

Bachelor Project by **Mick de Lange** and **Mick van Gelderen**
Coached by **H.G. Gross**



On developing **DizzyData Front-End**
Commissioned by **T. Paymans**

July 2013



Preface

This is the final report on the *DizzyData front-end* (DF) Bachelor Project for the Delft University of Technology by Mick de Lange and Mick van Gelderen. The project was performed in assignment of the company Newviews based in Rotterdam, the Netherlands.

The assignment was to create an easy-to-use client side application that exposes the versatile functionality of Newviews's digital document processing system called *DizzyData*. This project was performed at the offices of Newviews, where we had been given our very own desks. The Bachelor Project was started in April 2013 and finished in July 2013.

In this report all details about the process and results of the project will be described. Some parts of this document are quite technical, but it was kept in mind that the reader might not have the same expertise as the authors.

This document includes the following documents as appendices: DF – Project Approach, DF – Orientation Report, DF – Scrum Workflow and DF – Git Workflow. These documents have been written during the project and their purpose is explained in this report.

We hope you enjoy reading this report as much as we have enjoyed writing it and we would be honored if our documented experiences can serve a purpose even after the project.



van Gelderen



de Lange

Acknowledgements

We want to thank the following people in particular for their help:

Tim Paymans - the CEO and Founder of Newviews, our main contact and guidance during this project.

Wessel van Leeuwen - the interaction designer who helped us get a feeling for user interaction in the front-end.

Martin van Wezel and Amol Gathale - the key developers of the *DizzyData* system and API.

Hans-Gerhard Gross - our TU coach during the project.

Martha Larson - the coordinator of the TU Bachelor Project.

Also, we want to thank our colleagues at Newviews for supporting us and making working at Newviews an absolute pleasure.

Summary

With the growing use of digital systems, data processing and digital archives, the demand for digital document processing solutions grows. Newviews is a company that provides such a solution for digitally processing scanned invoices. And now Newviews has developed a new product: DizzyData, a flexible document processing solution. This solution can be implemented by other parties through an API and the user is free to design their own way that the documents are processed.

For this Bachelor Project a front-end was designed and developed that aims to expose the DizzyData API's functionality. The main objective was to create an environment for non-programmers to get a full understanding of the capabilities of the DizzyData system behind it. Thereby helping sales for the new product.

To guide the development process in a good fashion the Scrum methodology was used. This methodology aims at developing small bits of the end product in short sprints, thus making it easy to monitor the progress of the project. This method was new to both developers and therefore took some time to get used to, but was eventually a great way to manage the project. Scrum is also a method that the company wants to implement in their processes further, which made this a great testing ground.

To create an understandable interface, the help of an interaction designer was called in, who advised on suitable ways to visualize the application. This resulted in the design of an easy to understand interface that would allow anyone to create their own document processing workflow.

Implementation of this front-end was done in a young and new JavaScript framework: AngularJS. Incorporating HTML and JavaScript as main programming languages, the application is very flexible and can be run on any computer with a browser.

To guarantee code quality several testing technologies were applied, where unit testing and coverage results played key roles. The code was also rated on maintainability by SIG, which gave it an average rating on the first submission of code. This holds that the code is maintainable, but some improvements were made to improve this score.

The resulting product did not completely cover all parts that were originally intended. This was partly due to improvements that need to be made to the DizzyData API, which is still in a testing phase. The cooperation between the development teams of the front-end and DizzyData system was very good, so many of the issues could be easily fixed.

CONTENTS

Contents

1	Introduction	1
1.1	Company	1
1.2	Project background	1
2	Task description and Requirements	3
2.1	Project client	3
2.2	Problem definition	3
2.3	Objectives	4
2.4	Assignment formulation	4
2.4.1	Deliverables	4
2.4.2	Preconditions	4
2.5	Requirements	5
3	Process	6
3.1	Scrum	6
3.2	Planning	6
3.2.1	Realized planning	7
3.3	Tools	9
3.3.1	Jira	10
3.3.2	Bitbucket	10
4	System design	12
4.1	DizzyData API	12
4.2	Front-end	15
4.3	User interaction	15
4.4	Graphical design	16
4.5	Technical design	20
4.5.1	OOP in JavaScript	20
4.5.2	Class diagrams	20
4.6	AngularJS framework	22
4.7	Hosting the application	23
5	Implementation	24
5.1	A web application	24
5.1.1	HTML	24
5.1.2	CSS	25
5.1.3	JavaScript	25
5.2	AngularJS	26
5.2.1	Document structure and partials	26
5.2.2	Adding liveliness through controllers	27
5.2.3	Working with data in controllers	28
5.3	Promises	29
5.3.1	Introduction to asynchronous programming in JavaScript	29
5.3.2	Problems of the callback method	30
5.3.3	Introducing promises	33
5.4	DizzyData API communication	36
5.4.1	CRUD	36
5.4.2	Authentication	37
5.5	Models	39
5.5.1	Basic models	39
5.5.2	Workflow translation	39
5.6	The document previewer with OCR support	40

5.6.1	Laying the foundation	41
5.6.2	Navigating through the document	42
5.6.3	Adding pagination	43
5.6.4	Implementing zooming	43
5.6.5	The OCR overlay	47
5.6.6	Improvements	47
6	Code quality	49
6.1	Testing	49
6.1.1	JavaScript applications with Node.js	49
6.1.2	Dependency management with NPM	49
6.1.3	Automation with Grunt	50
6.1.4	Testing with Karma and Jasmine	50
6.1.5	Code coverage with istanbul	51
6.2	SIG feedback	52
7	Future work	54
7.1	API improvements	54
7.1.1	Statistics and billing	54
7.1.2	Sharing Workflows and templates	55
7.1.3	Edit Split and Classify Steps	55
7.1.4	Ordering Steps	55
7.2	Front-end improvements	55
7.2.1	Creating Workflows	56
7.2.2	Storage-specific settings management	56
7.2.3	Previewer	56
7.2.4	Graphical design	56
8	Conclusion	58
	References	60
	Appendix A: SIG feedback (Dutch)	61
	Appendix B: DF – Project Approach	62
	Appendix C: DF – Orientation Report	78
	Appendix D: DF – Scrum Workflow	95
	Appendix E: DF – Git Workflow	105

1 Introduction

This document is the final report on the *DizzyData front-end* (DF) Bachelor Project. The project was focused on creating an interface for the *DizzyData* software created by Newviews. The functionality of the *DizzyData* system is exposed through a *Representational State Transfer* (REST) *Application Program Interface* (API) that DF can use. Due to the extensive possibilities of *DizzyData* software and the characteristics of DF's typical users, the project leaned a bit to the field of *End-User Programming*.

In the coming sections we will discuss the assignment that was given to us by the company and the requirements analysis that followed. Then the process of this project will be illustrated by explaining what methods we used to manage the process, which tools were used and what the planning looked like compared to the realized planning. We will go into the system designing process and all aspects of the design. From the design follows the implementation, where the used methods and techniques will be discussed and illustrated using examples. Finally we will assess the code quality, based on testing and feedback from *Software Improvement Group* [4] (SIG).

The following sections will give a short introduction on the company and why the user interface is relevant to them.

1.1 Company

Newviews (BudgetBoekers B.V.) is the company that develops *DizzyData*, Newviews is a software company which currently delivers a *Software-as-a-Service* (SaaS) solution to digital invoice processing. The service processes scanned invoices provided by its clients, digitalizes the information and then sends that information to third-party accounting software solutions. In this process the important data on the invoice is collected using *Optical Character Recognition* (OCR) and smart recognition algorithms. When the data has been transmitted to the accounting software package of the clients' choice, accountants can further process the invoices.

Newviews is a small company with about twelve employees, based in Rotterdam. The employees consist of two directors, two salespersons, two support employees and six developers. Currently, Newviews has many customers in the Netherlands that use the software to integrate with their accounting software. There is a growing interest in solutions like Newviews for easier processing of invoices and other documents, this is one of the main reasons to start a new project: *DizzyData*.

1.2 Project background

Due to this risen demand for digital document processing, not just for invoices, the *DizzyData* project was started to provide users with the possibility to build their own digital document processing solution. As stated in the introduction, the functionality of *DizzyData* is available through a REST API which can be used in systems created by others. The *DizzyData* technology has great flexibility in the document processing Workflows, allowing for numerous different applications to be created. With this great flexibility comes a problem. Because of the vast amount of options and combinations of all these options, understanding the workings of the system can seem a daunting task. The complexity makes it hard to keep an overview and to explain to potential customers the capabilities of the solution and what they can use it for.

1 INTRODUCTION

This is where our project comes in. The front-end which we created should give the users an overview of the system, and make creating and editing the structure and properties of a document Workflow easy and clear. This helps to make the project accessible for users that lack experience in programming and development, thereby also helping the sales team by allowing them to give a graphical demonstration of the product to potential customers.

2 Task description and Requirements

For a description of the task we are going to perform, we will first describe the client and contact at the company for whom the project is performed. Next the problem definition will be given, along with the objectives. From this description the assignment formulation, deliverables and preconditions will be extracted. Based on the given task description requirements were formulated in agreement with the product manager, these requirements can be found at the end of the DF – Orientation Report document, see Appendix C.

2.1 Project client

As said, the client is Newviews, a small software company which develops and offers document processing solutions. Newviews has requested us to explore the possibilities to create a standalone front-end to their new product called DizzyData.

Our contact at the company is Tim Paymans, founder and CEO of Newviews (BudgetBoekers B.V.). He is the main contact and guidance to the development team.

2.2 Problem definition

DizzyData is an all-in-one solution to document processing and recognition, offering many options for processing flows and document management. The DizzyData API enables the user to design and create their own document Workflows, edit them and look at the usage statistics of these Workflows. The problem is that this not only makes the solution very flexible and adaptive, but also quite complex. To manage this complexity and provide with a easy and transparent interface for the users, a solution needs to be found. The problem challenges consist of three main parts.

The first part of the problem is making an understandable graphical representation of the capabilities of the DizzyData software. A wish from our client is to expose as much of the capabilities as possible. To visualize the system, an interface is required that can translate the capabilities of the system to a mental model that can be understood by everyone, not just programmers. There is an obvious challenge in wanting to cover as much of the functionality as possible while also maintaining simplicity.

The second part of the problem is that the interface should enable users to manage, edit and create Workflows. The challenge lies in the fact that the interface should be usable for developers/computer experts as well as people with no programming experience. This requires that the end-user development environment is simple and understandable for people with no programming experience, and also allows for more complex actions for the more experienced programmers.

The third part of the problem is the challenge to make the interface function standalone, and possibly place the created interface in several cloud systems and marketplaces. For easy access to the front-end application, the client wants to explore the possibility of adding the front-end to several marketplaces and cloud solutions. This is of lower priority than creating the front-end itself, but would be a great way to create an extra marketing channel for the application.

2.3 Objectives

The objective of this project is defined as creating a front-end to use as a marketing tool towards potential customers as well as an interface to provide with a less complicated way of using the capabilities of the API.

DizzyData is a versatile but complex system, with many possible applications. To market this and to visualize the possibilities to potential customers, even those with no programming skills, a good visualization is needed. This is the main objective for the project, required by the client. The client wants a visual interface to show customers the versatility of the DizzyData system.

The second objective is to enable customers with little to no experience in working with REST API's or programming, to manage, create and edit their own document processing Workflows. This level of end-user development is the second main objective for the project. Via an accessible interface users with minimal programming skills should be able to work with the DizzyData system.

2.4 Assignment formulation

The assignment is to create a user interface, which functions as a standalone visual front-end to communicate with the DizzyData API. This front-end fulfils the two main objectives, by creating an administrative environment for the user. In the project, the deliverables are more important than the time invested in the project. The project is expected to be an almost full-time effort for two months, starting at the end of April 2013. The deliverables give an overview of the specifications of the project.

2.4.1 Deliverables

1. A functional and understandable¹ front-end interface for the DizzyData REST API.
2. The interface should give a clear visualization of the capabilities of the DizzyData system.
3. The interface should function as a administrator tool to the DizzyData system, where account settings, Workflows, Jobs², statistics and billing are visually available. Without requiring the user to understand the API methods.
4. The interface should allow for users with little to no programming skills to create and edit Workflows.
5. The front-end should allow for the distribution of several template Workflows and possibly also the (publicly) sharing of user-created Workflows.
6. The front-end should work in a standalone environment, thus using only API calls for communication with the DizzyData system.

2.4.2 Preconditions

There are several preconditions to the project which need to be taken into account. These preconditions represent the limitations of method-use and on the final result.

¹Understandable is defined through user tests and interviews.

²Jobs are the instances of a Workflow, containing details about the processed documents and the resulting data.

1. For development the agile SCRUM method is used.
2. The design is done in coöperation with both a graphics and interaction designer.
3. The front-end is a web-based application, which runs standalone.
4. The server side of the web application has to be written in `C#`³ using `.NET`⁴.
5. The web application has to be modern in terms of web technology
6. The web application has to run on semi-modern web browsers: IE8+, Chrome 25+, Firefox 19+, Opera 12+ and Safari 5.1+ (preferably also Safari iPad).
7. The product software complies with the *Model-View-Controller* (MVC) architecture.

2.5 Requirements

Based on the task description these deliverables were translated to a requirements list. We used a technique called *requirement analysis* to determine the scope of this project. Common requirement analysis methods are outlined in the publication *Software Development Process – activities and steps* [5]. It gives a good overview of the aspects of requirement analysis such as *stakeholder identification* and defining “measurable goals”. The method we employed during the planning phase and also in the development phase is based on *SCRAM* as defined in *Scenario-based requirements engineering* by Sutcliffe [7]. The complete *SCRAM* approach is too extensive and time intensive for our small team and time resources so we will not be able to elaborately cover every step.

Determining all the stakeholders will be done by interviewing the *project manager* and doing the requirement analysis. In turn, specifying the stakeholders will open up possibilities of discovering additional requirements. The requirements analysis is an appendix of the DF – Orientation Report document, see Appendix C.

³<http://msdn.microsoft.com/en-us/library/vstudio/67ef8sbd.aspx>

⁴<http://microsoft.com/net>

3 Process

This section will cover all details about the process of the project. We will be discussing the use of the *Scrum* methodology, planning and the main tools we used for guiding the process. This section will then reflect on these aspects of the process and what elements were successful and where improvement was needed.

3.1 Scrum

For this project we chose the agile Scrum methodology, which was also a request from the company. The company is currently trying to convert their processes to the Scrum process and this seemed a good trial run. The Scrum methodology is aimed at developing in short periods of time, sprints, resulting in working parts of code. As explained in our adaptation of the Scrum guide, see Appendix D: DF – Scrum Workflow, we used the Scrum framework as a guide in this project.

One important decision was to shorten the sprint length to one week, instead of the usual length of two weeks. This was done mainly because our development team consists of just two people and the time for the project is limited. This way we could focus on delivering parts of code each week, although they might be a bit smaller than is common in Scrum.

The first few sprints were needed to discover how much time was needed for specific tasks. This meant that not all tasks we set for the first sprint were actually finished in the first sprint, because we underestimated their complexity or time consumption. In later sprints these complex tasks were split up in to several smaller tasks, to comply with the target of finishing tasks in a sprint.

Also, the meetings did not have to take a long time, because we work together the whole week. When there was a small issue, this could be discussed immediately, eliminating the need to recap the entire sprint in each meeting. We did focus on having a daily Scrum meeting in the morning, in which the product manager sometimes participated. This way both us as developers and the product manager would be up to date on which elements would be focused on that specific day and how far along we were in that sprint.

This meant that our interpretation of the Scrum process was a limited version, but still an effective method. Especially the daily Scrum meetings were a useful tool to keep track of the work that needed to be done.

3.2 Planning

After the assignment was defined clearly work started on writing a plan of approach, see Appendix B: DF – Project Approach, in which we drafted up an initial planning as well. This first planning is shown in Table 3.2, where each row represents one week of the project. Not all information was clear at the start of the project, which resulted in a bit of a rough planning.

3 PROCESS

Week	Date	Activities
1	22-04 - 28-04	Orientation phase Plan of approach finished
2	29-04 - 05-05	Orientation phase finished Orientation report finished Contacted the designers Interview stakeholders
3	06-05 - 12-05	First Scrum sprint
4	13-05 - 19-05	Second Scrum sprint
5	20-05 - 26-05	Third Scrum sprint
6	27-05 - 02-06	Fourth Scrum sprint
7	03-06 - 09-06	Fifth Scrum sprint SIG code evaluation
8	10-06 - 16-06	Final Scrum sprint SIG feedback Final code to SIG
9	17-06 - 23-06	Final report finished Presentation

Table 1: Original planning

3.2.1 Realized planning

At later moments in the project we edited the planning to fit the current situation. An example is that, after formulating our requirements and plan of approach, we decided in week 3 to go with a bit different approach to the implementation. This delayed the initial sprint, because we needed to change our implementation plan.

The following part will give a more detailed overview of our realized planning per week. In noteworthy cases additional information is added to explain the differences to the original planning.

Week 1: Orientation phase

During the orientation phase we focussed on researching the techniques and methods we were going to use. Because we did not have all the details on the system design yet, we started off researching many possible techniques, especially on the code testing part. Later on in the system design we decided on another implementation method, which did not involve as much server-side programming as we originally presumed. The results of this phase are a) the orientation report which includes the requirements analysis, see Appendix C; b) our adaption of Scrum, see Appendix D; and c) a guide on the use of *Git*⁵, see Appendix E.

Worked on:

- Plan of approach
- Planning
- Scrum guide
- Git guide
- Orientation report

⁵<http://git-scm.com>

Deliverables: Plan of approach, planning

Week 2: Orientation phase

Worked on:

- Orientation report
- Scrum guide
- Git guide

Deliverables: Orientation report

Week 3: First sprint

This week we got delayed due to redefining the plan for our implementation. We decided to adapt another approach to the implementation, where we would focus all implementation on the client-side. The decision was made to work with [JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript)⁶ and [HyperText Markup Language](https://developer.mozilla.org/en-US/docs/Web/HTML)⁷ (HTML) and not with the server-side languages C# and .NET, as originally planned. This new approach would lead to a more flexible product, not requiring a specific server set-up, but just a static file server. This option was discussed and decided together with the product manager of the company.

Worked on:

- New plan for implementation in JavaScript and HTML
- Researched several JavaScript libraries and frameworks
- Set up first basis for the implementation

Deliverables: New implementation plan

Week 4: Second sprint

In this week the decision was made to work with [AngularJS](http://angularjs.org)⁸, a framework developed by Google, which will be discussed further in Section 5 on implementation.

Worked on:

- Further research on AngularJS
- First implementations in AngularJS

Week 5 - 7: implementation sprints

Worked on:

- Further specifying the system
- Implementation
- Testing
- Report

Deliverables: System code

⁶<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

⁷<https://developer.mozilla.org/en-US/docs/Web/HTML>

⁸<http://angularjs.org>

Week 8: Final sprint

The first SIG deadline was one week later than we originally planned. This came in handy, as the delay from the first sprint had to be made up for in the later sprints.

Although we had some extra time, we did have to cut some elements from the original planning. Because we needed to send in complete code, some unfinished parts did not make it to the final product. In Section 7 future work will be discussed, there the unfinished parts will be explained and recommendations on the future work will be given.

Worked on:

- Implementation
- Testing
- Report

Deliverables: First SIG deadline

Week 9: Report

Due to a late planning, our final presentation was moved to week 11. The deadline for the final report was kept at week 9, to allow studying for the exams in week 10.

Worked on:

- Final report
- SIG feedback

Deliverables: Final report, finalized code to SIG

Week 10: Break due to exams

Week 11: Final presentation

The important lesson if we look at the realized planning is that we needed more time to plan and research the work we were going to do. This resulted in a small delay of a week, which cost us some much needed development time.

Over the course of the project we also found that not every deliverable specified in the assignment could be realized. The planning and deliverables had to be changed to handle this difference. There were several different reasons for this, we will discuss these points in Section 7 on Future Work.

3.3 Tools

To help us in the process of the project, a few tools were used that guided this process. The company behind DizzyData uses products by *Atlassian* [1], which help code developers in the implementation process. Atlassian offers systems that help plan, test, review and code. The two main tools we used to guide the process are *Jira* [3] and *Bitbucket* [2], we will elaborate on these two. Other tools that were used during the implementation (Section 5) and testing (Section 6) will be discussed in their respective sections.

3.3.1 Jira

Jira is an issue tracker created by Atlassian, which allows the user to edit and monitor all issues (tasks, epics, stories or bugs) that are created. By creating issues, these are added to the backlog and can be placed in a sprint planning. Issues describe the user story, bug or task that needs to be fixed and contain additional information to describe the details.

The developers can assign Story Points to issues, which represent an estimation of the required time to complete that task. Ten stands for the longest or most complex task imaginable, one is the simplest task in the project. These estimations are based on experience, which makes them vary a bit per developer. This was new to us, so it took some time to get the estimations accurate, which went better towards the end of the project.

At the beginning of each sprint the issues from the previous sprint were reviewed to check that they are finished. Unfinished issues are moved to the new sprint and reformulated, because apparently the tasks complexity was not estimated correctly. New issues are created and together with updated old issues they are added to the new sprint. Jira shows a Scrum board which gives an overview of the current sprint, an example can be seen in Figure 1.

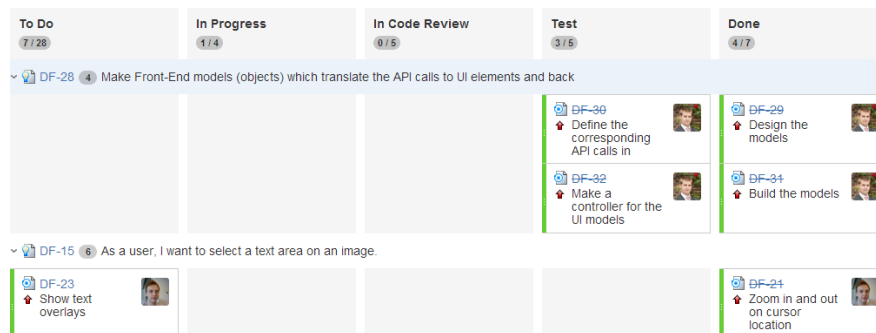


Figure 1: Part of the Scrum board in Jira

Keeping track of the issues in this manner gives a clear and simple overview of what needs to be done in a specific sprint. We used this system for almost all tasks that needed to be performed, but in some cases we did deviate to fix a minor bug with high priority. This is not the way the scrum sprints are intended, but some bugs needed to be taken care of quickly. Generally, we kept a good record of our issues in Jira and found this tool very helpful in the Scrum process.

3.3.2 Bitbucket

Bitbucket is a Git version control system, which hosts your code as well as provides with an interface to track all commits. The main purpose to use such a service is that the cooperation between multiple developers requires a central server to host the code. We wrote a small Git guide for documentation, which can be found in Appendix E. Bitbucket was acquired by Atlassian and integrated with their other services, such as Jira. This allowed for linking between commits in Git and the issues in Jira. Jira can then display which commits were made to a specific issue. Figure 2 shows a part of the commit history in Bitbucket, where the issue numbers (the *DF-XX* numbers) are links to the issue in Jira.

The version control that Bitbucket offers, provides a way to track all changes in the code. Developers are supposed to only push functional (non-disruptive) code to the repository and therefore only the latest, working version should

3 PROCESS

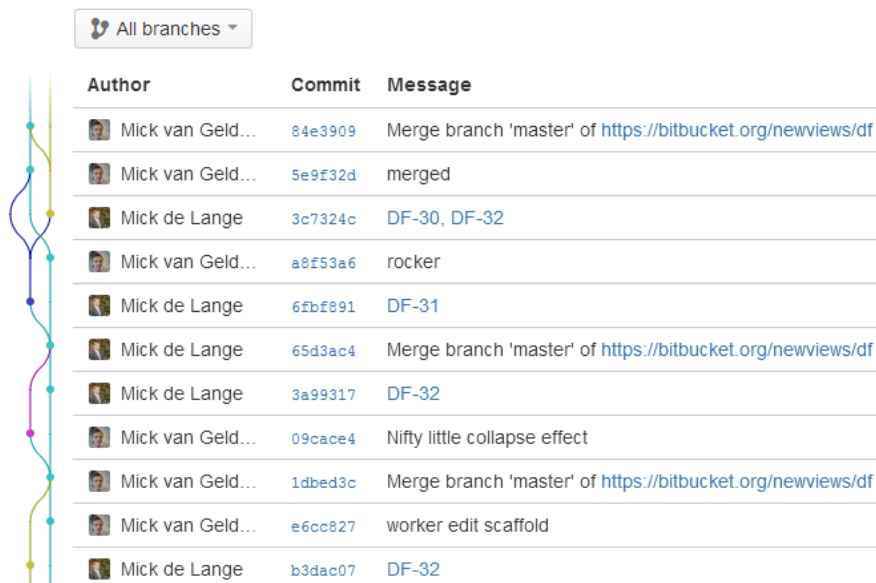


Figure 2: Part of the commit history in Bitbucket

be in the origin. When a developer works on a separate part of the project, a branch can be made, allowing the developer to push partial work to that branch, without disturbing the master branch. We only used the option of branching once because we wanted to switch tasks. One of us was working on the so called “previewer”, see Section 5.6, and needed to start working on something else. This meant that we needed to transfer the experimental code to the Git server so that another developer could continue to work on it. The previewer branch was merged with the master once the previewer code was stable and tested.

Although we did not completely utilize all possibilities that Git provides, we did get a good feeling for what we can do with it and how it can support the development process. The company will make the switch from *Apache Subversion*⁹ (SVN) to Git soon and also asked us to draw up a manual to help make the switch, see Appendix E. Creating this manual gave a good insight in Git and will definitely help us in future projects using Git.

⁹<http://subversion.apache.org>

4 System design

The main challenge in this project was visualizing the features in the DizzyData API. As explained, the DizzyData system is aimed at giving the users as much freedom as possible in setting up their document processing system. This results in a system with many options and possibilities, represented by different elements. The front-end needs to represent these elements in an understandable manner. To get a feeling for a good way to do this a brainstorm with an interaction designer was organized by the product manager. This interaction designer had many good tips, which helped design a good way to represent the, quite complex, system in a logical user interface. Before we explain how this representation came to be, we will first give an overview of the API itself and explain the main elements which we worked with.

4.1 DizzyData API

The DizzyData API is the interface used by developers to manage their document processing system. This API is designed, build and maintained by the software developers at Newviews itself. For our project we had direct contact with these developers for a good cooperation between the system and the front-end. Because the DizzyData system is still in development, many functions were under testing. This meant that some errors or incomplete functions were found during our project, which we relayed to the developers immediately. The development team is continually working on improving the system and is also processing our comments. To get an understanding of the system, we will discuss the important resources provided by the API and their respective functions in the system.

Clients

A Client in the DizzyData system represents an application or program that incorporates the API. This means that any system that uses the DizzyData API is registered as a Client. The Client can have multiple Users and Workflows for several different applications. The Client can be seen as the main registration for all settings in the DizzyData system.

Users

Users are, as mentioned, linked to a Client, they are represented by a name, email address and password. Upon logging in, all activities in the system are bound to that specific User. Rights and permissions can be set per User, to manage the Users capabilities in editing, viewing or creating Workflows, Steps, Jobs etc. Because permissions and activities are bound to a User, this can be easily monitored to keep track of the usage of the system.

Workflows

Workflows are the important part of the DizzyData system, they represent the processing which needs to be done on a document that is send in. The Workflows consist of consecutive Steps which perform the actual tasks on the document from end to finish. Just as a Client can have multiple Workflows, a Workflow can have multiple Steps to process the documents. When a file is send in to the API, it references the Workflow that should be used in that Job to process the file.

Steps

Steps define the actual processors in the system, they represent the parts of the system that should be invoked on the documents in the process. The Steps describe the settings that are used in the processing and the order of Steps is used to represent the Workflow. The DizzyData system currently has a total of about 25 different Step types., the main relevant Steps are:

- **OCR**

The *Optical Character Recognition* (OCR) Step is actually the main and most important function in the DizzyData system, this Step uses OCR software to convert scanned images to text. This Step is needed for almost all further processing Steps, because the text is needed for processing. Only a few Workflows can be used that don't require an OCR Step.

- **Convert to tiff**

The convert to tiff Step is a Step which is almost always used to convert the submitted scan to a standard grey tone for optimal OCR results. The Step is optional, but in all Workflows created by the company itself it is added, because it helps produce better results regardless of the quality of the input.

- **Merge**

Merge is a Step that simply combines all files that are send in to one specific Job and outputs a multi-page document containing all pages from the original documents. It is also the only processing Step that does not require the input to first go through an OCR Step, because no knowledge of the contents of the documents is required.

- **Extract**

Extract is a processing Step that uses Elements to specify the information that needs to be extracted from the documents that it receives. For instance, it is possible to create a Workflow that contains an Extract step that has an Element that defines how to find a bank account number. The extract Step uses the data of the bank account number Element to find all the bank account numbers in a document. The found values then become available to the system for output or further processing.

- **Split**

Split is another processing Step that allows a user to split documents based on predefined rule sets. These rules are based on found keywords or values on the page, like the Elements used in extract. This way a multi page document can be split into separate files based on certain rules. This Step is especially useful when you have multiple documents that have been put in a single file. The Split step can take the single file and cut it into separate documents that can then be further processed.

- **Classify**

Classify is the third processing Step, which can be used to detect the type of document that was send in. This does not mean the computer file type as in the file extension, but the type of real world document, for instance: invoice, tax form, letter or order form. Similar to the split Step, rule sets can be created that define what type of document is to be recognized, based on what values. Depending on the settings the classification can either be outputted or can result in the document being send to another Workflow which is specifically designed to process files of that type. This Step is particularly useful as a follow up to the Split step.

- **Input and output to several storage services**

The system also contains a selection of storage services like Dropbox¹⁰, Box.net¹¹ and Google Drive¹², which can be used as input or output for a Workflow. This way files can be automatically retrieved from a specified folder to be processed and then delivered to any other folder or storage service. These Steps need to be either the first or last Steps in a Workflow. Normal in- and output go through the API, no special Steps need to be selected for that.

Input Steps can also be placed in a separate Workflow, containing only this Step and point to another Workflow that actually processes the files. This allows the user to select multiple storage services to serve as input for a single Workflow.

Multiple output Steps can be put at the end of a Workflow, they will be completed in the order in which they are placed, so files will first be stored in one service and then the other.

Jobs

Jobs can best be interpreted as an instance of a Workflow. When documents are send in to a Workflow, a new Job is created. This Job resource monitors the status in the process, showing when the Job is working, finished or gave an error and on which Step the Job is currently working. A finished Job contains the results of the processing Step in the Workflow and possibly files that were generated in the process. For use of the API, outside of the creating and editing of Workflows, only this resource is needed for sending in files and retrieving results.

Elements

Elements are part of the processing Steps: Extract, Split and Classify and allow for the user to define keywords and values that are of interest. An Element contains information on how to search, where to search and what important characteristics the results need to have. The representation of the keywords and values is done by Anchors and Results respectively. An Anchor defines the keyword that has a location related to the searched value, this can also be the entire page. A Result gives a format for the resulting value in the form of a regular expression, this way only results of the preferred type can be found. For example, an Anchor could be “bank account”, the related area would be right next to that keyword and the Result would be a regular expression defining a set of numbers.

OAuth

OAuth¹³ is an open protocol to allow secure authorization, it is used in many API solutions. The DizzyData API also uses this protocol to authenticate users and verify their rights to each resource. The protocol provides a registered Client with special keys to retrieve an access token, this token is valid for a limited time period. Each call to the API needs to contain this access token to verify the identity of the user. For this project, a specific addition to the protocol had

¹⁰<http://www.dropbox.com>

¹¹<http://www.box.net>

¹²<http://drive.google.com/>

¹³<http://oauth.net/>

to be implemented to allow users to receive an access token based on a login using username and password.

In the standard OAuth flow the Client id, Client secret code and a refresh token can be used to retrieve an access token. Doing so in this project would present a dangerous leak in the system, because the front-end is build in JavaScript. A Client secret code should never be stored in a code that is accessible by others than the application itself, which would be the case when using JavaScript. Another problem is that a user must sign in to its own Client, not the Client id that could be used by the front-end. These are the reasons to implement a new authorization flow using a username and password combination.

4.2 Front-end

For the front-end that we were going to build, several choices needed to be made on how to display the resources from the API in a simple and understandable manner, without limiting the users possibilities. A direct visualization of all parts in the API would be simple and practical choice. In that case a Workflow could be represented by a collection of Steps and each Step would have its own specific settings page. This is the approach we chose for visualizing the Client, User and Job resources, but after a good brainstorm with the interaction designer a different approach was chosen for Workflows, Steps and Elements.

In the following sections we will explain this process, the choices that were made on interaction design and how the front-end was built to realize these choices.

4.3 User interaction

User interaction is a field that focusses on the way users interact or work with a system or product. In our case this relates to the way users can use the interface and in what way the users will be represented with information. For the interaction design the company called in the help of *Wessel van Leeuwen*, a senior interaction designer at *BackBase*¹⁴. We held a small brainstorm session with him. First the details of the DizzyData solution were explained to him, after which the goals of the front-end were discussed. Wessel continually gave feedback on our ideas and brought his own ideas and solutions to the table.

The main conclusion in this session was that the complexity of displaying the Workflows lay in the amount of Steps and Step types. A Workflow containing up to, or possibly even more than, ten Steps would be hard to visualize in a simple and understandable way. To solve this a simpler approach to the systems possibilities needed to be found. We started to distil the main functions of the system and break them down to the basics.

In this process the Merge, Extract, Split and Classify Steps came forward as obvious main Steps. All except the Merge Step require OCR before they can be processed, so the OCR and convert to tiff Step should be incorporated within these main Steps invisibly. The thought behind this is that most Users have little to no knowledge of settings that could optimize the character recognition, this could better be handled by us. After this decision was made, we were left with only four main Steps, which we decided to call “*Workers*” for the time being. All other Steps were only meant to use as input or output to these workers.

A special case occurs when multiple main Steps are used in one Workflow, which would extend the length of a worker and bring us back to the original

¹⁴<http://www.backbase.com/>

problem. The solution we thought up for this problem was to view each main Step as a separate worker and display the first one as to have output to the next specific worker and the other to only receive input from the previous worker. After we made these interaction choices, we could start on the next part of the design phase.

Because the main task in this project is to create a graphical interface we started designing from the perspective of the user. We tried to leave the details and possibilities of the API behind us. Therefore, after getting a rough idea about the workings and feel of the front-end based on the interaction design, the next step was to make mock-ups of the User interface. These mock-ups gave a clearer picture of the things that needed to be done to translate the API calls into an interface and vice versa.

4.4 Graphical design

Originally the plan was to work with a graphical designer to create the graphics and look of the application. Unfortunately the designer got ill the day before the meeting and rescheduling would delay the project too much. Therefore the professional design was moved to a later stage after this project, more on this in Section 7 on future work.

The ideas about the interaction design were visualized in mock-ups, to get a feeling for the interface and the usability of the front-end. The mock-ups play a big role in designing the models that would be used in the front-end, while the main goal in this project is visualization of the DizzyData system. Because the graphical look was not as important as the actual functionality of the system, we kept the graphics simple in the mock-ups. In the actual view elements of the implementation we used the *Bootstrap* library¹⁵ to handle the graphics.

One of the most interesting parts to create a mock-up for was the Worker overview. In this view the user can see the list of all his Workers and select one to edit. This overview would thus represent the Worker in a small and simple block, giving the user a clear impression of its key features. As mentioned in the Section on user interaction, this visualization required the most work.

In Figure 3 the mock-up created for the Worker overview can be seen and Figure 4 shows the actual overview page created in this project. A difference between the mock-up and the resulting solution is the fact that not all graphics were available, these were replaced with braces (`{ }`) for the time being. Also, the mock-up shows a Worker that is four parts wide, this was mocked to get a feel for the layout of a Worker. Workers were eventually cut down to be maximum three parts wide, in accordance with the interaction design.

¹⁵<http://twitter.github.io/bootstrap/>

4 SYSTEM DESIGN

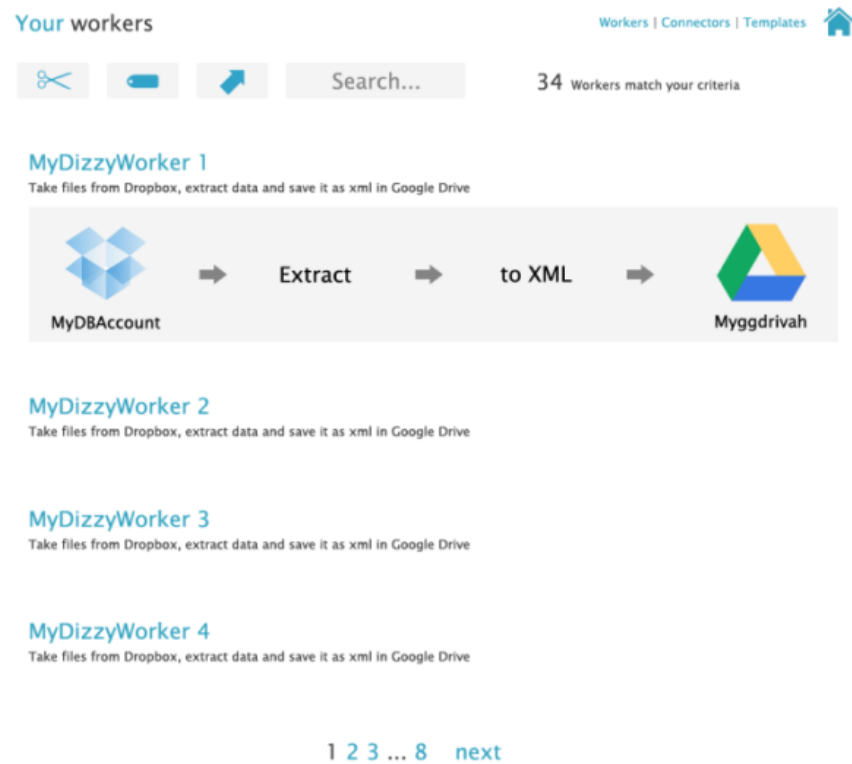


Figure 3: Mock-up of workers overview.

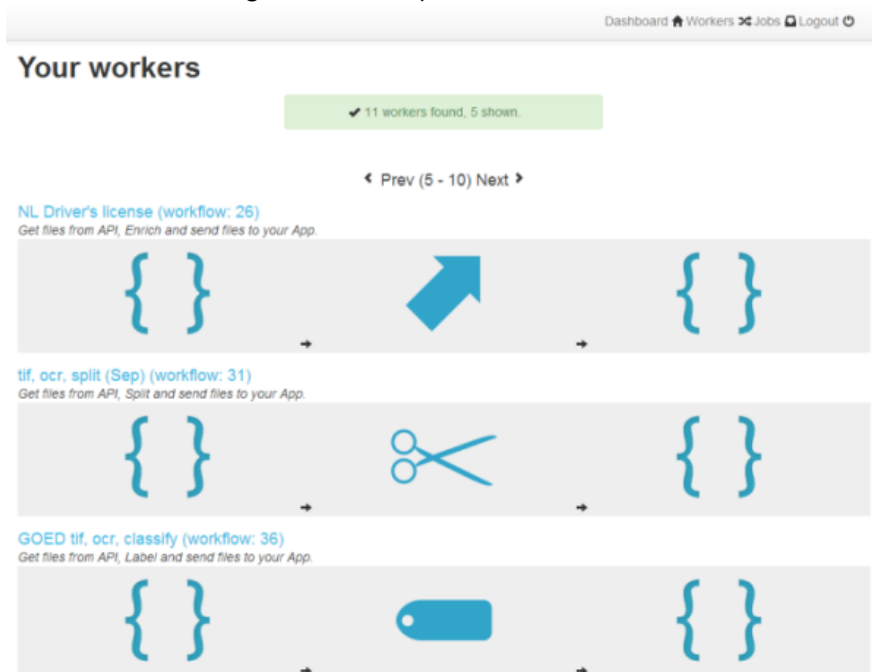



Figure 4: Actual workers overview.


4 SYSTEM DESIGN

Another important part of the system which needed a good design was the edit page for Workers. This page allows the user to edit their Worker settings and define the Elements that should be searched for in the document. A decision was made to place all Elements on the left side of the page and place an example document on the right side, with an overlay of the recognized words on the image. This way a document could be used to get a feeling for the working of the system and make editing Elements less abstract. The mock-up for the Worker edit screen is shown in Figure 5.

Creating a previewer to display an image and place recognized words on it proved quite a challenge, more on that in Section 5 on the implementation. The end result for the Worker edit page was very similar to what we had in mind, although we could not implement all parts which we intended. A screenshot of the page can be found in Figure 6. Pay special attention to the previewer, where the word “aprii” (*unfortunately, the OCR recognized “April” incorrectly*) is highlighted in the date. This is the function that displays the recognized words on top of an image of the document.

4 SYSTEM DESIGN

You are editing **mySplitWorker** Workers | Connectors | Templates 



 **How** **What** **Where**

What characteristics should be recognized so that a page is split?

kvk nummer 24404115

Anchor: **match** kvknr
area full page

Value: **area** right of anchor
type KVK-nummer

 Delete Advanced 

▼ **vervaldatum** 3-30-2012

▼ **rekeningnr** 154635863

+ Add

1 / 1









Figure 5: Mock-up of worker edit screen.

Dashboard     Logout

You are editing  **NL Tax invoice**

What **Where**

kvknummer

ibannummer

btwnummer

bankrekeningnummer



bedragen




Label
bedragen

Page number
0

Anchor
Add searchterm

Results
Format pattern:
([1-9])?([0-9])?([0-9])?
Add format pattern

 Remove  Save

← 2 → Search 75%   

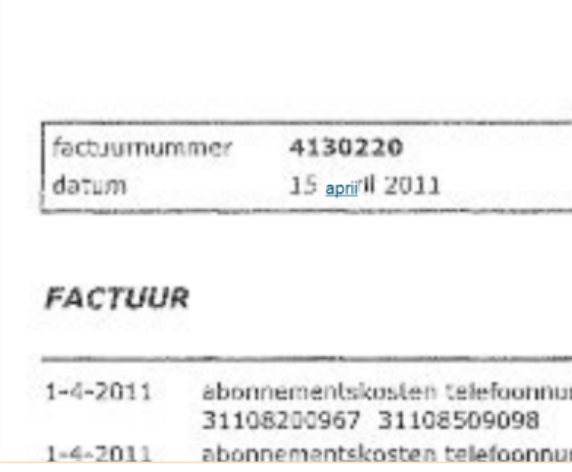


Figure 6: Actual worker edit screen.

4.5 Technical design

Based on the mock-ups created for the interface, a design of the technical background of the system could be made. The assignment required that we work with a MVC type system, which separates the models, views and controllers in a strict manner. Because our system is supposed to be a graphical representation of the DizzyData system, the models we created were more closely related to the views than one would expect in a strict MVC system. Our models are itself a representation (or translation) of the original models: the resources that can be retrieved from the API. In our set-up the views are HTML templates, that give a framework for displaying information. The controllers are defined in AngularJS through *ng-controller* (see Section 5.2.2), these handle the interaction between the views and the models.

The views could be designed after the mock-ups shown in the previous section and controllers had to be build matching that, therefore the technical system design focussed on the models that would make the API models displayable. The simpler parts to design were the Jobs, Users and Client details, because these could be almost directly copied from the API resources.

4.5.1 OOP in JavaScript

Although JavaScript does not actually support *Object Oriented Programming* (OOP) as a programming approach, we did want to design our models in an OOP manner. This can be done by using several approaches, for instance JavaScript supports nested functions and *prototypal inheritance*. Prototypal inheritance allows the developer to define a prototype object which contains variables and functions that each object of that type will have. But because JavaScript is not as strict in this implementation method as other languages, it is still possible to assign new variables and functions to an object and also overwrite the functions defined in the prototype. This means that to keep the objects correct, the development team must work according to strict –self-imposed– rules of OOP, to prevent corrupting the objects.

4.5.2 Class diagrams

Figure 7 shows the part of the class diagram containing the Jobs resource. An API call to the DizzyData system returns *JavaScript Object Notation*¹⁶ (JSON) that contains a list of Jobs and each Job contains a list of ResultFiles and Results. The class diagram shows that this object tree has been directly translated to classes. Thus, this class and resource could be easily created. The choice was made to keep a separate object that contains all Jobs, so all functions that work on multiple Jobs can be build here.

¹⁶<https://developer.mozilla.org/en/docs/JSON>

4 SYSTEM DESIGN

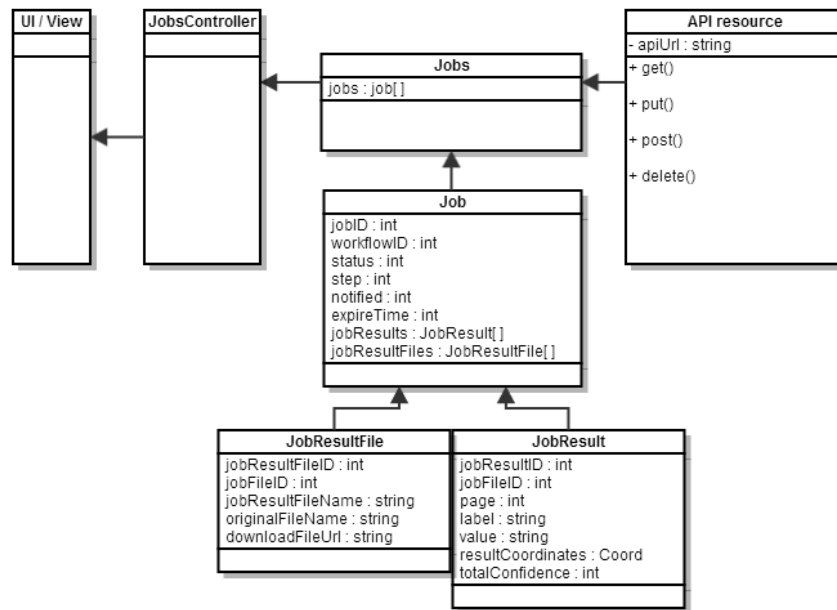


Figure 7: Jobs part of the class diagram

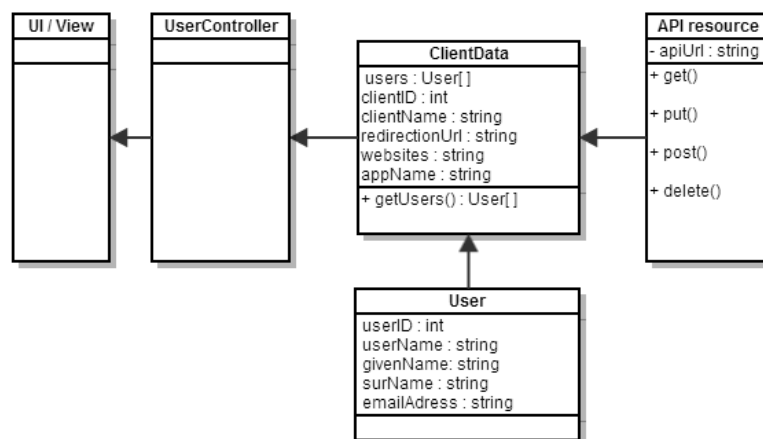


Figure 8: ClientData part of the class diagram

For the Client details a decision was made to alter the resources that the API returns just a little. In the API Clients and Users are separate resources, which have their own calls and are not per se related. But a Client actually is the owner of several Users, or put in other words: Users are bound to a Client. As shown in Figure 8 the Client was represented in a ClientData object, which has a list of Users. This method allowed for a better graphical representation in the interface and made more sense in the logic towards the User. After all, a User logs in to a specific Client and only has access to that Client and its Users, if the User had permission to those resources.

The most interesting part to design was the Workflows, Steps and Elements. These needed to be translated to the worker objects we decided on in the user interaction part. To achieve this, a new class was designed: WorkflowTranslator, which can be seen in Figure 9. This class contains the Workflows and Steps retrieved from the API and has a function that translates these to the

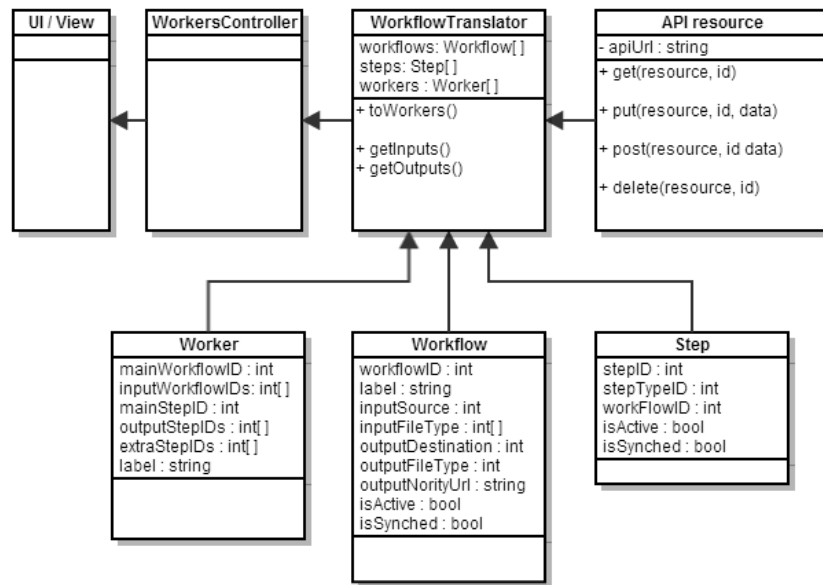


Figure 9: WorkflowTranslator part of the class diagram

displayable Worker objects. To limit the number of API calls the decision was made to retrieve all Steps in one call, not per Workflow. This is also why the Steps are not a list within a Workflow object, but rather a separate list in the WorkflowTranslator

The Worker class contains mainly id numbers that point to the actual objects that make up that Worker. This prevents an abundance of duplicate objects, such as Steps repeated in the Workers and Step list. Also, this allows for the editing of a Worker object to be done directly on the relevant resources, which could then be synchronized with the API. A Worker consists of a main Step, which is one of the main Steps as defined earlier in this Section, a list of input Steps and a list of output Steps. Other Steps that are kept invisible for the user are saved in a list of extra Steps, which makes sure that for instance the OCR Step is not lost in the editing process.

Elements, which are not displayed in the class diagrams, are not relevant for displaying the overview of Workers and were therefore left out. Instead, Elements are linked directly to the controller that handles the editing of a Worker. When editing a Worker the Elements that belong to the main Step are retrieved from the API based on the Step id.

4.6 AngularJS framework

After designing the models we had to decide on the JavaScript framework we wanted to use. There are a lot of client side JavaScript libraries that help with developing JavaScript intensive web applications. Examples of such libraries are [Backbone](#), [batman.js](#), [Ember.js](#), [Knockout](#), [Sammy.js](#), [Spine](#) and [AngularJS](#). We have chosen to use the relatively young AngularJS to develop DF.



Figure 10: AngularJS

The reason why we have chosen AngularJS over the other libraries is that AngularJS is unique in the sense that it allows you to define your own HTML tags and attach behaviour to them very easily. This makes it easy to reuse

components by just inserting the right tags in your HTML. Also, it does not impose any restrictions on how you code your models. The tutorials on the AngularJS website also demonstrate how to test your application. There will be a time in the career of a developer where you think that testing just takes time and doesn't add anything to your code directly. We had both already learned that testing can actually improve your code substantially while you are writing it. It may actually speed up development since it is so easy to re-run all the tests and confirm that everything still works as expected. Being able to easily combine testing with AngularJS development was another big motivation that led us to use AngularJS.

4.7 Hosting the application

A web application needs a server that hosts its code and assets. There are many types of hosting available and different levels of capabilities. The project requirements and the decisions we make during the system design affect our hosting requirements.

An example of a requirement that affects the hosting type is the wish that the application should be globally available by using a cloud hosting solution which was described in Section 2.2. Even if we currently do not really need global availability for our product to work, it is still wise to keep it in mind.

The decisions made in Sections 4.2 and 4.6 affect what our server should be capable of. The fact that the DF models are different from the DizzyData models means that we might have to use our own database to store additional data, see Section 7.1.2. This would be the case if not all the data that DF needs to store can be fit in the DizzyData models.

The fact that we use AngularJS means that our server does not need to have a templating engine configured. `achtml` elements are generated on the client from the AngularJS templates. The templates themselves are static files that will be (asynchronously) fetched from the server.

We figured out how to translate between the two model spaces so we did not need our own database and using AngularJS made a simple static file server enough to suit our needs. The communication flow is illustrated in Figure 11.

An advantage of the fact that we can use a simple static file server is that it can be fitted in any cloud hosting solution. Another advantage is that it is easy to set up a development environment. With the help of Node.js and some modules we could do cross-platform development by using a standard file server.

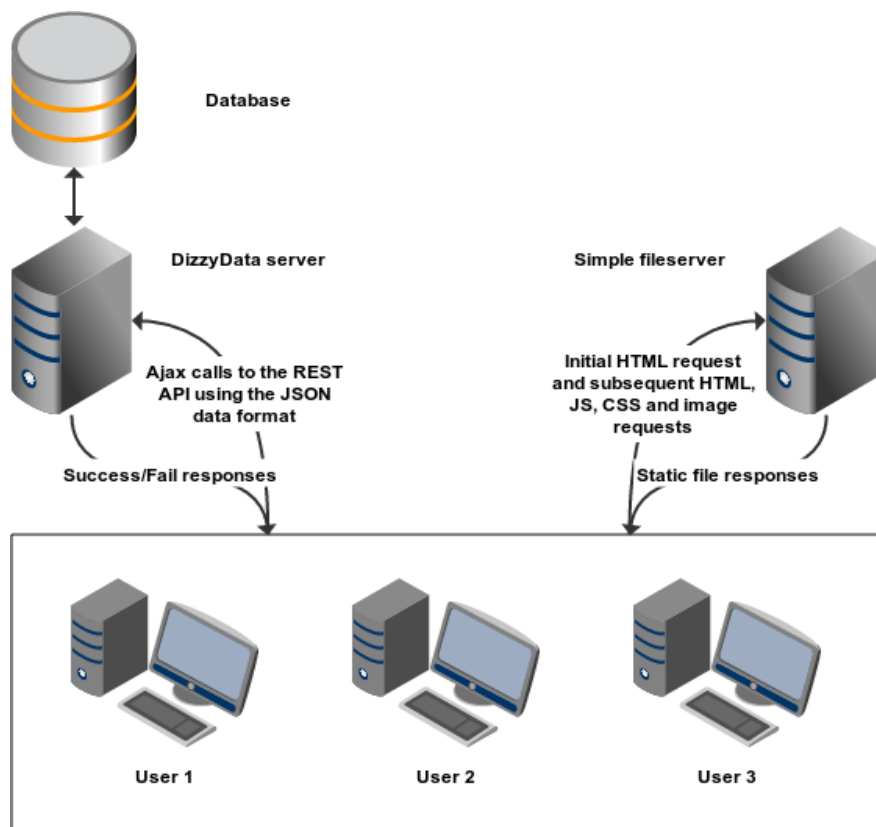


Figure 11: Server communication overview

5 Implementation

So we have our objectives set out for us and the idea of the system design is clear. This is where the real action takes place: the implementation phase. This section covers the most important aspects of the DF implementation. While having a technical background and experience with web development will ease the reading of this section, it has been written with less technical readers in mind.

5.1 A web application

First of all, let's get a quick overview of the components that make up a web application in 2013. Feel free to skip this subsection if you are already familiar with HTML, CSS and JavaScript.

5.1.1 HTML

A term that you will probably have heard of is *HyperText Markup Language*¹⁷ (HTML). HTML has a very long history and the purpose of the language has changed over time. Currently, the purpose of HTML is "to describe the content of a webpage in a structural and semantic manner".

```
<html> <!-- I'm a comment -->
  <head>
```

¹⁷<https://developer.mozilla.org/en-US/docs/Web/HTML>

```

        <title>Document Title</title>
    </head>
</html>

```

Listing 1: HTML code that defines an empty document with title “Document Title”

5.1.2 CSS

The styling of a web page is mainly captured in *Cascading Style Sheets*¹⁸ (CSS) files. Some time ago, the styling was applied by HTML. While this is still possible, it is considered good practice to separate styling from content by using CSS.

```

h1 {
    font-size: 3em;
}

```

Listing 2: CSS code that sets the font-size of h1 elements to 3em

5.1.3 JavaScript

The standard scripting language in browsers is *JavaScript*¹⁹. Scripting can be used in a lot of different ways. What comes to mind immediately is enhancing the application by creating interactive elements such as dialogs or image galleries. Another use-case is asynchronously sending and retrieving data from a server without reloading the page. There are however many other forms, if you will, of scripting.

One of the other forms is providing backwards compatibility with older browsers. As you might imagine, as personal computers became more powerful and the internet more widely used, the most widely used browsers started implementing new functional and graphical extensions. This led to incompatibilities and inconsistencies with respectively older and other browsers. There are many JavaScript libraries, such as the popular *jQuery*²⁰ and *Modernizr*²¹ that try to dynamically fix these problems by providing a consistent and compatible interface. It is usually impossible to achieve the same results with HTML and CSS alone.

The code below is a very simple JavaScript script to give you an idea of what a script may actually look like.

```

// I am a line-comment
/* I am a block comment, I can span multiple lines.
   The script is not affected by comments, they
   are used to document the code */

/* this is a function definition */
function square(x) {
    /* return the square of x by multiplying it with
       itself */
    return x*x;
}
/* declare a variable called 'n' and assign it the number
   3 */

```

¹⁸<https://developer.mozilla.org/en-US/docs/Web/CSS>

¹⁹<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

²⁰<http://jquery.com>

²¹<http://modernizr.com>

```
var n = 3;
/* print this message in an message box */
alert('The square of ' + n + ' is ' + square(n));
```

Listing 3: JavaScript code that prints the square of 3

When you run the code in Listing 3 it will show the famous alert box as can be seen in Figure 12.

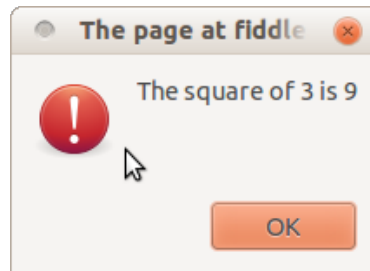


Figure 12: The alert box

5.2 AngularJS

It is a rare chance for a development team to work with a young and, according to the developers of AngularJS themselves, “a superheroic JavaScript *Model-View-Whatever* (MVW) framework”. Often times, the stakeholders will want to stick to more aged methods and frameworks that over the years have proven to be effective for projects that will be used in production. Knowing the ins and outs of a framework is crucial if you want to develop a well-written application. This subsection covers the lessons we have learned about AngularJS during the development of DF.

5.2.1 Document structure and partials

AngularJS encourages you to incrementally build your application. You start with defining the structure of a small piece of your application that very well may be reused in other parts of your website or even in other applications. The basis of these components is usually a piece of HTML code. AngularJS will automatically load these pieces asynchronously if needed.

```
<div class="my-component">
  <input type="text" ng-model="aTextValue">
  <span>You typed: {{aTextValue}}</span>
</div>
```

Listing 4: A basic AngularJS template

Listing 4 shows a basic AngularJS template. It looks like plain HTML at first sight but there are actually some special pieces of code, namely: the attribute `ng-model` and the binding `{{aTextValue}}`. Upon load, AngularJS actually parses your HTML documents and looks for these special attributes and bindings. This specific example will display an input field and a span with text, once the value in the input field gets updated by the user, the text in the span is simultaneously updated with it.

This behaviour might seem odd at first because until now, people advised against using JavaScript in your HTML. For example, this was considered bad practice:

```
<button onclick="myFunction()">Click me</button>
```

Listing 5: The onclick handler

With AngularJS however, you will see a lot of functionality weaved through the HTML code.

The AngularJS template from Listing 4 can be put in a single HTML file. A file containing a piece of HTML is called a partial since it defines a certain part of a webpage. The partial can now be included in multiple places on the website by AngularJS. To give an example, the following code just includes the code in the partial without doing anything special with it:

```
<div ng-include="'/partials/example.html'"></div>
```

Listing 6: A way to include partials

5.2.2 Adding liveliness through controllers

As you may have noticed, AngularJS adds a lot of functionality without having to write a single line of JavaScript. Once you get used to writing applications using AngularJS this can significantly cut down development time. Though the standard functionality in AngularJS already gives you great expressive power, you will sometimes find yourself in need of a custom component. To write such a component you will have to create a “directive”. A directive consists mainly of a partial, a so called controller and some other options.

Note: There is a lot more to writing directives, such as knowing the differences between the compile and link phase and the scope rules, but we are omitting these details for simplicity. Although not all of the information on directives and how to use them is comprehensively and completely documented, see <http://docs.angularjs.org/guide/directive> for the details.

An example of a simple directive is “zippy”. Zippy is a HTML element that can be expanded and contracted by clicking on it. When it is expanded it shows some additional content. The end result is shown in Figure 13. The zippy on the left is expanded and the zippy on the right is collapsed.



Figure 13: Zippy

To use a zippy in your page you would just have to include the following AngularJS template:

```
<zippy zippy-title="Details: Title of the zippy...">
  Some content that goes into the zippy. It is amazing!
</zippy>
```

Listing 7: Zippy usage

This is the partial that is used by the zippy directive:

```
<div>
  <div class="title">{{title}}</div>
  <div class="body" ng-transclude></div>
</div>
```

Listing 8: Zippy partial

In the controller we have to keep track of whether or not the zippy is expanded and that when the zippy is clicked, the expanded state is toggled.

Controllers can also be bound to elements in a webpage with the `ng-controller` directive if you need to expose behaviour to the view but there are no appropriate directives available that do this for you and you don't think you will re-use that particular element somehow.

5.2.3 Working with data in controllers

So you might ask: "How does a controller affect the view? How are they connected?". Well, the controller has access to a `$scope` object. The `$scope` object is essentially the glue between the view and the actual model. The controller decides what the view can do by attaching data and behaviour to the `$scope`

Figure 14 shows an overview of how an AngularJS template, the model and the view are connected according to AngularJS. The word "model" might be confusing because it seems to include all the scopes that are being used by the view in this figure. The actual business models that probably exist, judging from the figure, are the `PhonesHolder` and the `Phone` models.

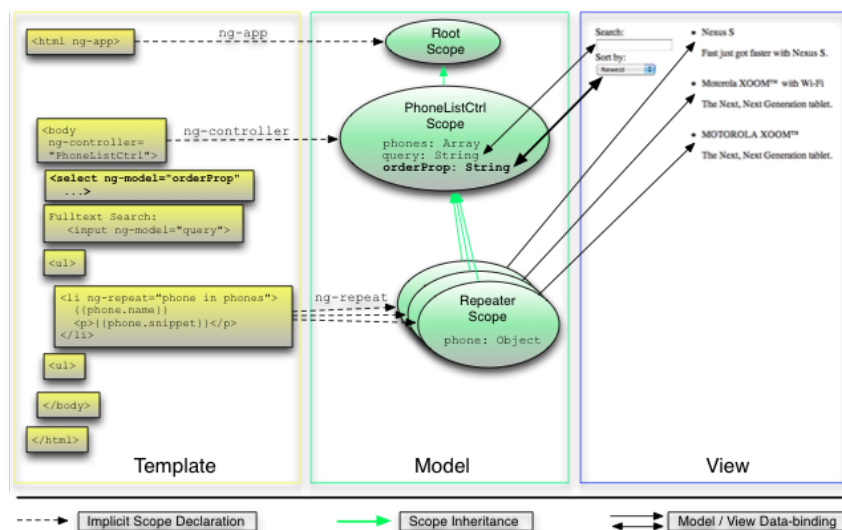


Figure 14: Relation between template, model and view (from <http://angularjs.org>)

AngularJS templates allow you to easily use values attached to the `$scope` by using the double-curly-bracket syntax like in Listing 8. There we used `{{title}}` to retrieve the title from the `$scope` object. Whenever `$scope.title` is changed, AngularJS will update the view for you. This is one of the two biggest powers of the `$scope` object. The `$scope` object can also be used to let the view exert certain behaviour by attaching a function to it.

For example if we wanted the view to be able to load some data when the user interacts with it we could do the following in the controller that is attached to that view:

```
angular.module('myModule').controller('myController', [
  '$scope', 'dataService', function($scope, dataService)
  {
```

```
// Attach a function to the scope
$scope.loadData = function() {
    /* Assuming that dataService will immediately
       return an object that will be filled with the
       data eventually and assuming that angular
       knows that the $scope object might have
       changed when that happens. See http://docs.
       angularjs.org/api/ng.$rootScope.Scope for
       details */
    $scope.data = dataService.loadData();
}
}]);
```

Listing 9: Exposing behaviour to the view

The controller defined above could then be used like this in the view:

```
<div ng-controller="myController">
  <button ng-click="loadData()">Click me to load data</
    button>
  <div>{{data}}</div>
</div>
```

Listing 10: Using functions bound to the \$scope

5.3 Promises

At some point when developing JavaScript applications, you will stumble upon asynchronous programming. Three common ways of handling asynchronous events are through callbacks, with events and with promises. The [Q²²](https://github.com/kriskowal/q) library originally developed by Kowal [6], and now maintained by him and around 30 other contributors, served as the main inspiration for our own promises library. First we will give a short refresher on asynchronous programming in JavaScript and how it has been done for the last decade. Then we will explain what problems occur when using the more simpler methods of handling synchronization. How promises can help with asynchronous programming and the reasoning behind choosing to write our own flavour of promises will be explained at the end of this section.

5.3.1 Introduction to asynchronous programming in JavaScript

Here we will provide a very short introduction to asynchronous programming in JavaScript. This introduction serves to give the reader a basic sense of what asynchronous programming is. Keep in mind that it is by no means a complete guide. The most basic asynchronous function in JavaScript is `setTimeout`. This function will run a piece of code that you provide after a delay that you can specify.

```
// Define the code to be delayed
function toBeDelayedCode() {
    alert("A late message. ");
}
// Execute the function after 1000 milliseconds
setTimeout(toBeDelayedCode, 1000);
```

Listing 11: Asynchronous programming

²²<https://github.com/kriskowal/q>

There are many other occasions where something will be done in the future. For example, let's say we have created a login dialog as in Figure 15 that is activated through JavaScript when the application needs the user to log in again.

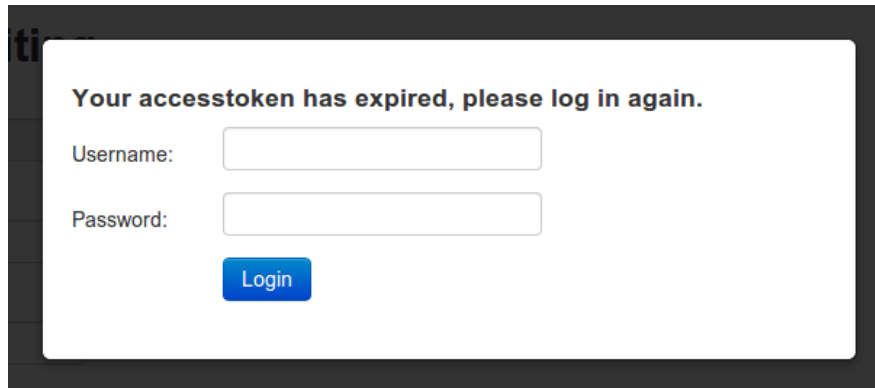


Figure 15: A login dialog

The JavaScript code on the webpage that we are currently on will want to know when the user has logged in. One common way to be notified of such an event is by passing a so called “callback function” when we create the login dialog. The code looks very similar to the `setTimeout` example in Listing 11 in that it also uses a callback.

```
var options = { // An object that contains some options
  title: 'Login Dialog' // such as the title of the
    dialog
};
createLoginDialog(options, function callback() {
  // This code is run when the login dialog is closed
});
```

Listing 12: Login dialog callback

A more common example of an asynchronous event in JavaScript is an *Asynchronous JavaScript and XML*²³ (Ajax) call response. The Ajax technology can be used from JavaScript and provides a method for exchanging data asynchronously between browser and server to avoid page reloads. The login dialog actually uses Ajax internally. When the user fills in his username and password and clicks on the “login” button, a request is sent to the API. The API then responds to this request with either a “success” or a “fail” message. The response is then passed to a callback function so that it can be handled appropriately.

5.3.2 Problems of the callback method

As we discussed in the previous section, the most basic way of handling asynchronous events is through callbacks. There are certain common problems that arise when using callbacks. We will give some examples of these problems so that we can use them later on to demonstrate how promises can help.

It is possible that multiple parts of your application will have to be notified of the result of an asynchronous operation. Let us call these parts “observers”. With callbacks, you would have to notify each observer in the place where the

²³<https://developer.mozilla.org/en/docs/AJAX>

5 IMPLEMENTATION

callback is registered. The example in Listing 13 shows an abstract example of this.

```
someAsynchronousOperation(function(result) {  
    // this is the callback  
    notifyObserverA(result);  
    notifyObserverB(result);  
    notifyObserverC(result);  
});
```

Listing 13: Observing with callbacks

Notice that the observers are now tightly coupled to the above code. If you can only change the observer A code, you can not prevent `notifyObserverA` from being called. The dependency of observer A on the result of the asynchronous operation is inverted: the result is notifying the observer that it is ready rather than the observer is listening to the result to see if it is ready.

Another problem is controlling the flow of your application. With callbacks it is possible to create some utility functions that can help handle the control-flow. Two often required functions are serializing a series of asynchronous operations and doing something after a set of asynchronous operations have finished.

It should be clear from Listing 14 why it would be nice to be able to sequentialize a list of asynchronous functions.

```
getUser(function callback(user) {  
    getWorkflows(user, function callback(workflows) {  
        getElements(workflows, function callback(elements)  
        {  
            // use all the elements of all the workflows  
            // of the user  
        });  
    });  
});
```

Listing 14: Callback pyramid

If you do not apply some clever technique you will end up with a gigantic “callback pyramid”. Things get even worse if you also add error handling to the mix.

Something like the following would be ideal:

```
sequentialize([  
    getUser,  
    getWorkflows,  
    getElements  
], function(user, workflows, elements) {  
    // Use the user, workflows and elements  
});
```

Listing 15: Sequentializing with callbacks

However, `getWorkflows` depends on the result of `getUser` and we do not explicitly define how the results are passed as arguments to the next function. This means that the `getWhatever` functions must have a standard signature such as:

```
function(input, next) {  
    someAsynchronousOperation(input, function(output) {  
        // asynchronous function is done  
        next(output)  
    })  
}
```



```
}
```

Listing 16: Sequential function signature

This already provides more clear and better control over the data flow than plain callbacks. However, it does require a certain format and there are some other drawbacks such as not being able to change the next step in the sequential chain dynamically.

The other common utility function is parallelization. Let's say we are developing a system that does some magical interweaving of the tweets a person wrote on [Twitter](https://twitter.com)²⁴ and the shots that that person made on [Dribbble](http://dribbble.com)²⁵. We will do some Ajax requests from the person's browser to the respective web services to retrieve the information. We obviously want these requests to happen simultaneously and not sequentially.

```
getTweets('email@example.com', function(tweets) {
  // We have the tweets, but do we have the shots?
})
getShots('email@example.com', function(shots) {
  // We have the shots, but do we have the tweets?
})
```

Listing 17: Simultaneous Ajax requests

From the example you can see the problem that arises: we want to know when both of these requests have completed. We could manually solve this by using a counter that keeps track of the number of requests that have finished.

```
var tweets, shots, counter = 0;
getTweets('email@example.com', function(t) {
  tweets = t;
  if (++counter === 2) {
    // we have both the tweets and the shots
  }
})
getShots('email@example.com', function(s) {
  shots = s;
  if (++counter === 2) {
    // we have both the tweets and the shots
  }
})
```

Listing 18: Parallelization with a counter

Note: This code does not suffer from race conditions because JavaScript is single-threaded.

This method is of course prone to error so we would like to automate this manual synchronization. It is possible to write a utility function that keeps track of the number of responses for us and then rewrite our code to something like this:

```
parallelize([
  function(callback) { getTweets('email@example.com',
    callback); },
  function(callback) { getShots('email@example.com',
    callback); }
], function(tweets, shots) {
```

²⁴<https://twitter.com>

²⁵<http://dribbble.com>

```
// we have both the tweets and shots
});
```

Listing 19: Parallelization with callbacks

Note that the `parallelize` function expects an array of functions with signature `function(callback)` where the callback has the signature `function(result)`. Again, this method is better than vanilla callbacks but it is still very inflexible because of the specific function signature and because it still does not solve the observer problem.

5.3.3 Introducing promises

A promise is a synchronization construct. The basic idea can be demonstrated with a few lines of JavaScript code by defining a function that returns a promise object. The example below is such a basic but naive implementation.

```
function defer() {
  var callback;
  return {
    resolve: function(value) {
      callback(value);
    },
    then: function(_callback) {
      callback = _callback;
    }
  };
}
```

Listing 20: Naive promise implementation

The above function allows us to defer the resolution of a value. We can use it to make a promise as follows:

```
function login(username, password) {
  var promise = defer();
  // Does an asynchronous database query
  database.login(username, password, function(user) {
    // This function is called when we have a response
    promise.resolve(user);
  })
  // Immediately return the promise
  return promise;
}
```

Listing 21: Using defer to wrap a callback in a promise

The function `login()` uses `defer()` to create a promise. The promise is resolved after the user is logged in to the database. It essentially wraps the `login(username, password)` function in a promise. To demonstrate how to actually use the `login()` function see the listing below:

```
var result = login('dragonSlayer14', 'D3a1*d#!');
result.then(function(user) {
  // print a message
  console.log(user.name + ' has logged in');
});
```

Listing 22: Using a function that returns a promise

From the code you can already see that the control is un-inversed. The `login(username, password)` function immediately returns a value. This value is then

5 IMPLEMENTATION

used to attach a function that is called when the result is ready to be used. With synchronous functions, it is often desired to have a useful return value. The following function is a simple example of this.

```
function add(a, b)
```

If we assume that the add computation must be done asynchronously, the function would look like this:

```
function addAsync(a, b, callback)
```

Notice that addAsync does not have a useful return value. If we were to use promises, the function would look like this:

```
function addPromise(a, b)
```

It has the same signature as the normal add function and its return value is a useful value. The only difference is that it will be resolved in the future.

So with the small amount of code in Listing 20 we already achieved an improvement over using plain callbacks. There are however some major drawbacks to our little `defer()` function that need to be addressed:

1. It does not allow you to attach multiple observers.
2. The promise can be resolved multiple times.
3. A promise handler is not run if it is attached after the promise has been resolved.
4. It still does not provide an improved way of handling sequentialization.
5. There is no explicit error handling.

The first three drawbacks are quite easy to remedy. The fourth and fifth drawbacks however are not.

In order to get a deep understanding of not only how promises are to be used but also of how they should be created, we decided to write our own version of a promise library that is directly integrated into AngularJS. It was developed with the design rationale of Q, which is well documented at <https://github.com/krisKowal/q/blob/master/design>, in mind. Doing so was a very valuable experience which led us to gain a better understanding of common pitfalls in asynchronous programming and of how promises can be used.

We decided to leave our own promises library in the project and use it instead of Q. Our reasoning behind this was that our own promise library was already integrated into AngularJS and that it covered our needs. An additional benefit was that we knew very well how to use it. An example of how we use promises can be found in the auth service. Listing 23 serves to illustrate how promises play a role there. Note that the actual auth service code is more powerful and correct than this simplified version.

```
// obtain a reference to our module
var myModule = angular.module('myModule');

// Register the auth service as a part of myModule
myModule.factory('auth', ['database', 'promise', function(
    database, promise) {

    // Some utility functions
    function getError(errorCode) { /* ... */ }
    function setUser(userData) { /* ... */ }
```

5 IMPLEMENTATION

```
function clearUser(userData) { /* ... */ }

var user = {}, auth = {
  authenticated: false,
  login: function(username, password) {
    return promise(function(resolver) {
      // Check if we are already logged in
      if (auth.authenticated) {
        // Immediately resolve and return
        resolver.resolve(user); return;
      }

      // Asynchronous database request to log in
      database.login(username, password,
        function(errorCode, user) {
          if (errorCode) {
            clearUser();
            resolver.resolve(getError(
              errorCode));
          } else {
            setUser(user);
            resolver.resolve(user);
          }
        });
    });
  }
};

return auth;
})();
```

Listing 23: Promises in AngularJS

The parts of Listing 23 that we are interested in now are where the promise is created and where it is resolved. Our promise library allows us to create a promise with the following syntax:

```
function makePromise() {
  return promise(function(resolver) {
    // Do something asynchronous here ...
    // and eventually resolve it.
    resolver.resolve('The value');
  });
}
```

The following code uses `makePromise()` and attaches two observers, one logs the value to the console and the other creates a dialog box with the value as the message:

```
makePromise().done(function(value) {
  console.log(value); // Log the value to the console
}).done(function(value) {
  alert(value); // Print a message box with the value
})
```

The use of our `.done(callback)` function is similar to that of Q's `.then(callback)` but the implementation and effects are actually totally different. The Q library allows you to sequentialize promises by using a chain of `.then(callback)` calls. The callbacks are required to return a promise if you wish to do so. Our library allows you to sequentialize your promises more explicitly

using `promise.sequentialize(promisors)`. This function accepts an array of functions returning a promise. These functions returning a promise are called promisors. The promisors are executed after the last promise has finished in order to sequentialize the execution of the promises.

For parallelization, Q provides the function `Q.all(promises)` and our library provides `promise.parallelize(promises)`. Both of these functions return a promise that is resolved after all of the passed promises have been resolved.

In conclusion, the Q library is a very complete and well tested library for promises in JavaScript. Using the lessons that the developers of Q learnt and documented we created our flavour of promises. Our own library covered the needs we had for this application and the simplicity of it made using and debugging easy for us. If needed, it is possible to switch to using Q but there was not enough reason to do so during the project because of the reasons mentioned before and because there were a lot more pressing concerns.

5.4 DizzyData API communication

To connect the front-end with the DizzyData system API calls are used. Because DizzyData utilizes a web API, these calls go through HTTP request which are made to the server. AngularJS provides with a standard function, `$http`, that sends a HTTP request to the specified url. An example of this function is shown in Listing 24, here a 'GET' request is made to '/someUrl'.

```
$http({method: 'GET', url: '/someUrl'}).
  success(function(data, status, headers, config) {
    // this callback will be called asynchronously
    // when the response is available
  }).
  error(function(data, status, headers, config) {
    // called asynchronously if an error occurs
    // or server returns response with an error status
  });
```

Listing 24: `$http` call in AngularJS

Because several different types of requests are available, the `$http` function takes input from an object containing these elements: `method` (required), `url` (required), `data` (optional), `params` (optional), `headers` (optional). `Method` tells the function what type of action should be performed, more about this in Section 5.4.1 about CRUD. The `url` points to the location where the request should be done. `Data` and `params` contain the information that should be send to the server along with the request. The `headers` can be used to define the format of the data that is send with the request and the requested format in the response, for instance JSON.

For our implementation we decided to build a service that manages all API calls. This way all calls could be handled in the same manner and information such as the API url could be kept in one single place. The `$http` function was wrapped in a promise, as described in Section 5.3, to have full control over the form of the response.

5.4.1 CRUD

CRUD stands for “Create, Read, Update and Delete”, these are the four different operations which can be done on objects or elements. In our case the methods

5 IMPLEMENTATION

that can be performed on the API resources are called: POST, GET, PUT and DELETE respectively.

POST allows the user to create a new object and send the details of that object to the server in the data part of the HTTP request.

GET allows the user to retrieve the resources that satisfy possible characteristics given in the params part of the HTTP request.

PUT allows the user to edit an existing resource by sending the new details in the data part and an identification number of the specific resource that needs to be updated.

DELETE allows the user to remove a resource by specifying the identification number of that resource.

These four operations were turned in to separate functions, which in turn all return a promise object that contains the actual HTTP request. This way a call can be made as in Listing 25, where a GET request is performed on the Jobs resource, with params: {'limit': 25, 'offset': 0}.

```
ddAPIService.get('jobs', {'limit': 25, 'offset': 0}).  
  done(function(response) {  
    // Handle the response  
  })
```

Listing 25: An API call using our own function and promise return

5.4.2 Authentication

As mentioned earlier in Section 4 the API uses an authentication system called OAuth. For this specific project the API needed to be extended to support authentication through a username and password combination. When the user is correctly authenticated, the API returns an access token that must be used in later calls to verify the identity of the user. These tokens expire after a given amount of time and therefore need to be refreshed at regular intervals. Also when a user logs in through another system or someone else uses the same credentials to log in and receives a new access token, the old token becomes invalid.

```
auth.login = function login(username, password) {  
  return promise(function(request) {  
    $http({method: 'POST',  
      url: 'http://dev2-api.dizzydata.com/v1/oauth2/  
        token',  
      data: {  
        'grant_type': 'password',  
        'scope': 'test',  
        'username': username,  
        'password': password  
      },  
      headers: {  
        'Accept': 'application/json',  
        'Content-Type': 'application/x-www-form-  
          urlencoded'  
      }  
    })  
  })  
  // ...  
}
```

Listing 26: Part of the access token request in auth.js

The authentication process was implemented in a separate service called auth.js, this was done because the request to the API for an access token is fundamentally different from the normal requests. Also the fact that the normal

5 IMPLEMENTATION

request need the access token to function and not the other way around, means that separating these two types of API calls is a logical choice. The `auth.js` service handles not only the access token request, but also stores this token in a cookie and manages the login dialog when needed. By keeping these parts of the authentication in one simple service, all other methods that require a user to be authenticated can simply check with this service to see whether that is the case. In Listing 26 a part of the token request can be seen, using the username and password as data to send to the API.

Because we had to take into account that the token could become invalid in the process, the response from the API needed to be checked for this error specifically. When the token was expired, the current request would be discarded and a new request for a token should be made. To prevent that a request is discarded and possibly lost when the token is expired, we implemented a function that requires the user to login again when their request gets denied. In the case that an access token is expired, the API responds with a 401 `StatusCode`, which indicates that access was denied to that resource. In Listing 27 part of the `$http` request function is shown illustrating how the request is repeated after successfully entering credentials in the login dialog.

```
$http(/* parameters */).
  success(function(response) { /* handle success */}).
  error(function(response) {
    if (response.StatusCode === 401) {
      // require the user to login again
      auth.requestLogin().done(function() {
        request(result) // repeat the request
      })
    }
  })
})
```

Listing 27: Part of the `$http` request method, showing the 401 error handling

Listing 28 shows the `auth.requestLogin()` function that prompts the user with a dialog. The dialog returns a promise that is resolved when the user is authenticated and received a new access token. This allows us to use this dialog to verify the login and repeat an API request when the token has been renewed.

```
// Returns a promise that is resolved when the user has
// logged in.
auth.requestLogin = function requestLogin() {
  if (requestLoginPromise) { return requestLoginPromise
  }
  return requestLoginPromise = promise(function(resolver) {
    auth.logout()
    requestLoginResolver = resolver
    loginDialog.open('/partials/login-dialog.html', '
      LoginDialogController')
  })
}
```

Listing 28: `requestLogin()` function that request the user to login through a dialog

5.5 Models

As explained in Section 4 we designed models to be able to visualize the resources of the API in the interface. Most models could be an almost direct representation of the objects returned by the API, but some special cases were handled in the implementation of these models, which we will discuss here.

5.5.1 Basic models

The basic models are the models that did not require much transformation to be displayable in the interface. These models contain some similar functions which we will use to illustrate the implementation of the models.

Most models contain their own call to the API service enabling the models to be retrieved from or send to the DizzyData API. These functions were used mostly in the higher level models, as shown in the class diagrams in Section 4.5.2. With higher level models we mean models that contain collections of other models. It makes sense to let these models retrieve the information from the API, because then the resources that belong to that specific model can be added instantly. For example: we use the Jobs model to represent all Jobs, JobResultFiles and JobResults. The Jobs model has all the functionality to retrieve the Jobs from the JSON returned by the API and add all these Jobs to the collection of Jobs. The same is done with the JobResults and JobResultFiles at the level of the Job.

Another implementation decision shared throughout all models is the decision to use the JavaScript object literal “{}” to represent the collection of sub-elements rather than using an array. For example: the WorkflowTranslator object contains a list of Steps, which are added to this object, with their StepID as the object key. This allows us to invoke a specific Step from this list using `WorkflowTranslator.steps[stepID]`. This functionality comes in very handy when a specific element needs to be retrieved from such a list when only the id is known, more on this in the following Section about Workflow translation.

5.5.2 Workflow translation

Translating Steps and Workflows to the displayable Worker object was one of the more complex tasks to perform. As discussed in Section 4 on System Design, the Worker would represent a collection of Steps, with input and output Steps and the most important Step: the main Step. This main Step is the Step that defines the work that is done by that Worker, the other Steps are of secondary importance to the user. The translation to Workers is performed the following way:

1. We loop through all Steps, to identify the main Steps.
2. For each main Step a new Worker object is created.
3. Each Worker retrieves a list of input Steps and a list of output Steps.
4. The Workflow to which the main Step belongs is retrieved to add details to the Worker.
5. The Worker is added to the list of Workers.

Action 3 is an interesting part of this translation. Retrieving all output Steps is an easy task, because these are just Steps that are in the same Workflow which are of the type output. Retrieving all input Steps on the other hand can be more challenging, because input Steps that monitor storage services can be placed in

another Workflow and then point to the Workflow that does the document processing. Listing 29 shows what check needs to be performed to verify if a Step is an input to a specific Workflow.

```
isInputTo: function(wfID) {
    return this.isInput() && (this.workflowID === wfID ||
        this.settings['OutputWorkflowID'] == wfID)
}
```

Listing 29: Verifying whether a Step is input to a specific Workflow

A special case occurs when a Workflow has multiple main Steps, in this case multiple Workers are created, but in their respective in- or output they point to the previous or next Worker instead of a list of Steps.

Action 4 is a good example why we used the ids of resources as keys in the listing of these resources. The Workflows are stored in an object in the WorkflowTranslator and the Step contains an attribute WorkflowID, pointing to the Workflow to which that Step belongs. Instead of searching for the Workflow, it can be easily retrieved using: `WorkflowTranslator.workflows[step.workflowID]`.

In first instance we sketched a system that would transform these Workflows and Steps into Workers, then allow the user to edit the Workers and transform these back to be send to the API. But JavaScript limited us in this aspect, because there is no easy method to save multiple variables between different pages in a JavaScript environment without the use of databases. Therefore we decided an another implementation approach.

The Worker objects would be used purely for visualization and generated by retrieving all Workflows and Steps from the API. When the user edits a Worker, the user is actually editing the specific Step belonging to that Worker. Because all changes that can be made to the Worker object are related to either settings in the Workflow or Step, these changes can be pushed directly to these resources. The same goes for updating an output or input Step. Elements that belong to the main Step can also be edited directly, because they are directly related to that Step and need no translation to be visualized to the user.

5.6 The document previewer with OCR support

The idea behind the front-end is that you teach DizzyData how to process your documents. Making it easy for the user to do this was one of our main goals so we put a lot of time in thinking of ways to realize this. One of the ideas we had was to let users upload test documents: documents that are instances of the type of document that needs to be processed. By letting the user make recognition rules based on the test document combined with its OCR data should be a very natural way of guiding the system through your document. Additional benefits would include the visual editing of search areas and easily creating recognition rules by clicking on the overlying OCR data.

So what we needed to build was our very own document previewer with OCR support. It had to work in older browsers and it had to be fast enough for small to medium sized documents. The previewer also had to be extensible because new features are constantly being added to DizzyData some of which we might have to support in the future. Next to these project specific requirements we also needed our document previewer to have standard features such as scrolling, pagination, zooming and searching. Some of the features were surprisingly easy to implement with the help of AngularJS and others proved to be quite a challenge. The complete previewer is shown in Figure 16. On the left you can see some Elements and the block with the toolbar on the right is the previewer.

5 IMPLEMENTATION

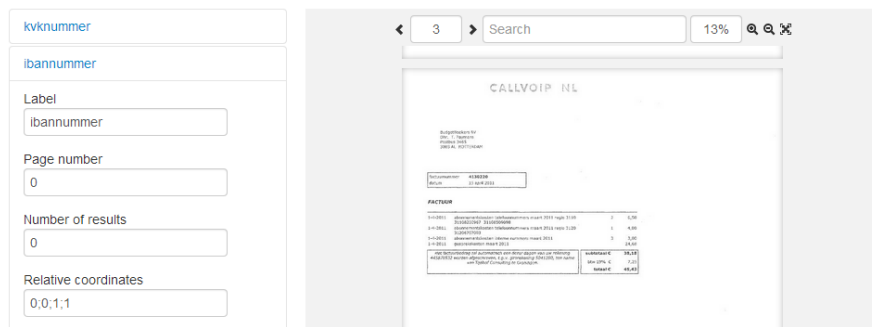


Figure 16: The previewer in action

This section starts off with a description of how the previewer is built up. Next, the most important aspects of the previewer are discussed in detail. Possible improvements on the previewer are listed at the end.

5.6.1 Laying the foundation

There are a couple of things we required in order to realise the previewer.

1. Images of the pages
2. Page meta-data
3. OCR data usable in JavaScript

It is obvious to see why the browser needs to have access to the images of the pages. The OCR software used by DizzyData takes a images or a scanned documents and does some processing on them. It outputs the generated images per page, as well as the extracted OCR data.

It is useful to have some additional meta data about each page such as the page number in the original document, the url and the width and height of the page image. Why this is true for the width and height is discussed in Section 5.6.3.

The OCR software provides a lot of data on each recognized character and on character groups. For our purposes we needed only a small subset of this information. We wanted to know the place and text of every word found on each page.

These DizzyData API requirements led to the JSON specification in Listing 30. Note that it is not valid JSON code. The `<-` denotes a textual definition rather than a value.

```
version: <- Version number of the JSON format as a string
           in the form of "major.minor.patch", will be helpful if
           we need to change the format
pages: [
{
  image: <- Full path to the image as an url
  width: <- The width of the page in pixels as a number
  height: <- The height of the page in pixels as a
            number
  number: <- The page number as a number
  words: [
    {
```

```

    left: <- distance between the left side of the
              page and the left side of the word in pixels
              as a number
    top: <- idem for top side and top side of the word
    right: <- idem for left side and right side of the
              word
    bottom: <- idem for top side and bottom side of
              the word
    value: <- the textual value of the word as a
              string
  }, ... <- the rest of the words if there are any
]
}, ... <- the rest of the pages if there are any
]
```

Listing 30: OCR data specification

The DizzyData API is not capable of providing OCR data in this format yet. For the duration of the project we have used an example document for which we manually created a JSON file that met the specification. At a later time this feature will probably be added to the API.

5.6.2 Navigating through the document

Navigating through the document is an very important aspect of a document viewer. When using a web browser you usually scroll by using the scroll wheel, pressing the space or arrow keys or by using the find function. For document viewers however, we noticed by experience that other forms of navigation are much more common.

Lets first talk a bit about how the display of the previewer actually works. We will not discuss it in detail because it is a common technique. Imagine a big plate that hold all of the content – in our case a series of images that are horizontally aligned on the center and vertically placed beneath each other. Now imagine a rectangular hole that has a fixed position through which you can look at the plate. If you want to view another part of the plate, you have to move the plate since the position of the hole cannot be changed. Just keep in mind that you have to move the plate in the opposite direction of what you want to view.

In our application the plate is called the “screen”, which might not have been the best choice, and the hole is the “viewport”. The viewport element is the parent of the screen element. The offset of the screen relative to the viewport is what determines what you can see. Vertically and horizontally scrolling now amounts to simply modifying respectively the vertical and horizontal offset.

So what user actions should trigger this scrolling behaviour? Well, obviously the mouse wheel and the arrow keys should work. Also, most document viewers often have some form of pagination and a find function, some even have bookmarks, automatic indexes and thumbnails. Having a basic form of pagination in addition to the mouse wheel and the arrow keys should be sufficient, or so we thought.

While viewing our test document we noticed that we tried to use the click-and-drag method to navigate through the document. This occurred mostly at times when the page was larger than the viewport which meant that we needed to scroll horizontally. It was possible to scroll horizontally by holding the `SHIFT` key combined with the mouse wheel, but somehow we subconsciously kept trying to use the click-and-drag method.

5.6.3 Adding pagination

Pagination is a standard feature of document viewers. Making sure that the correct page number is visible and letting the user jump to the next, previous or a page of their choice is the job of the “pager”. All these functions are nicely grouped together on the left side of the previewer toolbar as you can see in Figure 17. The pager exposes functions to the view that make it easy to go to the next, the previous or a specific page.

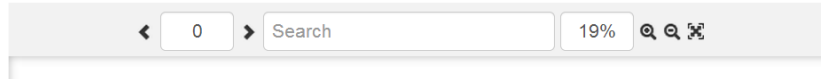


Figure 17: Previewer toolbar

The pager is a part of the previewer module. In order to successfully do any of its tasks, the pager must know what page is currently being viewed by the user. The most common rule used to determine this is by finding the page closest to the centre of the viewport. Listing 31 contains the pseudo code for finding the page closest to a y-coordinate.

```
function closestPage(pages, y)
  min = POSITIVE_INFINITY
  closest = null
  foreach page in pages
    if top or bottom of page is closer to y than min
      closest = page
  return closest
```

Listing 31: Finding the closest page

Since we have access to a list of pages sorted by vertical position we could use a binary search algorithm. For our purposes this was too complex the speed gain with respect to a simple iterative version is negligible. Since the bottom of a page is calculated by getting the top offset and adding its height we can think of another optimization. By visiting the pages from bottom to top we can prevent the expensive CSS lookup of the height property and only do a single check per page until we passed the y-coordinate. This might be added in the future but during development we were focused on getting it to work.

The `closestPage` function is used by the pager every time the screen offset changes to make sure that the “current page” input always contains the correct page number. This is easily realized using the `$watch(value, callback, deepCompare)` function provided by AngularJS. The function essentially allows you to keep an eye on an object or value and do something when it changes.

Currently the pages are contained in a zero-indexed array. This means that to go to the first page you will have to type “0” in the input box. The second page is then “1” and so on. In the future, the pager should use the page number specified in the page meta-data contained in the document JSON.

5.6.4 Implementing zooming

Another key component of the previewer was the zooming functionality, hereafter referred to as the “zoomer”. Figure 17 and 18 already reveal that we managed to implement this requirement but it was definitely not easy. There were a number of requirements for the zoomer of which the most important ones were that the zoomer must a) be able to zoom in and out; b) be able to find a zoom level for

5 IMPLEMENTATION

which the page fits exactly in the viewport; c) work in all major browsers that support at least CSS 2.0; d) zoom relative to the cursor position or the centre of the previewer if the cursor is not present; and e) not depend on the styling of the pages with exception of the fact that they are placed above one another. The first two are not as interesting as the other three so we will not discuss them in detail.

Not being able to use CSS versions later than 2.0, requirement c, meant that we could not use the CSS 3.0 zoom property. We had to create our own solution which used the automatic scaling of images to their element's width. This allowed us to simulate zooming by scaling all the elements in a page and the page itself using the zoom factor of the previewer. The elements in a page include the OCR data, see Section 5.6.5 for more information on this subject and how it handles zooming. The page elements are actually AngularJS directives that watch the zoom level provided by the previewer.



Figure 18: Fit page button

Figure 19 serves to illustrate requirement d. Since almost every application that allows you to zoom takes care of this it feels very odd when you encounter an application that does not. Cursor-aware zooming has to work perfectly because when it does, it feels very natural. Even to people who are not used to zooming.



Figure 19: Zooming towards the cursor

Requirement e made realizing the cursor-aware zooming even more difficult. The fact that the styling can change meant that we could do no assumptions on the size, margin and horizontal position of the pages. The only data we could fetch were the position and dimensions of the pages. Figuring out an algorithm solved only half of the problem since it had to be implemented using AngularJS. You might ask yourself why it was harder because we were using AngularJS. The answer is: AngularJS abstracts away flow-control. It is hard to specify a sequence of operations on elements that are contained in different directives because the philosophy of AngularJS is that directives, which could be described as functional elements, should update themselves.

The eventual pseudo code for zooming consist of two parts. The first part, Listing 32, handles getting the right zooming level and the second part, Listing 33, makes sure that the screen offset is corrected so that the cursor stays in place. By splitting the code up in these two parts we allow ourselves to use a step model for determining the next or previous zoom level while still allowing any zoom level to be used.

```
// a list of zoom levels
steps = [5, 2, 1, 0.5, 0.2]

// returns index of the step that is closest to level
function getClosestStep(level) {
```

5 IMPLEMENTATION

```
return steps.min(function(value) {
    // map steps to an array with these values
    return abs(level - value)
}) // and return the index of the minimum value
}
```

Listing 32: Finding the current zoom step

The `getClosestStep(level)` function is used to get the current index by the zoom in and out functions. The index is then decremented or incremented respectively if the resulting index is within bounds.

```
var mx, my, // abs coords rel to viewport
    rx, ry, // rel coords rel to page
    closest, // closest page
    numCalls // number of calls to after

/* calculates and/or stores mx, my, rx, ry and closest and
   resets numCalls */
function before() {

    if mouse is over viewport
        /* store absolute mouse position relative to the
           viewport */
        mx = viewport.mouse.x
        my = viewport.mouse.y
    else
        /* store mouse position as the center of the
           viewport */
        mx = viewport.width/2
        my = viewport.height/2

    /* translate mouse coordinates to be relative to the
       screen */
    var x = screen.offset.x + mx,
        y = screen.offset.y + my

    closest = closestPage(previewer.pages, y)

    rx = (x - page.left)/page.width
    ry = (y - page.top)/page.height

    numCalls = 0
}

/* uses the stored data to determine the correct offset
   after all the pages have been resized, the pages are
   responsible for calling this function */
function after() {
    increment numCalls
    if numCalls is not equal to the number of pages
        /* not all the pages have been resized yet: stop
           */
        return

    /* do the reverse operation of begin: translate the
       relative page coords back to absolute coords
       relative to the page and then back to absolute
       coords relative to the screen. It will absolute
```

5 IMPLEMENTATION

```
screen coordinates that are different from the
ones used in the before function because the page
dimensions have now changed. */
var nx = rx*page.width + page.left,
    ny = ry*page.height + page.top

/* using nx and ny as the screen offset would mean
that the center of zoom is the top left corner,
subtracting the stored mouse position from nx and
ny is the final touch. It moves the screen so that
the relative mouse point stays in place. */
screen.offset.x = nx - mx
screen.offset.y = ny - my
}
```

Listing 33: Correcting the screen offset

Note that the `after()` function just sets the screen offset to a new value. The scroller component takes care of clamping the screen offset to a valid value when it changes.

As can be read in the code comments, the `before()` function is run before the zoom level will change. This is illustrated in Listing 34:

```
zoom.in = function() {
  before()
  var index = getClosestStep(zoom.level) - 1
  if index is in bounds
    zoom.level = steps[index]
}
```

Listing 34: Usage of `before()` by the zoomer

The `after()` function is called from the page controller which is part of the page directive. Listing 35 shows the pseudocode of a piece of the page controller:

```
$watch(
  function() { return zoom.level },
  function(level) {
    /* the zoom level has changed and the level
       variable now holds the new value */
    image.css({
      width: page.originalWidth*factor,
      height: page.originalHeight*factor
    })
    /* notify the zoomer that this page is done
       resizing */
    zoom.after()
  })
})
```

Listing 35: Using `after()` in the page controller

The page controller watches the zoom level for changes and when it does, it updates the size of the image element and notifies the zoomer that the page has been updated.

This solution may not be the prettiest but it does its job very well. The translation to and from relative coordinates in the space of the page closest to the cursor algorithm works really well.

5.6.5 The OCR overlay

The OCR overlay has one purpose: easing the editing of Workflows. It shows what words have been recognized by the OCR system.

The overlay consist of elements that are positioned relatively to the page. Each word has its own element and contains the textual value of the word and a piece of context menu code. The words become visible when you hover over them with the mouse. A currently non-functional context menu will open if you click on a recognized word as is shown in Figure 20.

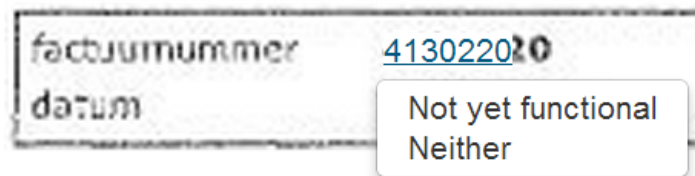


Figure 20: Utilizing OCR data by creating an interactive overlay

The word elements are AngularJS directives that behave similar to the page elements; they watch the zoom level provided by the previewer and scale their properties when it changes. The difference is that the words do not need to notify the zoomer of their size changes because they do not effect the size of the screen, they are contained within their page. The properties that need to be scaled are the word position, dimensions and the font size.

5.6.6 Improvements

There is a lot of room for improvement on the previewer code. For example, we noticed that AngularJS is not particularly suited for these more complex implementations. It might be a good idea to create a version of the previewer that does not use AngularJS but just JavaScript with some utility libraries. This is not worth the effort unless the zooming code breaks badly for some reason. There are however some other improvements that can and probably will be realized in the near future.

searching

An important requirement was being able to search through the document by using the OCR data.

context-menu

The context-menu that is opened when you click on a word is not functional at the moment. We will have to see what options can be useful there. For now it is a proof-of-concept.

selecting areas

One of the ideas was to be able to visually create and edit areas in the previewer. There are a number of recognition rules that allow you to specify an area in which the rule applies by providing the coordinates. Doing this graphically will be much easier.

click-and-drag

Something you *will* try to do while working with the previewer is using the mouse to drag the document around, unfortunately this does not do anything at the moment. By plain scrolling you can achieve the same results

5 IMPLEMENTATION

but dragging feels much more natural when the page you are looking at does not fit in the viewport.

scrollbars

It is, apart from the page number, hard to get a sense of where you are in the document. We wanted to have full control over the scroll bars so we used our own display solution. Having smart overlying scrollbars that appear only when it feels useful would be a great addition as well.

page number

The pager should be using the provided page number data. The pages should be sorted by their number. The implementation is halfway done. More pressing tasks prevented the continuation.

fit to closest page

It would be nice to have the “fit to page” button and hot key change the zoom factor so that the closest page fits precisely in the viewport. This is more of a nicety because it will not be useful in case the document pages are all about the same size, which is often.

event based

It may be worth the effort to rewrite the previewer to be more event based instead of watch based. By doing so we provide better control of when update handlers need to be run. It also allows you to cancel the events, a functionality we would have loved to have during development.

help screen

Something should trigger a help screen where the goal of the previewer, its functionalities and a description of how to use it is given along with the respective hot keys if there are any.

6 Code quality

6.1 Testing

As we explained in Section 4.6, being able to easily test components of our application was one of the reasons why we chose to use AngularJS. While it is quite easy to write tests once everything is set up, we did have some trouble configuring the required tools. In the end, we managed to get a decent set-up that could be easily installed on other machines. This section provides an short overview of all the tools that we had to use in order to write and execute tests.

6.1.1 JavaScript applications with Node.js

*Node.js*²⁶ is a cross-platform application built on Chrome's JavaScript runtime. It allows you to run JavaScript without a browser. This JavaScript code can interact with the file systems and networks through the cross-platform libraries that Node.js provides. The Node.js libraries are event-driven and non-blocking which, in combination with the single-threaded nature of JavaScript, allows you to write efficient enough data-intensive applications with ease.

For our project we needed a simple fileserver. We chose to use Node.js in combination with a module called *Express*²⁷ to generate a fileserver. This fileserver serves HTML, JavaScript and CSS files. Additionally, it automatically compiles *Stylus*²⁸ files into CSS for us. Stylus is a stylesheet language that is more powerful than plain CSS. It allows you to for example use variables, math expressions and abstract away browser inconsistencies.



Figure 21: Node.js

6.1.2 Dependency management with NPM

As you might imagine, a software development tool like Node.js would benefit from having a good dependency management tool and *Node Package Manager*²⁹ (NPM) does exactly that. NPM is a package manager that allows you to specify meta information on your Node.js application or, in NPM terms, your package. This information is to be stored in a JSON file called "package.json" that must be placed in the root directory of your package. The file describes properties such as the name of your package, who the author is, where its readme can be found, what other packages your package depends on to run and what other packages your package depends on when you want to develop it. Listing 36 shows our package.json at a certain time during the project.



Figure 22: NPM

```
{
  "name": "DF",
  "version": "0.3.6",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "~3.2.4",
    "stylus": "~0.32.1"
  },
}
```

²⁶<http://nodejs.org>

²⁷<http://expressjs.com>

²⁸<http://learnboost.github.io/stylus>

²⁹<https://npmjs.org>

```

    "devDependencies": {
      "grunt": "~0.4.1",
      "grunt-contrib-jshint": "~0.5.4",
      "grunt-karma": "~0.4.4",
      "istanbul": "~0.1.35",
      "grunt-docular": "~0.1.1"
    }
  }
}

```

Listing 36: package.json

The developers of Node.js noticed that NPM became inseparable from Node.js. As a result, the latest distributions of Node.js have NPM bundled with it so that you do not have to install it separately.

6.1.3 Automation with Grunt

*Grunt*³⁰ is a JavaScript automation tool that runs on Node.js. The installation of Grunt is very easy, especially if you already have Node.js set up on your machine. There are literally tons of Grunt scripts available because of its huge user base. Authoring your own Grunt plug-ins is easy and publishing them is done simply through NPM.

We used Grunt to automate testing, lint our JavaScript and generate documentation. Grunt is configured by creating a Node.js module and save it as “Gruntfile.js” in the root directory of your project. The fact that the configuration is actually a module that is loaded by Grunt means that you have way more flexibility than with an ordinary JSON file.



Figure 23: Grunt

6.1.4 Testing with Karma and Jasmine

We used *Karma*³¹ as a test runner for our JavaScript code. It was developed by the AngularJS team. Karma is just a test runner which means that you need another tool to actually write your test suites in. The default JavaScript test write library that has good support in Karma is *Jasmine*³².

Setting up testing was not as easy as we had hoped though because there were little guides available for Karma. Most of the time they provided a “project generator” that would generate a new web application that used AngularJS, had testing set up for you and was runnable by a simple Node.js web server. We had to figure out how to create our own Grunt and Karma configurations that would fit in our project by looking at the generated applications. There were however many different application generators such as the do-it-yourself *angular-seed*³³ and *Yeoman*³⁴ and finding a good up-to-date example was tough.

The actual configuration is not hard to understand. For Karma you can create multiple Karma configuration files and include them in the Grunt configuration. The naming convention for the Karma configurations is “karma.conf.js” and they are, just like the Grunt configurations, JavaScript files.

The unit tests we wrote used Jasmine as a testing framework. Jasmine is extremely powerful and the tests are easy to read. The test results can be output

³⁰<http://gruntjs.com>

³¹<http://karma-runner.github.io>

³²<http://pivotal.github.io/jasmine>

³³<https://github.com/angular/angular-seed>

³⁴<http://yeoman.io/gettingstarted.html>

6 CODE QUALITY

by various printers. By default, the output is printed in the console by Karma which was fine for us. An example of the test results can be found in Figure 25

```
INFO [watcher]: Changed file "/home/mick/projects/df/df/public/javascripts/modules/promise.js".
Chrome 28.0 (Linux) LOG: [ 'Resolved promise -1', '1/3' ]
Chrome 28.0 (Linux) LOG: [ 'Resolved promise 0', '2/3' ]
Chrome 28.0 (Linux) LOG: [ 'Resolved promise 2', '3/3' ]
Chrome 28.0 (Linux) LOG: [ 'Resolved promise 1', '4/3' ]
Chrome 28.0 (Linux) df promise should call done on the promise created by parallel after all the promises are executed FAILED
    Expected 3 to equal 2.
    Error: Expected 3 to equal 2.
      at null.<anonymous> (/home/mick/projects/df/df/test/unit/promise.js:92:26)
    Expected spy done not to have been called with [ undefined ] but it was.
    Error: Expected spy done not to have been called with [ undefined ] but it was.
      at null.<anonymous> (/home/mick/projects/df/df/test/unit/promise.js:94:20)
..
chrome 28.0 (Linux) LOG: [ 'Resolved promise -1', '1/2' ]
chrome 28.0 (Linux) LOG: [ 'Resolved promise 0', '2/2' ]
chrome 28.0 (Linux) LOG: [ 'Resolved promise 1', '3/2' ]
Chrome 28.0 (Linux) df promise should handle both promises and promisors for parallel promises FAILED
    Expected spy done not to have been called with [ undefined ] but it was.
    Error: Expected spy done not to have been called with [ undefined ] but it was.
      at null.<anonymous> (/home/mick/projects/df/df/test/unit/promise.js:157:20)
..
Chrome 28.0 (Linux): Executed 61 of 61 (2 FAILED) (0.827 secs / 0.435 secs)
INFO [watcher]: Changed file "/home/mick/projects/df/df/public/javascripts/modules/promise.js".
Chrome 28.0 (Linux): Executed 61 of 61 SUCCESS (0.857 secs / 0.46 secs)
```

Figure 25: Karma output

The beauty of Karma is that the tests are run whenever one of the implementation or test descriptions change. This makes for a really nice experience while developing because improving your code and tests does not require you to manually re-run the tests.



Figure 24: Jasmine

6.1.5 Code coverage with istanbul

To get a sense of how well our tests covered the implementations we use a code coverage tool called [istanbul](http://gotwarlost.github.io/istanbul)³⁵. It can be integrated with Karma by using the right configuration. There is a guide on this on the Karma website. The coverage reports are generated as a local website whenever you edit the files it depends on. Figure 26 shows an example of how the output of istanbul looks for a collection of files.

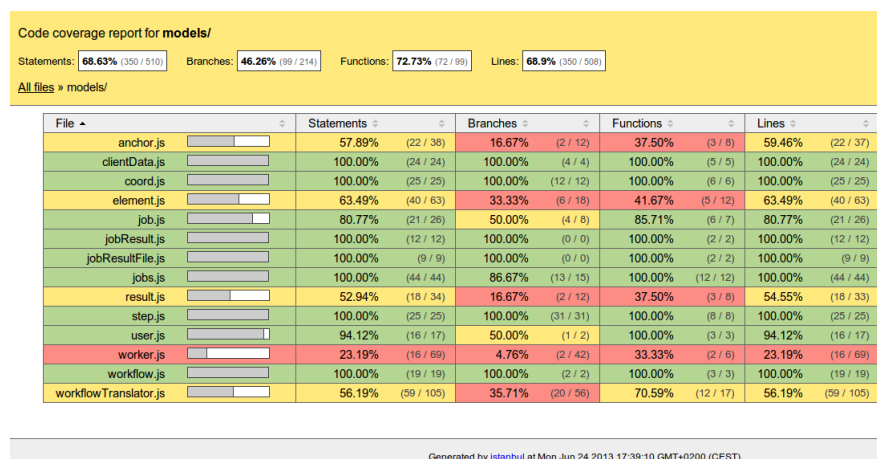


Figure 26: istanbul output

The individual files can be inspected as well to spot the lines that were not reached during the tests. These results sometimes reminded us that we needed to write additional test cases and thus have proven to be very useful. The

³⁵<http://gotwarlost.github.io/istanbul>

tool does something which computers are typically good at: doing an objective measurement (spotting untested areas) in a lot of data (the code). Although having 100% coverage does not necessarily mean that your tests are good, it does guarantee some level of quality.

6.2 SIG feedback

The first feedback from SIG can be found in Appendix A, the language of this feedback is Dutch. We will summarize the remarks that SIG mentioned and respond to these with either our corrections or justify why we chose a certain approach.

SIG mentioned that we received a score of 3 out of 5 stars, which means that our code is averagely maintainable. Our code is therefore maintainable, but could use improvement to make it easier for other developers to understand the workings of the system. The main remarks on the maintainability were:

The file structure is unclear

The files structure in the code is unclear, because many different files are in the same folder and not neatly sorted in their respective components. Due to this, the higher-level structure of the system is not clear to people who are new to the code. The advice is to distribute the files into separate functional components to make this easier.

We were aware of this problem and know that the structure needs cleaning up. The main reason why this structure became a bit messy is because we focussed on producing a working product rather than keeping a clean and neatly sorted file structure. After this feedback we cleaned up the structure, added new folders to represent specific parts of the system and keep a clearer overview of all components.

Some constructors have too many parameters

Unit Interfacing looks at the percentage of code in units with an above average amount of parameters. This normally relates to a lack of abstraction and leads to confusion and long methods. For instance the constructor in `element.js` has 21 parameters.

It is true that these constructors have a large amount of parameters, as they represent objects with many values. At first this seemed a good way to implement this, but after this feedback we came to the conclusion that this is not the case. Almost all of these constructors are only called through a function that dissects a JSON response from the API into separate variables that are send to the constructor. The decision was made to change the constructors of most of these objects to only accept JSON data as input. This reduces the number of parameters to only one and makes the constructors much more maintainable. A restriction that follows from this is that it is no longer possible to manually create an instance of these objects without writing it in JSON. This could cause complications when a new object needs to be created, while this would cause big chunks of JSON to be inserted in the code.

Some functions can also be found in common libraries

Some functions and methods that were written cover functionality that can also be found in standard libraries. For instance, in `preview.js` a min and max func-

tion are created, which are found in the standard JavaScript Math library. Also the cookie management is done in a self-created cookie.js, but there are libraries that can handle cookies for you.

There are two separate parts in this remark: the Math functions and the cookie.js. The Math functions were indeed written by our self while we could just as easily use the standard library. In this case we initially chose for our own function because many calls needed to be made to these, and we wanted the code to be neat and not include too many extra text. A better solution would be to use the standard Math.min and Math.max by changing the definition of min and max like so: `var min = Math.min, max = Math.max`. This solution is actually neater than our original solution, thus we changed our implementation to fix this.

The cookie.js file was created by us because the standard libraries in AngularJS do not provide the right options for us. We wanted to be able to store the user details and access token as JavaScript object, preferably as JSON, in a cookie. The \$cookieStore service in AngularJS is actually a service that utilizes the browser sessions to store information. Closing the browser would result in loss of data. The \$cookie service in AngularJS on the other hand does not allow for JavaScript objects or JSON to be stored to or read from a cookie. We could not find other libraries that work in AngularJS and provide the functionality we searched. This resulted in creating a custom solution which was tailored to our needs, but did implement some elements that could have been found elsewhere.

Good amount of test code

There is a promising amount of test code in the current project. This is good and SIG hopes that the test code will grow accordingly when new functionality is added. In some places, the separation of test code and functional code is unclear, this should be fixed. For example, test.json should not be in the public folder.

Test code should indeed be completely separate from functional code, but in fact this is the case. The file test.json is used by the previewer as input in order to replace JSON data which was not yet available through the API. This file will be renamed to indicate it is used as an example until that feature in the API is functional. All actually test related files are placed in the separate test folder. During the cleaning up of the file system these remarks were also be taken into account to make sure there is no unclear separation between test code and functional code.

7 Future work

During the project several deliverables and parts that were originally planned to be created could not be realized. There were different reasons why this was not possible, some were related to functionality in the API, others to a lack of time to finish that part. In the following sections we will address the parts that could not be realized, the issues that caused this and an advise on how to implement them in a later stage of development.

7.1 API improvements

As said, some deliverables could not be realized, because the functionality of the API was not yet completed. We worked side-by-side with the developers working on the API, so many small issues could be addressed quickly, but bigger issues need to be fixed later on. The main issues that still need to be fixed are mentioned here.

7.1.1 Statistics and billing

In the project description and deliverables the option to show billing and statistics was mentioned. This is an important part for the interface in its final form, but these options were not yet available in the API and are worked on at the moment. Therefore, we could not yet implement these options. In the design phase however, we did take the statistics options in to account in the design of the dashboard, shown in Figure 27. But because the exact information given in statistics was still unclear, this was not as detailed as other designs and eventually left out in the implementation.



Figure 27: Mock-up of worker edit screen.

7.1.2 Sharing Workflows and templates

Another deliverable described in the assignment was the option to get template Workflows or possibly even support a system where the user could share their own Workflows. This options was already a nice-to-have in the assignment and after puzzling on the system design it was decided that this required some major changes to the DizzyData system. The decision was based on the fact that these options require a special type of Workflows that can be retrieved by every user, but not changed by every user.

There are two options on how to implement this, either the front-end needs to provide a special collection of Workflows on a separate database or the DizzyData system needs to be changed to allow for this. The first option, to let the front-end host its own set of Workflows, we found not to be suitable, because the front-end functions in JavaScript completely and having a separate database with Workflows would cause bigger problems when changes are made to the DizzyData system.

Templates could be build in to the DizzyData system quite simply, by altering the API calls and for instance adding a special client that owns the template Workflows. This way the current infrastructure can be kept on and only minor changes are needed.

For the option to share Workflows more changes are needed. Some kind of community structure should be build where users can upload their Workflows, possibly rate other Workflows and search for Workflows that they might find useful. During the design phase this was discussed with the product manager and interaction designer and it was decided that this would be outside of the scope of the bachelors project. It still remains an option which is nice to have, but could be implemented at a later moment.

7.1.3 Edit Split and Classify Steps

The current front-end does not enable the user to edit Classify or Split Steps, an important change to the system was required to make this possible. Split and Classify work in a way that Elements can be combines in rule sets that describe the action to take when a certain set of Elements is found on a document. The current API does not allow the user to edit these rules, at the moment of writing this text this function is being added and tested. In the design process we did incorporate this functionality, but it is not yet available in the implementation.

7.1.4 Ordering Steps

Another problem in the current DizzyData API is that Steps are ordered in the way they are created. If a user would want to edit the ordering of Steps, they would need to delete and recreate the Steps involved and all Steps that come after that. This is not a pressing problem at the moment, but would be a good addition to allow more editing options and help ease the process of creating a Worker, which we will discuss later on in this section.

7.2 Front-end improvements

The previous improvements were all related to changes to the DizzyData system itself, but there are also some elements that could not be (completely) implemented in the front-end due to other reasons. Here we will list these improvements and explain what needs to be done to make that possible.

7.2.1 Creating Workflows

The current front-end does not yet enable the creation of a new Worker (or Workflow). Because the translation function that visualizes a Workflow in the front-end was a complicated function, this part could not yet be completed. In the design some basic steps were thought of that could enable this feature, but still some work remains to prevent errors.

Because of the translation to Workers, the user should be able to add a Worker and freely edit all aspects of that Worker. This means that for every edit to the Worker should be check for validity; whether it can be translated to a collection of Steps and Workflows. Also, the Steps that are kept invisible to the user should be predefined in the front-end, so they can be added immediately upon creating the new Worker. As mentioned earlier, if it would be possible to edit the ordering of Steps in a later stage, these invisible Steps could be more easily inserted when needed.

7.2.2 Storage-specific settings management

Currently it is only possible to textually edit the storage service settings which are already in place. This means a user should manually edit access tokens or secret keys that are used by some of these services. This is not very user friendly, but was implemented in this way to have at least a simple variant of this feature working. Also, for services such as an (S)FTP server, the setting fields are more easy to edit. For services such as Dropbox, the settings require much more knowledge of the service.

For adding a new storage service with these more complex tokens and keys, a graphical interface is needed that allows the user to login to their service account and retrieve these keys. This is a complex task to complete and requires knowledge of all available storage services. In this project, not enough time was available to perform the needed research and work to implement this more neatly. In a future update to the system this would be a very good and useful addition to really enable a user to edit all Workflows completely.

7.2.3 Previewer

The previewer, as mentioned in Section 5.6, allows you to view and work with a document and the words recognized on it by OCR software. Currently this document is an example image we processed manually. The final front-end should allow the user to upload their own document to use as example in this screen. One problem is that the API currently has no functionality that returns the needed information to display such an example. This is a feature which we have not yet come to a decision about with the product manager, because it should return very basic information, which not every user should be able to retrieve. And due to the nature of this project there is no way to limit this feature to be only used by our front-end.

The complete list of improvements on the previewer was already discussed in Section 5.6.6. The most important ones were searching, having a functional context-menu, allowing the visual creation and editing of areas and adding click-and-drag scroll behaviour.

7.2.4 Graphical design

Ans a final part that could not be completed in this project is the graphical design of the interface. As mentioned in earlier sections, we did manage to work with an interaction designer to improve the interaction of the system, but

7 FUTURE WORK

did not work with the graphics designer. This was mostly due to the fact that unfortunately the designer got ill and had to cancel our scheduled meeting. Of course, this was somewhat of a set back, but we did not have enough time left in the project to postpone the design until after a new meeting. Therefore we decided on working with standard graphics libraries that helped create an interface without spending too much time on the graphics.

The system and website do still need a better design and this is still an important element in a later stage. Due to the fact that we kept the design of our views quite basic it would not be too complicated to change graphical elements. Changes to the layout would bring with them some more challenges, but these should not be too great as the current layout was developed with the interaction designer.

8 Conclusion

As a final part of this report we will reflect on the work that was done, how this relates to the original assignment and whether we were successful at completing the project. First, as a recap we will summarize the deliverables, as stated in Section 2:

1. A functional and understandable front-end interface for the DizzyData REST API.
2. The interface should give a clear visualization of the capabilities of the DizzyData system.
3. The interface should function as a administrator tool to the DizzyData system, where account settings, Workflows, Jobs, statistics and billing are visually available. Without requiring the user to understand the API methods.
4. The interface should be usable by non-programmers.
5. The front-end should allow for the distribution of several template Workflows and possibly also the (publicly) sharing of user-created Workflows.
6. The front-end should work in a standalone environment, thus using only API calls for communication with the DizzyData system.

Deliverable 1 is completed for the most part, because there is an understandable interface, developed in accordance with an interaction design. Although it is not completed, because not all functionality is in place, a good strategy is ready for implementing the remaining functionality.

Deliverable 2 is, just as the previous deliverable, completed for the most part. There is a system that begins to show a clear visualization, but there is still more that could be added to visualize all capabilities of the API

Deliverable 3 also has some parts remaining, that require changes to the DizzyData system before they can be implemented in the front-end. The front-end does currently provide a way to update and view Users and Workflows and view Jobs, without knowledge of the API methods.

Deliverable 4 is successful, because no programming experience is required to operate the front-end we created. A part that still remains to be build is the option to create Workflows, this requires some more development than this project allowed for.

Deliverable 5 could not be completed, because it was decided that several changes to the DizzyData system would need to be made. These changes will be made at a later time, our thoughts on how this should be done will be taken into account.

Deliverable 6 was successful, the front-end runs in a JavaScript and HTML environment, requiring no special environment other than a simple file server to be served to a user's browser.

The product we delivered at the end of the project was an interface that allows a user to edit their Workflows. The product was not complete in the sense that there are still parts that were not finished or not functional when the project ended. The parts that we were not able to complete were discussed in Section 7, here we also explained per part why they could not be completed.

In the end we can say that the project was a succes. We created the basis for, what we believe will be, an amazing web application. Even though some parts of the system could not be completed, partly because the API did not yet

8 CONCLUSION

offer the required functionality, the application is well thought out and can be improved upon when time and resources permit. The considerable amount of work put in researching the system architecture, the application design and the tools, libraries and frameworks that were used are, together with this document and the tests, invaluable to the future developers.

References

- [1] Atlassian. URL <http://www.atlassian.com>.
- [2] Bitbucket, Git version control hosting. URL <http://bitbucket.org>.
- [3] Jira, issue tracker by Atlassian. URL <http://www.atlassian.com/software/jira>.
- [4] Software Improvement Group. URL <http://www.sig.eu/nl>.
- [5] Software Development Process – activities and steps, 2010. URL http://www.uacg.bg/filebank/acadstaff/userfiles/publ_bg_397_SDP_activities_and_steps.pdf.
- [6] Kris Kowal. Q, A JavaScript promise library. URL <https://github.com/krisnowal/q>.
- [7] Alistair Sutcliffe. Scenario-based requirements engineering. In *Requirements engineering conference, 2003. Proceedings. 11th IEEE international*, pages 320–329. IEEE, 2003.

Appendix A: SIG feedback (Dutch)

De code van het systeem scoort bijna 3 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code gemiddeld onderhoudbaar is. De hoogste score is niet behaald door lagere score voor Component Balance en Unit Interfacing.

Wat opvalt bij het bekijken van de code is dat er geen duidelijke componentenstructuur zichtbaar is op het file-systeem. Dit maakt het voor een ontwikkelaar in eerste instantie lastiger om een algemeen beeld te krijgen van de functionaliteit die het systeem aanbied. Wij raden aan om kritisch te overwegen om de code in verschillende (functionele) componenten op te delen om zo een eerste indruk te geven van de high-level structuur van het systeem. De meting Component Balance kijkt naar de component-indeling en de verdeling van het codevolume over de componenten, en jullie scoren hier momenteel ondergemiddeld.

Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden. In jullie code gebeurt dit in bijvoorbeeld `element.js` (21 parameters), `workflow.js` (11 parameters) en `anchor.js` (11 parameters).

Wat tijdens het bekijken van jullie code verder opviel is dat er in een aantal gevallen code is geschreven voor functionaliteit die normaal gesproken door libraries wordt afgehandeld. Voorbeelden zijn het beheren van cookies in `cookie.js` en de functies `min/max` in `preview.js`. Voor algemene functionaliteit als deze is het beter om een library te gebruiken, je hoeft de code en bijbehorende tests dan zelf niet meer te schrijven. Daarnaast heeft een library meer gebruikers, waardoor de kans groter is dat obscure randgevallen goed afgehandeld worden.

Over het algemeen scoort de code gemiddeld, hopelijk lukt het om dit niveau te verhogen tijdens de rest van de ontwikkelfase.

De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt. Het valt ook op dat de opdeling tussen productie- en testcode op sommige plaatsen wat rommelig is. In grote lijnen hebben jullie dat goed aangepakt, maar ik zie in de public-directory een `test.json` staan.

Appendix B: DF – Project Approach

DF – Project Approach

Mick van Gelderen
4091566

Mick de Lange
1534068

June 27, 2013

Contents

1	Introduction	2
1.1	Company	2
1.2	Project background	2
2	Task description	3
2.1	Client	3
2.2	Problem definition	3
2.3	Objectives	3
2.4	Assignment formulation	4
2.4.1	Deliverables	4
2.5	Preconditions	4
3	Software development approach	6
3.1	Initial planning	6
3.1.1	Methods	6
3.1.2	Reasoning	6
3.1.3	Responsibilities	7
3.2	Development	7
3.2.1	Methods	7
3.2.2	Reasoning	7
3.2.3	Responsibilities	8
3.3	Deployment	8
3.3.1	Methods	8
3.3.2	Reasoning	9
3.3.3	Responsibilities	9
4	Project environment	10
4.1	People	10
4.2	Organization	10
4.3	Resources	10
5	Quality control	11
5.1	Documentation	11
5.2	Version control	11
5.3	Evaluation	11
5.3.1	Code testing	11
5.3.2	SIG code evaluation	12
5.3.3	Pilots	12
	References	13
	Appendix A: Planning	14

1 Introduction

This project will concern itself with creating a graphical interface, to facilitate end-user development in the DizzyData REST *Application Program Interface* (API). DizzyData is a new solution to digital document processing, which allows for users to develop their own document processing system. In this document we will describe what the project is about, what approaches we are going to use to achieve the project goals and how we are going to guarantee quality in the project. But first we will give a short introduction to the company which facilitates this project and the background for this project.

1.1 Company

Newviews (BudgetBoekers B.V.) is the company that develops DizzyData, Newviews is a software company which delivers a *Software-as-a-Service* (SaaS) solution to digital invoice processing. The service processes scanned invoices into accounting software solutions by third parties. In this processing the important data on the invoice is collected using *Optical Character Recognition* (OCR) and smart recognition options. This data is then sent to the third party accounting software, for the accountants to process the invoices further.

Newviews is a small company with about 10 employees, based in Rotterdam. The employees consist of two directors, two salespersons, a support employee and five developers. Currently, Newviews has many customers in the Netherlands that use the software to integrate with their accounting software. There is a growing interest in solutions like Newviews for easier processing of invoices and other documents, this is one of the main reasons to start a new project: DizzyData.

1.2 Project background

Due to this risen demand for digital document processing, not only invoices, the DizzyData project was started to provide users with the possibility to build their own digital document processing solution, using the knowledge gathered by the Newviews company. As said, DizzyData is a REST API solution which can be integrated in systems created by other companies. This API has great flexibility in the document processing *workflows*, allowing for numerous different applications to be created. With this great flexibility comes a problem. Because of the enormous amount of options and combinations of all these options, the workings of the system can seem a daunting task. The complexity not only makes it hard to keep an overview, but also to explain to potential customers the capabilities of the solution and what they can use it for.

This is where our project comes in. The *DizzyData front-end* (DF) which we are going to create should give the users a good overview of the system, and make the end-user development of a document workflow easy and clear. This helps to make the project accessible for users that lack experience in programming and development. Thereby also helping the sales team by allowing them to give a graphical demonstration of the product to potential customers.

2 Task description

For a description of the task we are going to perform, we will first describe the client and contact at the company for whom the project is performed. Next the problem definition will be given, along with objectives and the assignment formulation. Based on this description the deliverables will be listed as well as the preconditions and risks involved in the project.

2.1 Client

As said, the client is Newviews, a small software company which develops and offers document processing solutions. A more detailed description of the company was given previously. This client has requested us to explore the possibilities to create a standalone front-end to their new product called DizzyData.

Our contact at the company is Tim Paymans, founder and CEO of Newviews (BudgetBoekers B.V.). He is the main contact and guidance to the development team.

2.2 Problem definition

DizzyData is an all-in-one solution to document processing and recognition, offering many options in the workflow design by the user. The DizzyData REST API enables the user to design and create their own document workflows, edit them and look at the usage statistics of these workflows. The problem is that this not only makes the solution very flexible and adaptive, but also quite complex. To manage this complexity and provide with a easy and transparent interface for the users, a solution needs to be found. The problem challenges consist of three main parts.

The first part of the problem is making an understandable graphic representation of the capabilities of the DizzyData REST API. To visualize this, an interface is needed which shows (almost) all options and capabilities to the user. This must be done in an understandable and clear manner, but on the other hand, the versatility of the underlying software should be visible as well. The big amount of options and possible combinations makes this a difficult challenge.

The second part of the problem is that the interface should enable users to manage, edit and create workflows. The challenge lies in the fact that the interface should be usable for developers/computer experts and people with no programming experience. This requires that the end-user development environment is simple and understandable for people with no programming experience, and also allows for more complex actions for the more experienced programmers.

The third part of the problem is the challenge to make the interface function standalone, and possibly place the created interface in several cloud systems and marketplaces. For easy access to the front-end application, the client wants to explore the possibility of adding the front-end to several marketplaces and cloud solutions. This is of lower priority than creating the front-end itself, but would be a great way to create an extra marketing channel for the application.

2.3 Objectives

The objective of this project is defined as creating a front-end to use as a marketing tool towards potential customers as well as an interface to provide

2 TASK DESCRIPTION

with a less complicated way of using the capabilities of the REST API.

DizzyData is a versatile but complex system, with many possible applications. To market this and to visualize the possibilities to potential customers, even those with no programming skills, a good visualization is needed. This is the main objective for the project, required by the client. The client wants a visual interface to show customers the versatility of the DizzyData system.

The second objective is to enable customers with little to no experience in working with REST API's or programming, to manage, create and edit their own document processing workflows. This level of end-user development is the second main objective for the project. Via an accessible interface users with minimum amount of programming skills should be able to work with the DizzyData system.

2.4 Assignment formulation

The assignment is to create a user interface, which functions as a standalone visual front-end to communicate with the DizzyData API. This front-end fulfills the two main objectives, by creating an administrative environment for the user. In the project, the deliverables are more important than the time invested in the project. The project is expected to be a (almost) full-time effort in the period of end of April to June/July.

The deliverables give an overview of the specifications of the project.

2.4.1 Deliverables

1. A functional and understandable¹ front-end interface for the DizzyData REST API.
2. The interface should give a clear visualization of the capabilities of the DizzyData system.
3. The interface should function as a administrator tool to the DizzyData system, where account settings, workflows, jobs², statistics and billing are visually available. Without requiring the user to understand the API methods.
4. The interface should allow for users with little to no programming skills to create and edit workflows.
5. The front-end should allow for the distribution of several template workflows and possibly also the (publicly) sharing of user-created workflows.
6. The front-end should work in a standalone environment, thus using only API calls for communication with the DizzyData system.

2.5 Preconditions

There are several preconditions to the project which need to be taken into account. These preconditions represent the limitations of method-use and on the final result.

1. For development the agile SCRUM method is used.

¹ Understandable is defined through user tests and interviews.

² Jobs are the instances of a workflow, containing details about the processed documents and the resulting data.

2 TASK DESCRIPTION

2. The design is done in coöperation with both a graphics and interaction designer.
3. The front-end is a web-based application, which runs standalone.
4. The server side of the web application has to be written in C# using .NET.
5. The web application has to be modern in terms of web technology
6. The web application has to run on semi-modern web browsers: IE8+, Chrome 25+, Firefox 19+, Opera 12+ and Safari 5.1+ (preferably also Safari iPad).
7. The product software complies with the *Model-View-Controller* (MVC) architecture.

3 Software development approach

Our approach to the complete development process is discussed in this section. The process consists of a planning phase, a development phase and a deployment phase. The methods and techniques used in these phases are listed accompanied by why we chose to use them. Our role in each phase is discussed as well.

3.1 Initial planning

The initial planning phase consists of determining what the assignment is, who the stakeholders are what resources are available to us and under what conditions the project will be viewed as a success. There are several ways to cover these aspects. To ensure a quick and useful starting phase we have carefully selected a few techniques.

3.1.1 Methods

In order to determine what the assignment is we will be interviewing the *project manager*. This gives us a basic idea of what the project manager wants to have as a product, which is of great importance to the initial planning phase, and what he wants to achieve with the product when it reaches the deployment phase which is at least as important. The latter will be discussed in detail in section 3.3, the deployment phase. Most of the stakeholders can be identified by the project manager. The resources that are available to us should become clear from the interview as well.

When the basic idea is clear, we will use a technique called *requirement analysis* to determine the scope of this project. Common requirement analysis methods are outlined in the publication *Software Development Process – activities and steps* [9]. It gives a good overview of the aspects of requirement analysis such as *stakeholder identification* and defining *measurable goals*. The method we will employ during the planning phase and also in the development phase is based on *SCRAM* as defined in *Scenario-based requirements engineering* by Sutcliffe [11]. The complete SCRAM approach is too extensive and time intensive for our small team and time resources so we will not be able to elaborately cover every step.

Determining all the stakeholders will be done by interviewing the project manager and doing the requirement analysis. In turn, specifying the stakeholders will open up possibilities of discovering additional requirements. We want to create a non-software prototype of the interface and use it to interview some of the end-users to observe what they expect to be able to do with the system.

Once we have a detailed view of what we are building and what requirements it needs to satisfy, we can specify the *win conditions*. The product can be viewed as a success if these conditions are met when the deployment phase starts.

3.1.2 Reasoning

The reason for using a scenario based *requirements analysis* methodology such as SCRAM is simply that it has proven to work quite well. Uncovering the possible uses of a product is a good starting point for defining functional and non-functional requirements. A *functional requirement* describes what a product has to be able to do while a *non-functional requirement* describes criteria that can be used to judge the product. By writing scenarios, most of the *functional requirements* will be discovered. The *non-functional requirements* are on the other hand mostly found by interviewing the stakeholders.

3.1.3 Responsibilities

In order to successfully execute the initial planning phase we have to do a multitude of things. For the interviews, we have to contact the interviewees and schedule as well as prepare and execute the interviews. In order to properly use scenario based requirement analysis, we have to learn how to use SCRAM and devise our own strategy from it.

3.2 Development

The development phase will kick in immediately after we are finished with the initial planning. In this section we discuss a software development framework that we use to guide the development process.

3.2.1 Methods

The software development framework that we will use is *Scrum*. The definition of Scrum that we will adhere to is outlined in *The Scrum Guide* by Schwaber and Sutherland [10]. In short, Scrum is a framework to manage complex product development. It forces involvement of stakeholders and managers and makes the progress as well as what exactly is being built visible to them. This allows stakeholders to steer the product in the right direction and it allows managers to anticipate better on possible future problems.

To facilitate using Scrum we use a project tracker called *Jira* [2]. Atlassian has developed Jira as well as several other useful products such as *Greenhopper* to allow teams to efficiently plan, build and launch products and their development.

A revision control system is indispensable in software development. The revision control system of our choice is called *Git*³.

Since we are working on the software as a team it is useful to have a central place that permanently runs a git server for us. This means that it is always possible to update from and push updates to this central server. *Bitbucket* [1] provides this service for us and on top of that integrates nicely with Jira.

The documentation is written in \LaTeX which is a high-quality typesetting system. It is used mostly in scientific documents since it allows you to write mathematical formulas with relative ease once you are used to doing so. On top of that it does not let you mess with the document styling as easy as other systems.

3.2.2 Reasoning

Because of the nature of the assignment we wanted to use an iterative software development methodology. Since our project manager wanted us to use Scrum in order to gain some experience with the method within the company, we did not have much of a choice. Luckily, the agile software development method is perfectly suited for a Bachelor project because we have to periodically write reports on the project progress for our coaches. The reports can be written without much effort by using the Scrum Board and the information presented during Sprint Reviews. Also, Scrum gives insight in the progress which is vital for a project with a strict deadline. The Scrum methodology is a framework and should be adjusted to your needs. We decided for example that the *scrum sprint* length should be a week. This allows us to adjust the project coarse weekly and

³<http://git-scm.com>

prevents us from having to diverge from the Scrum methodology in such cases. These decisions are documented in *DF – Scrum Workflow*.

We use Jira as a project management application simply because the newviews has experience with using it and it satisfied their needs.

Bitbucket will stop supporting the revision control software *Apache Subversion*⁴ (SVN) and its users will have to switch over to either Git or *mercurial*. Since one of us has experience with Git and sees it as a reliable and flexible piece of software, the choice to work with Git was an obvious one.

Choosing \LaTeX to write the project documentation was because we valued consistency over visual prowess. \LaTeX ensures a consistent document markup but on the other hand does not allow you to create beautiful documents unless you put a lot of effort into it. We will satisfy our urge to make things pretty by putting our efforts into the interface design.

3.2.3 Responsibilities

Setting up the project and the accounts for the project management tools is mostly our responsibility. We will have to create a collection of resources and a small description for these resources with which someone who is new to the tools can start working on our project. This includes for example links to how to use Git and what the work flow is that we use for this project with respect to Git.

3.3 Deployment

The deployment of the product will be mostly the responsibility of the project manager. However, to comply with his wishes we will have to keep a number of things in mind. Hypothetical examples of such wishes are:

1. the code needs to be maintainable by someone new to the project.
2. the support department needs to be able to inspect a client's account.
3. the software needs to be accessible through a software channel such as *google play*.

3.3.1 Methods

By interviewing the project manager we have to get an overview of what he has in mind for the deployment phase. In bigger projects, it might be useful to talk for example with the sales or distribution departments to uncover additional aspects we have to keep in mind during development. For our small scale project, the project manager will be able to provide all the product requirements regarding deployment.

A very important requirement we have to keep in mind is that this project will actually be used in a business environment. This implies that the software will have to be maintained after the deployment phase and this is not necessarily done by the original software authors. This in turn means that new developers should be able to pick up where we left off without spending a lot of time trying to figure out how the software works. To facilitate this, we will be documenting the code as well as the assignment and the choices that have been made. The exact code documentation software we will use depends on the languages that we will use. After the initial planning phase is done, we will know what languages we have to use and which code documentation tools are available for those languages.

⁴<http://subversion.apache.org>

3.3.2 Reasoning

Interviewing the project manager, stakeholders and specialists from the companies' departments seems like a logical thing to do. In fact it is, since the only other option is to guess what the requirements are. Note that this is a serious issue because it is not uncommon for developers to make assumptions about product requirements which turn out to be wrong. How succesful this approach is highly depends on the time you put into preparation and your interviewing skills.

Since we have to write a detailed description of the project and our approach to creating the product for the Bachelor project, the problem of documentation is partly solved. Adding high quality code documentation on top of that will ensure that new developers can start working on the project without much effort.

3.3.3 Responsibilities

Our responsibilities for the deployment phase are not directly present. We do have the indirect responsibility to make sure that we know exactly what the project manager is planning to do in this phase and to facilitate the means to do so or to point out problems in a timely fashion.

4 Project environment

The project environment will be covered here, we will discuss the people involved, the organization and available resources.

4.1 People

The project team consist of two team members, Mick van Gelderen and Mick de Lange, a supervisor from Newviews, Tim Paymans, and a TU coach, *yet to be defined*. The team members will perform the tasks required to fulfill the project and spend about nine weeks working on the project. They are expected to perform their tasks as an independent software developing team.

Next to the development team a graphical and an interaction designer will be working on the design of the application. These designers are located outside of the company, but there will be a cooperation between the team members and the designers.

4.2 Organization

The organization is as follows. The team members will work on the project and report back to the supervisor and TU coach. The company supervisor and TU coach will monitor the progress and serve as a feedback reference for the team members. The input and feedback from the supervisor and TU coach are used to steer the project in the correct direction. This reporting is done on a regular basis, preferably weekly with the supervisor and/or TU coach. Daily short meetings with the team members and possibly the supervisor are used to monitor the day-today progress. This organization structure abides to the schematics of the Scrum development process.

4.3 Resources

There are not that many resources needed to perform the project, those needed are mentioned here. Two workspaces in the office in Rotterdam are provided by the company, where the team members can work the whole week. Also, the necessary software and development tools are provided by the company. The team members use their personal laptops to work on during the project.

The company provides in some human resources as well. For instance the designers mentioned earlier, but also contacts with potential customers or stakeholders for interviewing purposes.

5 Quality control

Quality control is an important aspect of the project. As said, the application will be distributed and used in a business environment and needs to be maintainable by developers that were not involved in the building of the application. To guarantee the quality of the project result, several procedures and tools will be used. These focus on specific aspects of the quality assessment: documentation, version control, evaluation and pilots.

5.1 Documentation

Documentation is important to deliver a fully functional and maintainable product. The documentation should hold information on the specific design choices and implementations applied in the process. Also should there be a manual, or instruction documentation on the working of the interface. This manual should be understandable for the end-users of the product.

To create good documentation, the documentation needs to be maintained during the entire process. The Scrum sprint planning meetings, stories and issues should therefore be carefully written down, so these can be used as documentation. In accordance with this, the interviews and other discussions should be noted and recorded as well.

5.2 Version control

For version control Git is used, in a Bitbucket environment. This allows for an accurate registration of file history and version history. By using branching and other built-in git features, version control becomes an easy to manage task.

5.3 Evaluation

Evaluation consists of a few parts. Firstly, the code needs to be evaluated, tested and monitored on a continuous basis. Secondly the *Software Improvement Group* [6] (SIG) does an evaluation of the maintainability and quality of the code and gives feedback on their findings. And finally, on a higher level, user tests need to be done in the form of a pilot, or beta tests.

5.3.1 Code testing

To test the coding during the process, we will use several automated techniques to monitor the code. Unit testing will be one main part of this testing. For unit testing, extra test classes need to be created that test and verify the actual code. Since we will be performing our coding in the .NET framework, using C#, HTML, CSS and JavaScript as main programming language, we will use the following test tools.

NUnit [4] is a tool developed for performing unit tests on .NET/C# code. Unit tests are automated tests that verify the outcome of a function with the expected result. These tests can be used to check that methods behave as expected, but do require the developer to define the expected outcome manually. This tool is similar to the Junit tool (for Java unit testing) we worked with earlier in the Software Quality and Engineering course in the bachelor's second year.

QUnit [5] is a similar tool for performing unit tests on JavaScript code.

Visual Studio [7] is a developer tool that allows for debugging, coverage testing and functions as the code editor. This tool will be used to run debug tests and coverage checks on the developed code.

Moq [3] is a tool that enables developers to mock or imitate a class to control and observe interactions between classes. We will use Moq for interaction testing. A simple example would be to imitate API calls to produce either desirable or erroneous results.

W3C validation [8] can be used for testing HTML and CSS for errors and misuse. The validation will be used to confirm the correct implementation of these languages.

Browser support We will manually test several different browsers that were specified in the preconditions.

5.3.2 SIG code evaluation

SIG is a company that analyzes code, partly automated, partly by hand. This company will evaluate the created code for the bachelor project and give a good measure (1 through 5 stars) on the maintainability of the code. This is a good quality check and gives an accurate indication if the product is in fact ready to be transferred to other developers.

5.3.3 Pilots

The final, but crucial, quality control will be user testing, or pilots. This will be done at least once the project is finished, but preferably also during the project. The user will be presented with the interface and instructions and the experience will be documented. Interviews need to be taken from the test users, to monitor their expectations and findings, respectively before and after testing the environment.

The product owner will also function as a test user to control whether the results match his expectations.

References

- [1] Bitbucket, Git version control hosting. URL <http://bitbucket.org>.
- [2] Jira, issue tracker by Atlassian. URL <http://www.atlassian.com/software/jira>.
- [3] Moq, mocking library for .NET. URL <https://github.com/Moq>.
- [4] NUnit, unit testing framework for .NET languages. URL <http://www.nunit.org/>.
- [5] QUnit, unit testing framework for JavaScript. URL <http://qunitjs.com/>.
- [6] Software Improvement Group. URL <http://www.sig.eu/nl>.
- [7] Visual Studio, a .NET code development tool. URL <http://www.microsoft.com/visualstudio>.
- [8] W3C Markup Validation Service. URL <http://validator.w3.org/>.
- [9] Software Development Process – activities and steps, 2010. URL http://www.uacg.bg/filebank/acadstaff/userfiles/publ_bg_397_SDP_activities_and_steps.pdf.
- [10] Ken Schwaber and Jeff Sutherland. *The Scrum Guide*. 2011.
- [11] Alistair Sutcliffe. Scenario-based requirements engineering. In *Requirements engineering conference, 2003. Proceedings. 11th IEEE international*, pages 320–329. IEEE, 2003.

Appendix A: Planning

This is a basic outline for the project planning:

Week	Date (Monday)	Activities
1	22-04	Orientation phase Plan of approach finished
2	29-04	Orientation phase finished Orientation report finished Contacted the designers Interview stakeholders
3	06-05	First Scrum sprint
4	13-05	Second Scrum sprint
5	20-05	Third Scrum sprint
6	27-05	Fourth Scrum sprint
7	03-06	Fifth Scrum sprint SIG code evaluation
8	10-06	Final Scrum sprint SIG feedback Final code to SIG
9	17-06	Final report finished Presentation

Appendix C: DF – Orientation Report

DF – Orientation Report

Mick van Gelderen
4091566

Mick de Lange
1534068

June 26, 2013

Contents

1	Introduction	2
2	Requirements	2
3	Scrum	2
4	Git	2
5	MVC	3
5.1	The MVC model	3
5.1.1	Model	3
5.1.2	View	3
5.1.3	Controller	4
5.1.4	Communication	4
5.2	Project specific implementation of MVC	4
6	Testing methods	5
6.1	<i>NUnit</i>	5
6.2	<i>QUnit</i>	5
6.3	<i>Visual Studio</i>	5
6.4	<i>Moq</i>	5
6.5	<i>W3C validation</i>	5
	References	6
	Appendix A: Requirements Analysis	7

1 Introduction

This is the orientation report, in this report we will discuss several important techniques and methods which we will use in the project. We will discuss not only techniques that are relevant for the implementation, but also methods that are used in this project. For instance, we will be working with an agile *Scrum* project management method, which we will explain here.

The concept of the different methods and techniques are explained and references are made to other work concerning the topic. Then we will discuss our stance on the matter and the way we will implement the method or technique in our project.

2 Requirements

Next to determining and exploring methods and techniques, we also did a requirements analysis.

These requirements can be found in Appendix [A](#).

3 Scrum

Scrum is an agile software development method. We will use this method to develop the software in our project.

For this, we made an extensive manual on the SCRUM method and our application, which can be found at the end of this report. *DF – Scrum Workflow*.

4 Git

[Git](#)¹ is a version control system, which we will use for the software development.

For Git we also made a manual on how we will use this software, which can be found at the end of report. *DF – Git Workflow*.

¹<http://git-scm.com>

5 MVC

For the implementation of the front-end interface we will use the *Model-View-Controller* (MVC) model. This is a structured method to setting up web-applications and was requested by the client as a pre-condition. This model is used as architectural guide to building an application with strict separation of program logic (*Model*), user interface (*View*) and interaction handling (*Controller*). Here we will explain the basic theory behind the MVC model and how we are going to apply it.

5.1 The MVC model

For the theory behind MVC we used Deacon [6] as a source. The MVC model aims to make the user interface interchangeable, by separating the front and back of the software application. This means that all programming logic is done separate from the user interface and all graphics are irrelevant to the programming logic. We will briefly explain the three parts of the MVC model and how these separate parts communicate or interact with each other.

We will use a simple example to illustrate the MVC model. We have a basic HTML page with a CSS stylesheet attached to it. The HTML contains the information or data that needs to be represented. The CSS contains the styling information and therefore knows how to represent the data, but has no knowledge of the data itself. The browser allows the user to interact with the HTML page.

5.1.1 Model

The Model part is the main functionality of the program. In this layer all objects and classes are defined that make up the application. The Model is not aware of any outside environments or user interactions. It only provides with the aspects of the underlying application and solution.

In our example, the HTML page is the Model, because it contains the information that needs to be represented.

In some cases two separate models are used [6]. These two are the *Application Model* and *Domain Model*. The Domain Model is what is classically described as the model, like the definition given above. The Application Model is a Model which has knowledge of the View and understands that data needs to be represented. This type of Model is specifically useful to facilitate connectivity with the Views.

5.1.2 View

The View part is centered about giving a (visual) representation of the system to the user. This means that the view of an application often consists of for example a *Graphical User Interface* (GUI) or an *Application Program Interface* (API). The View layer only knows how the information should be formatted to represent it to the user. The information itself is unknown to the Views, this is retrieved from the Model. The View is aware of the Model, but the Model is not aware of the View.

The CSS stylesheet in our example is the View, because it contains information about styling, but has no knowledge of the content.

Mostly the View is a GUI, which graphically represents the information of the system to the user. The example of an API is used in the DizzyData system and our project consists of building a GUI for the DizzyData API.

5.1.3 Controller

The Controller is the part that allows the user to manipulate the view. In other words, the Controller is used to interact with the information represented in the View. In strict terms the Controller is aware of the View, but the View is not aware of the Controller.

In the example given above, the browser represents the Controller. It allows the user to manipulate the View.

Often the Controller and View are combined for simplification, in which case the strict separation does not hold. This way the user interface is used to both represent the data and interact with it. Obvious reasons can be thought of for this implementation, whilst most interfaces provide the buttons and controls require visual representation.

5.1.4 Communication

The different parts of the MVC model are clearly separated, but they need to communicate to function. Communication in the MVC model is done through messages. Here we will explain how the different parts communicate.

The View and Controller interact with the Model through requests and updates, to request the Model data or to post an update to the data. In the case where the Model is split into an Application Model and a Domain Model, the Application Model handles these interactions and translates these to requests to the Domain Model.

The Model can also communicate to the View. But because the Model is not aware of the view's existence it can only use events to broadcast changes. If the View is listening to this type of event, it can update the appropriate fields.

5.2 Project specific implementation of MVC

Here we will briefly discuss our view on the MVC model and the way we plan to implement it in this project.

We will implement the View in our HTML and CSS and use a template structure to insert any data in this View. This way the View is a strictly separate part of the system.

The Controller will be mostly implemented in Javascript, which will handle all interactions with the interface. Again, this is a strictly separate part of the system, handling only interactions and placing calls to the Model.

We will also use just one Model at the back-end of the software. This will be an .NET implementation that will be used as a translation between the API calls to DizzyData and the graphical representation in the View. In a more loose definition one could say that our Model will be an Application Model and that we use the DizzyData API as the Domain Model.

6 Testing methods

As previously mentioned in the plan of approach, we will be using several different automated testing methods. These methods are summed up again here, with references to their relative websites and a brief explanation of the method.

6.1 *NUnit*

NUnit [2] is a tool developed for performing unit tests on .NET/C# code. Unit tests are automated tests that verify the outcome of a function with an expected result, as provided by the user. These tests will be used to check that methods behave as expected, but do require that we define the expected outcome manually. This tool is similar to the Junit tool (for Java unit testing) we worked with earlier in the Software Quality and Engineering course in the bachelor's second year.

6.2 *QUnit*

QUnit [3] is a similar tool as NUnit, only for performing unit tests on JavaScript code.

6.3 *Visual Studio*

Visual Studio [4] is a developer tool that allows for debugging, coverage testing and functions as the code editor. This tool will be used to run debug tests and coverage checks on the developed code.

6.4 *Moq*

Moq [1] is a tool that enables developers to mock or imitate a class to control and observe interactions between classes. We will use Moq for interaction testing. A simple example would be to imitate/mock API calls to produce either desirable or erroneous results.

6.5 *W3C validation*

W3C validation [5] can be used for testing HTML and CSS for errors and misuse. The validation will be used to confirm the correct implementation of these languages.

References

- [1] Moq, mocking library for .NET. URL <https://github.com/Moq>.
- [2] NUnit, unit testing framework for .NET languages. URL <http://www.nunit.org/>.
- [3] QUnit, unit testing framework for JavaScript. URL <http://qunitjs.com/>.
- [4] Visual Studio, a .NET code development tool. URL <http://www.microsoft.com/visualstudio>.
- [5] W3C Markup Validation Service. URL <http://validator.w3.org/>.
- [6] John Deacon. Model-view-controller (mvc) architecture. 2009.

Appendix A: Requirements Analysis

DF Requirements

Target release	DF 1.0
Theme	DizzyData Front-End project
Document status	DRAFT
Document owner	Mick van Gelderen, Mick de Lange
Designer	-
Developers	-
QA	-

Goals

- Create a visual interface for interaction with the DizzyData API.
- Allow users to manage their workflows.
- Allow users to manage their account settings.
- Allow users to manage their third party account settings.
- Allow users to view template WorkFlow and use them as a basis for a new WorkFlow.
- Allow users to view template Element and use them as a basis for a new Element.
- Allow users to view their usage statistics.
- Allow users to view job details.

Background and strategic fit

The objective of this project is dened as creating a front-end to use as a marketing tool towards potential customers as well as an interface to provide with a less complicated way of using the capabilities of the REST API.

DizzyData is a versatile but complex system, with many possible applications. To market this and to visualize the possibilities to potential customers, even those with no programming skills, a good visualization is needed. This is the main objective for the project, required by the client. The client wants a visual interface to show customers the versatility of the DizzyData system.

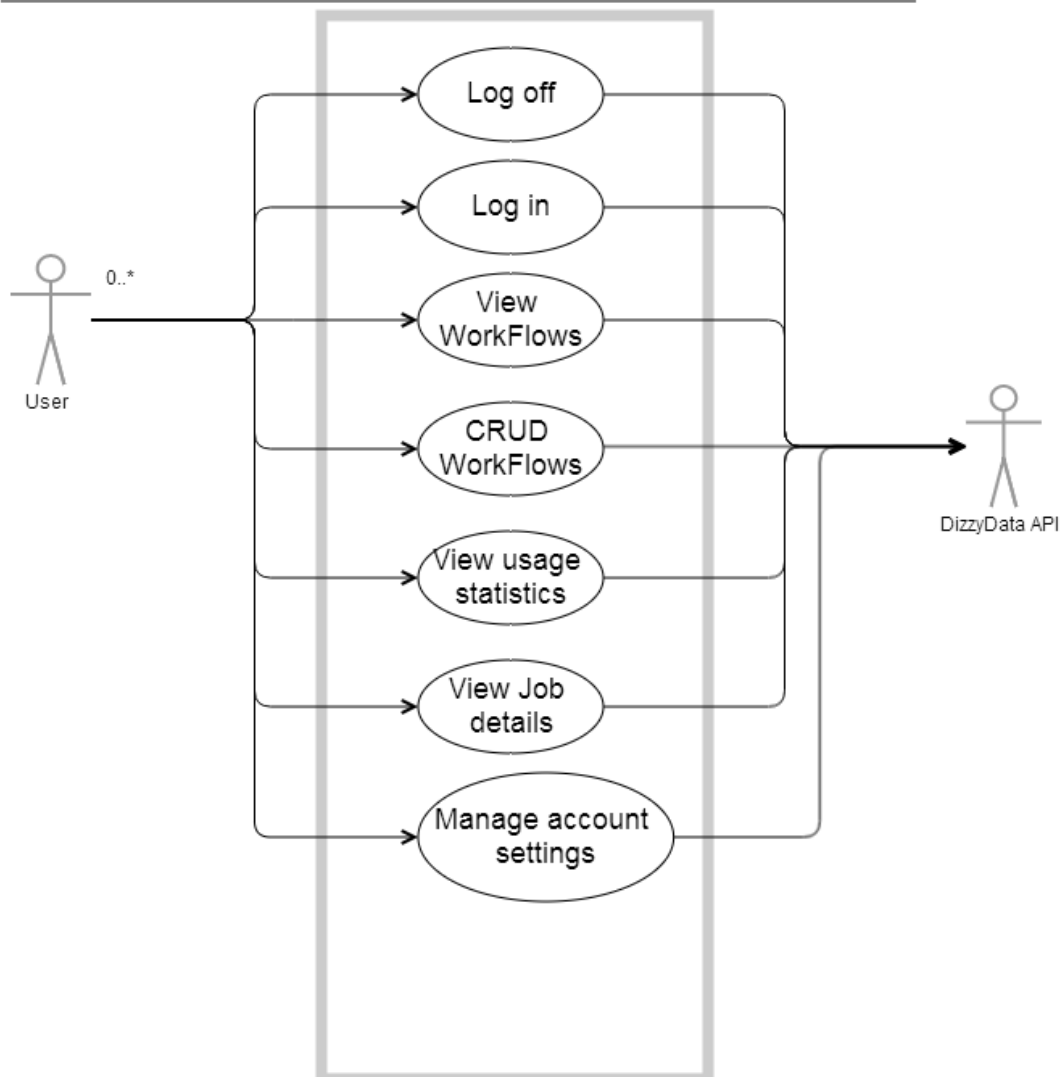
The second objective is to enable customers with little to no experience in working with REST API's or programming, to manage, create and edit their own document processing work ows. This level of end-user development is the second main objective for the project. Via an accessible interface users with minimum amount of programming skills should be able to work with the DizzyData system.

Assumptions

- The user uses the interface through a browser.
- The user has no knowledge of programming.
- The API is constantly available.

User interaction and design

Use Case



Users

For this application we expect one type of user, which we will describe here.

- The user is an employee (possibly administrator) of the consumer applications. This means that they work for the software company that implements the DizzyData system in their product.
- The user has knowledge of the document type they wish to process.
- The user knows how to use a computer and web browser.
- The user does not necessarily understand the DizzyData API.

There is one aspect in which the users may differ, this is their skill in software development. Some user may have little to no programming skills, they are users that mainly use the front-end to get an insight in the possibilities and create and manage their workflows in a visual manner.

The users that do have skills in programming / software development will probably be more accustomed to working with an API and will mainly use the front end to the same means as the user with lesser programming skills. These users might want to have more control over the system, they could use direct API interaction for this extended functionality.

Requirements

#	User Story Title	User Story Description	Priority
1	User logs in	As a user of the DizzyData API, I want to log in to the DF to use the functionality.	Must have
2	User logs off	As a user, I want to log off, because I'm finished.	Must have
3	View WorkFlows	As a user, I want to be able to view my WorkFlows and filter them by name.	Must have
4	CRUD WorkFlow	As a user, I want to be able to create, read, update and delete a WorkFlow.	Must have
5	View templates	As a user, I want to be able to view WorkFlow templates.	Must have
6	Statistics	As a user, I want to be able to view my WorkFlow and account statistics.	Must have
7	View Job	As a user, I want to be able to view the running Jobs and their details.	Must have
8	Manage account	As a user, I want to be able to manage my account.	Must have

Generated from requirements.ted:

#	Activity Diagram	Requirement	Priority
1	Create WorkFlow	A user should be able to create a new WorkFlow	Must have
2	Create WorkFlow	A user should be able to create a WorkFlow from a template	Must have
3	Delete WorkFlow	A user should be able to delete a WorkFlow	Must have
4	Edit WorkFlow	A user should be able to edit a WorkFlow	Must have
5	Edit WorkFlow	A user should, while editing a WorkFlow, be able to save his WorkFlow	Must have
6	Edit WorkFlow	A user should, while editing a WorkFlow, be able to stop editing his WorkFlow	Must have
7	Edit WorkFlow	A user should, while editing a WorkFlow, be able to restore his WorkFlow to a previous version	Must have
8	Edit WorkFlow	A user should, while editing a WorkFlow, be able to edit his WorkFlow settings	Must have

9	Edit Workflow	A user should, while editing a Workflow, be able to add a Step to his Workflow	Must have
10	Edit Workflow	A user should, while editing a Workflow, be able to view a Step	Must have
11	Edit Workflow	A user should, while editing a Workflow, be able to edit a Step	Must have
12	Edit Workflow	A user should, while editing a Workflow, be able to delete a Step	Must have
13	Edit Workflow	A user should, while editing a Workflow, be able to connect Steps	Must have
14	Edit Workflow	A user should, while editing a Workflow, be able to disconnect Steps	Must have
15	Edit Workflow	A user should, while editing a Workflow, be able to copy and paste Steps	Should have
16	Log in	A user should be able to log in with his DizzyData account	Must have
17	Log in	A user should be able to log in with his account from any external OpenID supporting service	Must have
18	Log in	A user should be able to restore his password	Must have
19	Log out	A user should, while being logged in, be able to log out	Must have
20	Edit Step	A user should, while editing a Step, be able to change the Step type	Must have
21	Edit Step	A user should, while editing a Step, be able to edit the settings specific to the Step type	Must have
22	Edit Step	A user should, while editing a Step, be able to apply his changes	Must have
23	Edit Step	A user should, while editing a Step, be able to stop editing	Must have
24	Edit Step	A user should, while editing a Step, be able to save and stop editing	Must have
25	Edit Step Settings	A user should, while editing an input storage service Step, be able to select which storage service I want to use	Must have

26	Edit Step Settings	A user should, while editing an input storage service Step, be able to select an input and processed folder	Must have
27	Edit Step Settings	A user should, while editing an input storage service Step, be able to enter the storage service account and its credentials	Must have
28	Edit Step Settings	A user should, while editing an output storage service Step, be able to select which storage service I want to use	Must have
29	Edit Step Settings	A user should, while editing an output storage service Step, be able to select an output folder	Must have
30	Edit Step Settings	A user should, while editing an output storage service Step, be able to enter the storage service account and its credentials	Must have
31	Edit Step Settings	A user should, while editing an Extract Step, be able to view the Extract Elements	Must have
32	Edit Step Settings	A user should, while editing an Extract Step, be able to remove an Extract Element	Must have
33	Edit Step Settings	A user should, while editing an Extract Step, be able to add a new Extract Element	Must have
34	Edit Step Settings	A user should, while editing an Extract Step, be able to add a new Extract Element based on template	Must have
35	Edit Step Settings	A user should, while editing an Extract Step, be able to edit an Extract Element	Must have
36	Edit Step Settings	A user should, while editing an Element, be able to add an Anchor	Must have
37	Edit Step Settings	A user should, while editing an Element, be able to edit an Anchor	Must have
38	Edit Step Settings	A user should, while editing an Element, be able to remove an Anchor	Must have
39	Edit Step Settings	A user should, while editing an Element, be able to change the order of the Anchors	Must have
40	Edit Step Settings	A user should, while editing an Element, be able to view the Anchors	Must have

41	Edit Step Settings	A user should, while editing an Element, be able to set the name of the Element	Must have
42	Edit Step Settings	A user should, while editing an Element, be able to set the description of an Element	Must have
43	Edit Step Settings	A user should, while editing an Element, be able to save the Element as a user template	Must have
44	Edit Step Settings	A user should, while editing an Anchor, be able to set the name of the Anchor	Must have
45	Edit Step Settings	A user should, while editing an Anchor, be able to edit the matching expression which is either a literal string or a regular expression	Must have
46	Edit Step Settings	A user should, while editing an Anchor, be able to edit the exclude expression	Must have
47	Edit Step Settings	A user should, while editing a Split Step, be able to view the Split Rules	Must have
48	Edit Step Settings	A user should, while editing a Split Step, be able to remove a Split Rule	Must have
49	Edit Step Settings	A user should, while editing a Split Step, be able to add a new Split Rule	Must have
50	Edit Step Settings	A user should, while editing a Split Step, be able to edit a Split Rule	Must have
51	Edit Step Settings	A user should, while editing a Split Rule, be able to add Elements	Must have
52	Edit Step Settings	A user should, while editing a Split Rule, be able to edit Elements	Must have
53	Edit Step Settings	A user should, while editing a Split Rule, be able to remove Elements	Must have
54	Edit Step Settings	A user should, while editing a Classify Step, be able to view the Classify Rules	Must have
55	Edit Step Settings	A user should, while editing a Classify Step, be able to remove a Classify Rule	Must have

56	Edit Step Settings	A user should, while editing a Classify Step, be able to add a new Classify Rule	Must have
57	Edit Step Settings	A user should, while editing a Classify Step, be able to edit a Classify Rule	Must have
58	Edit Step Settings	A user should, while editing a Classify Rule, be able to add Elements	Must have
59	Edit Step Settings	A user should, while editing a Classify Rule, be able to edit Elements	Must have
60	Edit Step Settings	A user should, while editing a Classify Rule, be able to remove Elements	Must have
61	Edit Step Settings	A user should, while editing an Enrich Step, be able to select an Enrich Module from a predefined set	Must have
62	Edit Step Settings	A user should, while editing an Enrich Step, be able to view the input fields	Must have
63	Edit Step Settings	A user should, while editing an Enrich Step, be able to select the input values	Must have
64	Edit Step Settings	A user should, while editing an UBL2 File Step, be able to edit the settings of this Step	Must have
65	Manage account	A user should, while logged in, be able to manage his account	Must have
66	Manage account	A user should be able to view his plan details	Must have
67	Manage account	A user should be able to view his invoices	Must have
68	Manage account	A user should be able to view his account details	Must have
69	Manage account	A user should be able to edit his account details	Must have
70	Manage account	A user should be able to request a new API key	Must have

Technical requirements

#	Short	Description	Priority
1	Server-side implementation	The server-side programming is done in a .NET environment.	Must have

2	Client-side implementation	The latest Javascript, HTML and CSS are used to present the interface to the client.	Must have
3	MVC	The system is build using the Model-View-Controller model.	Must have
4	Browser support	The web interface is compliant with popular browsers: IE8+, Chrome 25+, Firefox 19+, Opera 12+ and Safari 5.1+.	Must have
5	iPad support	The web interface is compliant with Safari iPad.	Should have
6	Stand-alone front-end	The front-end functions standalone, without direct (database) access to the DizzyData system.	Must have
7	Minimum data transfer	The data sent between server and client is kept to a minimum.	Must have

Appendix D: DF – Scrum Workflow

DF – Scrum Workflow

Mick van Gelderen
4091566

Mick de Lange
1534068

June 26, 2013

Contents

1	Introduction	2
2	The Scrum Team	3
2.1	Product Owner	3
2.2	Development Team	3
2.3	Scrum Master	3
2.3.1	Scrum Master service to the Product Owner	3
2.3.2	Scrum Master service to the Development Team	4
2.3.3	Scrum Master service to the Organization	4
3	Scrum events	4
3.1	The Sprint	4
3.2	Sprint Planning Meeting	4
3.2.1	What	5
3.2.2	How	5
3.2.3	Goal	5
3.3	Daily Scrum	5
3.4	Sprint Review	5
3.5	Sprint Retrospective	6
4	Scrum artifacts	6
4.1	Product Backlog	6
4.2	Monitoring progress towards a goal	6
4.3	Sprint Backlog	6
4.3.1	Monitoring sprint progress	6
4.3.2	Increment	6
5	Definition of Done	6
6	Conclusion	7
	References	8

1 Introduction

This document defines the project workflow for *DizzyData front-end* (DF). The basis of our workflow comes from *Scrum* as described in *The Scrum Guide* by Schwaber and Sutherland [2].

2 The Scrum Team

The *Scrum Team* consists of the *Product Owner*, the *Development Team* and the *Scrum Master*. Scrum Teams choose how best to accomplish their work and consists of members whose skills combined cover all the skills required to complete the project.

2.1 Product Owner

The Product Owner is responsible for “maximizing the value of the product and work of the Development Team” [2, p. 5]. For DF, the Product Owner is the *project manager*: Tim Paymans. His tasks include:

- Clearly expressing the *Product Backlog* items
- Assigning priorities to the Product Backlog items to best achieve goals
- Ensure that the Product Backlog is updated and reflects the project state

In our case the Product Owner assigns part of the Product Backlog management tasks to the Development Team but as stated in The Scrum Guide, the Product Owner will always remain accountable for the Product Backlog.

2.2 Development Team

The Development Team consists of “professionals who do the work of delivering a potentially releasable increment of *Done* product at the end of each *Sprint*” [2, p. 6].

Our Development Team is not as cross functional as we would like. Our Development Team consists of two software developers: Mick de Lange and Mick van Gelderen. We have limited access to other human resources such as a interaction designer and graphical designer. Ideally they would participate directly in our Development Team so that they can participate in an agile manner. This is not the case however which makes the agile development a bit harder since they need more complete information on what we require them to do.

2.3 Scrum Master

The Scrum Master is “responsible for ensuring Scrum is understood and enacted” [2, p. 6].

Our Scrum Master is Mick van Gelderen. He will be a servant-leader for the Scrum Team. This guards the Scrum process by helping the Product Owner, Development Team and those outside the Scrum Team.

The following list is largely adopted from The Scrum Guide.

2.3.1 Scrum Master service to the Product Owner

The Scrum Master serves the Product Owner in several ways, including:

- Finding techniques for effective Product Backlog management.
- Clearly communicating vision and goals to the Development Team.
- Teaching the Development Team to produce clear and concise Product Backlog items.
- Facilitating Scrum events as requested or needed.

2.3.2 Scrum Master service to the Development Team

The Scrum Master serves the Development Team in several ways, including:

- Coaching the Development Team in self-organization and cross-functionality.
- Removing impediments to the Development Team's progress.
- Facilitating Scrum events as requested or needed.
- Teaching the Development Team how to act in organizational environments in which Scrum is not yet fully adopted and understood

2.3.3 Scrum Master service to the Organization

The Scrum Master serves the *Organization* in several ways, including:

- Leading and coaching the organization in its Scrum adoption.
- Planning Scrum implementations within the Organization.
- Helping employees and stakeholders understand and enact Scrum and empirical product development.
- Causing change that increases the productivity of the Scrum Team.
- Working with other Scrum Masters to increase the effectiveness of the application of Scrum in the organization.

3 Scrum events

There are a number of Scrum events defined in The Scrum Guide. They are aimed at creating regularity and minimizing the need for meetings not defined in Scrum. The events are time-boxed so that every event has a maximum duration. "This ensures an appropriate amount of time is spent planning without allowing waste in the planning process" [2, p. 7].

Our take on the events will be described in this section. The details on each event can be found in The Scrum Guide.

3.1 The Sprint

A Sprint will consist of the *Sprint Planning Meeting*, the *Daily Scrums*, the development work, the *Sprint Review* and the *Sprint Retrospective*. Each Sprint will have a time-box of one week. Reasons for this choice are having a small Development Team that works closely together and the very limited time that is available to work on the project. We have to be able to refocus on particular goals during the process.

3.2 Sprint Planning Meeting

Our Sprint Planning Meeting is time-boxed to two hours since our Sprints occur every week. The Sprint Planning Meeting is aimed at answering two questions:

- What will be delivered in the *Increment* resulting from the upcoming Sprint?
- How will the work needed to deliver Increment be achieved?

3.2.1 What

The Product Owner and Development Team will decide what Product Backlog items will be treated during the upcoming Sprint. But, only the Development Team will decide how many of the selected items it can process during the Sprint. This will be done in a one hour period.

After the Product Backlog items have been chosen, the Scrum Team crafts a *Sprint Goal*. “The Sprint Goal is an objective that will be met within the Sprint through the implementation of the Product Backlog, and it provides guidance to the Development Team on why it is building the Increment” [2, p. 7].

3.2.2 How

The Development Team will make a plan that describes how to approach creating a solution for the *Sprint Backlog* items. This plan encompasses for example the people that will be put on the design of a system. In this phase of the Sprint Planning Meeting it should become clear how the functionality will be built into a Done product Increment during the Sprint. The Product Owner and possibly domain experts may or may not be present during this part of the meeting.

3.2.3 Goal

The Sprint Goal gives the Development Team some flexibility regarding the functionality implemented within the Sprint. If the functionality and technology turns out to be different from what the Development Team expected, then they collaborate with the Product Owner to negotiate the scope of the Sprint Backlog within the Sprint.

3.3 Daily Scrum

Our Daily Scrum will be held in the morning from 9:15 to 9:30. We will shortly discuss what has been accomplished since the last meeting, what will be done before the next meeting and what obstacles are in the way. If possible, the Product Owner will attend these meetings as an observer to get a feeling for what Scrum is like. Officially this should not be the case since the Daily Scrum is not a status meeting but in our case this project is also a test project for agile development.

3.4 Sprint Review

The Sprint Review will be an informal one hour meeting held at the end of a Sprint. The following elements are included in the Sprint Review:

- The Product Owner identifies what has been Done and what not.
- The Development Team discusses what went well and what problems it ran into and how these were solved.
- The Development Team demonstrates the work that it has Done and answers questions about the Increment
- The Product Owner discusses the Product Backlog as it stands and tries to give a projected completion date based on the progress made so far if an educated estimate can be made.
- The Scrum Team collaborates on what to do next so that the Sprint Review provides valuable input to subsequent Sprint Planning Meeting

3.5 Sprint Retrospective

The Sprint Retrospective focuses on the Scrum process as a whole and provides a formal opportunity to give feedback on it. This meeting will be scheduled between the Sprint Review and the Sprint Planning Meeting and take a maximum of 45 minutes.

4 Scrum artifacts

“Scrum’s artifacts represent work or value in various ways that are useful in providing transparency and opportunities for inspection and adaptation. Artifacts defined by Scrum are specifically designed to maximize transparency of key information needed to ensure Scrum Teams are successful in delivering a Done Increment. ” [2, p. 12].

4.1 Product Backlog

The Product Backlog is hosted on *Jira* [1]. The initial Product Backlog will be constructed from the results of the orientation phase of the project, see *DF – Project Approach* for details on the project phases.

4.2 Monitoring progress towards a goal

“At any point in time, the total work remaining to reach a goal can be summed. The Product Owner tracks this total work remaining at least for every Sprint Review. The Product Owner compares this amount with work remaining at previous Sprint Reviews to assess progress toward completing projected work by the desired time for the goal. This information is made transparent to all stakeholders.” [2, p. 13].

4.3 Sprint Backlog

The Sprint Backlog will hold the backlog items that are planned to be treated in the current Sprint in addition to the plans used or required to work on the Sprint Backlog items. Working on the Product Backlog or Sprint Backlog items should not be considered as items themselves.

4.3.1 Monitoring sprint progress

The total remaining work in the Sprint Backlog items can be summed. The Development Team tracks these sums daily and projects the likelihood of achieving the Sprint Goal.

4.3.2 Increment

The Increment is the sum of all the Product Backlog items that have been completed, that means they are Done, in a Sprint.

5 Definition of Done

Everyone in the Scrum Team must know what it means for work to be complete to ensure transparency. This is what we call Done. Initially this definition may be slightly loosely formulated but, as the Scrum Team matures, it is expected that their definition of Done will become more strict to enforce higher quality.

6 Conclusion

Our approach to Scrum heavily relies on the definitions provided in The Scrum Guide. There are however a number of choices and trade-offs that have had to be made when we decided to use Scrum such as selecting the length of a Sprint.

REFERENCES

References

- [1] Jira, issue tracker by Atlassian. URL <http://www.atlassian.com/software/jira>.
- [2] Ken Schwaber and Jeff Sutherland. *The Scrum Guide*. 2011.

Appendix E: DF – Git Workflow

DF – Git Workflow

Mick van Gelderen
4091566

Mick de Lange
1534068

June 26, 2013

Contents

1	Introduction	2
2	Initial setup	3
3	Daily workflow	3
3.1	Grab an item from the Sprint Backlog	3
3.2	Develop	3
3.3	Collaborate	3
3.4	Test	3
3.5	Complete	3
4	Sprint workflow	3
	References	4

1 Introduction

This document describes the [Git](#)¹ workflow used in the development of *Dizzy-Data front-end* (DF). The basis of our workflow comes from Git as described in *Pro Git* by Chacon [2].

To start working on git you should get a general idea of what git is. I recommend you skim over [this online manual](#) and read the parts that you think are interesting and the topics below. Do not start by reading everything in detail. Just make sure you have an idea of what Git is and of what it can do. Only when you have a working Git setup, experiment with a test repository using the basic commands given below. When you feel comfortable, dive into topics such as branching.

Note: Installing git might not be necessary depending on the tool you will use, I recommend learning what Git is before installing something. I also recommend installing something before learning how to use Git so that you can learn by doing it yourself.

- [Git basics, especially about the workflow.](#)
- [Changing the repository.](#)
- [Branching.](#)
- Required basic commands
 - `git clone`².
 - `git pull`³.
 - `git status`⁴.
 - `git add`⁵.
 - `git commit`⁶.
 - `git push`⁷.
- Other commonly used and advanced commands
 - `git log`⁸
 - `git branch`⁹
 - `git merge`¹⁰
 - `git tag`¹¹
 - ...

¹<http://git-scm.com>

²<http://git-scm.com/docs/git-clone>

³<http://git-scm.com/docs/git-pull>

⁴<http://git-scm.com/docs/git-status>

⁵<http://git-scm.com/docs/git-add>

⁶<http://git-scm.com/docs/git-commit>

⁷<http://git-scm.com/docs/git-push>

⁸<http://git-scm.com/docs/git-log>

⁹<http://git-scm.com/docs/git-branch>

¹⁰<http://git-scm.com/docs/git-merge>

¹¹<http://git-scm.com/docs/git-tag>

2 Initial setup

Configure your git installation to use your name, email and other settings. Then create a clone of the repository you wish to work on with `git clone` so you can work on the project on your machine. The repository will probably have to be fetched from a central server only a limited number of people have read and write access to. Common hosting platforms include [github](https://github.com)¹² and *Bitbucket* [1].

3 Daily workflow

This section describes the general item-to-item workflow for our project.

3.1 Grab an item from the Sprint Backlog

Take an item from the *Product Backlog* and create a new branch using `git branch`. This will be the branch on which you develop functionality for that particular item. If you have to switch and work on another item, create a new branch for the other item so that you always start with a working version of the software: the master branch.

3.2 Develop

While writing a solution to the Product Backlog item be sure to commit each time you are able to write a meaningful commit message.

3.3 Collaborate

If you need to collaborate with someone else you will have to make your local branch public by pushing it to the central repository.

3.4 Test

Write if you have not done so already and run tests for the new piece of functionality. Be sure to move the Product Backlog item from in progress to testing.

3.5 Complete

If the tests ran successfully and everything is working correctly you can merge the master branch with the Product Backlog item branch with `git merge`. Be sure to move the Product Backlog item from testing to done.

4 Sprint workflow

Every *Sprint*, the master branch should be tagged as being a releasable version. This can be done using `git tag`.

¹²<http://github.com>

REFERENCES

References

- [1] Bitbucket, Git version control hosting. URL <http://bitbucket.org>.
- [2] Scott Chacon. *Pro Git*. Apress, 2009.