }
4 template list {
5   for( x in (from T) ){
6     row {
7       col { ~x.id }
8       col { ~x.b }
9     }
10  }
11 }
12 page root {
13   table { list }
14 }
15 htmlwrapper {
16   table table
17   row tr
18   col td
19 }

```

---

Figure 8.6 Equivalent entity declaration in WebDSL

*Persistence* A table declaration in Ur/Web declares a SQL table with rows of a particular record type. The embedded SQL syntax enforces that all queries and updates are typechecked. It prevents executing invalid SQL queries, and the marshaling and unmarshaling happens implicitly without the risk of making wrong assumptions about the types of tables. The system generates a file with SQL statements to initialize the tables in the database. Figure 8.5 shows a simple table definition with two columns for an integer value and a string value. The integer column is manually declared as primary key. This fragment is taken from the Ur/Web SQL demo [Chlipala 2023]. The `queryX` function takes as arguments a SQL query, and a function that maps the result rows of the query to a list of XML fragments, enclosed by `<xml>` tags. Accessing the column data in the mapping function is typechecked. Ur/Web performs escaping on the values that are inserted from the database into the output XML to avoid XSS issues.

Figure 8.6 shows an equivalent WebDSL fragment. Entity declarations used to persist data will create database tables automatically when deploying, and update the tables when entities change. Primary keys are implicit in WebDSL, the runtime implicitly creates primary keys for all entities. The value of a primary key, which is of type `UUID`, can be accessed through the `id` property. Using `htmlwrapper` simple HTML tags such as `table`, `tr`, and `td`

```

table t : { Id : int, Name : string, Parent : option int }
PRIMARY KEY Id,
CONSTRAINT Name UNIQUE Name,
CONSTRAINT Id CHECK Id >= 0,
CONSTRAINT Parent FOREIGN KEY Parent REFERENCES t(Id)
fun main () =
  list <- queryX (SELECT * FROM t)
    (fn r => <xml><tr>
      <td>{[r.T.Id]}</td>
      <td>{[r.T.Name]}</td>
      <td>{case r.T.Parent of
        None => <xml>no parent</xml>
        | Some id => <xml>{[id]}</xml>}</td>
    </tr></xml>);
  return <xml><body>
    <table>
      {list}
    </table>
  </body></xml>

```

---

Figure 8.7 Table Constraints Ur/Web demo

```

1 entity T {
2   name : String
3   parent : T
4 }
5 page root {
6   table {
7     for( x in (from T) ){
8       row {
9         col { ~x.id }
10        col { ~x.name }
11        col {
12          if( x.parent == null ){ "no parent" }
13          else { ~x.parent.id }
14        }
15        col { ~x.parent.name }
16      }
17    }
18  }
19 }

```

---

Figure 8.8 Equivalent entity references in WebDSL

can be syntactically used as templates. The property accesses `x.id` and `x.b` are typechecked. Similar to Ur/Web, queries are part of the language, data marshaling happens implicitly, and data is escaped to avoid XSS issues.

For the next comparison, we include references to other tables. The fragment in Figure 8.7 is taken from the Ur/Web Constraints demo [Chlipala 2023]. Foreign key constraints are explicitly declared in Ur/Web. A type error will be reported when a foreign key column refers to a property that is not a primary key or does not have a uniqueness constraint. Retrieving and updating data always requires explicit select and update queries in Ur/Web. The parent reference in the example is the primary key, retrieving the parent data requires a join in this query or an additional separate query. The interesting feature of these queries is that they are integrated into the language and typechecker. The typechecker understands the structure of the result set of a query that for example contains 'join' and 'group by' clauses.

The equivalent WebDSL code is shown in Figure 8.8. In WebDSL queries

are only needed to select entities that are not directly related to the page or template arguments, and updating always happens implicitly by setting new values for object properties. WebDSL employs an ORM solution to avoid having to write SQL queries for most data retrieval cases. When an object is loaded from the database (typically the initial objects loaded are the page arguments), the object graph can be traversed by accessing the members, which might trigger additional queries in the background to load the required data (unless there is prefetching active which avoids additional queries). In practice, this means explicit queries are usually not necessary, however, they are supported in case a page component benefits from loading data with an explicit join. Doing an output of a value that does not exist, will skip rendering that call. For example, when the value of `x.parent` is null, the rendering of `x.parent.id` and `x.parent.name` is skipped (see Section 2.5.4).

In general there is no concept of Objects or an ORM in Ur/Web because it follows a functional programming style, using records to hold data instead. These records do not have a built-in identity like Objects, two records that have the same values are indistinguishable. Identities need to be handled explicitly using a primary key field when executing queries. The WebDSL entity abstraction simplifies this aspect for the developer. The application developer does not have to make design decisions about primary key fields and foreign keys. Database primary key identities are implicitly tracked and kept in sync with object identities in the request handling session. Multiple queries that select the same objects will get references to the same in-memory objects in the result list (in the request handling session a particular entity instance is only instantiated once). Changes to objects are recorded and implicitly flushed with update queries at the end of a request handling session. References to other entities automatically get the required schema with foreign keys to ensure data integrity (every persisted entity reference refers to a persisted entity), including support for bidirectional one-to-one and one-to-many relations.

*User Interface Form and Action Handling* As indicated in the language features of Ur/Web, it prevents mismatches between forms and the fields expected by their action handlers. The fragment in Figure 8.9 taken from the Ur/Web Constraints demo [Chlipala 2023] illustrates this feature. In this example the form fields `Id`, `Name`, `Parent` are tracked by the type system, and it enforces that the handler function `add` expects a record containing exactly those fields with their types. To store the entered data, the handler performs an insert query through the `dml` (Data Manipulation Language) function. Antiquoted Ur code can be inserted into queries and DML commands, the typechecker catches mistakes in the expression types of the inserted fragments. The `readError` function will throw a runtime error if the submitted data is incorrectly formatted with respect to the type.

An equivalent WebDSL fragment is shown in Figure 8.10. A mismatch between form fields and action handler can also not occur, because form ids are handled by the runtime system implicitly. The `databind` abstraction in WebDSL using deterministic template identifiers avoids having to specify input names. An update query is not necessary because the persistence mechanism



```

fun main () =
  return <xml><body>
    <form>
      <table>
        <tr> <th>Id:</th> <td><textbox{#Id}/></td> </tr>
        <tr> <th>Name:</th> <td><textbox{#Name}/></td> </tr>
        <tr> <th>Parent:</th> <td><textbox{#Parent}/></td> </tr>
        <tr> <th/> <td><submit action={add}/></td> </tr>
      </table>
    </form>
  </body></xml>
and add r =
  dml (INSERT INTO t (Id, Name, Parent)
      VALUES ({{readError r.Id}}, {{r.Name}},
              {{case r.Parent of
                 "" => None
                 | s => Some (readError s)}}));

```

---

Figure 8.9 Form inputs Ur/Web demo

```

1 page root {
2   var t := T{}
3   form {
4     table {
5       row { header { "Name:" } col { input( t.name ) } }
6       row { header { "Parent:" } col { input( t.parent ) } }
7       row { header col { submit action{ t.save(); }{ "Add" } } }
8     }
9   }
10 }

```

---

Figure 8.10 Equivalent form inputs in WebDSL

commits persisted data automatically at the end of the request. The `save` call is sufficient to mark the new entity instance for it to be persisted. The input widget created for `input( t.parent )` is not a text field input, but a dropdown where the only options are the allowed entity instances. Data validation for types, e.g. `valid integer`, is built-in for WebDSL inputs. These would not require the developer to insert a check like `readError`. It will also not be a runtime error like in the case of the Ur/Web example, instead it will display a form input error message with customizable styling.

*Security* The security model, as reported in the manual, states that there is a pragmatic approach to security. Some things cannot be secured reasonably, such as any code invoked using the foreign function interface (FFI). Strings are never implicitly interpreted as programs, to avoid code-injection attacks, this is enforced by having specific language constructs for all tiers, there are no embeddings of code in string values necessary.

To protect against Cross-Site Request Forgery (CSRF) attacks, cookie values are signed cryptographically. Signing and signature checking are inserted by the compiler. The form input names themselves are readable in the page source and equivalent to the declared names in the application code. A form that is available without logging in (or any other cookies), would be vulnerable to CSRF. A difference with WebDSL is that WebDSL implicitly makes the form input names unique based on the specific data it references (which already provides some protection), and additionally adds the user identity in the input

name hash, if a login session (based on cookie) is available.

The Ur/Web manual also states that there is no guarantee that all function calls experienced by the application are possible according to legit traversal of links and forms. WebDSL provides stronger guarantees about which actions and functions can be invoked. Form actions in WebDSL are not direct endpoints for request handling, they are always executed in the context of the request handling of a page. Actions that are no longer available according to the page flow and current state of the data, cannot be executed.

*Experience Report* Van Casteren [Van Casteren 2019] reports on his experiences with Ur/Web in a one-person commercial web application project. The listed benefits are that it is trivial to keep front-end and back-end code in sync. Similarly, keeping database query code up-to-date is enforced by the language. Explicit control is possible for handling rendering and client-side calculation. The runtime is performant. The language provides an option to mark a function as pure, to enforce that there are no side-effects.

There were also negative experiences while using the tool in practice. Compiler errors were often confusing. There is a small library ecosystem and no package management. Build times can get annoying, taking minutes to build an application from scratch.

When comparing with the WebDSL experiences, the error reporting in WebDSL has been one of the design goals. Errors are represented in domain concepts that relate directly to the code, and do not include type system mechanism details. Similar to Ur/Web, WebDSL has a very limited user base and small library ecosystem. Build times are also a problematic point in WebDSL, and improvements for incremental building have made a huge difference in the practical applicability (see Section 6.7).

### 8.3.2 Links

The Links [Cooper et al. 2006] web programming language provides a single language from which code for all tiers (client, server, and database) is generated. Functions are labelled as intended to be executed on client or server. Client-side code compiles into JavaScript to run on the client. Queries are written in Links notation and compiled into SQL. The database abstraction provides transparent optimized database queries derived from the code. This differs from solutions where SQL is embedded in the language or other query syntax tree solutions as used in WebDSL, Ur/Web, or LINQ for .NET [Meijer, Beckman, and Bierman 2006]. Queries are written using Links list comprehension syntax and can be built out of reusable components. A notable feature of Links is that all server state is serialized and passed to the client, and restored to the server when required. The used approach is that continuation objects are defunctionalized, functions are replaced with unique identifiers. The execution context is then represented by identifying the closures which need to execute in the future of the computation, and the dynamic data needed as part of these closures.

In WebDSL, server calculation state is not exposed to the client. The only output of data comes from inserting values into the generated HTML code.

The only input of data comes from accepting a URL and form parameter values, which get checked by the input components to contain only valid options. These are single instances or collections of primitive values and entity identifiers. These parameters do not encode calculations. Additionally, the input identifiers are protected from CSRF attacks by making them unique based on the referenced data and logged-in user information. WebDSL does not provide an explicit mechanism for continuing a calculation at a later point. Complex operations with multiple steps are typically backed by an entity to store intermediate state in the database. From a security perspective, storing the whole execution context on the client raises concerns, as this state would be vulnerable to tampering without additional security measures.

Cooper et al. [Cooper et al. 2008] address a shortcoming in the initial work on Links by providing a mechanism for abstracting over form components called formlets. This work identifies the following problems in conventional web programming solutions:

- no static association between form inputs and handlers;
- fields retrieved as strings individually, need to be parsed and recombined into data structures;
- two form definitions cannot be merged without concern for name clashes, making it hard to abstract over form components, e.g. using a form component twice in a larger form.

The formlets solution provides static association (consistency check between form parameters and action handler), supports processing raw form data into structured values, and allows composition by generating fresh field names at runtime.

The WebDSL solution for forms and inputs provides similar benefits. The check between form input and action handler is made obsolete by the provided template identifiers from the runtime, and databinding abstraction. Input templates provide abstraction over transforming field values back into data model structures. Form composition is effortless, templates containing inputs can be reused within conditional sections and loops without risking name clashes. Customization is supported in several ways, such as changing class attributes with attribute overrides, overriding a template globally or locally within a certain context, or simply taking the definition of an input template and creating a variant of it with a different name. For example, in the built-in library, there are variants for the set of entities input, one based on the `<select multiple>` HTML input component, and one based on a list of checkbox inputs.

Fowler et al. [2021] performed a larger case study using Links. They reimplemented an existing Java web application in Links, ensuring functional correctness by crawling both applications, and evaluated the differences in performance. The studied application is a web front-end for a curated scientific database. An important feature of Links for this case study is the shredding technique for implementing nested queries by Cheney, Lindley, and Wadler

[2014]. This technique gives an upper bound guarantee on the number of SQL queries: the upper bound is the number of occurrences of collections in the query result type, which is independent of the number of records returned by a query. Another relevant Links feature is translation of user-defined functions to SQL code where possible, which relies on query normalization [Cooper 2009]. The queries Links generated in the case study were both fewer and faster to execute than the ones in the original Java application. However, the ordinary code execution speed for generating pages was worse than the equivalent Java code, making the overall performance typically comparable.

WebDSL generates code for the Java platform, and it benefits from the general maturity of that underlying platform, in particular for performance. Note that WebDSL also generates the database schema entirely from the entity definitions, so for a similar experiment as performed by Fowler et al. [2021], the original database would have to be migrated first. WebDSL uses an ORM with lazy loading for persistence, which, unfortunately, does not give an upper bound to the number of queries. When iterating a collection, a query could be executed for every element in the collection to retrieve additional data (N+1 query problem). A potential solution is prefetching of required data based on static analysis of the program, which we explored in Gersen [2013]. We found that having more complex join queries up front, which intuitively should give a performance boost, does not guarantee a performance improvement. Additionally, due to the many different types of caches in the DBMS, it is very hard to benchmark the results accurately. A possible reason for the mixed results is that tiny queries that were generated received more benefit from the caches than large join queries. For our deployed applications, we provide our DBMS with a large amount of memory to benefit as much as possible from various caches. In our real-world applications, we have used manual prefetch directives in some cases to preload data on pages where the queries became too numerous and slow. WebDSL provides a debug feature for logging all queries executed on a page render or in an action, which can pinpoint an issue quickly. Even with a detailed query that retrieves all required data at once, a page might also simply load too much data, which is a situation where all the discussed solutions would suffer in performance.

*Persistence Example* We will now look at a code fragment comparison between Links and WebDSL. This provides an indication of the abstraction differences provided by the languages. The fragment in Figure 8.11 is taken from the Citeseer data example of the Links demos [Links 2023]. First, the tables for authors and papers are declared. The join table to connect authors to papers is also explicitly declared. In the data retrieval query written in Links, the join requirements of `paperauthorTable` are explicitly indicated, matching `paperid` with the `id` field of `paper`, and matching `authorid` with the `id` field of `author`.

This example is a good illustration of the benefit of the ORM abstraction in WebDSL. Figure 8.12 shows an example of a similar data retrieval in WebDSL. An inverse relation can be used to automatically keep both collections synchronized, which would also be the behavior of a single join table. The code

```

var authorsTable = table "authors"
  with (id : Int, name : String) where id readonly from db;
var papersTable = table "papers"
  with (id : Int, title : String) where id readonly from db;
var paperauthorTable = table "paperauthor"
  with (paperid : Int, authorid : Int) from db;
sig getAuthors : (Paper) -> [Author]
fun getAuthors(paper) server {
  query {
    for (r <- paperauthorTable)
      where (paper.id == r.paperid)
        for (a <- authorsTable)
          where (r.authorid == a.id)
            [(id=a.id, name=a.name)]
  }
}

```

---

Figure 8.11 Data retrieval example in Links

```

1 entity Author {
2   name : String
3   papers : {Paper}
4 }
5 entity Paper {
6   name : String
7   authors : {Author} ( inverse = papers )
8 }
9 // function for equivalence with Links example, typically would just use paper.authors
10 function getAuthors( paper: Paper ): {Author} {
11   return paper.authors;
12 }

```

---

Figure 8.12 Equivalent data retrieval example in WebDSL

generator will also generate table mapping to a single join table in this case. The convenience this provides becomes clear when implementing the `getAuthors` function, which is actually not required because a simple `papers.authors` property access would retrieve the list of authors. Implicit (lazy) loading of additional referenced data enables a more natural style of writing application code. The application developer does not have to think about which data to load in advance. Any fetching of related data happens automatically through traversing the object graph.

*Form input handling and data update* In this next fragment from the Links demos, we will look at form handling and updating persisted data. This fragment is shown in Figure 8.13. Displaying data while escaping HTML tags is performed with the `stringToXml` function. This example uses a fixed name `paperTitle` for the input which is problematic for reuse and CSRF prevention. The action handler `updatePaper` is invoked and supplied with a record containing the paper id and the entered title. The `server` keyword marks the function to be executed on the server rather than the client, which is required for the database update. The update statement selects the relevant paper from the database and updates the title column.

Figure 8.14 shows an equivalent WebDSL fragment. It is easy to observe the benefit gained from the WebDSL abstractions in terms of lines of code. The application developer needs to take fewer decisions in implementing simple

```

typename Paper = (id:Int, title:String);
var papersTable = table "papers"
  with (id : Int, title : String) where id readonly from db;
mutual {
  fun updatePaper(paper) server {
    update (p <- papersTable)
      where (p.id == paper.id)
      set (title=paper.title);
    showPaperInfo(paper)
  }
  sig showPaperInfo : (Paper) ~> Page
  fun showPaperInfo(paper) server {
    page <html><body>
      <h1>{stringToXml(paper.title)}</h1>
      <form l:action="{updatePaper((id=paper.id, title=paperTitle))}"
        method="POST">
        <input type="text"
          class="input"
          l:name="paperTitle"
          value="{paper.title}"/>
        <button type="submit">update title</button>
      </form>
    </body></html>
  }
}

```

---

Figure 8.13 Form example in Links

```

1 entity Paper {
2   title : String
3 }
4 page showPaperInfo( paper: Paper ){
5   h1 { ~paper.title }
6   form {
7     input( paper.title )
8     submit { "update title" }
9   }
10 }

```

---

Figure 8.14 Equivalent form example in WebDSL

forms like this, which reduces the number of bugs that can be created. The pages in WebDSL automatically construct `html` and `body` tags with required includes. Automatic data binding takes care of displaying the current paper title in the input field, and updating the data model with the newly entered title on submit. Updating properties of persisted entities automatically get committed in a submit action, no additional commands are required in the action in this case.

### 8.3.3 Hop.js

Hop.js [Serrano and Prunet 2016; Serrano 2006] is a multitier programming environment for JavaScript. A single JavaScript program describes both client-side and server-side components. The HopScript language provided in Hop.js is a super set of JavaScript. The major additions it provides are HTML syntax support, integrated server-side web workers and native websockets. Hopscript is compiled to JavaScript for client-side execution, and compiled to a multi-threaded variant of the Scheme language for server-side execution. Hopscript is

a dynamically typed language and does not come with IDE support. In popular client-side libraries like React, GUI updates are automatically derived from changes in state. Hop.js does not support this style of client-side programming, instead GUI updates are programmed explicitly.

The design goals for Hop.js are quite different than WebDSL. Hop.js is intended for applications with few users, as the website [Serrano 2023] explains that users must be declared before executing a program, and the system only accepts requests from authenticated users. It is designed for light-weight applications like multimedia applications, ubiquitous computing, home automation, mashups, and office tools. It does not target full-stack web applications, and does not provide any specific support for database persistence, access control, and data validation. The security requirements are also much lighter, because only predefined trusted users can access the applications. Because of the dynamically typed nature of the language and lack of IDE, static verification of the application code and error reporting quality are not considered.

## 8.4 Modeling and Low-Code Tools

Modeling solutions have been applied to web programming problems for a long time, as seen in tools like WebML (Web Modeling Language) [Brambilla et al. 2008], UWE (UML-based Web Engineering) [Koch, Kraus, and Henicker 2001], OOHDM (Object-Oriented Hypermedia Design Model) [Schwabe and Rossi 1995], Object Oriented Web Solution (OOWS) [Pastor, Fons, and Pelechano 2003], and Hera [Houben et al. 2003]. The tools attempt to bring a more structured approach to the development process of web applications. Web programming concerns such as page composition and navigation are represented in a graphical model. A graphical editor is required to interact with the application definition. Some classes of errors are avoided through the graphical application editor, e.g. a link or reference can only be created as an arrow between existing boxes. Typically, the modeling languages target conventional programming languages, such as Java and C#, for their code generation. The focus in these methods is often on the high-level application components, generating only a partially complete application. Detailed behavior tweaking relies on extending generated code, or falling back on a scripting language. In those cases there will be a disconnect between the graphical models and the custom code, weakening the provided abstraction, and often not providing static verification.

There are several commercial platforms such as Mendix [2023] promoting low-code (or no-code) solutions, which conceptually are the same as modeling languages. Applications can be “clicked together” in a graphical environment, hosted in a locally installed editor or directly online in the cloud. The tools enable non-programmers to build relatively simple applications. Complex business logic is often delegated to extension components and requires hiring a programmer or consultant from the company behind the tool.

Comparing modeling and low-code tools to WebDSL, the main difference is in the graphical tools used and the perspective of enabling business experts

to provide (partial) implementation of applications. WebDSL is a textual programming language, and enables programmers to become more effective in creating complete web applications.

### 8.4.1 WebML

Brambilla and Fraternali [2014] report on the experience of the WebRatio Model-Driven Engineering tool. The tool employs transformation to applications from models expressed in the DSL called WebML (Web Modeling Language). This approach has been applied in many industrial projects, resulting in practical experience and lessons learned.

Applications are created in a graphical tool using high-level page components such as login, show all items, search items, item detail. High-level components have the benefit of allowing fast prototyping, and enabling non-programmers to contribute to the implementation. A notable feature of WebML is that there is no sub-model for business logic. This requires an extension mechanism in the DSL in the form of custom units, encoding user-defined business logic. The approach is based on multiple people or roles taking part in the development process. High-level models of the process are created by a business analyst, realistic prototypes are created by an application analyst, detailed web layout templates by a web designer, and custom business logic by a Java developer. A downside of these separated abstraction levels, is that they become their own source of potential inconsistencies. For fine-grained behavior the models can be extended with scripts written in Groovy [Apache 2023], or a new component can be programmed in Java. This happens quite soon, e.g. for writing custom operations on data and data validation rules. A WebRatio expert with knowledge of the underlying architecture is required to write essential parts of the web application, given that the scripts need to interface with the generated code and runtime. Changes to the higher-level model can invalidate custom written components, where the high-level designer is oblivious to such inconsistencies.

## 8.5 Conclusion

In this chapter, we have compared WebDSL to existing solutions and research, and identified key differences. Compared to conventional full-stack web programming frameworks, WebDSL provides improvements in the areas of abstraction, static verification, and security. The closest comparison can be made with tierless programming languages proposed in research, that provide a single language abstraction to generate code for all tiers. The comparisons in this chapter show that WebDSL abstractions are more concise and are able to hide more of the web programming accidental complexity (e.g. input identifiers and table mappings), while providing similar (e.g. automatic XSS protection), or better (e.g. automatic CSRF protection) security.





# Conclusion

---

In this concluding chapter we revisit the thesis in Section 9.1, and revisit the design principles in Section 9.2. Finally, we will discuss directions for future work in Section 9.3.



## 9.1 Thesis Revisited

**New web programming abstractions integrated in a domain-specific language improve web programming by avoiding boilerplate code, providing timely and accurate feedback on problems in application source code, and ensuring reliability (robustness, performance, scalability, and security) of applications. The design and implementation of such a language is feasible and has practical applicability.**

The thesis contains several topics that have been addressed in this work. The topics are new web programming abstractions, domain-specific language design, static verification and providing timely and accurate feedback, web application security, performance and robustness of the runtime, and finally feasibility and practical applicability.

New web programming abstractions have been designed in WebDSL that avoid boilerplate code and associated possibility errors and security flaws. Chapter 2 discusses the base components of WebDSL: persistence handling, data objects, functions on that data, and user interface components for rendering the data. Chapter 3 shows the design of the user interface abstraction, including the essential support for input of data and data validation. This is the most important abstraction in WebDSL, which differentiates it from other web programming solutions by using a notion of deterministic template identifiers supplied by the runtime system. Chapter 4 describes the design of the access control language and modeling of policies in WebDSL. This is another unique aspect of WebDSL, other web programming languages have little or no support for access control policy specification through semantics of the language itself.

The next topic is domain-specific language design. The WebDSL language is a large case study into domain-specific languages. Chapter 5 describes our process of incremental DSL design, which evolved the WebDSL compiler to its current state. Chapter 6 describes our compiler implementation strategy, where both compiler and IDE are developed in conjunction based on the same analysis implementation. The implementation combines rewriting and analysis in order to reuse language elements in the implementation of other higher level of abstraction language elements.

The third topic is static verification and providing timely and accurate feedback on problems in application specifications. Chapter 6 describes im-

plementation aspects for creating the analysis in WebDSL. By leveraging the Spoofox language workbench, checking application constraints is performed live on every edit action in the IDE. Because the WebDSL language has specifically designed syntax for web abstractions, errors can be connected to these elements, and describe problems in terms of the abstractions.

The fourth topic is web application security in order to ensure reliability. Security plays a role in many of the abstractions found in WebDSL, but in particular in the user interface component. Web application vulnerabilities such as Cross-Site Scripting and Cross-Site Request Forgeries are addressed in the abstractions provided by WebDSL. Chapter 7 describes and reflects on our experiences with running real-world applications. In practice we have seen that keeping applications secure is a continuous task, zero-day vulnerabilities in underlying libraries do occur even in conceptually harmless features such as logging, as seen with the Log4j vulnerability in 2021. Vulnerabilities also occur in other areas of the application stack, look for example at the Heartbleed bug in OpenSSL discovered in 2014.

The fifth topic is performance, scalability, and robustness of the runtime in order to ensure a reliable application. The majority of the work on this topic is contained in the source code of WebDSL. As described in Chapter 7, for many years we have improved these aspects while testing it with new real-world applications along the way. We have made robustness improvements by solving discovered bugs, and adding features that improve the behavior in exceptional situations. Application performance is good enough out of the box for medium-sized web information systems in the university and academia domain, in some cases it required manual tweaks for optimizing database access when loading many entities.

Finally, feasibility and practical applicability is part of all our work on WebDSL and web applications. In writing the compiler and runtime, managing builds with automatic regression testing, creating web applications with WebDSL, configuring web servers, operating the web applications, supporting the usage of the web applications, and the continuous improvement of all these aspects.

## 9.2 Design Principles Revisited

In the introduction we listed the 5 core design principles of WebDSL (Section 1.7), based on the problems we observed in web programming languages, and our experiences in designing and developing the WebDSL language and applications. In this section we revisit those design principles and relate them back to the topics discussed in this thesis.

1. *Linguistic abstractions should enable direct expression of intent.* Language concepts are designed with as much or little flexibility as required for the essential complexity. The syntax and semantics are tailored to exactly what they need to cover. Accidental complexity is removed, only essential complexity is expressed. In order to determine what the essential complexity is, the scope

of the language needs to be clear, meaning what type of applications are going to be covered by the language. If you design for everything, you end up with a general-purpose programming language, and cannot provide benefit for domain-specific contexts. The design decisions made for WebDSL to achieve better abstraction favor convenience for application developer over pure performance or more control over implementation details. Boilerplate code is generated or hidden in the runtime as much as possible. Design decisions have to be made up front about what is boilerplate code or accidental complexity. This allows providing many aspects for free, however, it takes away control over these aspects.

Boilerplate code is avoided in multiple web application concerns. Any persistence code related to database setup, table creation, and constructing queries is hidden. Primary key fields for entities and their foreign key references are implicit and automatically created. An important aspect of web programming for forms and actions, is the handling of input name identities. These need to be deterministic for detecting what the contents of input fields are when handling the submit request, however, they should also be unique to the user and related data to avoid Cross-Site Request Forgery problems. WebDSL hides this name generation aspect from application programmers entirely, which avoids many potential bugs, without specific static analysis. Data validation is completely integrated into the user interface abstraction, using a separate request handling phase that can conveniently access submitted data through the entity model as well as query against the database. In most programming languages, access control rules need to be inserted across the whole application. In WebDSL, the access control policy can be specified in rules separate from the rest of the application code. The boilerplate code for weaving in checks at all the right spots is avoided, avoiding many potential bugs from manually inserting these checks.

The lack of control makes WebDSL not an ideal tool for every web application. We have applied WebDSL in practice for several information systems with thousands of users, where it enabled us as to make real-world impact with a small team. The WebDSL tooling enables fast prototyping, which is essential for discovering the specific requirements for an application, and adjusting the application when these requirements inevitably change.

2. *Linguistic abstractions should ensure reliability and security.* Applications should keep working when deployed in a real setting. This means the runtime should ensure robustness, performance, scalability, and also security, protecting against malicious web technology exploits (e.g. Cross-Site Scripting or remote code evaluation). Security should not add boilerplate code, exploit countermeasures are enforced in the runtime without adding complexity to application code.

A general problem for web applications is the possibility and ease of tampering with client-side JavaScript code and state, or form submit request parameters. Updating entity data and function code always happens server-side in WebDSL. Form submit request parameters are checked to confirm that they select only options that are (still) available, which is implemented in the

input handling components of the standard library. Access control rules are checked for every server request, and data validation happens server-side. Tampering with JavaScript or form parameters cannot provide any additional unintended functionality.

Typical web security problems arise from injection attacks, where a user-defined value becomes code that is executed. In the case of SQL queries, injections are avoided in WebDSL by integrating queries in the language syntax and enforcing automatic escaping of values. The entity persistence abstraction that allows traversal of the object graph with implicit loading of data also avoids the need for direct querying in many cases. Cross-Site Scripting is avoided by automatically escaping values that are inserted in the page output, which is also enabled by having page rendering syntax integrated in the language. Cross-Site Request Forgery problems are avoided with automatically generated input name identifiers that are not predictable, as they are based on the data it concerns and the logged in user.

General robustness, performance, and scalability of WebDSL applications is partially gained from targeting a mature platform: the Java language, the Java Virtual Machine, and Java application servers. The Java ecosystem is large, and not all libraries and frameworks contribute to better robustness, performance, scalability, and security. Initially the WebDSL project also targeted an existing web programming framework, which turned out to be too limiting for further development of the language, as well as having problems in these areas. The recent Log4Shell vulnerability shows that even though a platform is considered mature, there is no guarantee that it is free from security problems. Operating web applications requires constant attention for security problems and updates.

In practice, applications often require integration with other code such as invoking web services, invoking libraries, and including JavaScript widgets. There is a trade-off in expressivity and security, security becomes harder to enforce when there is more expressivity. For example, having an escape to Java and JavaScript code in WebDSL has been essential for application development, however, it takes extra attention from the application developer to ensure security with these extensions.

A better programming language can do a lot to avoid security vulnerabilities in web programming. However, security cannot be completely ensured by the programming language runtime, it also relies on a large software stack underneath. Attackers can focus on any link in the connection chain between the client and the server. For example, attacks might happen on DNS servers, networking, operating system, proxy servers, and application servers. The application is only as secure as its weakest link.

3. *Static checking should present errors in terms of the domain.* WebDSL is designed from the ground up with static analysis and cross-language consistency checking in mind. The intent of code is precisely described by using abstractions for a limited domain. In a general-purpose programming language it can become very hard to detect patterns that are encodings of certain web application concerns. An example of this are navigation links, in many

solutions these would be based on a dynamic string value and harder to detect and check statically.

Part of developing abstractions for the WebDSL language, is thinking about static analysis. In a good abstraction, the remaining possible errors become small. However, even small mistakes can consume a lot of development time when they are discovered late. For example, unused declared variables are reported as warnings by WebDSL. Together with an error marker for an unknown variable usage, it becomes trivial to fix such an inconsistency immediately without searching. Regular template arguments cannot be assigned to, and immediately generate errors when trying to assign new values or passing them to an input template. In some cases, static analysis may actually not be the best solution, and a more dynamic runtime solution provides better abstraction. This is seen with the template identifier generation, where the abstraction in WebDSL can automate this task entirely.

Because of the explicit syntactic constructs for language concepts, semantic errors can be precise and messages in terms of the domain concepts. Error messages should be easy to follow and fix for application developers. These messages should not contain technical terms from the type system mechanics, and should provide a clear link to the violating language constructs.

The first version of the WebDSL compiler was developed as a command-line compiler in the Stratego language. The Spoofox language workbench is an evolution of the Stratego language to generate IDEs from language definitions. The WebDSL language is the largest language developed in the Spoofox language workbench. The WebDSL compiler and IDE share analysis code, the IDE can report errors immediately with error markers in the editor view. In addition to error markers, the IDE provides syntax highlighting, semantic code completion, and reference resolving.

*4. Extensibility should be explicit.* The WebDSL language achieves flexibility by providing several language features for customization, such as the option for global and local template overrides, and attribute set overrides. However, the DSL cannot cover everything an application might need. In practice we have seen several application-specific features that required integration with existing libraries. In these cases it is important to avoid that abstractions become leaky. Generated code could be modified to include integration with additional libraries, however, this causes an integration problem where explicit mechanisms are required to keep generated code and extensions synchronized. Additionally, the application code would not show any indication that an extension is used, which makes reasoning about the application more difficult. Instead of modifying generated code, extension with external components is done through explicit foreign function interfaces in the language, such as for invoking server-side Java or client-side JavaScript libraries. A clean interface is created for these extension components. Java code is typically invoked through a static function, or through a declaration of the Java class with properties and functions mapped to a WebDSL type. JavaScript components can usually be entirely encapsulated in a template definition.

There are several examples of extensions in our applications. An often used component in several of our applications is a client-side sortable and filterable table, which is a JavaScript widget wrapped in a WebDSL template definition. The MyStudyPlanning system imports data from an existing system for course descriptions. This importer is a small Java program developed separately from the WebDSL application. The importer is invoked through a global function, and the resulting data structure is mapped to the WebDSL type system. Single sign-on integration is essential for our university applications in order to allow students to log in and automatically retrieve their data. This involves configuration on the server, as well as calling into Java libraries to retrieve the data. The WebLab application enables online programming assessment, using an integration of the Ace [2023] and Monaco [2023] code editors.

*5. Lessons learned should be consolidated in the language.* Language and applications should co-evolve, reflecting experiences from requirements engineering and application development in the language design. General problems found and fixed in applications should become language improvements, so that other applications automatically reap the benefits. Instead of creating a library or framework, a language feature can be created with static analysis and full integration with other language semantics, to enable better reuse of gained knowledge. The WebDSL development process of discovering new abstractions, domain-specific language abstraction, and reimplementing using new core abstractions is described in Chapter 5.

Runtime fixes can be applied to get security issues fixed for all applications. For example, the Markdown rendering provided by the WikiText type in WebDSL is a commonly used component in our applications. This component has received several iterations to improve security and performance. An example feature added to the language based on application requirements is multitenancy support in Conf Researchr. To host multiple conferences on different domains, a mechanism was required to customize URL patterns and perform additional lookups before deciding the arguments to a page. An extension was created that provides control over the generation of navigation links, and the decoding of requests to pages with parameters.

### 9.3 Directions for Future Work

The scope of the WebDSL project covers many areas of web programming, which each provide directions for future work. In this section, we discuss future work topics where we have done initial exploration or work, in the areas of language design, compiler and runtime, compiler and IDE tooling, and application maintainance.

*Client-side code abstraction* A missing feature in the current WebDSL is an abstraction for client-side calculation and rendering. From a security perspective, this can also be considered a benefit, as the application developer does not have to worry about which application fragments are running in the client, and which are on the server. There is no risk of tampering with any of the

function code or calculation state, which are associated with code running on the client. The current WebDSL language provides sufficient coverage for most of the real-world application features. However, there are a few instances in our applications where we rely on client-side programming using the JavaScript extension. A good example of this is the client-side faceted filtering in the Conf Researchr conference programs. Initially this was handled server-side using the search features [Van Chastelet 2013]. Because the experience was not great with wireless at conferences often being unreliable, we decided to move this to the client [Van Chastelet 2020]. This client-side implementation is not ideal, as it needs to search the DOM and perform manual DOM manipulation. A feature like this would become easier to customize, maintain and optimize when client-side rendering is integrated into the language. Technologies for client-side rendering like React [2023] have become very mature, and would be a good target for code generation. In addition to client-side rendering, features for generating server-side APIs for data retrieval and updating need to be improved. This has been partially explored [Melman 2013], although in that work the client-side component was written in a separate language. Another feature to explore related to client-side calculations is WebSocket communication, which is heavily used in popular communication platforms like Slack, Mattermost, and Discord.

*Persistence performance* The current persistence abstraction has proved to be reliable and fast enough for practical applications. There is room for improvement, because the current implementation still heavily relies on the Hibernate ORM framework, instead of having a persistence library tailored to the WebDSL semantics and requirements. The default behavior is currently lazy loading of referenced entities and properties. Our investigation of prefetching strategies [Gersen 2013] provided practical speed improvements, however, there is still room for further improvement. Potential improvements are deriving detailed queries from application code (as seen in the Links related work in Section 8.3.2), to avoid unnecessary query counts and loading unused data. Such improvements should not impact the persistence abstraction negatively.

*Formal model of dynamic semantics* Based on collaboration with William Cook, we have made an initial attempt to describe the dynamic semantics of WebDSL in a formal model. Besides a clean description of the behavior of WebDSL programs, such a semantic description enables automated testing of the compiler. Semantically correct test programs can be generated, and the output of the model implementation can be checked against the actual compiler implementation. One problem we experienced is that the more details are included in this model of WebDSL, the more work it is to maintain. To reduce the amount of work required for developing and maintaining semantic models, there are several formalisms and tools which could be used, such as reduction semantics in PLT Redex [Felleisen, Findler, and Flatt 2009] and component-based semantics in CBS [Mosses 2019; Mosses 2021].

*Static semantics specification* The typesystem rules in WebDSL are currently described in Stratego code. This is an encoding of typesystem rules in a func-



tional programming language. There is room for improvement in readability and maintainability of this code by moving to higher level languages in specifying type systems. In particular, we have investigated a migration to specify the type system using scope graphs in the Statix language [De Krieger 2022].

*Incremental compilation* Improving performance and incrementality of the WebDSL compiler is an ongoing goal. Incrementality in compilation was partially explored in earlier work [Bruning 2013]. This resulted in code generation caching and IDE caching, which were vital to the practical usability of WebDSL. Later we extended this with a compile unit cache, which was a more language-specific caching of top-level definitions in WebDSL, such as pages and templates. This is still a relatively coarse-grained incrementalization, more fine-grained incrementalization could provide additional build-time performance. Pipelines for Interactive Environments (PIE) [Konat 2019] can provide an interesting opportunity for improving the general handling of incrementalization in the compiler and IDE processes.

*Deployment speed* The time between saving the application code and observing the results in the browser should be minimized as much as possible to provide the best development experience. The compiler is a Java application, which benefits from running a longer time, because the initial class loading causes overhead and the JVM performs optimizations on frequently executed code. We currently run the WebDSL compiler as a daemon using the Nailgun library [Lamb 2023] to get the benefits from keeping the JVM alive. The deployment of the web application in an application server can still take around 10 seconds, even if the incremental build of the application code was fast. This is also caused by the JVM initialization and class file loading overhead. There are solutions for performing hot reloading of classes in the JVM, such as JRebel [Perforce 2023] and Spring Loaded [2023]. Such a solution would have to be customized to the features of the WebDSL language. A custom solution for class reloading is also currently being used in the JVM-based LabBack back-end for WebLab for hot reloading student code.

# Bibliography

---

- Ace (2023). *Ace: the High Performance Code Editor for the Web*. <https://ace.c9.io/> (cited on pages 167, 214).
- Alex, B. (2008). *Acegi Security, Reference Documentation 1.0.7*. <https://web.archive.org/web/20110518155758/http://www.acegisecurity.org/guide/springsecurity.pdf>. Accessed: 2011-05-18 (cited on pages 95, 110, 116).
- Anderson, A. (2004). "XACML Profile for Role Based Access Control (RBAC)". In: *OASIS Access Control TC Committee Draft 1*, page 13 (cited on page 95).
- Angular (2023). *Angular: One framework. Mobile & desktop*. <https://angular.io/> (cited on pages 87, 189).
- Apache (2023). *Apache Groovy, a multi-faceted language for the Java platform*. <https://groovy-lang.org/> (cited on page 207).
- Bertino, E., Ferrari, E., and Atluri, V. (1999). "The specification and enforcement of authorization constraints in workflow management systems". In: *ACM Transactions on Information and System Security* 2.1, pages 65–104. ISSN: 1094-9224. DOI: 10.1145/300830.300837 (cited on page 119).
- Bloch, J. (2008). *Effective Java*. Addison-Wesley Professional (cited on page 37).
- Bootstrap (2023). *Build fast, responsive sites with Bootstrap*. <https://getbootstrap.com/> (cited on page 29).
- Brabrand, C., Møller, A., and Schwartzbach, M. I. (2001). "Static validation of dynamically generated HTML". In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE 01, Snowbird, Utah, USA, June 18-19, 2001*. ACM, pages 38–45. ISBN: 1-58113-413-4. DOI: 10.1145/379605.379657 (cited on page 8).
- Brambilla, M., Comai, S., Fraternali, P., and Matera, M. (2008). "Designing Web Applications with Webml and Webratio". In: *Web Engineering: Modelling and Implementing Web Applications*. Edited by G. Rossi, O. Pastor, D. Schwabe, and L. Olsina. Human-Computer Interaction Series. Springer, pages 221–261. ISBN: 978-1-84628-922-4. DOI: 10.1007/978-1-84628-923-1\_9 (cited on page 206).
- Brambilla, M. and Fraternali, P. (2014). "Large-scale Model-Driven Engineering of web user interaction: The WebML and WebRatio experience". In: *Science of Computer Programming* 89, pages 71–87. DOI: 10.1016/j.scico.2013.03.010 (cited on page 207).
- Bravenboer, M., Dolstra, E., and Visser, E. (2010). "Preventing injection attacks with syntax embeddings". In: *Science of Computer Programming* 75.7, pages 473–495. DOI: 10.1016/j.scico.2009.05.004 (cited on pages 12, 45, 114).
- Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2008). "Stratego/XT 0.17. A language and toolset for program transformation". In: *Science*

- of *Computer Programming* 72.1-2, pages 52–70. DOI: 10.1016/j.scico.2007.11.003 (cited on page 143).
- Bravenboer, M. and Visser, E. (2004). “Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions”. In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004*. Edited by J. M. Vlissides and D. C. Schmidt. Vancouver, BC, Canada: ACM, pages 365–383. ISBN: 1-58113-831-8. DOI: 10.1145/1028976.1029007 (cited on pages 114, 152).
- Bruning, N. (2013). *Separate Compilation as a Separate Concern: A Framework for Language-Independent Selective Recompilation*. Master’s thesis. <http://resolver.tudelft.nl/uuid:d2ebe038-fc1e-47b4-a708-e043e9d3ca74> (cited on pages 25, 161, 216).
- Buck, J. (2006). *Skinny Controller, Fat Model*. <https://web.archive.org/web/20220808063147/http://weblog.jamisbuck.org/2006/10/18/skinny-controller-fat-model>. Accessed: 2022-08-08 (cited on page 6).
- Burns, E. and Kitain, R., editors (2006). *JavaServer Faces Specification. Version 1.2*. Sun (cited on page 8).
- Cheney, J., Lindley, S., and Wadler, P. (2014). “Query shredding: efficient relational evaluation of queries over nested multisets”. In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. Edited by C. E. Dyreson, F. Li, and M. T. Özsu. ACM, pages 1027–1038. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2612186 (cited on page 202).
- Chlipala, A. (2015). “Ur/Web: A Simple Model for Programming the Web”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Edited by S. K. Rajamani and D. Walker. ACM, pages 153–165. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2677004 (cited on page 196).
- Chlipala, A. (2020). *The Ur/Web Manual*. <https://web.archive.org/web/20211225145747/https://www.impredicative.com/ur/manual.pdf>. Accessed: 2021-12-25 (cited on page 196).
- Chlipala, A. (2023). *Ur/Web Demo*. <http://impredicative.com/ur/demo/> (cited on pages 197–199).
- Chlipala, A. J. (2010). “Ur: statically-typed metaprogramming with type-level record computation”. In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. Edited by B. G. Zorn and A. Aiken. ACM, pages 122–133. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806612 (cited on pages 4, 8, 196).
- Conf Researchr (2014). *Conf Researchr*. <https://conf.researchr.org> (cited on pages 20, 163).
- Cooper, E. (2009). “The Script-Writer’s Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed”. In: *Database Programming Languages - DBPL 2009, 12th International Symposium, Lyon, France, August 24,*

2009. *Proceedings*. Edited by P. Gardner and F. Geerts. Volume 5708. Lecture Notes in Computer Science. Springer, pages 36–51. ISBN: 978-3-642-03792-4. DOI: 10.1007/978-3-642-03793-1\_3 (cited on page 203).
- Cooper, E., Lindley, S., Wadler, P., and Yallop, J. (2006). “Links: Web Programming Without Tiers”. In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*. Edited by F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever. Volume 4709. Lecture Notes in Computer Science. Springer, pages 266–296. ISBN: 978-3-540-74791-8. DOI: 10.1007/978-3-540-74792-5\_12 (cited on pages 4, 201).
- Cooper, E., Lindley, S., Wadler, P., and Yallop, J. (2008). “The Essence of Form Abstraction”. In: *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*. Edited by G. Ramalingam. Volume 5356. Lecture Notes in Computer Science. Springer, pages 205–220. ISBN: 978-3-540-89329-5. DOI: 10.1007/978-3-540-89330-1\_15 (cited on page 202).
- Crielaard, B., Bruin, C., and Aerts, T. (2017). *Native WebLab: Safe Execution of Native Code in WebLab*. Bachelor’s thesis. <http://resolver.tudelft.nl/uuid:f45d3e78-95a6-4c8b-9ca4-723680513f59> (cited on page 168).
- Damianou, N., Dulay, N., Lupu, E., and Sloman, M. (2001). “The Ponder Policy Specification Language”. In: *Policies for Distributed Systems and Networks: Int. Workshop, Policy 2001, Bristol, Uk, January 29-31, 2001: Proceedings* (cited on pages 95, 115).
- De Jonge, M. (2014). “Language-parametric Techniques for Language-Specific Editors”. PhD thesis. Delft University of Technology. DOI: 10.4233/uuid:5b485a4a-e502-42d9-8bd2-21c02226ed91 (cited on page 150).
- De Jonge, M., Kats, L. C. L., Visser, E., and Söderberg, E. (2012). “Natural and Flexible Error Recovery for Generated Modular Language Environments”. In: *ACM Transactions on Programming Languages and Systems* 34.4, page 15. DOI: 10.1145/2400676.2400678 (cited on page 150).
- De Krieger, M. M. (2022). *Modernizing the WebDSL Front-End: A Case Study in SDF<sub>3</sub> and Statix*. Master’s thesis. <http://resolver.tudelft.nl/uuid:564b8471-631f-4831-a049-58b187425aed> (cited on pages 26, 162, 183, 216).
- Django (2023). *Django: The web framework for perfectionists with deadlines*. <https://www.djangoproject.com/> (cited on pages 2, 13, 178, 189).
- Django Tutorial (2023). *Writing your first Django app*. <https://docs.djangoproject.com/en/4.1/intro/> (cited on page 190).
- Elm (2023). *elm: A delightful language for reliable web applications*. <https://elm-lang.org/> (cited on pages 87, 189).
- Emscripten (2023). *Emscripten is a complete compiler toolchain to WebAssembly, using LLVM, with a special focus on speed, size, and the Web platform*. <https://emscripten.org/> (cited on page 168).

- Erdweg, S. T. (2013). “Extensible Languages for Flexible and Principled Domain Abstraction”. PhD thesis. Philipps-Universität Marburg. doi: 10.17192/z2013.0280 (cited on page 137).
- EvaTool (2012). *EvaTool*. <https://evatool.tudelft.nl> (cited on pages 20, 163).
- Evered, M. and Bögeholz, S. (2004). “A case study in access control requirements for a Health Information System”. In: *ACSW Frontiers*. Dunedin, New Zealand: Australian Computer Society, Inc., pages 53–61 (cited on page 114).
- Felleisen, M., Findler, R., and Flatt, M. (2009). *Semantics Engineering with PLT Redex*. MIT Press (cited on page 215).
- Ferraiolo, D., Kuhn, D., and Chandramouli, R. (2003). *Role-based Access Control*. Artech House (cited on page 109).
- Fowler, S., Harding, S. D., Sharman, J. L., and Cheney, J. (2021). “Cross-tier Web Programming for Curated Databases: a Case Study”. In: *Int. J. Digit. Curation* 16.1, page 21. doi: 10.2218/ijdc.v16i1.735 (cited on pages 202–203).
- Gamma, E., Johnson, R., Helm, R., Johnson, R. E., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley (cited on page 47).
- Gersen, C. M. (2013). *ORM Optimization through Automatic Prefetching in WebDSL*. Master’s thesis. <http://resolver.tudelft.nl/uuid:597b318c-a1af-4fde-865f-4422f548336b> (cited on pages 25, 40, 177, 203, 215).
- Groenewegen, D. M. (2008). *Declarative Access Control for WebDSL*. Master’s thesis. <http://resolver.tudelft.nl/uuid:4d1844c8-89df-4787-b777-c742b4a27217> (cited on page 23).
- Groenewegen, D. M., Hemel, Z., Kats, L. C. L., and Visser, E. (2008a). “WebDSL: A Domain-Specific Language for Dynamic Web Applications”. In: *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*. Edited by N. Mielke and O. Zimmermann. (poster). Nashville, Tennessee, USA: ACM, pages 779–780. ISBN: 978-1-60558-220-7 (cited on page 232).
- Groenewegen, D. M., Hemel, Z., Kats, L. C. L., and Visser, E. (2008b). “When Frameworks Let You Down. Platform-Imposed Constraints on the Design and Evolution of Domain-Specific Languages”. In: *Proceedings of the 8th OOPSLA Workshop on Domain Specific Modelling (DSM’08)*. Edited by J. G. Gray, J. Sprinkle, J.-P. Tolvanen, and M. Rossi. Nashville, Tennessee, USA (cited on pages 24, 232).
- Groenewegen, D. M., Hemel, Z., and Visser, E. (2010). “Separation of Concerns and Linguistic Integration in WebDSL”. In: *IEEE Software* 27.5, pages 31–37. doi: 10.1109/MS.2010.92 (cited on pages 23, 232).
- Groenewegen, D. M., Van Chastelet, E., and Visser, E. (2020). “Evolution of the WebDSL runtime: reliability engineering of the WebDSL web programming language”. In: *Programming’20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23-26, 2020*. Edited by A. Aguiar, S. Chiba, and E. G. Boix. ACM, pages 77–83. ISBN: 978-1-4503-7507-8. doi: 10.1145/3397537.3397553 (cited on pages 23, 231).

- Groenewegen, D. M., Van Chastelet, E., De Krieger, M. M., and Pelsmaeker, D. A. A. (2023a). "Eating Your Own Dog Food: WebDSL Case Studies to Improve Academic Workflows". In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Edited by R. Lämmel, P. D. Mosses, and F. Steimann. Volume 109. Open Access Series in Informatics (OASICS). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 13:1–13:11. ISBN: 978-3-95977-267-9. DOI: 10.4230/OASICS.EVCS.2023.13 (cited on pages 23, 231).
- Groenewegen, D. M., Van Chastelet, E., De Krieger, M. M., Pelsmaeker, D. A. A., and Anslow, C. (2023b). "Conf Researchr: A Domain-Specific Content Management System for Managing Large Conference Websites". In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Edited by R. Lämmel, P. D. Mosses, and F. Steimann. Volume 109. Open Access Series in Informatics (OASICS). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 12:1–12:6. ISBN: 978-3-95977-267-9. DOI: 10.4230/OASICS.EVCS.2023.12 (cited on pages 23, 231).
- Groenewegen, D. M. and Visser, E. (2008). "Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns". In: *Proceedings of the Eighth International Conference on Web Engineering, ICWE 2008, 14-18 July 2008, Yorktown Heights, New York, USA*. Edited by D. Schwabe, F. Curbera, and P. Dantzig. IEEE, pages 175–188. ISBN: 978-0-7695-3261-5. DOI: 10.1109/ICWE.2008.15 (cited on pages 23, 163, 232).
- Groenewegen, D. M. and Visser, E. (2009a). "Integration of Data Validation and User Interface Concerns in a DSL for Web Applications". In: *Software Language Engineering, Second International Conference, SLE 2009*. Edited by M. G. J. van den Brand, D. Gasevic, and J. G. Gray. Volume 5969. Lecture Notes in Computer Science. Springer, pages 164–173. ISBN: 978-3-642-12106-7. DOI: 10.1007/978-3-642-12107-4\_13 (cited on pages 23, 232).
- Groenewegen, D. M. and Visser, E. (2009b). "Weaving web applications with WebDSL (demonstration)". In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 797–798. DOI: 10.1145/1639950.1640020 (cited on page 232).
- Groenewegen, D. M. and Visser, E. (2013). "Integration of data validation and user interface concerns in a DSL for web applications". In: *Software and Systems Modeling 12.1*, pages 35–52. DOI: 10.1007/s10270-010-0173-9 (cited on pages 23, 163, 231).
- GTmetrix (2023). *How fast does your website load? Find out with GTmetrix*. <https://gtmetrix.com/> (cited on page 184).
- H2 Database Engine (2023). *H2 Database Engine*. <https://www.h2database.com/html/main.html> (cited on page 40).
- Halfond, W., Viegas, J., and Orso, A. (2006). "A classification of SQL-injection attacks and countermeasures". In: *Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, USA*, pages 13–15 (cited on page 10).

- Halfond, W. G. J. and Orso, A. (2008). "Automated identification of parameter mismatches in web applications". In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*. Edited by M. J. Harrold and G. C. Murphy. ACM, pages 181–191. ISBN: 978-1-59593-995-1. DOI: 10.1145/1453101.1453126 (cited on page 8).
- Harkes, D. (2019). "Declarative Specification of Information System Data Models and Business Logic". PhD thesis. Delft University of Technology. DOI: 10.4233/uuid:5e9805ca-95d0-451e-a8f0-55decb26c94a (cited on pages 24, 177).
- Harkes, D., Groenewegen, D. M., and Visser, E. (2016). "IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs". In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. Edited by S. Krishnamurthi and B. S. Lerner. Volume 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. ISBN: 978-3-95977-014-9. DOI: 10.4230/LIPIcs.ECOOP.2016.11 (cited on pages 24, 177, 231).
- Harkes, D., Van Chastelet, E., and Visser, E. (2018). "Migrating business logic to an incremental computing DSL: a case study". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*. Edited by D. P. O'Keefe, T. Mayerhofer, and F. Steimann. ACM, pages 83–96. ISBN: 978-1-4503-6029-6. DOI: 10.1145/3276604.3276617 (cited on pages 24, 177).
- Hemel, Z. (2012). "Methods and Techniques for the Design and Implementation of Domain-Specific Languages". PhD thesis. Delft University of Technology. <http://resolver.tudelft.nl/uuid:c3ca8bef-ecda-4f71-9fda-bfc4bd353660> (cited on pages 2, 24).
- Hemel, Z., Groenewegen, D. M., Kats, L. C. L., and Visser, E. (2011). "Static consistency checking of web applications with WebDSL". In: *Journal of Symbolic Computation* 46.2, pages 150–182. DOI: 10.1016/j.jsc.2010.08.006 (cited on pages 2, 7, 23, 163, 232).
- Hemel, Z., Kats, L. C. L., Groenewegen, D. M., and Visser, E. (2010). "Code generation by model transformation: a case study in transformation modularity". In: *Software and Systems Modeling* 9.3, pages 375–402. DOI: 10.1007/s10270-009-0136-1 (cited on pages 24, 141, 163, 232).
- Hemel, Z., Verhaaf, R., and Visser, E. (2008). "WebWorkFlow: An Object-Oriented Workflow Modeling Language for Web Applications". In: *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*. Edited by K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter. Volume 5301. Lecture Notes in Computer Science. Springer, pages 113–127. ISBN: 978-3-540-87874-2. DOI: 10.1007/978-3-540-87875-9\_8 (cited on page 119).
- Hemel, Z. and Visser, E. (2011). "Mobl: the new language of the mobile web". In: *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. Edited by C. V.

- Lopes and K. Fisher. ACM, pages 23–24. ISBN: 978-1-4503-0942-4. DOI: 10.1145/2048147.2048159 (cited on page 25).
- Hibernate ORM (2023). *Hibernate ORM: Your relational data. Objectively*. <https://hibernate.org/orm/> (cited on pages 4–5, 36, 129, 143, 177).
- Hibernate Search (2023). *Hibernate Search: ORM. Lucene. Elasticsearch. Integrated*. <https://hibernate.org/search/> (cited on pages 123, 129).
- Hoare, T. (2009). “Null references: The billion dollar mistake”. In: *Presentation at QCon London 298*, page 88 (cited on page 47).
- Houben, G.-J., Barna, P., Frasinca, F., and Vdovjak, R. (2003). “Hera: Development of Semantic Web Information Systems”. In: *Web Engineering, International Conference, ICWE 2003, Oviedo, Spain, July 14–18, 2003, Proceedings*. Edited by J. M. C. Lovelle, B. M. G. Rodríguez, L. J. Aguilar, J. E. L. Gayo, and M. del Puerto Paule Ruíz. Volume 2722. Lecture Notes in Computer Science. Springer, pages 529–538. ISBN: 3-540-40522-4 (cited on page 206).
- JavaEE (2023). *The Java EE Tutorial: Expression Language*. <https://javaee.github.io/tutorial/jsf-el.html> (cited on page 4).
- Johnson, R. et al. (2005). *Professional Java Development with the Spring Framework*. Wrox Press Birmingham, UK (cited on page 95).
- Kats, L. C. L. (2011). “Building Blocks for Language Workbenches”. PhD thesis. Delft University of Technology. <http://resolver.tudelft.nl/uuid:c3b17264-a7ed-4f6d-aca7-88c34f2f6958> (cited on pages 24, 149).
- Kats, L. C. L. and Visser, E. (2010). “The Spoofox language workbench: rules for declarative specification of languages and IDEs”. In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Edited by W. R. Cook, S. Clarke, and M. C. Rinard. Reno/Tahoe, Nevada: ACM, pages 444–463. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869497 (cited on pages 143, 149).
- Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). “An overview of AspectJ”. In: *Lecture Notes in Computer Science 2072.327-355*, pages 110–121 (cited on pages 96, 101).
- Koch, N., Kraus, A., and Hennicker, R. (2001). “The Authoring Process of the UML-based Web Engineering Approach”. In: *First International Workshop on Web-Oriented Software Technology* (cited on page 206).
- Konat, G. (2019). “Language-Parametric Methods for Developing Interactive Programming Systems”. PhD thesis. Delft University of Technology. DOI: 10.4233/uuid:03d70c5d-596d-4c8c-92da-0398dd8221cb (cited on page 216).
- Lamb, M. (2023). *nailgun*. <https://github.com/facebook/nailgun> (cited on page 216).
- Laravel (2023). *Laravel: The PHP Framework for Web Artisans*. <https://laravel.com/> (cited on pages 2, 189).
- Latif, U. (2005). “A Generalized Temporal Role-Based Access Control Model”. In: *IEEE Transactions on Knowledge and Data Engineering 17.1*, pages 4–23 (cited on page 119).



- Let's Encrypt (2023). *Let's Encrypt*. <https://letsencrypt.org/> (cited on page 171).
- Links (2023). *Links: Linking Theory to Practice for the Web*. <https://links-lang.org/> (cited on page 203).
- Livshits, V. B. and Lam, M. S. (2005). "Finding Security Vulnerabilities in Java Applications with Static Analysis". In: *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. Edited by P. McDaniel. USENIX Association (cited on page 11).
- Lopes, G. (2014). *Bug 56684 - java7: java.net.SocketTimeoutException: Accept timed out*. [https://web.archive.org/web/20211119211541/https://bz.apache.org/bugzilla/show\\_bug.cgi?id=56684](https://web.archive.org/web/20211119211541/https://bz.apache.org/bugzilla/show_bug.cgi?id=56684). Accessed: 2021-11-19 (cited on page 175).
- Lucene (2023). *Apache Lucene*. <https://lucene.apache.org/> (cited on pages 123, 129).
- MathJax (2023). *MathJax, beautiful and accessible math in all browsers*. <https://www.mathjax.org/> (cited on page 178).
- Meijer, E., Beckman, B., and Bierman, G. M. (2006). "LINQ: reconciling object, relations and XML in the .NET framework". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. Edited by S. Chaudhuri, V. Hristidis, and N. Polyzotis. ACM, page 706. ISBN: 1-59593-256-9. DOI: 10.1145/1142473.1142552 (cited on page 201).
- Melman, C. (2013). *A Generative Approach for Data Synchronization between Web and Mobile Applications*. Master's thesis. <http://resolver.tudelft.nl/uuid:e3b70e0e-ef65-4b17-9e28-2c29a1e40972> (cited on pages 25, 215).
- Mendix (2023). *Mendix: Low-code. High impact*. <https://www.mendix.com/> (cited on page 206).
- Microsoft (2023a). *Language Server Protocol*. <https://microsoft.github.io/language-server-protocol/> (cited on page 183).
- Microsoft (2023b). *Visual Studio Code*. <https://code.visualstudio.com/> (cited on page 183).
- Mikkonen, T. and Taivala, A. (2007). *Web Applications: Spaghetti Code for the 21st Century*. Technical report TR-2007-166. Sun Microsystems (cited on page 114).
- Monaco (2023). *Monaco*. <https://microsoft.github.io/monaco-editor/> (cited on pages 167, 214).
- Moses, T. et al. (2005). "eXtensible Access Control Markup Language (XACML) Version 2.0". In: *OASIS Standard 200502* (cited on pages 95, 115).
- Mosses, P. D. (2019). "Software meta-language engineering and CBS". In: *Journal of Computer Languages* 50, pages 39–48. DOI: 10.1016/j.jvlc.2018.11.003 (cited on page 215).
- Mosses, P. D. (2021). "Fundamental Constructs in Programming Languages". In: *Leveraging Applications of Formal Methods, Verification and Validation - 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 17-29, 2021, Proceedings*. Edited by T.

- Margarita and B. Steffen. Volume 13036. *Lecture Notes in Computer Science*. Springer, pages 296–321. ISBN: 978-3-030-89159-6. DOI: 10.1007/978-3-030-89159-6\_19 (cited on page 215).
- MyStudyPlanning (2016). *MyStudyPlanning*. <https://mystudyplanning.tudelft.nl> (cited on pages 21, 163).
- National Vulnerability Database (2013). *CVE-2013-0156 Detail*. <https://nvd.nist.gov/vuln/detail/CVE-2013-0156> (cited on page 14).
- National Vulnerability Database (2021). *CVE-2021-44228 Detail*. <https://nvd.nist.gov/vuln/detail/CVE-2021-44228> (cited on page 179).
- Node.js (2023). *Node.js is an open-source, cross-platform JavaScript runtime environment*. <https://nodejs.org/> (cited on page 4).
- Open Web Application Security Project (2017). *OWASP Top Ten 2017*. <https://web.archive.org/web/20211215030551/https://owasp.org/www-project-top-ten/2017/>. Accessed: 2021-12-15 (cited on page 9).
- OWASP (2022). *Cross Site Request Forgery (CSRF)*. <https://web.archive.org/web/20221205130427/https://owasp.org/www-community/attacks/csrf>. Accessed: 2022-12-05 (cited on page 13).
- Park, J. and Sandhu, R. (2004). “The UCON ABC Usage Control Model”. In: *ACM Transactions on Information and System Security* 7.1, pages 128–174 (cited on page 119).
- Pastor, O., Fons, J., and Pelechano, V. (2003). “OOWS: A method to develop web applications from web-oriented conceptual models”. In: *Web Oriented Software Technology (IWWOST’03)*, pages 65–70 (cited on page 206).
- Perforce (2023). *JRebel*. <https://www.jrebel.com/products/jrebel> (cited on pages 161, 216).
- Racket (2023). *Racket*. <https://racket-lang.org/> (cited on page 137).
- Rails Guides (2022). *Rails Guides: Active Record Associations*. [https://web.archive.org/web/20220118080032/https://guides.rubyonrails.org/association\\_basics.html](https://web.archive.org/web/20220118080032/https://guides.rubyonrails.org/association_basics.html). Accessed: 2022-01-18 (cited on page 36).
- React (2023). *React: A JavaScript library for building user interfaces*. <https://reactjs.org/> (cited on pages 87, 184, 189, 215).
- Reenskaug, T. (2003). *The Model-View-Controller (MVC) Its Past and Present*. [https://web.archive.org/web/20200914164659/https://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC\\_pattern.pdf](https://web.archive.org/web/20200914164659/https://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf). Accessed: 2020-07-28 (cited on page 6).
- Reposerch (2011). *Reposerch*. <https://codefinder.org/> (cited on page 63).
- Ruby on Rails (2023). *Ruby on Rails: Compress the complexity of modern web apps*. <https://rubyonrails.org/> (cited on pages 2, 36, 130, 189).
- Sails (2023). *Sails: The MVC framework for Node.js*. <https://sailsjs.com/> (cited on pages 2, 189).
- Samarati, P. and di Vimercati, S. D. C. (2001). “Access Control: Policies, Models, and Mechanisms”. In: *Foundations of Security Analysis and Design on Foundations of Security Analysis and Design (FOSAD’00)*. London, UK: Springer-Verlag, pages 137–196. ISBN: 3-540-42896-8 (cited on pages 95, 106–107, 109).

- Sandhu, R., Ferraiolo, D., and Kuhn, R. (2000). "The NIST model for role-based access control: towards a unified standard". In: *Proceedings of the fifth ACM workshop on Role-based access control*, pages 47–63 (cited on pages 95, 109).
- Sandhu, R. S. (1998). "Role-Based Access Control". In: *Advances in Computers*. Edited by M. Zerkowitz. Volume 48. Academic Press (cited on pages 95, 109).
- Sandhu, R. S. (1993). "Lattice-Based Access Control Models". In: *Computer* 26.11, pages 9–19. ISSN: 0018-9162. DOI: 10.1109/2.241422 (cited on page 106).
- Sandhu, R. and Samarati, P. (1994). "Access control: principle and practice". In: *Comm. Magazine, IEEE* 32.9, pages 40–48. ISSN: 0163-6804. DOI: 10.1109/35.312842 (cited on pages 95, 107–109).
- Schwabe, D. and Rossi, G. (1995). "The Object-Oriented Hypermedia Design Model". In: *Communications of the ACM* 38.8, pages 45–46 (cited on page 206).
- Selenium (2023). *Selenium automates browsers*. <https://www.selenium.dev/> (cited on page 184).
- Serrano, M. (2006). *Hop, multitier Web Programming* (cited on page 205).
- Serrano, M. (2007). "Programming web multimedia applications with hop". In: *Proceedings of the 15th International Conference on Multimedia 2007, Augsburg, Germany, September 24-29, 2007*. Edited by R. Lienhart, A. R. Prasad, A. Hanjalic, S. Choi, B. P. Bailey, and N. Sebe. ACM, pages 1001–1004. ISBN: 978-1-59593-702-5. DOI: 10.1145/1291233.1291450 (cited on page 4).
- Serrano, M. (2023). *Hop.js*. <http://hop.inria.fr/home/index.html> (cited on page 206).
- Serrano, M. and Prunet, V. (2016). "A glimpse of Hopjs". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Edited by J. Garrigue, G. Keller, and E. Sumii. ACM, pages 180–192. ISBN: 978-1-4503-4219-3. DOI: 10.1145/2951913.2951916 (cited on page 205).
- Shneiderman, B., Byrd, D., and Croft, W. B. (1997). "Clarifying Search - A User-Interface Framework for Text Searches". In: *D-Lib Magazine* 3.1, pages 1–15. DOI: 10.1045/january97-shneiderman (cited on page 136).
- Spolsky, J. (2002). *The Law of Leaky Abstractions*. <https://web.archive.org/web/20220103161223/https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/>. Accessed: 2022-01-03 (cited on page 3).
- Spring (2023). *Spring: Build the apps that make the world run*. <https://spring.io/> (cited on pages 2, 189).
- Spring Loaded (2023). *Spring-Loaded*. <https://github.com/spring-projects/spring-loaded> (cited on page 216).
- Sun, C. (2010). *Operational Transformation Frequently Asked Questions and Answers*. <https://web.archive.org/web/20210606202842/https://www3.ntu.edu.sg/scse/staff/czsun/projects/otfaq/>. Accessed: 2021-06-06 (cited on page 77).
- Synopsys (2020). *The Heartbleed Bug*. <https://heartbleed.com/> (cited on pages 65, 179).

- Tilro, J. (2023). *Comparing Static Semantics Specifications for the IceDust DSL: A Case Study of Statix*. Master's thesis. <http://resolver.tudelft.nl/uuid:9a4875c0-6af7-49c4-b050-6c855b40857c> (cited on page 24).
- Tschantz, M. C. and Krishnamurthi, S. (2006). "Towards reasonability properties for access-control policy languages". In: *Proceedings of the eleventh ACM symposium on Access control models and technologies*. Lake Tahoe, California, USA: ACM, pages 160–169. ISBN: 1-59593-353-0. DOI: 10.1145/1133058.1133081 (cited on page 115).
- Van Antwerpen, H., Néron, P., Tolmach, A. P., Visser, E., and Wachsmuth, G. (2016). "A constraint language for static semantic analysis based on scope graphs". In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Edited by M. Erwig and T. Rompf. ACM, pages 49–60. ISBN: 978-1-4503-4097-7. DOI: 10.1145/2847538.2847543 (cited on page 25).
- Van Antwerpen, H., Poulsen, C. B., Rouvoet, A., and Visser, E. (2018). "Scopes as types". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA. DOI: 10.1145/3276484 (cited on pages 25, 162).
- Van Casteren, S. (2019). *Using Ur/Web: Pro's and Con's*. <https://web.archive.org/web/20211225143835/https://frigoeu.github.io/urweb1.html>. Accessed: 2021-12-25 (cited on page 201).
- Van Chastelet, E. (2020). *Client-side faceted filtering*. <https://github.com/webdsl/client-side-faceted-filtering> (cited on page 215).
- Van Chastelet, E. (2013). *A Domain-Specific Language for Internal Site Search*. Master's thesis. <http://resolver.tudelft.nl/uuid:61f7f022-60a4-4935-af13-6b5438d89c04> (cited on pages 25, 63, 137, 215).
- Van der Lippe, T., Smith, T., Pelsmaeker, D. A. A., and Visser, E. (2016). "A scalable infrastructure for teaching concepts of programming languages in Scala with WebLab: an experience report". In: *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*. Edited by A. Biboudis, M. Jonnalagedda, S. Stucki, and V. Ureche. ACM, pages 65–74. ISBN: 978-1-4503-4648-1. DOI: 10.1145/2998392.2998402 (cited on page 167).
- Vergu, V. A. (2012). *LabBack: An extendible platform for secure and robust in-the-cloud automatic assessment of student programs*. Master's thesis. <http://resolver.tudelft.nl/uuid:8c683733-546a-4fd2-8303-a2cf2edf3cd8> (cited on page 168).
- Vermolen, S. (2012). "Software Language Evolution". PhD thesis. Delft University of Technology. DOI: 10.4233/uuid:93988a21-5be3-4181-b471-b5a941a3641b (cited on pages 24, 41, 176).
- Visser, E. (1997). "Syntax Definition for Language Prototyping". PhD thesis. University of Amsterdam (cited on page ix).
- Visser, E. (2002). "Meta-programming with Concrete Object Syntax". In: *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*. Edited by D. S. Batory, C. Consel, and W. Taha. Volume 2487. Lecture Notes

- in Computer Science. Springer, pages 299–315. ISBN: 3-540-44284-7. DOI: 10.1007/3-540-45821-2\_19 (cited on page 152).
- Visser, E. (2004). “Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9”. In: *Domain-Specific Program Generation*. Edited by C. Lengauer et al. Volume 3016. LNCS. Springer-Verlag, pages 216–238 (cited on page 114).
- Visser, E. (2007). “WebDSL: A Case Study in Domain-Specific Language Engineering”. In: *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007*. Edited by R. Lämmel, J. Visser, and J. Saraiva. Volume 5235. Lecture Notes in Computer Science. Braga, Portugal: Springer, pages 291–373. ISBN: 978-3-540-88642-6. DOI: 10.1007/978-3-540-88643-3\_7 (cited on pages 27, 163).
- Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C., Visser, E., and Wachsmuth, G. (2013). “DSL engineering-designing, implementing and using domain-specific languages”. In: (cited on page 121).
- Wachsmuth, G., Konat, G., Vergu, V. A., Groenewegen, D. M., and Visser, E. (2013). “A Language Independent Task Engine for Incremental Name and Type Analysis”. In: *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*. Edited by M. Erwig, R. F. Paige, and E. Van Wyk. Volume 8225. Lecture Notes in Computer Science. Springer, pages 260–280. ISBN: 978-3-319-02653-4. DOI: 10.1007/978-3-319-02654-1\_15 (cited on pages 24, 162, 231).
- WebLab (2012). *WebLab*. <https://weblab.tudelft.nl> (cited on pages 20, 163).
- Weststrate, M. (2009). *Abstractions for Asynchronous User Interfaces in Web Applications*. Master’s thesis. <http://resolver.tudelft.nl/uuid:355f297a-bc22-445e-a489-934582d1d1d2> (cited on page 25).
- Weststrate, M. (2023a). *Immer*. <https://immerjs.github.io/immer/> (cited on page 25).
- Weststrate, M. (2023b). *MobX: Simple, scalable state management*. <https://mobx.js.org/> (cited on page 25).
- Wiedermann, B., Ibrahim, A., and Cook, W. R. (2008). “Interprocedural query extraction for transparent persistence”. In: *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*. Edited by G. E. Harris. ACM, pages 19–36. ISBN: 978-1-60558-215-3. DOI: 10.1145/1449764.1449767 (cited on page 25).
- Wieringa, R. and Morali, A. (2012). “Technical Action Research as a Validation Method in Information Systems Design Science”. In: *Design Science Research in Information Systems. Advances in Theory and Practice*. Edited by K. Peffers, M. Rothenberger, and B. Kuechler. Berlin, Heidelberg: Springer Berlin Heidelberg, pages 220–238. ISBN: 978-3-642-29863-9 (cited on page 17).
- YAML (2023). *YAML: YAML Ain’t Markup Language*. <https://yaml.org/> (cited on page 14).

- Yuan, M. and Heute, T. (2007). *JBoss Seam: Simplicity and Power Beyond Java EE*. Prentice Hall PTR Upper Saddle River, NJ, USA (cited on pages 95, 110, 116).
- Zhang, L., Ahn, G. J., and Chu, B. T. (2003). "A rule-based framework for role-based delegation and revocation". In: *ACM Transactions Information and System Security* 6.3, pages 404–441 (cited on page 119).
- Zhang, X., Oh, S., and Sandhu, R. (2003). "PBDM: a flexible delegation model in RBAC". In: *Proceedings of the eighth ACM symposium on Access control models and technologies*, pages 149–157 (cited on page 119).
- Zwaan, A., Van Antwerpen, H., and Visser, E. (2022). "Incremental type-checking for free: using scope graphs to derive incremental type-checkers". In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2, pages 424–448. DOI: 10.1145/3563303 (cited on page 162).



# Curriculum Vitae

---

Danny Maria Groenewegen

Born March 23rd 1984 in Nootdorp, the Netherlands

## **2012 - present**

Research Software Engineer

*Delft University of Technology*

Programming Languages research group

Computer Science & Engineering Teaching Team

## **2008 - 2023**

Ph.D. in Computer Science

*Delft University of Technology*

Programming Languages research group

## **2005 - 2008**

M.Sc. in Computer Science

*Delft University of Technology*

Specialization: Software Engineering

## **2002 - 2005**

B.Sc. in Computer Science

*Delft University of Technology*

Bachelor Technische Informatica

## **1996 - 2002**

Gymnasium diploma (*cum laude*)

*Sint-Maartenscollege* in Voorburg

Nature & Technology Profile





# List of Publications

---

Groenewegen, D. M., Van Chastelet, E., De Krieger, M. M., and Pelsmaeker, D. A. A. [2023a]. “Eating Your Own Dog Food: WebDSL Case Studies to Improve Academic Workflows”. In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Edited by R. Lämmel, P. D. Mosses, and F. Steimann. Volume 109. Open Access Series in Informatics (OASICs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 13:1–13:11. ISBN: 978-3-95977-267-9. DOI: 10.4230/OASICs.EVCS.2023.13

Groenewegen, D. M., Van Chastelet, E., De Krieger, M. M., Pelsmaeker, D. A. A., and Anslow, C. [2023b]. “Conf Research: A Domain-Specific Content Management System for Managing Large Conference Websites”. In: *Eelco Visser Commemorative Symposium (EVCS 2023)*. Edited by R. Lämmel, P. D. Mosses, and F. Steimann. Volume 109. Open Access Series in Informatics (OASICs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 12:1–12:6. ISBN: 978-3-95977-267-9. DOI: 10.4230/OASICs.EVCS.2023.12

Groenewegen, D. M., Van Chastelet, E., and Visser, E. [2020]. “Evolution of the WebDSL runtime: reliability engineering of the WebDSL web programming language”. In: *Programming’20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23-26, 2020*. Edited by A. Aguiar, S. Chiba, and E. G. Boix. ACM, pages 77–83. ISBN: 978-1-4503-7507-8. DOI: 10.1145/3397537.3397553

Harkes, D., Groenewegen, D. M., and Visser, E. [2016]. “IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs”. In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. Edited by S. Krishnamurthi and B. S. Lerner. Volume 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. ISBN: 978-3-95977-014-9. DOI: 10.4230/LIPIcs.ECOOP.2016.11

Groenewegen, D. M. and Visser, E. [2013]. “Integration of data validation and user interface concerns in a DSL for web applications”. In: *Software and Systems Modeling* 12.1, pages 35–52. DOI: 10.1007/s10270-010-0173-9

Wachsmuth, G., Konat, G., Vergu, V. A., Groenewegen, D. M., and Visser, E. [2013]. “A Language Independent Task Engine for Incremental Name and Type Analysis”. In: *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*. Edited by M. Erwig, R. F. Paige, and E. Van Wyk. Volume 8225. Lecture Notes in Computer Science. Springer, pages 260–280. ISBN: 978-3-319-02653-4. DOI: 10.1007/978-3-319-02654-1\_15

Hemel, Z., Groenewegen, D. M., Kats, L. C. L., and Visser, E. [2011]. “Static consistency checking of web applications with WebDSL”. in: *Journal of Symbolic Computation* 46.2, pages 150–182. DOI: 10.1016/j.jsc.2010.08.006

Groenewegen, D. M., Hemel, Z., and Visser, E. [2010]. “Separation of Concerns and Linguistic Integration in WebDSL”. in: *IEEE Software* 27.5, pages 31–37. DOI: 10.1109/MS.2010.92

Hemel, Z., Kats, L. C. L., Groenewegen, D. M., and Visser, E. [2010]. “Code generation by model transformation: a case study in transformation modularity”. In: *Software and Systems Modeling* 9.3, pages 375–402. DOI: 10.1007/s10270-009-0136-1

Groenewegen, D. M. and Visser, E. [2009a]. “Integration of Data Validation and User Interface Concerns in a DSL for Web Applications”. In: *Software Language Engineering, Second International Conference, SLE 2009*. Edited by M. G. J. van den Brand, D. Gasevic, and J. G. Gray. Volume 5969. Lecture Notes in Computer Science. Springer, pages 164–173. ISBN: 978-3-642-12106-7. DOI: 10.1007/978-3-642-12107-4\_13

Groenewegen, D. M. and Visser, E. [2009b]. “Weaving web applications with WebDSL (demonstration)”. In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 797–798. DOI: 10.1145/1639950.1640020

Groenewegen, D. M., Hemel, Z., Kats, L. C. L., and Visser, E. [2008b]. “When Frameworks Let You Down. Platform-Imposed Constraints on the Design and Evolution of Domain-Specific Languages”. In: *Proceedings of the 8th OOPSLA Workshop on Domain Specific Modelling (DSM’08)*. Edited by J. G. Gray, J. Sprinkle, J.-P. Tolvanen, and M. Rossi. Nashville, Tennessee, USA

Groenewegen, D. M., Hemel, Z., Kats, L. C. L., and Visser, E. [2008a]. “WebDSL: A Domain-Specific Language for Dynamic Web Applications”. In: *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*. Edited by N. Mielke and O. Zimmermann. (poster). Nashville, Tennessee, USA: ACM, pages 779–780. ISBN: 978-1-60558-220-7

Groenewegen, D. M. and Visser, E. [2008]. “Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns”. In: *Proceedings of the Eighth International Conference on Web Engineering, ICWE 2008, 14-18 July 2008, Yorktown Heights, New York, USA*. edited by D. Schwabe, F. Curbera, and P. Dantzig. IEEE, pages 175–188. ISBN: 978-0-7695-3261-5. DOI: 10.1109/ICWE.2008.15

## Titles in the IPA Dissertation Series since 2020

---

- M.A. Cano Grijalba.** *Session-Based Concurrency: Between Operational and Declarative Views.* Faculty of Science and Engineering, RUG. 2020-01
- T.C. Nägele.** *CoHLA: Rapid Co-simulation Construction.* Faculty of Science, Mathematics and Computer Science, RU. 2020-02
- R.A. van Rozen.** *Languages of Games and Play: Automating Game Design & Enabling Live Programming.* Faculty of Science, UvA. 2020-03
- B. Changizi.** *Constraint-Based Analysis of Business Process Models.* Faculty of Mathematics and Natural Sciences, UL. 2020-04
- N. Naus.** *Assisting End Users in Workflow Systems.* Faculty of Science, UU. 2020-05
- J.J.H.M. Wulms.** *Stability of Geometric Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2020-06
- T.S. Neele.** *Reductions for Parity Games and Model Checking.* Faculty of Mathematics and Computer Science, TU/e. 2020-07
- P. van den Bos.** *Coverage and Games in Model-Based Testing.* Faculty of Science, RU. 2020-08
- M.F.M. Sondag.** *Algorithms for Coherent Rectangular Visualizations.* Faculty of Mathematics and Computer Science, TU/e. 2020-09
- D. Frumin.** *Concurrent Separation Logics for Safety, Refinement, and Security.* Faculty of Science, Mathematics and Computer Science, RU. 2021-01
- A. Bentkamp.** *Superposition for Higher-Order Logic.* Faculty of Sciences, Department of Computer Science, VU. 2021-02
- P. Derakhshanfar.** *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03
- K. Aslam.** *Deriving Behavioral Specifications of Industrial Software Components.* Faculty of Mathematics and Computer Science, TU/e. 2021-04
- W. Silva Torres.** *Supporting Multi-Domain Model Management.* Faculty of Mathematics and Computer Science, TU/e. 2021-05
- A. Fedotov.** *Verification Techniques for xMAS.* Faculty of Mathematics and Computer Science, TU/e. 2022-01
- M.O. Mahmoud.** *GPU Enabled Automated Reasoning.* Faculty of Mathematics and Computer Science, TU/e. 2022-02
- M. Safari.** *Correct Optimized GPU Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03
- M. Verano Merino.** *Engineering Language-Parametric End-User Programming Environments for DSLs.* Faculty of Mathematics and Computer Science, TU/e. 2022-04

- G.F.C. Dupont.** *Network Security Monitoring in Environments where Digital and Physical Safety are Critical.* Faculty of Mathematics and Computer Science, TU/e. 2022-05
- T.M. Soethout.** *Banking on Domain Knowledge for Faster Transactions.* Faculty of Mathematics and Computer Science, TU/e. 2022-06
- P. Vukmirović.** *Implementation of Higher-Order Superposition.* Faculty of Sciences, Department of Computer Science, VU. 2022-07
- J. Wagemaker.** *Extensions of (Concurrent) Kleene Algebra.* Faculty of Science, Mathematics and Computer Science, RU. 2022-08
- R. Janssen.** *Refinement and Partiality for Model-Based Testing.* Faculty of Science, Mathematics and Computer Science, RU. 2022-09
- M. Laveaux.** *Accelerated Verification of Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2022-10
- S. Kochanthara.** *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving.* Faculty of Mathematics and Computer Science, TU/e. 2023-01
- L.M. Ochoa Venegas.** *Break the Code? Breaking Changes and Their Impact on Software Evolution.* Faculty of Mathematics and Computer Science, TU/e. 2023-02
- N. Yang.** *Logs and models in engineering complex embedded production software systems.* Faculty of Mathematics and Computer Science, TU/e. 2023-03
- J. Cao.** *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN.* Faculty of Mathematics and Computer Science, TU/e. 2023-04
- K. Dokter.** *Scheduled Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2023-05
- J. Smits.** *Strategic Language Workbench Improvements.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06
- A. Arslanagić.** *Minimal Structures for Program Analysis and Verification.* Faculty of Science and Engineering, RUG. 2023-07
- M.S. Bouwman.** *Supporting Railway Standardisation with Formal Verification.* Faculty of Mathematics and Computer Science, TU/e. 2023-08
- S.A.M. Lathouwers.** *Exploring Annotations for Deductive Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09
- J.H. Stoel.** *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software.* Faculty of Mathematics and Computer Science, TU/e. 2023-10
- D.M. Groenewegen.** *WebDSL: Linguistic Abstractions for Web Programming.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11