Final Report Magnet.me IT Infrastructure Reorganization

T. Langerak A. Walterbos

August 22, 2013

Abstract

The Bachelor Project assignment of internet startup Magnet.me, fulfilled by Tiddo Langerak and Alex Walterbos, consisted of the replacement of the IT infrastructure in the company. Before designing the new system, the old system was analyzed. Based on this analysis, a list of requirements was formed.

The system has been designed so that it fulfills a significant amount of the requirements per definition: Using modern techniques like the Node.js platform, the AngularJS framework, Redis caching and an Nginx webserver, a high performance RESTful IT infrastructure was built.

This design includes a server side service for business logic calculation and a Content Management System for internal usage. Using this system, Magnet.me hopes to grow substantially without being held back by technology.

(Performance) tests have shown a significant improvement over the old system. The system is set up to be flexible, maintainable and reliable. Its performance is of a grade that, according to early estimations, should even be able to support international traffic. The Magnet.me management has already expressed their satisfaction with the systems performance, even before the system has been implemented completely.

Introduction

Magnet.me is a young internet startup. The company provides an online platform where both students and companies can create profiles, entering information about themselves. The student also provides information about his or her preferences in the work field. Companies provide a so-called Potential Employee Profile (abbreviated as PEP), which contains demands and preferences about potential employees. Based on these this information and these preferences, the students and companies are automatically matched with the Matching System, as explained in Appendix C.

Early March, 2013, a bachelor project assignment was formed by Magnet.me. The assignment requested the replacement of the company's existing Content Management System (CMS), as can be seen in Appendix I. However, the existing CMS was built in a way that did not allow replacing a part of the system: The entire IT infrastructure had to be redesigned, and replaced. This included amongst other components a fully redesigned backend, and a replacement of the database. This monster assignment was accepted by Tiddo Langerak and Alex Walterbos, whom have worked on the project since late April 2013.

As the project progressed through June and July, designs were made, and code was written; the system began to take shape. The database was ready to support Magnet.me's extensive data set, with over 80 tables and a couple of hundred fields. Nearing the project's finish, in early August, the system came close to completion; be it in a tuned down form due to time constraints. Using techniques like Test Driven Design [28], and modern techniques like REST [30], Node.js [19] and AngularJS [3], the system was built to last.

In this report, we shortly describe the assignment, and walk you through the requirements. The design of the system is motivated. The implementation process is described, as are the difficulties we encountered. Our methods of developing and testing are portrayed, including the tools we used along the way. Finally, we reflect on the requirements; have they been fulfilled? How, if so, and otherwise: Why not?

Contents

1 Problem Analysis

1.1 Motivation

The motivation for the replacement of the IT Infrastructure within Magnet.me arises from the company's desire to grow. The current system is holding Magnet.me down, both in growth speed and possibilities: It does not allow scaling over multiple servers, does not support multiple languages, as explained in Section 2.2.

Magnet.me intends to attract more users as it grows. The current system will not be able to handle the increase in traffic which can be expected from this expansion, as it is struggling already; this is described in Section 6.1.

Another reason for the system replacement is the need for an in-house developed system. With the out-sourced system, Magnet.me's IT-team often finds it self lacking control or access to vital parts of the system. To reduce external dependencies, the system must be fully maintainable from within Magnet.me.

Magnet.me's existing IT infrastructure *is* the Caret CMS, a system developed by Click [2]. The system, designed in the late 1990s, is actually built up from a framework for building websites, a database and a web interface for content management (which is, confusingly, the actual CMS). This structure lacks modularity, and is therefore intensively interconnected, as further explained in Section 2.2. This results in a high dependency in the infrastructure, which is the reason why the system had to be replaced entirely.

This Bachelor Project is the first step in the process of replacing and upgrading the current IT infrastructure. After this project, the website will be transferred to the newly developed system. A mobile application will be connected later, and in the future third party software will be able to connect with the system as well.

1.2 Description

In this project, the student team (consisting of Alex Walterbos and Tiddo Langerak) have designed and implemented the web backend and Content Management System (CMS). The CMS has been called the 'Magnet.me Dashboard', not only to prevent confusion with the Caret CMS, but also because the Dashboard will later on be expanded with other non-CMS functionality.

The web back-end consists of the web server that will route all incoming requests, the database that will store all data, and the matching system. The matching system can be explained shortly as the Magnet.me business logic that matches students to company's Potential Employee Profile. The system is explained in more detail in Appendix C.

The Dashboard will form an interface for data object manipulation. It will be accessible for Magnet.me employees only, and it will create an abstraction of the database

entities, to make the process more intuitive for the user.

The student team has *not* implement the main website (the portal for students and recruiters) within the Bachelor Project. Instead, the currently live website will be connected to the new system after the Bachelor Project has been completed.

1.3 Requirements

This section will discuss the requirements of the system. It will be divided in two parts: functional and non functional requirements. These requirements will be prioritized and categorized in four categories: system wide, web back-end, matching system and the Magnet.me Dashboard. These categories are shown in the tables below.

1.4 Functional Requirements

Nr.	Description	Must/Could
	System wide	
1	Support for multiple languages	m
	Web Back-End	
1	It should provide an interface for the web applications including:	m
1.1	An abstraction for the data layer	m
1.2	An authentication interface	m
2	The following types of software should be able to be build on top of the back-end:	m
2.1	The website	m
2.2	The Magnet.me Dashboard	m
2.3	A mobile app	m
2.4	3rd party software	c
3	Requests should be authenticated	m
4	Requests should be authorized	m
4.1	Different actors should have different access rights	m
	Matching system	
1	It should match students with companies according to the model provided by Magnet.me (see Appendix C)	m
2	It should always act on the most recent data available	m
3	It should recalculate matches within 30 minutes after a change have been made which could influence the matches	m
	The Magnet.me Dashboard	
1	The Dashboard should only be accessible for Magnet.me employees	m
2	All data provided by the companies and students can be viewed and edited, with the exception of passwords	m
3	The Dashboard should always show the most recent data available	m
4	Different employees can have different access rights to parts of the Dashboard	m
5	Subscriptions for companies can be managed	m
6	Translations can be added for the website	m
7	Statistics can be viewed from the Dashboard, including:	m
7.1	New subscriptions over time	m
7.2	Accepted and rejected matches per company	m
7.3	Accepted and rejected matches per student	m
7.4	Geographical distribution of companies and students	m
8	Matches can be made manually	m

1.5 Non Functional Requirements

Nr.	Description	Must/Could
	System wide	
1	Security measurements should be automatically enforced where possible	m
1.1	All data should be sanitized by default before being used as output	m
2	The systems should be robust	m
2.1	Every request should receive a response	m
2.2	An error of a subsystem should never let the entire system crash	m
2.3	All systems should recover from any non fatal error[1]	m
3	The systems should be scalable across multiple servers	m
4	Subsystems should not interfere with eachother	m
5	New data should be visible to all subsystems within 10 minutes of an update	m
6	The system should support an international release of Magnet.me	c
	WHOOPS	
1	Requests should be processed within 100ms	c

[1] Only fatal errors from critical dependencies are considered to be fatal errors for the backend. Any subsystem error which is generated by a subsystem itself is non fatal. Examples of such errors are hardware failures or inaccessible databases.

2 Background

2.1 Magnet.me

Magnet.me is a young platform for students, graduates and employers. The goal of Magnet.me is to provide a platform in which students can find employers, and employers can find students.

With the Magnet.me platform companies can create one or more network of potential employees by defining some criteria for these potential employees in the so called potential employee profile, or PEP for short. Linked with a companies profile can be one or more recruiter accounts, which can be used by the recruiters to contact students. Students can also create a profile and they can fill in their resume. Students which pass the criteria of a PEP will automatically get an invitation to join the companies network, but manual requests can be made as well. As such companies can build a network of qualified and interested students.

Companies can do several things with their profile and network. First of all, a company can not view a students resume as long as the student is not in the network of that company, unless the student explicitly sends it to the company. When a student accepts a companies invitation its resume will become visible for the company immediately.

Companies can also create a so called minisite: a section on the Magnet.me website where company can display some information about themselves. Also, companies can post events, vacancies and internships, or EVIs for short, on their website, and they can allow students to register for these EVIs. Students who are part of the companies network will see new EVIs when they login on their overview page.

Finally companies and recruiters can communicate with students using private and group message as well as with public posts on the overview pages.

Using these features companies and students can effectively be connected with each other.

2.2 Current Situation

Before the new requirements and a new design could be created it was important to understand what the (then) current situation of the system was. Therefore, we studied the situation at that time, which proved to be an important part of our orientation phase.

The old Content Management System application is built by a third party vendor called Click [2]. Unfortunately, large parts of the design of this system are not documented, incomplete or are not available to people outside of Click. Therefore, all information below is acquired by analyzing the system from outside, unless explicitly stated otherwise.

2.2.1 Design

The core of the old system is the Caret CMS. The Caret CMS is a commercial CMS developed and maintained by Click[1]. It is originally developed by one of the co-founders of Click in the late 1990s and is mainly used by Click to develop websites for customers, including Magnet.me.

The CMS consists of three parts: a framework on top of which websites are built, a web interface for managing the website and the content (the actual CMS), and a PostgreSQL database for storing all information. A top level overview of this design can be found in figure 2.1. The following subsections will describe the design in more detail.

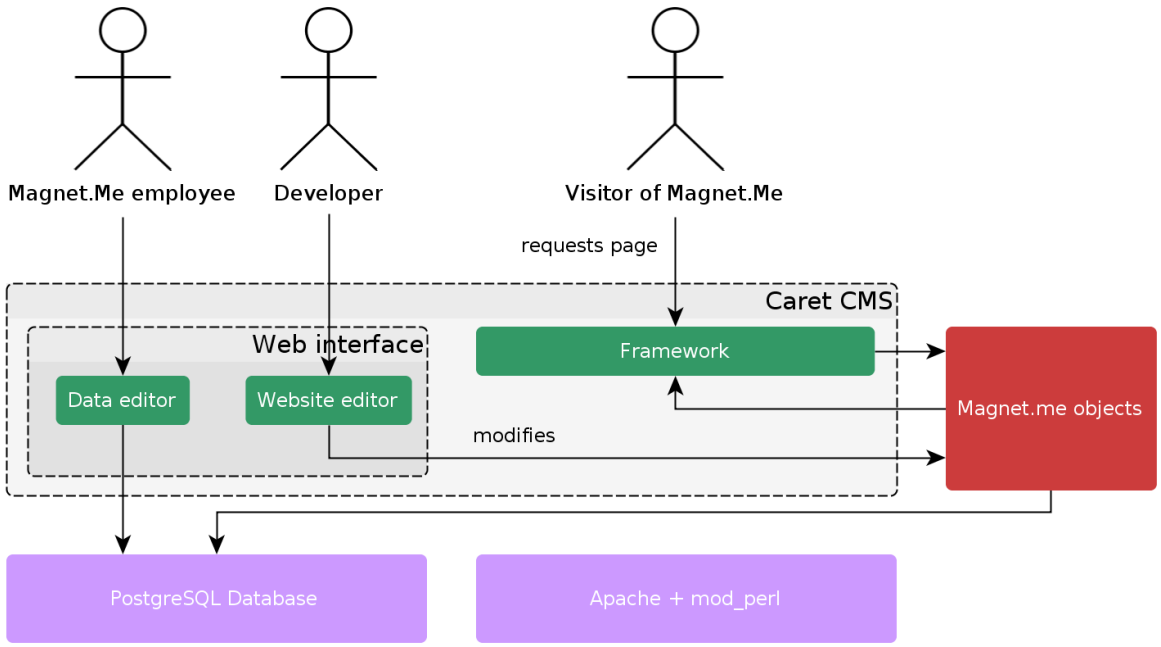


Figure 2.1: Current situation

Framework

Instead of using regular source files the framework uses a tree of special objects created using the web interface. These objects are very similar to regular files and folders, with a few notable differences.

First of all, each object has a predefined type which determines how it is treated by the framework. For example, an object with the type 'DataForm' can be used to create forms whose input will be stored in the database. These objects can also have string properties which can specify some behavior of the object. For example, a DataForm object can have a property called 'DataFormTable' which specifies the table to store the

[1]Caret and Click are highly linked, but separate companies. For simplicity's sake, we will refer to 'Click' from now on, when talking about these third party companies

input in. Most object can also contain text, e.g. HTML, CSS or JavaScript, which will be parsed by the framework. Most of the text will simply be sent to the browser, but special keywords can be used which instruct the framework to do some preprocessing such as inserting other objects, repeating blocks of code or inserting data from the database. Finally, objects can have other objects inside them such that the object tree can be constructed.

URLs are directly mapped to objects in the object tree. For example, a request to the fictional URL `https://magnet.me/site/foo/bar` will try to find the bar object inside the foo object and uses that object as a starting point of the application. The object will be processed by the framework and the result will be sent to the browser.

The framework does not allow custom server side code. It does allow queries to the database for some type of objects, but other than that only client side code can be written. Also, only functionality provided by the CMS can be used. Custom server side code can only be written by a few employees of Click, who can upload Perl script as plugins for the CMS.

Web interface

The web interface is the place where the entire application is built, apart from a few custom server side scripts. It provides a few different editors, of which the data editor and website editor are the most important ones.

The website editor is the interface in which the objects as described above can be created and edited. It shows a tree of objects and allows to browse trough these objects as if they represent a file tree. The actual contents of these objects can be edited using simple multiline text fields inside this editor. There is some very basic revision control implemented. Older versions of individual objects can be restored and compared, but merging or branching is not supported. Since the objects are only accessible from inside the framework it is not possible to use external tools such as minifiers, version control or unit testing frameworks without constantly having to copy the content of the objects manually between the CMS and actual files.

The other important part of the web interface is the data editor. This is a basic interface to raw information stored inside the database. This editor simply allows Magnet.me employees to edit any field inside the database. Foreign keys and cross references are automatically resolved by the Caret CMS, such that employees don't manually have to manage these references. Employees can use dropdown menus or checkboxes to create references. The database editor allows to do some very basic filtering, a filter on only one column can be applied, and the editor also allows to sort on any column. The database editor does not provide any other high level functionality, such as updating multiple records simultaneously or displaying the data in a more meaningful way to the Magnet.me employees.

Database

The final part is the database. The database management system used is PostgreSQL, as stated above, and the database is custom made for the Magnet.me website. No real

foreign keys are specified in the database, references are handled by the CMS instead. The design of the database for the Magnet.me website is unnecessarily complex and a thorough study of this design goes way beyond the scope of this report. The ERD provided to us can be found in appendix A. A larger version can be provided by Alex or Tiddo upon request.

2.2.2 Additional Drawbacks

The Caret CMS does not provide any straightforward way to synchronize the application with any other computer. Therefore, the only way to make changes to the application is by using the web interface provided by Caret, as mentioned above. This introduces a few problems: the web interface is a simple text area, and therefore it is not an ideal environment to program in. There is no syntax highlighting, code completion, or any other useful functionality which can help in the development process. In practice it works best to move the code between a decent IDE and the CMS by copying and pasting the source code.

Another problem is that you can only have multiple objects open at once by launching the CMS in different tabs in your browser. Creating objects and switching between them takes a very long time, it usually takes more than 2 seconds for the CMS to respond to a request. This makes it very slow to navigate through objects and this slows down the entire development process significantly.

Since it is not possible to synchronize the objects with a file system, it is also not possible to use any form of version control, unit testing framework, or any other type of automatic build tool. Testing has to be done manually, and 'locks' to objects can only be acquired by verbal communication with team members. This also slows down the development process, and limits the number of people which can work on the application at the same time significantly.

Finally the database editor has problems of its own as well. First of all, the editor simply shows the tables which exists in the database. For a software developer that's fine, but for the other employees of Magnet.me, the once who use this system most, this is not ideal. Many changes requires updates in multiple tables, which can be easily forgotten. Also, relationships between tables aren't clear from the CMS, and therefore it is easy to create an invalid state of data. It also provides very limited options for viewing detailed and useful statistics. Finally the database editor is not secure as well. A few security leaks, mostly XSS, have been found already in a few weeks time. It can be expected that more leaks will be found in the near future.

2.2.3 Problems To Overcome

The biggest issues that arise from this system are the lack of support for multiple languages, not allowing the system to scale over multiple servers, and the lack of control that Magnet.me employees have over the system.

How we want to overcome these problems is described mostly in Section 3. There, it will be clear that the modularity of the system directly allows scaling over multiple servers. Also, because the new system is developed within Magnet.me, the lack of control by Magnet.me employees has been solved.

3 System Design

As can be seen in Section 1.3 the new system should support multiple different ways of interacting: Magnet.me employees should be able to interact with the system using the Magnet.me dashboard, students and recruiters want to interact using the website and in the future a mobile app, and third party software should be able to interact with the system using some sort of API.

Traditional solutions in which the server renders the entire HTML page and sends it, complete with all data embedded, would require that the server needs seperate logic to render pages for each of the different applications. Different server side logic needs to be written for each new client application, a solution which is not very flexible.

Therefore a different solution has been chosen, which has also been explained in our orientation report [33]: The server will expose a REST[1] API which can be used to interact with the system. All clients will be implemented as thin clients[2] using this REST API. Therefore only one web service is needed for many different applications, including the dashboard, website, mobile application and third party applications.

Also two distinct responsibilities of the web service can be identified. Firstly, the service should provide the REST API as mentioned above. Secondly, the service should be able to match students with companies. These responsiblilities have very different characteristics and requirements. For example, the REST API mainly does a lot of input/output operations, whereas matching students with companies is mostly limited by available CPU power. Therefore these two parts are separated as well into two distinct applications, resulting in a heterogenous system.

The design of the new system now consists of several major separate parts; the server, the database, thin clients and the matching system. As shown in Figure 3.1, the server connects the components of the system, except for the matching system. It provides an API for the thin clients, which allows the thin clients to request, amongst other things, data from the database.

[1] REST: REpresentational State Transfer [30].

[2] A thin client is a standalone application which does not perform most of its computations by itself. In the case of Magnet.me the server performs most calculations and runs the business logic, whereas the clients merely provides a user accessible interface for this system, as can be seen in figure 3.1.

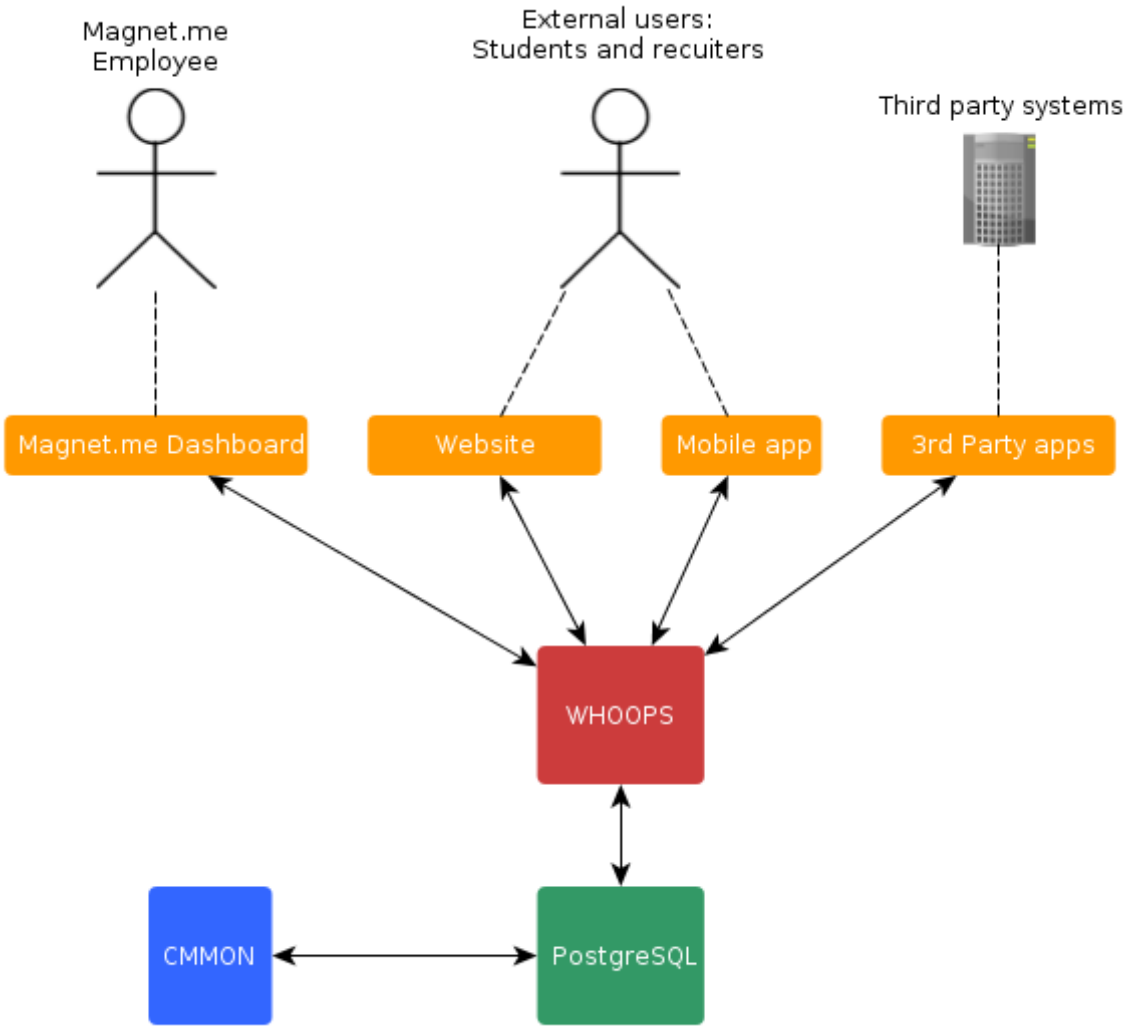


Figure 3.1: System design

4 Analysis of Available Platforms

To provide a good solid application it is important that a good foundation is used. Most of the research for the available platforms is already outlined in the orientation report [33]. Below the most important aspects of the research, as well as some additional information which has been acquired along the way will be described.

4.1 Database

The database is one of the most important components of the system. If the database is not chosen properly it can cause a lot of problems during the design of the system and it can seriously limit the scalability of the system. Some research into different databases have been done using Brewer's Theorem [32] have been done to find the most suitable database engine. However, during a meeting with the bachelor project supervisor, A. Bozzon[1], he advised to chose an database based on previous experience and personal preference within the boundaries of the requirements. Most modern database can be scaled both vertically and horizontally one way or another. Based on this advice a PostgreSQL database will be used for persistant data storage.

4.2 WHOOPS

WHOOPS (short for *Web-Hosted Omnipotent On-demand Providing Server*) is the web server in the system. The web server consists of three parts: a reverse proxy, a static file server and a REST server, which can als be seen in Figure 4.1. The REST Server is depicted in Figure 5.1.

Both the reverse proxy and the static file server are provided by Nginx, software build for exactly that purpose [20]. Nginx is configured such that it will provide the static content directly, but it proxies requests for the REST API[2] to the REST server. Nginx was chosen over other webservers, such as Apache, due to the ease to setup Nginx and its high performance [27, 29].

For the REST server there were several options as well. Its main responsibility is to correctly respond to requests from clients by mapping requests onto database operations. This requires little CPU power, but it does require that it can do input/output very efficiently, without blocking other requests. Traditional thread-per-request solutions, such as the Apache PHP combination, are not very suitable for this. The threads created for each request will be idle most of the time while waiting for input/output to complete, but the threads will give a memory overhead. This seriously limits the amount of requests the server can handle at the same time.

[1] Alessandro Bozzon, PhD, from the EEMCS department 'Web Information Systems'

[2] API: Application Programmable Interface

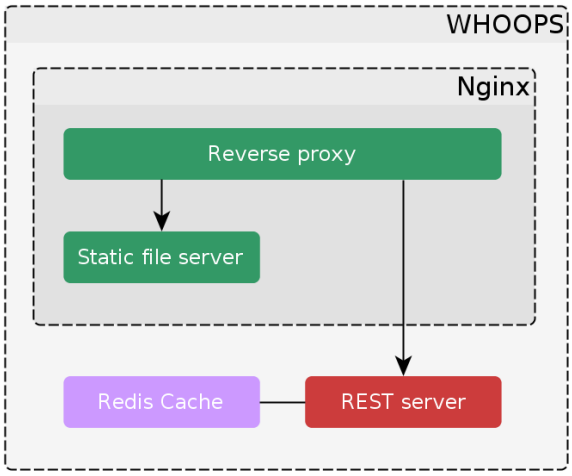


Figure 4.1: Components of WHOOPS

Event-driven solution, such as Node.js (Javascript) [19] or Twisted (Python) [17], are much better suited for these type of applications. Instead of creating a separate thread for each request it will use just one thread[3] which will wait for an event to occur and execute the appropriate handler(s).

Multiple language and platforms have been considered to build the web server in such as Python with Twisted or Django [16], C#, PHP, Java and Node.js. As mentioned before due to performance reasons an event-driven solution was preferred. Based on previous experience with the different platforms and the research done as outlined in the orientation report a choice for Node.js was made.

Node.js is an event driven non-blocking platform built for fast scalable network applications [19] and optimized for non-blocking input/output and thus very suitable for our application. On top of Node.js we use a web application framework called Express, which provides a set of features for building web applications [21].

Alongside these main technologies some additional tools and libraries are used to speed up the development. The most important ones can be found in Appendix H

4.3 Dashboard

The Magnet.me dashboard is one of the interfaces to the underlying system, and runs fully in the webbrowser. This dashboard is the interface Magnet.me employees will use to interact with the system. It will initially serve as the content management system for the website, but in the future more features might be added to support the employees with their daily work.

Since the dashboard runs in the browser there are limited options as to which technologies can be used: HTML, CSS and JavaScript should be used to build the

[3]Multiple threads can be used as well, however these threads are not related to individual requests and thus not relevant here.

dashboard. However, there are many great libraries and frameworks available to work with.

4.3.1 Javascript Frameworks

As explained in the beginning of this section the dashboard is a thin client, running in a web browser. Therefore it is desirable to use a library or framework which provides functionality to implement a thin client which communicates with a REST API. Such a library should be able to help retreiving data from a REST API using Javascript, displaying that information using the DOM[4] and sending user submitted data back to the REST API using Javascript. The main features such a library should have are as followed:

1. An easy interface for communicating with a REST API over Ajax.
2. two-way data binding between JavaScript and the HTML.
3. Client-side routing support.

A few different libraries have been considered. The first library considered is the Backbone.js library [1]. Backbone is a framework which provides an interface to build rich client-side applications using the model-view-presenter design paradigm. Backbone also has excellent build in support to synchronize data from the client-side application over a REST API with a server. Unfortunately, Backbone provides little support for updating the DOM based on the models in the JavaScript application: all HTML should be rendered manually, Backbone will only put the result in the correct place. Also, Backbone is very minimalistic. As a result of this Backbone is really fast, but is also requires the developers to write much more code than with other platforms.

The next two frameworks considered are Ember.js [4] and AngularJS [3]. Both these frameworks provide a rich set of features to build rich client-side applications, but in a different way. Ember.js provides an interface to build applications using the model-view-viewmodel paradigm. Data is binded to the HTML using handlebars templates, defined in HTML script elements. These templates will be parsed by Ember and Ember will provide automatic two-way data binding based on these templates. Angular.js on the other hand uses an entirely different approach, different from most other frameworks. The first difference is that with AngularJS templates are defined inside the HTML code, using custom elements and attributes, or directives as they are called in AngularJS. The result is that the HTML which is send to the browser is not valid HTML by itself, but AngularJS transforms it into something the browser can understand. Angular parses the HTML itself, and provides two-way databinding using these directives. On the JavaScript side of the application controllers can be created which act upon (parts of the) templates. From these controllers REST calls can be made to the REST API.

[4]Document Object Model. This is an object representation of HTML which can be used to programmatically change a user interface.

The killer feature of AngularJS however has nothing to do with all of the above: AngularJS allows developers to define custom directives, which can be used as if they were already present in the HTML specs. This allows the developers to easily create rich usable HTML components which reduces duplicate code significantly.

Due to the vast amount of features AngularJS provides to build rich client-side apps AngularJS is used as the underlying framework for the Magnet.me Dashboard.

4.3.2 HTML Frameworks

Next to the Javascript frameworks there are also a few HTML frameworks, which provide a rich set of features to quickly build nice user interfaces. The two most well known frameworks are Twitter Bootstrap [5] and ZURB's Foundation [6]. Both these frameworks provide a grid to build responsive pages, a set of standard components, a default style, and a lot of options to extend and customize the frameworks. The differences between these frameworks are very minor and mostly boils down to a personal preference[5]. Due to earlier experience with Twitter Bootstrap that has become the HTML framework used for the Dashboard.

4.3.3 Other Tools/Libraries

Next to the two main frameworks used to build the Dashboard there are a few more tools and libraries used to build the dashboard.

The first one is jQuery [11], a well known library for manipulating the DOM. This library mainly provides a cross-browser interface to update the DOM, and is mostly used for small user interface related scripts.

The second library used is the RequireJS [9] library. RequireJS is a JavaScript module loader, which makes it easier to split up the scripts over multiple files without having to include many scripts from the HTML. RequireJS also provides some optimization tools, which can be used when preparing the application for the production environment.

The final important library used is the utility library Lo-Dash [12], which is also used for WHOOPS 4.2.

[5]There are a few important differences between these frameworks when developing for mobile as well. However, these differences are not relevant for the Magnet.me dashboard as we do not support the mobile platforms for the dashboard.

5 Implementation

5.1 Web Server: WHOOPS

The REST server provides most of the functionality of WHOOPS. It provides authentication, authorization, caching and data request handling functionality, each serving their obvious purpose. The Express framework used to build the REST API provides support for middleware, an approach in which distinct components can be chained, each serving a specific purpose. By implementing the distinct pieces of functionality using a middleware-based approach a pipeline for processing requests is created, as can be seen in Figure 5.1. The stages of the pipeline itself is managed by the Express such that the individual components do not need to know about each other. This decouples the components from each other resulting in a system in which components can be easily replaced, updated, removed or added without needing to interfere with other components[1].

5.1.1 Authentication

The authentication middleware is built with the help of the Passport module for Node.js [26]. Passport is highly configurable authentication middleware for Node.js with support for many authentication strategies such as OATH, Facebook and session based login. For this bachelor project we only implemented session based logins, but in the future other strategies might be added as well. Session data, such as the ID of the logged in user, is stored using a Redis database, such that multiple threads and/or processes can access the same sessions and sessions are not lost when the REST server is restarted. Figure 5.2 shows a sequence diagram of a user logging in.

5.1.2 Routing

The routing is done by Express itself. The routes are defined inside the application and Express implements the routing functionality. For example, to route a GET request to the `/api/v1/student/<id>` resource to a function called `getStudent` the following route can be defined: `app.get("/api/v1/student/:id", getStudent)`, in which app is an instance of an Express based server. Express will now handle the parsing of the requested URL and calling the appropriate request handlers. Other URL-specific middleware is routed using the same strategy[2].

[1] Some components do require the input to be transformed in a certain way or depend on some additional data generated by another components. However, these dependencies are on the data, not on the components.

[2] Most routes are not defined manually but generated automatically by the middleware components. For example, the authorization middleware can automatically configure these routes based on the schema definitions as explained in Appendix D

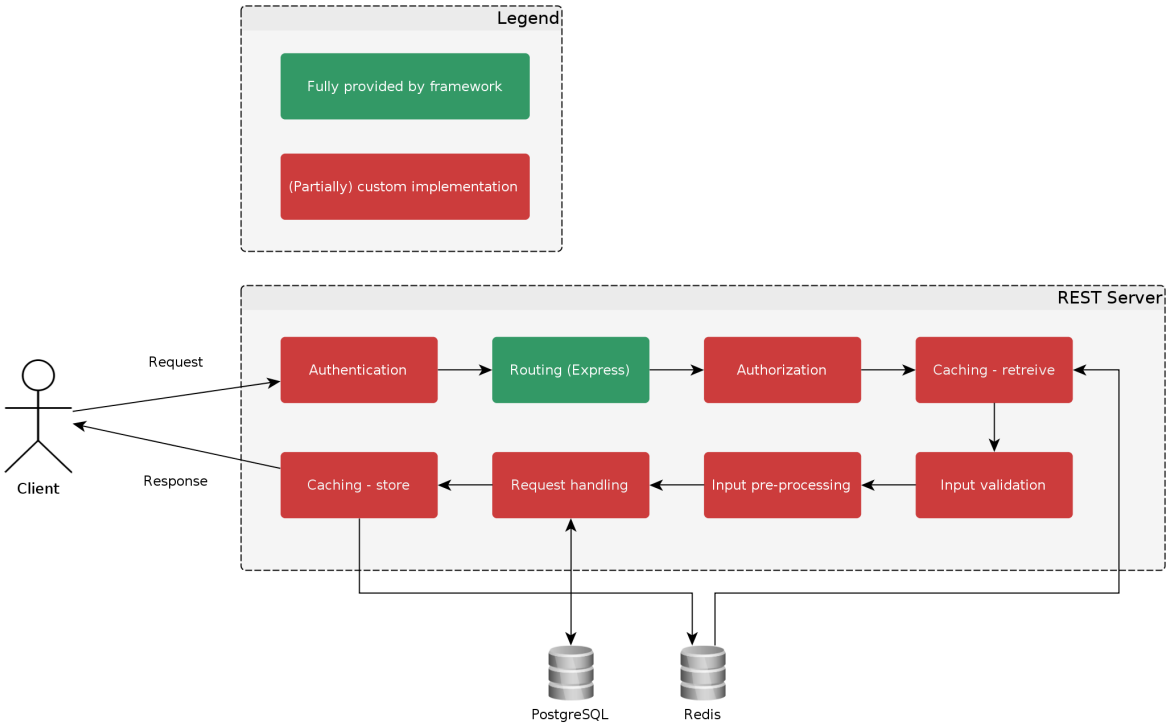


Figure 5.1: REST server pipeline

5.1.3 Authorization

Authorization is done on a per resource level. Some resources require different permissions from others, and also permissions can differ for the different HTTP methods, as specified in the functional requirements in section 1.4. For example, student might request company information using HTTP GET, but they can not change company information using HTTP PUT.

In the current system individual resources are not accessed directly. URLs can return different results and will processes submitted data differently based on which user is currently logged in. The information sent to the server is not directly linked to a specific resource, instead the server determines which resource needs to be updated when a form is submitted, based on session data. For example, when a student updates its personal information a form is submitted to a general URL (e.g. `https://magnet.me/site/student`). The server determines, based on session data, which rows needs to be updated.

With the REST API however, the client requests individual resources directly. This is done in compliance with the REST principles, which states that each resource should have a uniquer identifier [30]. Therefore a client can issue a request to a resource to which it should have no access. Because of this, authorization on a resource level is required to prevent users from accessing information to which it should not have access.

Another challenge is that permissions not only differ based on the role of the user,

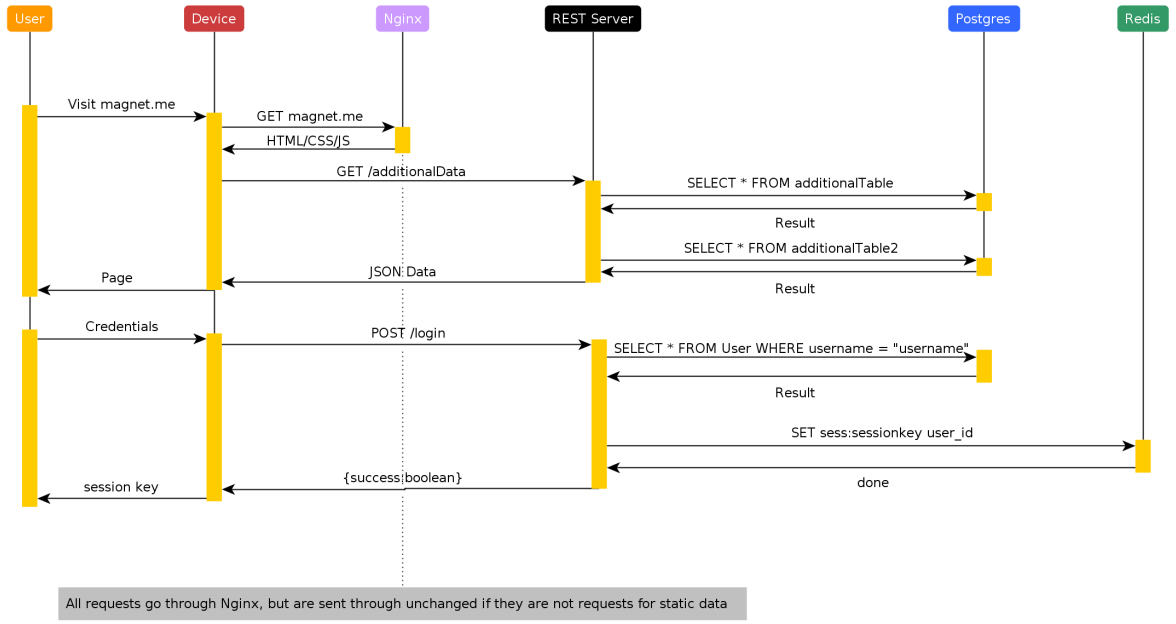


Figure 5.2: Authentication Sequence Diagram

but also on the individual user itself. For example, a student can only edit its own data, not the data of other students.

Permissions are configured in one file and enforced using custom built middleware. In these configuration files these permissions can be denied or granted per role and per route, either directly or via a custom authentication function. The custom authentication functions are needed for permissions which can not be defined using routes and roles alone, such as permissions to edit user data as mentioned above.

An explanation of the configuration file options can be found in Appendix D.

5.1.4 Caching

Caching is done using a Redis database. Response bodies of GET requests are cached in Redis hashes. Each hash is named by the requested resource, without domain and querystring. Inside each hash the requests are indexed by querystring, or "/" if no querystring was provided. By separating the querystring from the resource identifier both an entire resource cache and an individual request cache can be accessed in O(1) [18]. Each resource will be deleted after a specified amount of time, or when its underlying resource is updated.

The following list shows which commands will be send to redis when different requests are made:

GET /api/v1/example

Load from cache: `HGET /api/v1/example /`

Store in cache: `HSET /api/v1/example / {response body}`

GET /api/v1/example/1

Load from cache: `HGET /api/v1/example/1 /`

Store in cache: `HSET /api/v1/example/1 / {response body}`

GET /api/v1/example?_limit=20

Load from cache: `HGET /api/v1/example _limit=20`

Store in cache: `HSET /api/v1/example/1 _limit=20 {response body}`

POST /api/v1/example {request body}

Entire /api/v1/example cache needs to be emptied, since the collection is changed now[3]: `DEL /api/v1/example`

PUT /api/v1/example/1 {request body}

Both the collection and item cache need to be deleted, since they've both changed[3]: `DEL /api/v1/example /api/v1/example`

DELETE /api/v1/example/1

Both the collection and item cache need to be deleted, since they've both changed[3]: `DEL /api/v1/example /api/v1/example/1`

5.1.5 Input validation

All data which is send to the REST server should be validated before it is passed onto the request handlers. The REST server requires that the data is send as a JSON object with the application/json content-type header. If the correct header is set Express will aready check if the input is valid JSON and convert it to a Javascript object. To validate the actual content of the input a module called Amanda [15] is used. Amanda uses JSON Schema definitions [31] to validate Javascript objects. Amanda also allows to add custom properties to the schemas, which makes it possible to add custom validation. For each route both a POST and a PUT definition are specified such that Amanda can validate the input. If no schema is found for a route then its input will always be rejected.

5.1.6 Input pre-processing

The final step before the request can actually be processed is the pre-processing of the input. Some input requires to be parsed before it can be used. For example, the query string can contain filters, extra fields to be requested, and a few more options. The query string is already parsed into a key-value format by Express, but some more processing needs to be done to convert the pairs into descriptive objects which can be used by the request handlers. The pre-processing middleware will take care of that.

[3] If a subcollection is updated, such as `/api/v1/student/1/education`, then also the parent collections will be removed from the cache.

5.1.7 Request handling

The final step of the pipeline is the request handling itself. Thanks to the previous steps the request handlers do not need to validate their input anymore, any invalid input is already rejected. The request handlers usually map REST API resources onto database records and visa versa. Due to the size of the system and the limited time available it is not feasible to create all handlers manually. However, it is not easy to refactor these handlers. Each handler need to request different pieces of information from the database, some of which is spread over multiple tables. Fortunately we did manage to create a solution which could generate request handlers based on configuration objects.

In total there are five HandlerFactories created: there are HandlerFactories for updating, deleting, creating and retrieving individual resources, as well as a factory for retrieving collections. To generate the request handlers these factories require a resource definitions, whose format can be found in Appendix E.

The factory for generating request handlers for requesting collections will create a function which does the following: The handler will first determine which fields needs to be requested based on the "defaults" property of the definition and the extra fields requested through the REST API, which will be explained in section 5.2. Using the "properties" property of the definition it can determine how to map properties from the REST API to fields in the database, and thus constructing the "SELECT" part of the query. Based on the "extensions" property of the definition left joins are added to the query such that properties from multiple tables can be requested as well. Finally filters, which are also provided through the REST API, are added to the query. If a property has a load function defined it will be called once for each row, and the result of this function will be used for this property. Using these load functions it is possible to implement custom functionality for individual properties, without having to implement the entire resource handler by hand.

The handler generated for requesting individual resources simply uses the handler for collections with some predefined filters to select only one item.

Creating new items is done in a similar way. The generated handler determines in which order rows should be inserted by examining which tables have foreign key dependencies on other tables. It then inserts these rows in order. Again, it is possible to use custom functions for each property to execute some custom functionality. These save functions are executed before data is inserted, such that data can be transformed when needed.

The handler generated for deleting resources only has a simple implementation. It tries to delete the row from the basetable of that resource and relies on triggers defined in the Database, such as the "ON CASCADE" triggers, to delete rows from other tables.

Creating a factory for handlers which respond to PUT requests turned out to be the most challenging. One of the main problems is that the handler needs to find out if there already exist rows for a certain resource in the extension tables. To do so the PUT handler will first create a "SELECT" query to find the correct ids for each table. The construction of this query is done in a similar way as with the handlers for requesting

resources. Using the result of this query the handler can determine for each table if there is already a row for that resource: if the query returns an id for that table, there is already a row present, if it returns null, then there is not row yet. Based on these results it can determine for each property send with the request if it should update a row, or insert a new one. Also, the custom save function described for the creation handlers are also used here.

These factories do have their limit however: they cannot resolve one-to-many joins automatically and they require that each table has an id field. However, due to the possibility to inject load and save functions it is possible to work around these limitations by implementing these joins manually.

5.2 The API

The API is the interface provided by WHOOPS, through which the clients can request data information.

Applications will be able to request data, made available by the WHOOPS API. These requests will follow a predetermined form, as suggested below: Certain objects will be available as 'root' elements. The root elements of the Magnet.me dataset are the standalone elements which have a meaning on their own. Any element which has a meaning on its own will be a root element. For example, it makes sense to request an individual student resource, and thus "Student" is a root element and can be requested via `/api/v1/student/<id>`. However, it does not make sense to request a "job experience" element on its own: a job experience is always linked with a student and thus it will be a sub element of student. Since it is a sub element it needs to be requested via an URL like `/api/v1/student/<student id>/job/<job id>` Usually these root elements will be the starting points for retrieving data from the system, but exceptions can be made. Non-root elements can be used to find items which are related to the root element.

The modular buildup of the requests allows us to model it as a context-free grammar [34]. Quickly described, the format starts with a perspective, followed by an optional query string. The perspective may consist of two subsequent perspectives separated by a '/', or from an entity and an optional ID. The optional query string can contain key-value pairs which define filters on collections. For example, when the resource `/api/v1/company?location=Delft` is requested, only companies located in Delft will be returned.

Options starting with an underscore are threated as special options and their format is inspired by the Field Expansion feature of the Facebook Graph API [25]. The first of these options are `_limit` and `_offset`, which specifies the size and limit for collection results. The `_fields` options can contain a comma separated list of extra fields to be requested. If the requested fields are collections themselves, then the offset and limit filters can be defined on them as well[4]. The same syntax as with the Graph API is

[4]Note that this is not supported (yet) by the handlers generated from the factories, since they do not support one-to-many joins. Therefore, this needs to be implemented manually for each handler.

used: `.offset(x)` and/or `.limit(x)` can be appended to a field name to limit or offset an included collection. Nested field expansion, such as defined in the Facebook Graph API, is not supported (yet).

Based on this logic, the context-free grammar below is formed:

$S \rightarrow /api/V/EQ$
$V \rightarrow$ `api version`
$E \rightarrow P$
$E \rightarrow P/I$
$E \rightarrow P/I/E$
$P \rightarrow$ `perspective`
$I \rightarrow$ `resource id`
$Q \rightarrow ?F$
$F \rightarrow F\&F$
$F \rightarrow$ `parameter=value`

This can render the following examples:

`/api/v1/interest/` returns a list of all interests in the database

`/api/v1/event/13` returns the specific event with id=13

`/api/v3/student/421/education` returns the educations that 'belong' to the student with id=421

`/api/v2/conversation/5/member?_field=id,name&_limit=10` returns the id and name of at most ten members of the conversation with id=5

5.2.1 One-to-One

To keep the API as semantically correct as possible it will not be allowed that individual resources are requested via other root elements if these resources are also root elements. For example, one might request a list of universities of a student via the resource `/api/v1/student/<id>/university`. However, the details of a university do not belong to a student. A university is not a part of a student, nor is it a property of a student, and thus it does not make sense to model it as such by making it a sub element of student. Therefore, the details of university can only be accessed via its own URL, for example `/api/v1/university/<id>`. If such a resource is requested as a sub element a 303 HTTP status code will be returned, which instructs the client to look elsewhere.

This strategy not only has the advantage of being semantically more correct, but it can also improve the performance of caching. When each resource has only one URL then the cache does not need to know about multiple URLs pointing to the same resource. Therefore the cache can always returned cached results without having to know about aliases for certain resources. When multiple URLs could be used, the cache either needs to know which URLs correspond to the same resource, or it should cache each URL individually which results in redundancy in the cache.

5.3 Database

The database is designed using some (naming) conventions in mind. These conventions are explained before the database design itself is discussed.

5.3.1 Naming Conventions

These conventions are not only created to help developers quickly understand the tables, but also to make it easier to generalise database programming, using code generation for example.

- The underscore is a reserved character and should not be used in field or table-names, unless explicitely stated in another convention.
- Table names are always singular and in camelcase starting with an uppercase letter.
- Field names are in camelcase, starting with a lowercase letter.
- Junction tables are named after the target tables, seperated by an underscore character.
- Foreign keys fields are preferably named after the table and column name they refer to, starting with a lowercase letter and seperated by an underscore. This convention may be ignored when the resulting name is semantically incorrect, the name is not descriptive, or when multiple foreign keys to the same column are defined in one table.

5.3.2 International Standards

Some fields contain values for which an international standard, such as an ISO standard, is already widely used. If such a standard already exists, it should be used in the database.

The standards which we use in the database are as followed:

Country codes
ISO 3166-1 alpha-2 [22]

Language code
ISO 639-3 [24]

Currency names
ISO 4217-alpha [23]

5.3.3 Database Design

Since the database has more tables than will fit on a single page, the database will be describes hierarchically, beginning by describing the larger parts of the table and ending in a detailed explanation of the tables. The entire database design can be found in Appendix B.

5.3.4 Overview

As can be seen in Figure 5.3 the database can mostly be divided into 13 logical groups, leaving a 14th "Other" group. The two most important groups are the Company (green) and Student (red) groups, which represent the main actors for the Magnet.me website.

There are a few important groups connected to company. The first one is the "Minisite" group, the group which contains all data to build the minisites for each company.

Next there is the "EVI" group. EVI is an abbreviation for Events, Vacancies and Internships. These three entities have very similar properties and thus they are grouped together. EVI is also connected to Student with a dotted line: there is a connection between Student and EVI, but it is not a very strong one (as will be seen later on).

The following group connected to company is the "Recruiter" group. The Recruiter group contains the information of the recruiters which will use the Magnet.me website.

Connected to Company, Recruiter and Student is the "Message" group. This group is responsible for all data regarding direct communication between students and companies, such as group messages or personal messages between recruiters and students.

The final group directly connected with company is the "Network" group. This group contains all data regarding the Networks of a company, such as its members, name and invitation message.

Directly attached to the Network group is the "CMMON" group. This group contains all the company settings to match students to networks.

The other important group is the "Student" group. This group is obviously also connected to the Network group, since a network needs members.

The next group is the "Curriculum Vitae" group. This group contains all student data regarding their Curriculum Vitae, such as education, side activities and previous jobs. Note however that although both the Curriculum Vitae and the CMMON groups are needed for the matching process, there is no direct connection between the rows of these groups.

The following group is the "MagnetStore" group, which contains information about the Magnet.me reward system ("Magnets").

A table which is not directly linked to one of the key groups is the "I18N" group, in which I18N is an abbreviation for 'Internationalization'. This group contains tables to support internationalization of the website from a recruiters point of view. These tables are used to enable support for international, multi-language companies.

The final group linked to the others is the group "File". This group contains all the meta data needed to map files from the filesystem onto entities in the database.

Now there are only two groups left: "MM Dashboard" and "Other". The MM Dashboard group merely contains data used for the Magnet.me Dashboard. This includes login data and access rights for individual employees of Magnet.me. This group is not connected to any other group in the system, it is solely used for the Magnet.me dashboard and has no meaning for the rest of the system.

The final group is the Other group. This group contains all tables which do not

belong in any of the other groups.

Below, we will explain the contents of each group, when significant. We do not provide an explanation for every group, however, because not all information in the database is relevant for this report.

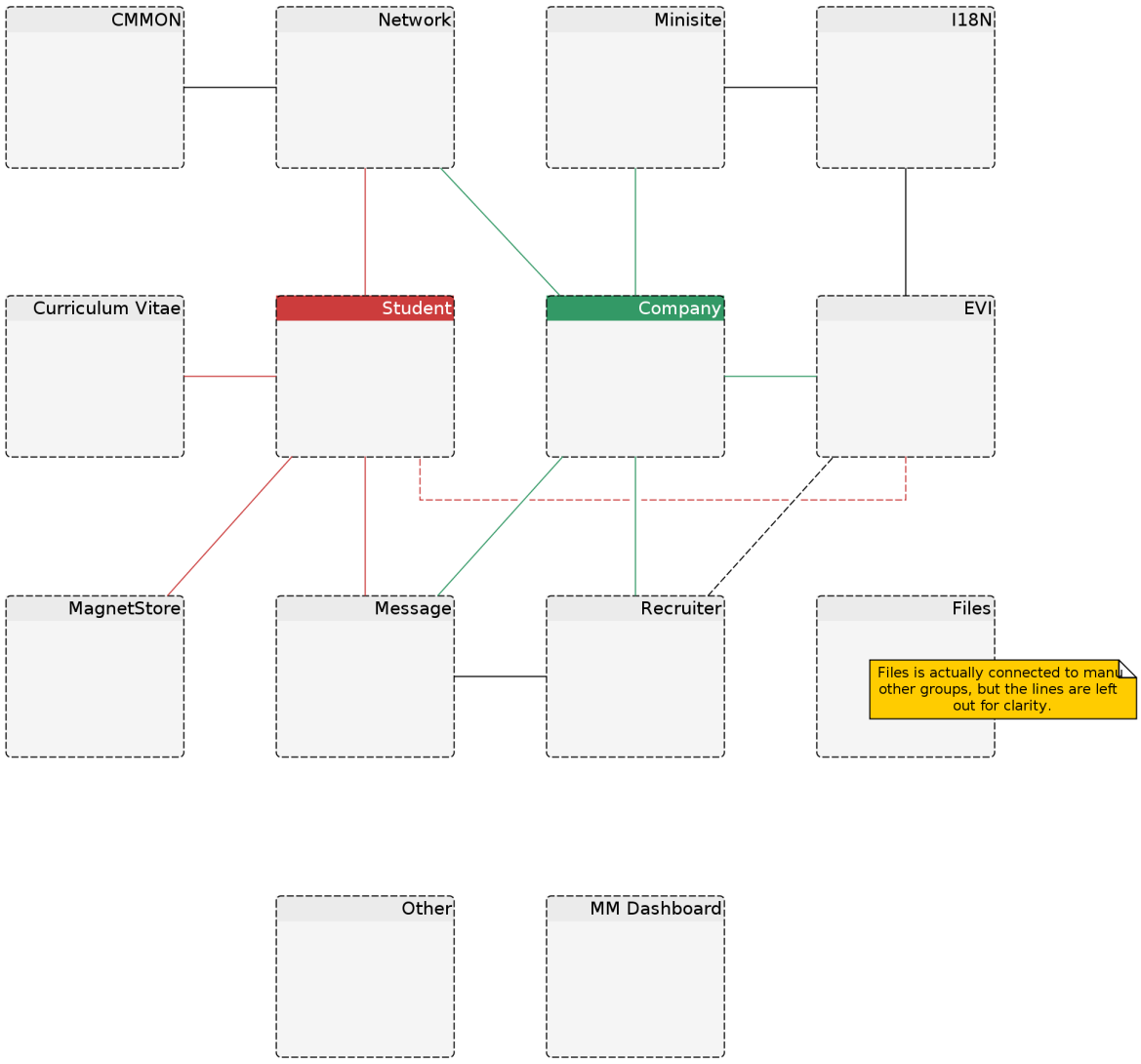


Figure 5.3: Database overview. Colors are only added for clarity

5.3.5 I18N

Internationalization support consists mostly of text translation. Therefore, the 'I18N' group in the database is a group of tables which, together, reference translations of the same text.

This is done with a two-table structure, as shown in Figure 5.4, where the first stores the original text with data about the language. The second table stores translations

with a foreign key to the original text table, to link the translations. Using this structure, translations of a text can be found, while information about the original text is stored.

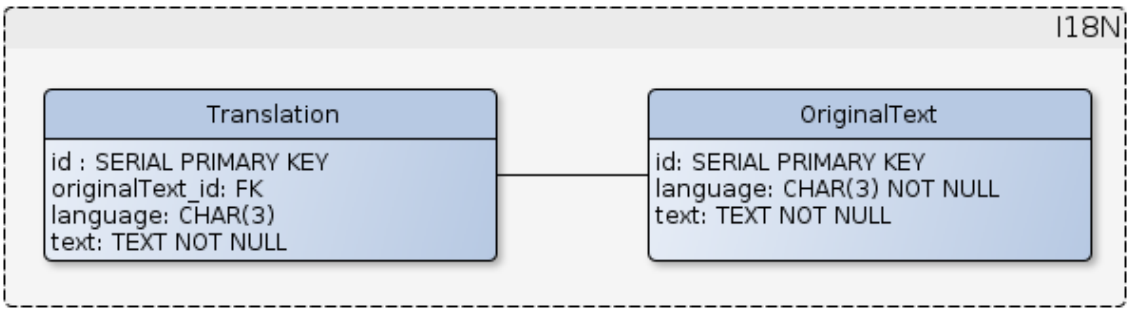


Figure 5.4: Overview of Database Internationalization table group.

5.3.6 File

The 'File' group (see Figure 5.5) contains tables that, between them, set up a structure that resolves to a file path for a certain resource. These resources can be (profile) photos and videos, but also student resumé files, articles about companies and message attachments. The wide variety of files that can be uploaded to the system create a somewhat complex structure, with the simple function of referring to a file path on the server, so a client can download the file. Junction tables are used for connections with elements that (can) reference to multiple files; examples are product photos in the MagnetStore,and applicable object photos (vacancies, etc.). Other tables, like Video, have (at most) a single file linked to them, and therefore do not require a junction table.

5.3.7 Resumé

The student's resumé is spread over several tables in the system. These tables include 'SideActivity', 'Employment', and information like that in 'Student_Interest' to store information about a student's work branch preferences.

The most complex part of this group (and perhaps even the entire database), however, is the storing of information about the student's educational career. In Figure 5.6, the Education table group is given. As one can see, a difference is made between levels of education.

Primary education (also known as 'elementary education') is not included, because this information is not used for matching students and companies in CMMON. Secondary education (latter part of middle school and high school) and tertiary education (college, or university) are stored.

Because tertiary education is standardized more than secondary education, for example in the international Bachelor/Master structure, this educational phase is considered most important, and most information is stored about this phase.

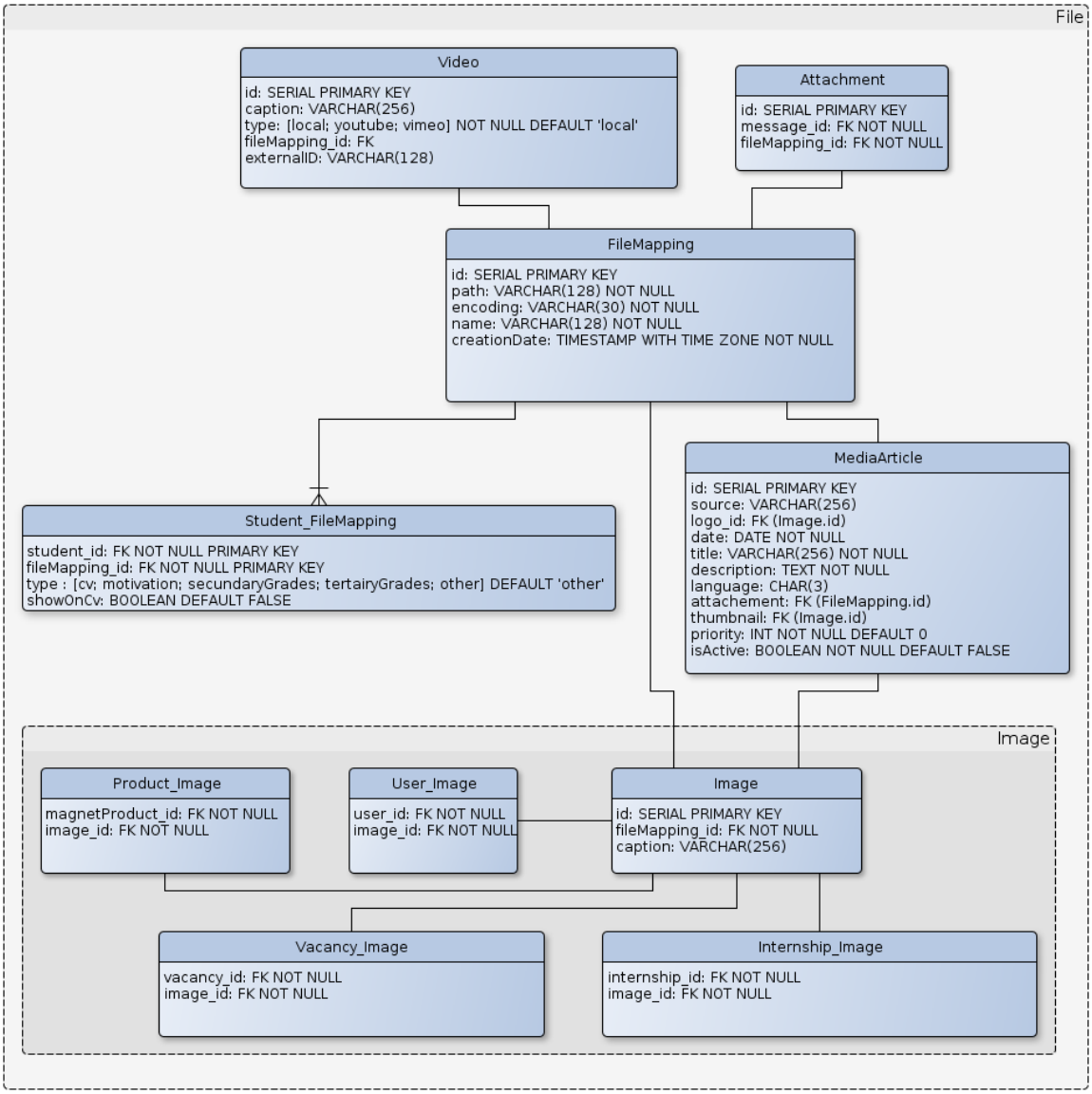


Figure 5.5: Overview of Database File table group. The junction tables (recognized by the '_') indicate connections to other tables, which are not depicted here.

The 'TertiaryEducation' table stores information about the study itself, where the cross-section with the student about fulfilling this information is stored in the junction table 'Student_TertiaryEducation'. EducationBranch stores information about the field of the study, for example 'IT'. The EducationalInstitution stores information about institutions, like universities. These 'provide' studies; for example, the institution 'Delft University of Technology' provides the study 'Computer Science'. The level of educational programmes provided is stored per institution as well, in the hasAcademic (the dutch 'HBO') and hasProfessional (the dutch 'WO') fields.

Secondary education is stored more freely, and therefore the structure is less complex. Therefore, only the 'SecondaryEducation' table is necessary here, and fields for specialization and level are even optional to support more education types.

The 'OtherEducation' table stores information about extracurricular courses that a student might have followed.

Finally, a followed exchange programme is stored with links to the institution where this programme was followed.

5.3.8 EVI

Students can apply to certain objects in the system; vacancies, events or internships. These events, abbreviated with EVI, are provided ('posted') by companies, and stored in the database in the applicables group, shown in Figure 5.7. The three EVIs inherit their shared properties from the 'ApplicableObject' table. Applicable objects can have an internal application, which stands for an automated application process within the magnet.me system, instead of an external process via the company. In this case, the 'InternalApplication' table stores information about this process. Any questions a company wants to ask, including optional multiple choice options and an answer provided by a student are stored respectively in 'Question', 'QuestionOption' and 'Answer'.

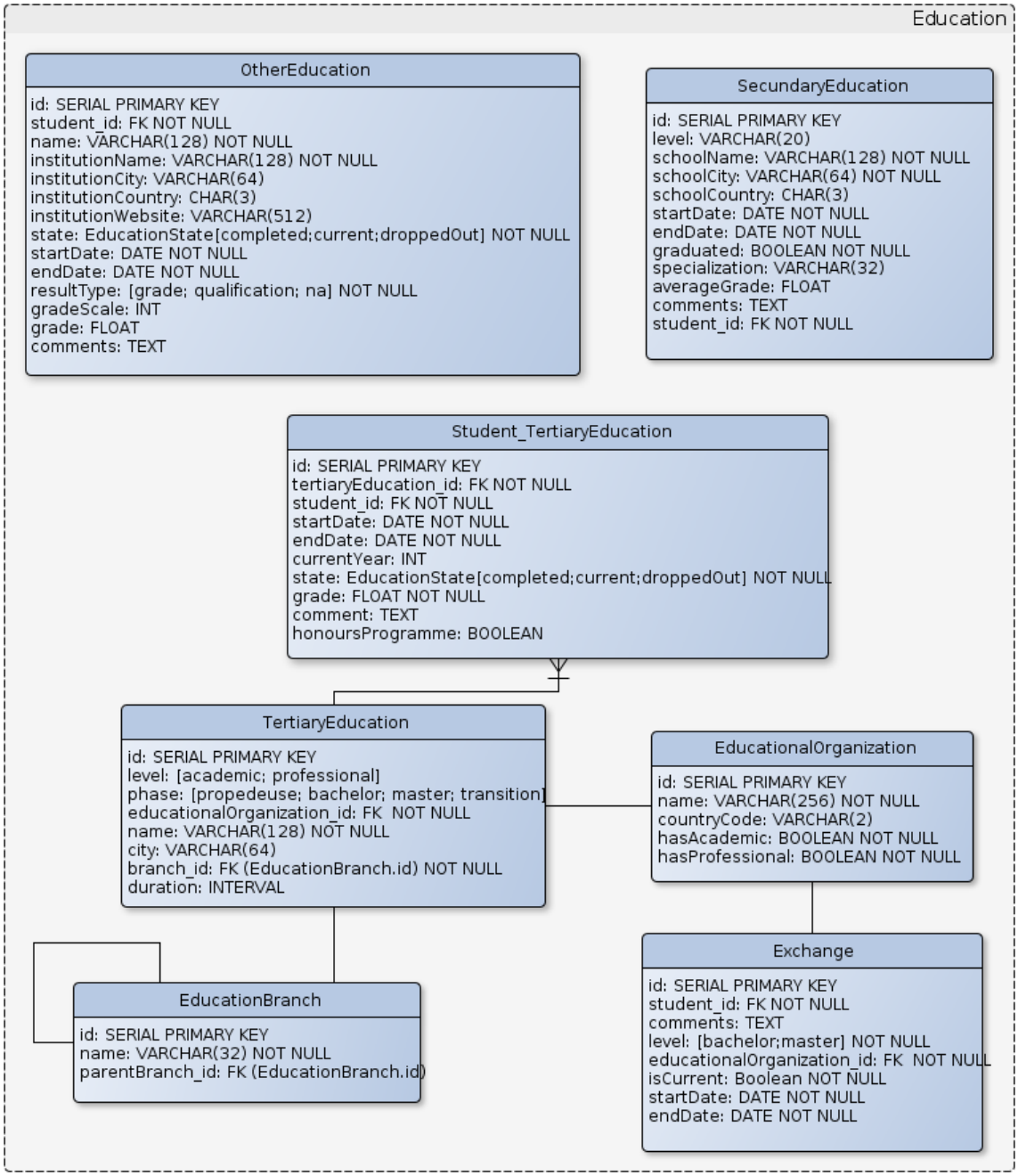


Figure 5.6: Overview of Database Education group.
OtherEducation, SecundaryEducation, Student_TertiaryEducation and Exchange are linked to the Student table.

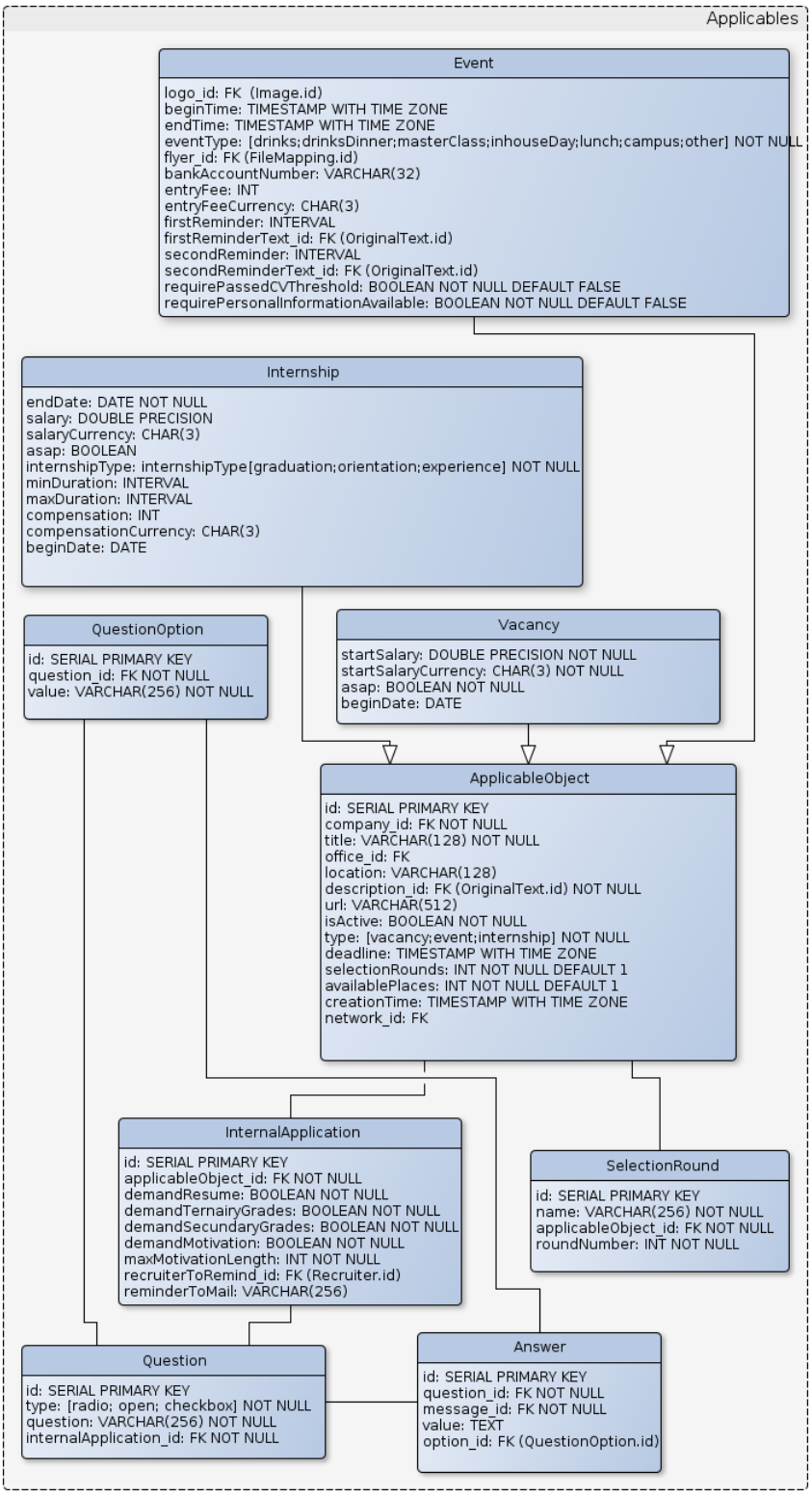


Figure 5.7: An overview of the applicable object table group, or EVI group. EVI stands for Event, Vacancy and Internship; the applicable objects in the system.

5.3.9 Messages

The database stores messages in the 'Messages' table group as seen in Figure 5.8. Messages are modelled as a 'message' object with possibly multiple 'message receiver' objects, depending on the amount of receivers the message has. This model is used to support private messages, group messages and 'broadcast' messages sent to, for example, all members of a network. The separation of the model into two objects is required to monitor the status of the message in general (whether it has been deleted, for example), and the status of a message in relation to a single receiver, like the read status. Conversations are supported by providing a conversationID in the 'MessageReceiver' table. This information is placed in the MessageReceiver table instead of the Message table, because the conversation depends on the relation between the message and its receiver, instead of just on the message.

The messages optionally have attachments. These attachments can be found through a link with the FileMapping table (outside the Messages table group), through the 'Attachments' table.

Finally, a message can be an application message, in which case the message is linked to an applicable object through the 'ApplicationMessage' table. If this applicable object is an event, there is an optional fee to be paid. In this case, a direct debit transfer can be used, for which the data is stored in the fields in 'ApplicationMessage' table, starting with 'debit'.

5.4 Magnet.me Dashboard

By request of mr. Karremans, the supervisor from Magnet.me, not too much time was spend to develop a great graphical design. The lead designer of Magnet.me, mr. Holleboom, will design the final graphical user interface of the Magnet.me dashboard before the system will go live. For the bachelor project only minimal design choices were made in cooperation with mr. Holleboom and mr. Karremans. In cooperation with them, we have chosen a Bootstrap theme to quickly set up an efficient graphical interface. Since the final graphical user interface will be designed by mr. Holleboom the design will not be explained much further in this report. Implications on the implementation of the graphical interface are limited to the usage of classes in HTML code. These classes, like `inline-input`, simply arrange predefined field templates from Bootstrap.

5.4.1 AngularJS directives

The Magnet.me dashboard has been implemented using the AngularJS framework. As described in Section 4.3.1, AngularJS allows developers to create custom directives, which we have gratuitously used in the dashboard. A great example is the 'Currency' directive (displayed in Figure 5.9), which consists of a dropdown for currency choice, and a number field for the amount of money. This directive has proven to be useful in several templates, returning as a basic property in numerous entities in the system.

We have defined directives like the currency directive for pagination (to spread the

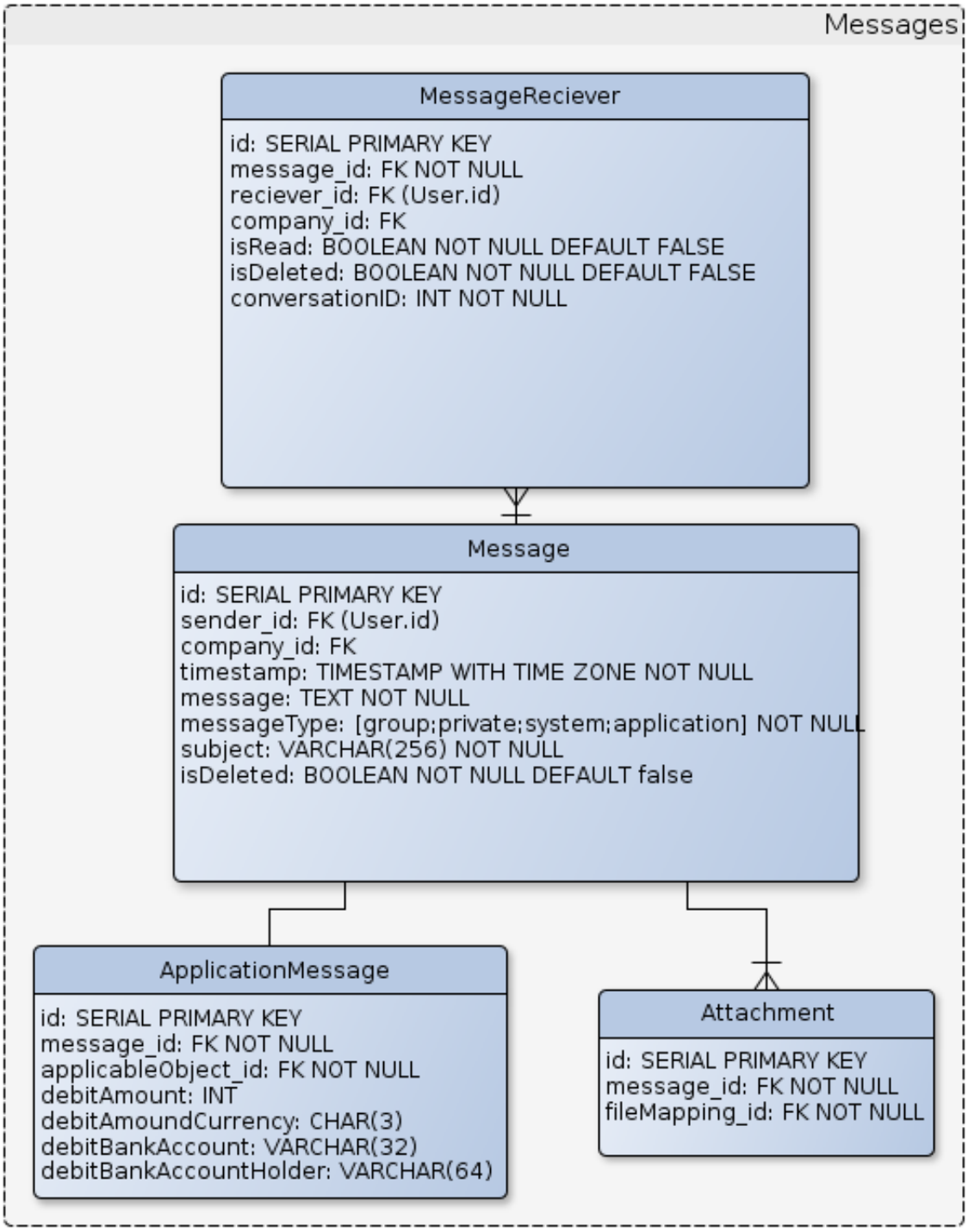


Figure 5.8: Overview of Messages table group.

overview of, for example, all companies, over several pages instead of in a single massive

Figure 5.9: The custom-defined currency directive, as displayed in the Magnet.me dashboard

list), but an interesting one is the mm-elements directive. This directive generates directives for data fields of an entity by checking the data type, and replacing itself with the appropriate directive. These directives can result in default html elements or custom directives; for example, the company's 'name of employees' field results in a default 'text input' element. On the other hand, a company's 'starter salary' field generates the custom defined 'currency' directive, as described earlier.

This mm-elements directive allows us to generate templates simply by using angular to create an mm-elements directive for all fields in the data object. The directives then replace themselves by the appropriate directive for the field data type, which will in turn be replaced by plain html elements, creating a template for the data object. In short, we use one auto-repeated directive to generate an entire template by using a versatile, custom-defined directive.

Combining these powerful directives with the Twitter Bootstrap template system has enabled us to rapidly but thoroughly set up pages of the Magnet.me dashboard. The separation of the template system and the directive (generation) system also result in great maintainability, because an adaption to the dashboard is as simple as changing a single block of code, without tracing the use of this code through the system.

6 Testing

6.1 Performance Test

One of the main objectives of replacement of the IT infrastructure is to prepare the software for an international release. Therefore, performance should be tested as well. Using performance tests we can verify that this system is faster than the current running one and that it can handle enough requests. Unfortunately, we did not have access to the source code of the current system, and we did not have direct access to the server as well. Therefore, it was impossible to setup a controlled test environment for the current system, we could only test on the live site.

6.1.1 WHOOPS

A controlled environment could be setup for the new system. For this test two virtual servers in a virtual lan have been used: one which acts as the REST server, the other for sending requests to this REST server and measuring performance. By having the servers in the same virtual lan the latency due to distance is (almost) zero. Therefore the measured latencies are only created by the REST server, and thus the peformance of the REST server can be measured directly. Both virtual servers had one core and one gigabyte of ram assigned to it.

The REST server has been prepared with 10.000 student resources, and caching was initially disabled. Each student has a row in the User table and a row in the Student table and thus requests to this resource need to perform a join. Now using a linux tool called `httperf` [10] random[1] requests to student resources were made. These requests were made in sessions of ten seconds at different rates. The rate at which requests were made was initially set at 50 requests/second and was increased on each test by 50, up until a few steps after the server could not handle the traffic anymore. The actual request rate however did differ from the request rate set, the rate set was merely a target rate for `httperf`. The allowed timeout was set at 10 seconds.

The results of this tests can be seen in figure 6.1. The graph might be a little surprising at first: near the end the response times decreases when the request rate increases. However, with a little more information the cause of this behaviour can be found. As can be seen in the graph, below a rate of 50 requests per second the server responded very quickly: the average request time was near ten milliseconds. When more requests were made the response time increased, just as expected. However, after a certain point the response times actually decreases. The reports of `httperf` showed that starting at the peak of the graph the server starts return HTTP status codes starting with 5, the server error codes, instead of the HTTP status code 200

[1] 1.000 resource urls were randomly generated in advance and used for the tests

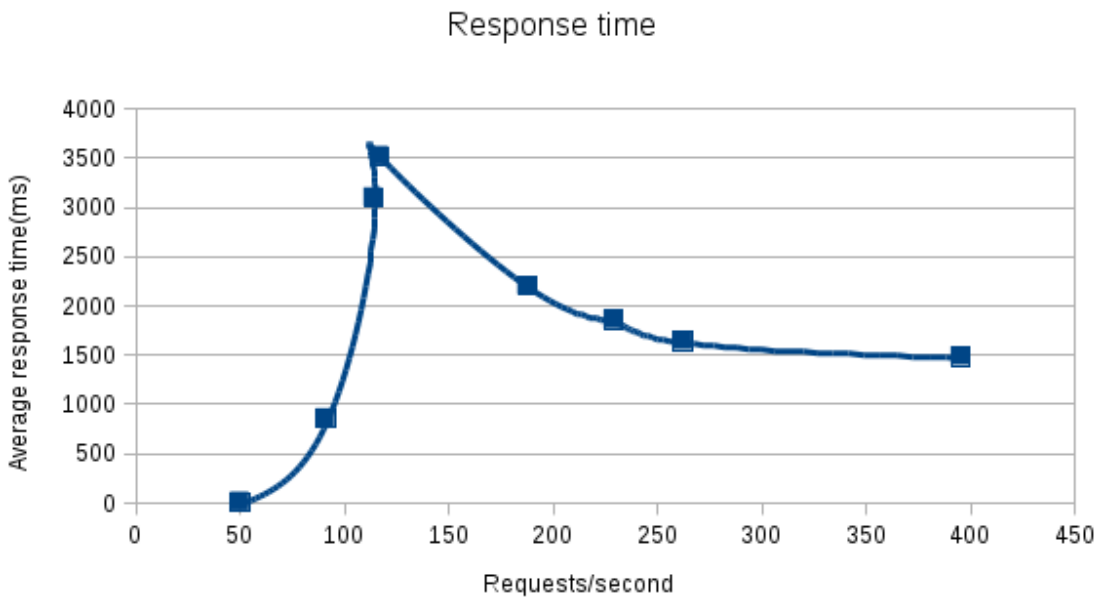


Figure 6.1: Response time WHOOPS

(OK). These errors are most likely generated before requests to the database could be made, and thus the time needed to generate these error responses is much smaller than the amount of time to generate the correct responses. When the amount of requests increased, the percentage of errors increased, resulting in a lower average response time. Of course at this point the server is already highly overloaded and such a situation should never happen, but it was interesting to see this behaviour.

However, at this point the data was requested from the database using an ORM layer for Node.js. This ORM layer did already caused some problems, one of which was that it did not support left joins. Therefore left joins had to be done in the Node.js part of the application. This is a bad situation, the database can do joins much better and faster, and thus we decided to run the same tests, but now with the ORM library removed. The result can be found in figure 6.2. As can be seen WHOOPS can serve up to twice the amount of requests per second before it is not fast enough anymore. Also, the same pattern as before can be seen: after a certain point the server start responding fast with error codes.

Finally we also tested how much requests the cache could handle. As can be seen in figure 6.3 the cache could respond to at least 350 requests per second before it started to slow down noticeably.

6.1.2 Current system

Since a controlled environment could not be setup using the current software, our best option was to simply test the response times for a single page under different loads. For this test requests were made to the homepage of Magnet.me using the same strategy

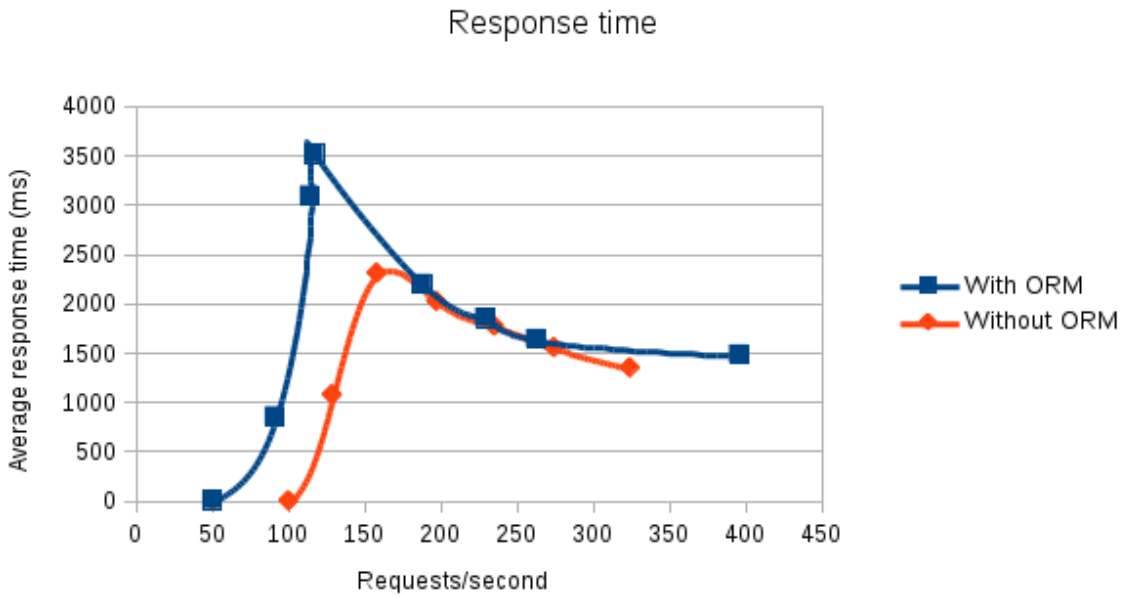


Figure 6.2: Response time WHOOPS - with and without ORM

as with the test for WHOOPS. The homepage was chosen because it does not require authentication, but does use the database for finding icons of companies. Only the raw HTML page was requested, other assets, such as the icons, were not loaded. This test was peformed late at night at a time when (almost) no other requests were made to the website.

Unfortunately there were still a few obstacles to overcome: First of all we had not other server near the live server and thus we had to deal with network latency. Ping could not be used either to detect the response time, since the server did not respond to ping. Therefore, 10 requests to the robots.txt file, a static file on the server, were made and the average response time was taken as the network latency. This is not a very accurate measurement, but a decent educated guess with the limited options available.

Secondly, there are quite a few differences between the virtual servers used for testing WHOOPS, and the live server. The live server sends all its traffic encrypted over https, which was not setup for WHOOPS yet, which will delay the response a little more, and also the live server had to generate an entire HTML page, as opposed to only a JSON object as the REST server does. On the other hand, information from only one table needed to be displayed and this table only had a few hundred records, as opposed to 10.000 for the WHOOPS test. Also, the live server has 16 cores and 64 gigabytes of ram (as opposed to only 1 core, 1 gigabyte for the virtual server).

Clearly the live server has a huge advantage over the virtual server, but unfortunately this was impossible to eliminate. This advantage however made the results only more interesting: Just as with the WHOOPS test the test on the live system also started

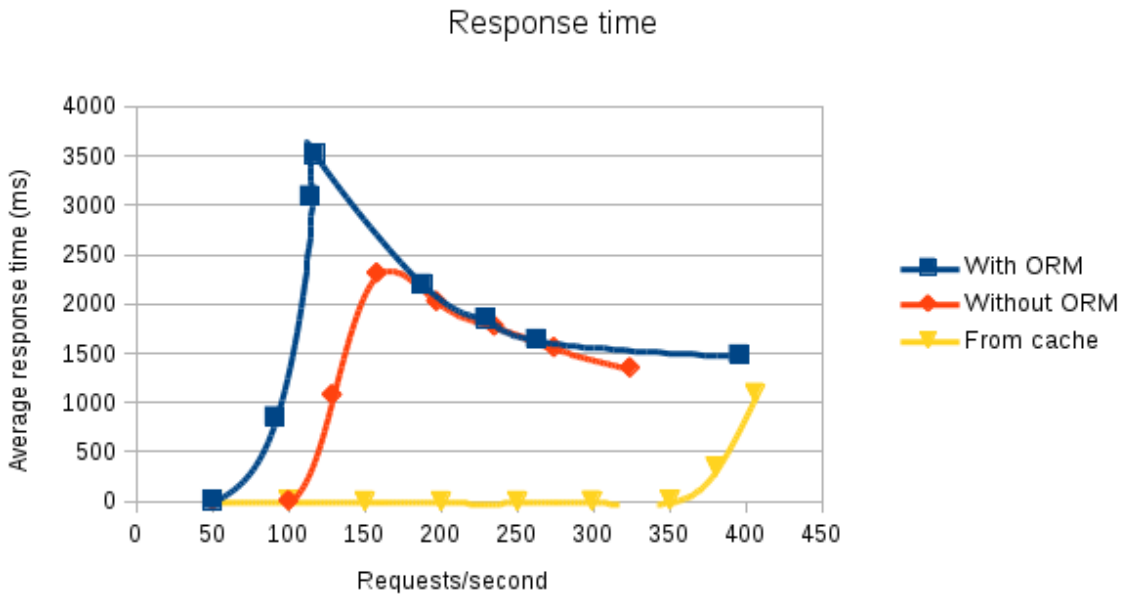


Figure 6.3: Response time WHOOPS - from cache

with 50 requests per second for a period of 10 seconds. However, at this first test the live server already responded to more than half of the requests with an HTTP error code. These results were realy surprising, but reruns of the test resulted in the same results. Also, twice shortly after we started a run an external monitoring tool notified us that the server took too long to respond (more than five seconds), while everything was fine just before and just after the test. This showed that the current live server really can not handle 50 requests per second, despite its 16 cores and 64 gigabytes of ram.

6.1.3 Interpretation of the results

Clearly the newly developed REST server can handle a lot more traffic than the current system, and thus it is a vast improvement. However, using a 'back of the envelope' calculation we can actually say a little bit more about the limitations of the new system.

According to the statistics of Google Analytics at peak times a little less than 1000 page views per hour are made. There are currently around 7000 user accounts, students and recruiters combined, registered on the Magnet.me website, so at the peak the number of requests per hour is about 15% of the number of users in the system. Even without caching the virtual server running WHOOPS can respond to roughly 100 requests per second within a few milliseconds each. This equals 360.000 requests per hour. If this equals 15% of the number of users in the system than it means that it could handle over two million users.

This number is however most likely way off for several reasons: with two million users the database is much bigger and will most likely give some performance issues. On

the other hand, the cache will increase the number of users it can handle significantly. If enough resources are cached it can more than double the amount of requests, and thus the amount of users, this system can handle. Also, this system is tested on a very limited server. If a dedicated database server is used, caching is properly tuned and WHOOPS runs on a more powerful server, the number of users this system can handle might as well be much higher than 2 million. Unfortunately due to the lack of proper resources the real limit can not be established yet. Further research is required to do so.

7 Process

The process of implementing the system design has been a 'bumpy road'. Setting up the IT infrastructure within Magnet.me was, and is a difficult challenge, as there is no earlier experience within the company regarding any IT infrastructure, let alone the fact that there is little to no knowledge about this subject present. This challenge has proven to be even larger than we estimated in advance. In this section we will review on this process, and while we are doing so, we will reflect on implementation challenges we have encountered.

When the assignment was first formed, it stated that "the (Caret) CMS had to be replaced by an in-house developed CMS", and was not specified any further. At this time, it was not clear that the entire system was build on top of the Caret CMS and replacing the CMS was impossible without replacing the entire infrastructure. This meant that, if we were going to replace the CMS, we had to build an entire new infrastructure; a much larger assignment.

7.1 Project Management

In our project process, we have monitored our project and the process by using the project management tool Trac, extended with Agilo for Trac. We have hosted this browser-accessible tool on the magnet.me development server, to keep it in private management. On this same server, we have also hosted git which provided a so-called origin for our repositories. In the process, this origin is where we have pushed our branches, which triggered the automated testing further explained in Section 6.

Using Trac, and Agilo for Trac to adapt the tool to the Scrum-based process we aimed for, we have monitored our project with the ticket structure provided in the tool. Embedding Trac in our process has been a little difficult; it is easy to forget to report your progress in a ticket. However, we do think it has proven to be a useful tool to monitor our progress, as we can review upon it easily.

7.1.1 Behaviour Driven Design

An always important part of software development is testing. To ensure testing is done regularly, we have used a technique called 'Behaviour Driven Design'. Derived from Test Driven Design [28], Behaviour Driven Designs prescribes a test for functionality is written before the actual functionality is implemented.

To include this in our process, we have used the 'Mocha' module for Node.js. This module provides a testing framework, including a structure for the BDD tests. In Appendix F, an example of this structure has been given, and in Figure 7.1 you can see a resulting auto-generated webpage showing the test results. In our project, we have used git hooks [13] to perform testing automatically, on the server side. These hooks

are activated when a team member pushes his git commits to the server, triggering the tests. The server then generates the test result pages as shown in Figure 7.1, which we could access internally to reflect on our functionality.

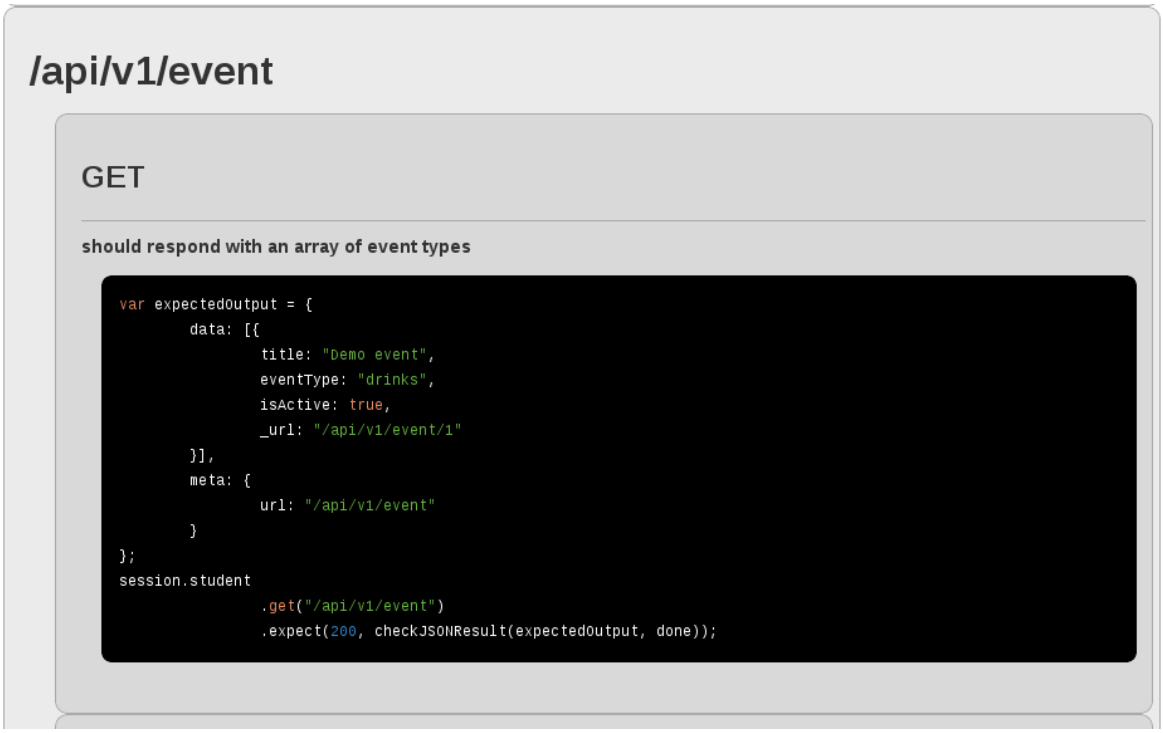


Figure 7.1: Part of an auto-generated Mocha test result page

7.2 Difficulties

In this process, we have encountered several significant difficulties which have caused delays, and/or have forced us to redesign/add parts of the system. The way we encountered these difficulties, and how we overcame them, is described below.

7.2.1 Size

As we started designing and implementing the project, we already knew that the assignment was larger than we had expected, because an entire IT infrastructure had to be developed. However, we did not realize the vast amount of data entities in the system, nor the implications this would have on the project duration. We first noticed this problem in the database design step, which included over 80 tables, and an estimated 700+ fields. However, it began to cause major planning problems in the design and implementation of the API, where http request handlers had to be implemented for every data entity.

As we started implementing these request handlers, it quickly became obvious that there was not enough time to implement all the request handlers manually. Fortunately,

we did notice in the few request handlers we implemented, that there was a common structure which we could generalize to some extent. This gave us the idea of writing a request handler 'generator', which could provide the request handler. These request handlers are based on hand-written configuration objects which define what data the request handler should retrieve from the database. These configuration objects were much smaller and easier to write, which saved us significant amounts of time.

Unfortunately, the problem that the size of the system formed has not been solved entirely, since the amount of entities is simply too large for a Bachelors Project. At the time of our Bachelor Presentation, the system does not yet support all data entities. This functionality will be implemented after the end of the Bachelor Project. However, the underlying structure of the system will be completed for the Bachelors Project.

7.2.2 Overengineering

Another problem we encountered was our own idealistic goal to make the best decision in every part of the system, making it ultimately scalable, performing and maintainable. This is simply not possible, as even these three form a trade-off amongst each other. We focussed too much on the design decisions, locking up the process; we were overengineering.

The solution to this was to choose for an option that was 'good enough', sufficient for the demands. Because the requirements that were given to us were vague, determining a 'sufficient' solution was, at times, a challenge itself.

7.2.3 Planning

As we both lacked experience with projects like these, making the planning was difficult to start with. Partly due to this lack of experience, but also due to the size of the project, the planning was not realistic. The design phase took more time than we had planned for this phase, resulting in a major early delay. We have not been able to catch up with the planning after this.

Matching System: CMMON

Because of the delay that formed early in the project, it became clear that certain parts of the project had to be postponed. The Matching System was deemed a too large component of the system to be implemented in the bachelor project, and did not form a dependency for the Magnet.me Dashboard. For these reasons, CMMON's implementation has been postponed until after the bachelor project has been finalized. Of course, this decision was made with mister Karremans, the Magnet.me supervisor, after discussing the problem with him.

7.2.4 Malfunctioning Libraries

Using Node.js, several ORM[1] libraries were available to us. These libraries are meant to provide an extra abstraction layer between the application code and the database,

[1]Object Relationship Mapping, simply put a translation layer between the database and the system implementation

with the extra benifit of allowing a more code-like syntax to communicate with the database. We initially decided to use the `node-orm2` [8] library for our system.

However, soon we ran into quite a few issues. For example, the library did not provide any way to peform left joins (only full joins), which resulted in having to peform multiple queries to retreive results. Since Node.js uses callbacks for asynchronous operations this resulted in a so called callback hell, in which the developer has to manage many asynchronous callbacks.

The library also contained a few bugs, some of which we could patch ourselves, which slowed down the development process even further. When later on in the process peformance tests showed that the ORM library was the main bottleneck for our application we decided to abandon it altogether in favor of a tailer made system for generating the correct queries, as described in section 5.1.

It is unfortunate that these problems were not known beforehand, which is a risk taken when developing with a such young platform. Choosing a library can therefore be a hard decision as libraries may not be reviewed thoroughly by the community.

8 Requirement Evaluation

As our Bachelor Project is nearing its end, we look back to see whether the requirements we have determined at the start of the projects have been fulfilled. We have evaluated the requirements in collaboration with the product owners to check sufficiency, and we have used performance testing for quantifiable requirements. We will not evaluate every requirement separately, because most requirements have been met. The requirements that have not been fulfilled (completely) are named, followed by an explanation and the possible solution.

In Section 6, we have widely covered the performance testing, and will therefore not focus on those test results here.

8.1 Functional requirements

The system-wide multiple language support is provided by storing multiple translations under a same text id, allowing for language-specific text retrieval.

8.1.1 Web Backend

An interface for the web applications (demand no. 1 in 8.1.1) is provided by the RESTful design in the form of the API, as explained in Section 5.2. Authentication, which is required to access the system, is provided by the Passport module, as explained in Section 5.1.1 under 'Authentication'. This interface is accessible for all type of web applications through standard HTTP requests, and therefore the software listed in demand no. 2 in 8.1.1 can be built on top of the backend. The API is an abstraction of the data layer, and requests sent to our server are authorized first (demand no. 3 in 8.1.1). This authorization is explained in Section 5.1.3 under 'Authorization'.

Web Backend Requirements

Nr.	Description	Explained in Section:
1	It should provide an interface for the web applications including:	5.2
1.1	An abstraction for the data layer	5.2
1.2	An authentication interface	5.1.1
2	The following types of software should be able to be build on top of the back-end:	5.2
2.1	The website	5.2
2.2	The Dashboard	5.2
2.3	A mobile app	5.2
2.4	3rd party software	5.2
3	Requests should be authorized	5.1.3

Many of these requirements are fulfilled by the functionality described in Section5.2, as this section provides a connection for external applications, and an abstraction for the data layer.

8.1.2 Dashboard Requirements

Unfortunately, not all functionality of the Dashboard has been implemented. Our delay has forced us to postpone the implementation of some functionality to a moment after the project. During the bachelor project, at least the following functionality has been implemented: The Dashboard is indeed only available for magnet.me employee´s. The system shows the most recent data to all users as a result of the caching strategy used, which is described in Section 5.1.4 under 'Caching'. The Dashboard allows data manipulation of company entities and student entities, except for passwords and auto-generated fields such as a creation time stamp.

Dashboard Requirements

Nr.	Description	Explained in Section:
1	The Dashboard should only be accessible for Magnet.me employees	5.1.3
2	All data provided by the companies and students can be viewed and edited, with the exception of passwords	
3	The Dashboard should always show the most recent data available	5.1.4
4	Different employees can have different access rights to parts of the Dashboard	*
5	Subscriptions for companies can be managed	*
6	Translations can be added for the website	*
7	Statistics can be viewed from the Dashboard, including:	*
7.1	New subscriptions over time	*
7.2	Accepted and rejected matches per company	*
7.3	Accepted and rejected matches per student	*
7.4	Geographical distribution of companies and students	*
8	Matches can be made manually	*

* These requirements have not been fulfilled completely. However, the backend fully supports this functionality; it has only not been implemented in the dashboard due to time constraints.

8.1.3 Matching System

The requirements for the matching system have not been fulfilled, because it has not been implemented yet. However, the backend does provide the functionality for the matching system to be attached to it; after implementing CMMON, the connection with the system will be made quickly.

8.2 Non-functional requirements

All non-functional requirements have been fulfilled, except for no. 4 in 8.2: 'Subsystems should not interfere with each other'. Because the implementation of CMMON was postponed, the implemented systems do not influence each other. We can therefore not test whether the systems truly do not interfere, but we can however predict that there is very little chance that this will happen.

This prediction is based on the structure of the system, which enables us to spread each of the system's components onto a separate machine. Furthermore, CMMON will be working on a duplicate of the working database, copied with a non-blocking method to prevent from bothering WHOOPS in the process. This approach will allow the

systems to be non-interfering.

Below, we have added the table of non-functional requirements, where we have added a column referring to the section where the fulfillment of the requirement is explained when applicable.

Non-functional Requirements

Nr.	Description	Explained in Section:
	System wide	
1	Security measurements should be automatically enforced where possible	5.1.3
1.1	All data should be sanitized by default before being used as output	
2	The systems should be robust	
2.1	Every request should receive a response	
2.2	An error of a subsystem should never let the entire system crash	
2.3	All systems should recover from any non fatal error	
3	The systems should be scalable across multiple servers	4.2
4	Subsystems should not interfere with eachother	
5	New data should be visible to all subsystems within 10 minutes of an update	5.1.4
6	The system should support an international release of Magnet.me	
	WHOOPS	
1	Requests should be processed within 100ms	6.1

As can be seen in the table above not all non-functional requirements are explicitely mentioned in this paper, and some require a little more explanation. This is mostly since some of these requirements were automatically fulfilled by certain design and/or platform choises.

All data should be sanitized by default before being used as output

This non-functional requirement is fulfilled at multiple parts of the application. At the REST server this requirement is fulfilled by the Node.js runtime: internally the output is always represented as an object. Javascript has a built in function, `JSON.stringify`, which can transform a Javascript object into a JSON string, already fully sanitized.

At the Magnet.me Dashboard this requirement is fulfilled by Angular.js. The HTML is never edited directly by our code. Instead data is bound to variables, and Angular.js updates the DOM based on these variables. Angular.js already makes sure that this output will not be interperted as HTML.

Every request should receive a response

This requirement is also fulfilled on multiple levels. The REST server itself implements a timeout for all of its requests. When the server takes to long to respond to a query, an error code will be send back to the browser. If for some reason the REST server fails to do so, then Nginx, the reverse proxy, will also send an error code after a certain timeout. Only when the entire server is down requests will not receive a response anymore.

An error of a subsystem should never let the entire system crash

Due to the strict seperation of the subsystems a subsystem can never let another subsystem crash directly.

All systems should recover from any non fatal error

Since CMMON is not implemented yet, and the dashboard can always be recovered by refreshing a page, only the recovering of the WHOOPS system will be explained here. In the worst case scenario the Node.js application will crash entirely due to some programming error. However, by running the application using a tool such as PM2 [7] the application will automatically be restarted in a clean state as soon as it crashes. As long as all critical dependencies are online, PM2 will keep the application running.

New data should be visible to all subsystems within 10 minutes of an update

Due to the caching technique used the cache will never contain old data. It will either contain the newest data for a resource, or no data at all. Also, writes are always done directly to the database. Therefore currently new data is directly visible to all subsystems.

When CMMON is implemented however, data can be updated without going through the REST API. This can however be solved very easily by setting an expiration time of less than 10 minutes for resources that might be updated outside of the REST API.

9 Conclusion

This project's result is a complex system, built to be flexible, maintainable and reliable. It will function as the IT infrastructure for Magnet.me, and will be extended with the public website, the

Since April, a complex IT infrastructure has been setup to replace the current Caret CMS used by Magnet.me. It consists of the following parts:

The Database has been built with PostgreSQL. With over 80 tables and several hundreds of fields, the dataset it represents is massive.

WHOOPS , short for Web-Hosted Omnipotent On-demand Providing Server, is the server system that provides a RESTful API for clients to communicate through. It also provides authentication and authorization to allow certain actors within the system limited access to the data stored in the database. WHOOPS does this using Node.js modules like Express for routing, and Passport for authentication. All requests sent to WHOOPS are routed by an Nginx web server, which provides static resources, such as the HTML templates, and routes requests for dynamic data through to the API.

The Magnet.me Dashboard , where Magnet.me employees have internal access to the systems data. Employees can look up, edit and delete close to all data in the system here. The Dashboard will also provide useful statistics about user activity and demographical information, and will be an extensive tool for the employees to manage the systems content.

The system has been built to be flexible, maintainable and reliable. Performance tests have already pointed out that it performs significantly better than the system it replaces, the Caret CMS. Early estimations indicate that it performs so good that it will be able to support an international release of Magnet.me.

Because the assignment was so much larger than originally intended, not all functionality has been implemented. However, with the finished backend, this functionality can be implemented rapidly. When the public website for students and recruiters is ready to be supported by the new backend, the entire infrastructure developed by Click will be replaced by the new system.

In the end, we are proud of what we have managed to achieve. It is a pity that we have not been able to implement the entire system, but considering the size of our team and the size of the assignment, this was to be expected. Reflecting on our project, we should have foreseen that our planning was not realistic. This project has taught us a lot, and we both see the experience that we have gained as a valuable part of our further careers.

10 Reflection

Looking back on the project and the process we have gone through, we can reflect on our decisions and actions.

10.1 Positive Points

The points in our process that we see as positive are the following:

Research

Because of intensive research, we have been able to consider all options in important decisions well. Therefore we feel confident that we have chosen an at least sufficient solution for the problems we have encountered, and we believe that this has helped us get to the result we have achieved.

Generalizing complex logic

Due to the unexpected size of the project we were forced not only to work hard, but also to work very smart. This resulted in solutions in which relatively complex logic was generalized to speed up our process. Of course generalizing logic is very, very common in software development, however we have been able to generalize some non-trivial complex logic such that large parts of this complex logic could be defined using simple configurations instead of custom implementation. This has proven to be very effective solution that we're quite proud off.

Communication

We believe that our intensive, open communication with both the management of Magnet.me and each other have helped us to catch problems early. This prevented having to spend time redesigning or rewriting system components, thus making the process more efficient.

10.2 Lessons Learned

Do not underestimate the assignment!

At the start of the project, we severely underestimated the size of the assignment. When beginning a similar project, we will study the assignment more closely, and try to get a grip on the size of the system and the assignment before 'diving in'.

Start with the end earlier

We have made the mistake of being so busy with the programming work, that we started on the final document too late. Describing the process and design of this massive project was, like the project itself, more work than we anticipated. Because of that, we could not hand in the report in time to give mister Bozzon

enough time for reviewing, but we were forced to rush the last improvements on the day of the deadline. We hope to solve this by planning the finalization of the project more strictly in future projects.

10.3 Software Improvement Group Reflection

In our first SIG reflection, some duplicate code was pointed out to us, and the Unit Size was too large in some components of the system. We were praised for the use of testing code, and the conclusion was that our code scored "above average".

We fixed the problems described by SIG, and later in the development process, we submitted our code for a second evaluation by SIG. They acknowledged that we had taken their comments into account, and had improved what they pointed out to us. They did however find some duplicate code in the parts of the system that were implemented after the first evaluation, which kept the score for duplication equal to the first evaluation score. Also, some of the new code came out to be too large in a few places. Our score for Unit Size was higher, but not significantly.

Finally, the volume of test code that we wrote was praised, which concluded in a score equal to that of the first evaluation; 4 out of 5.

Bibliography

[1] Backbone.js. http://backbonejs.org.

[2] Click. http://click.nl, Accessed April 22th 2013.

[3] Angularjs - superheroic javascript mvw framework.js. http://www.angularjs.org, Accessed April 24th 2013.

[4] Ember.js - about. http://emberjs.com, Accessed April 24th 2013.

[5] Bootstrap. http://getbootstrap.com, Accessed April 26th 2013.

[6] Foundation: The most advanced responsive front-end framework from zurb. http://foundation.zurb.com, Accessed April 26th 2013.

[7] Unitech/pm2. https://github.com/Unitech/pm2, Accessed August 1st.

[8] Node orm library. https://github.com/dresende/node-orm2, Accessed August 5th 2013.

[9] http://requirejs.org, Accessed August 6th 2013.

[10] httperf man page. http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.txt, Accessed August 6th 2013.

[11] jquery. http://jquery.com, Accessed August 6th 2013.

[12] Lo-dash. http://www.lodash.com/, Accessed August 6th 2013.

[13] Git hooks. http://www.githooks.com, Accessed August 8th 2013.

[14] Promises/a+. http://promises-aplus.github.io/promises-spec/, Accessed Juli 13th 2013.

[15] Amanda. http://redis.io/commands, Accessed June 10th 2013.

[16] The web framework for perfectionists with deadlines. https://www.djangoproject.com/, Accessed June 25th.

[17] What is twisted? http://twistedmatrix.com/trac/, Accessed June 25th.

[18] Command reference - redis. http://redis.io/commands, Accessed June 3rd 2013.

[19] Node.js. http://nodejs.org/, Accessed May 14th 2013.

[20] Nginx. http://nginx.com/, Accessed May 15th 2013.

[21] Express. http://expressjs.com/, Accessed May 16th 2013.

[22] Country codes - iso 3166. http://www.iso.org/iso/country_codes, Accessed May 20th 2013.

[23] Currency codes- iso 4217. http://www.iso.org/iso/home/standards/currency_codes.htm, Accessed May 20th 2013.

[24] Language codes - iso 639. http://www.iso.org/iso/home/standards/language_codes.htm, Accessed May 20th 2013.

[25] Field expansion - facebook-developers, Accessed May 21th 2013.

[26] Passport. http://passportjs.com/, Accessed May 21th 2013.

[27] The Organic Agency. Apache vs nginx performance comparison. http://www.theorganicagency.com/apache-vs-nginx-performance-comparison/, Accessed June 23th.

[28] Dave Astels. A new look at test-driven development. *http://blog.daveastels.com/files/BDD_Intro.pdf*, 13(06):2010, 2006.

[29] DreamHost. Web server performance comparison. http://wiki.dreamhost.com/Web_Server_Performance_Comparisonhttp://wiki.dreamhost.com/Web_Server_Performance_Comparison, Accessed June 23th.

[30] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

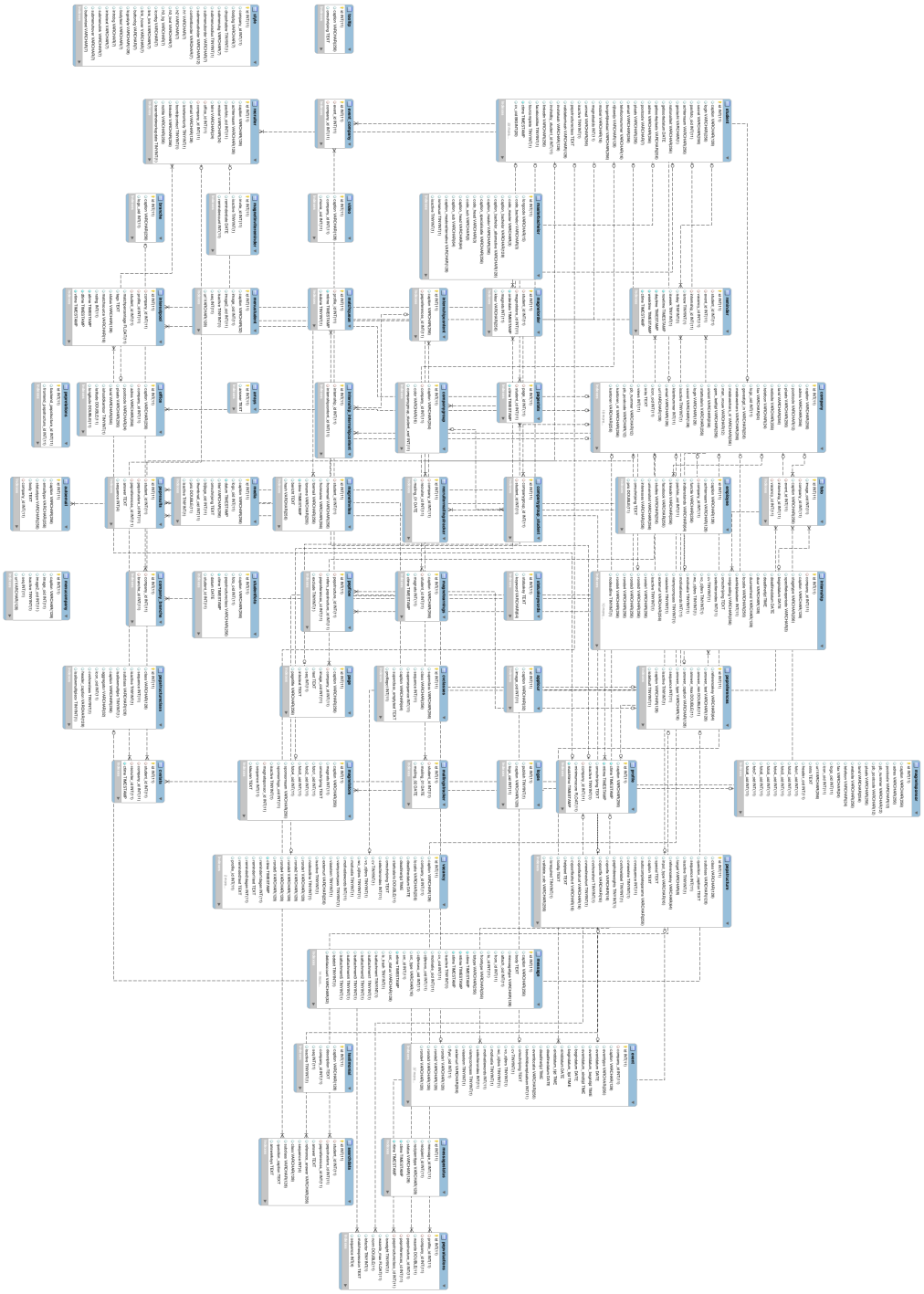
[31] Internet Engineering Task Force. A json media type for describing the structure and meaning of json documents. http://redis.io/commands, Accessed June 10th 2013.

[32] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

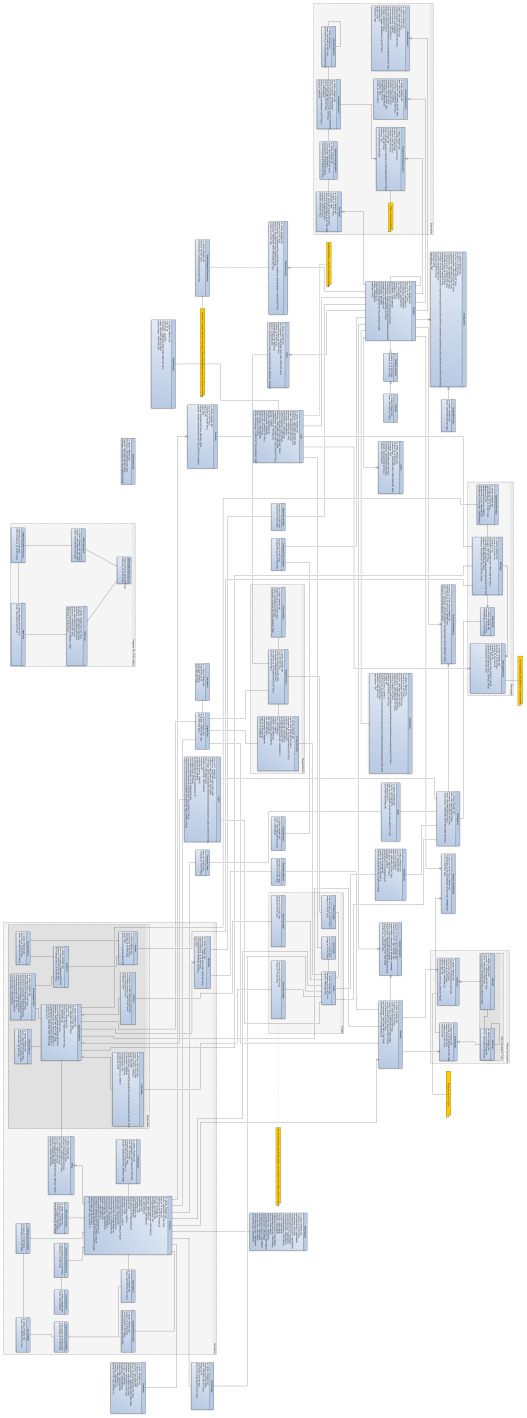
[33] Tiddo Langerak and Alex Walterbos. Orientation report replacing the magnet.me it infrastructure and content management system. 2013.

[34] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing, 1997.

A Old database design



B New database design



C Matching System

The matching system of Magnet.me uses a set of linear operations based on matching criteria provided by companies. The data provided by students is categorized. For example, there are categories for education and relevant working experience.

A category can have subcategories. Different categories can have different levels of nesting. Each category has a weighting factor. Inside the deepest subcategories are rules.

Each rule has a condition and a score associated with it. Students profiles can score points in each (sub)category by fulfilling a condition. All points scored inside a (sub)category are summed and the result is multiplied by the weighting factor of the category. The resulting score will be the students score for this category.

By recursively summing scores and multiplying the intermediate scores with the weighting factors of their parent categories a final score will be achieved. If this score is higher than the threshold score provided by a company, a match will be made.

Some rules can specify an early exit: if students do not fulfill those conditions, it will not be matched with that company regardless of its score.

D Authorization Schema

Permissions are defined in a separate module using a Javascript object literal. Each permission has its route as a key and an permission object as value. The permission object has http methods as keys and an allowed-roles definition as value (see later on). If all http methods require the same role(s) a shorthand notation can be used: simply assign the allowed-roles definition to the url.

The allowed-roles definition can be one of the following:

- A binary combination of roles. If a user has one of these roles he will be granted access to the resource.
- An authorization function. The authorization function will be passed the express request object as its only parameter. It should return a promise which will resolve to either true if the user is allowed access to the resource or false if the user is not.
- A array of tuples. Each tuples first value is the role and it's second value should be either a boolean value or a deferred function which resolves to a boolean value. If a users role resolves to true, permission is granted. Otherwise, it is denied. In case of get requests the second value can also be an array containing the extra fields which may be requested. Default fields should be omitted. Permission is granted when no fields other than the ones defined in the array are requested.

An example can be found on the next page.

Example

```
1
2  var permissions = {
3    "/api/v1/company" : {
4      //Everyone can request a list of companies
5      GET : roles.ALL,
6
7      //Only unregistered users and Magnet.me employees can
           create a new company account
8      POST : roles.VISITOR | rolse.MAGNET_ME
9
10     //Deleting and putting to a collection is not allowed, and
           thus omitted.
11   },
12   "/api/v1/company/:id" : {
13     //Everyone can request the details of a company
14     GET : roles.ALL,
15     PUT : [
16       //Magnet.me employees can update any company
17       [roles.MAGNET_ME, true],
18       //Recruiters can only update data for the company they
             are working for
19       //isEmployee is a function which checks if that
20       [roles.RECRUITER, isEmployee]
21     ],
22     //Only maget_me employees can fully delete a companies
           account,
23     //to prevent accidental loss of data.
24     DEL : roles.MAGNET_ME
25   }
26 }
```

E Handler Factory Definitions Specification

The definitions passed into the Handler Factories should be objects with the following fields:

baseTable (required)
: This is the name of the base table for this resource. By default properties will be mapped onto this table.

properties (required)
: A list of properties which can be requested. Properties which are to be found in other tables than base table should be prepended by "¡Tablename¿.", and the correct join rules should be defined. Instead of a property name it is also possible to define the property using an object, which provides some more flexibility. These object should always have a name property, which tells the factories the name of the property for the REST API. A few more options can be defined as well: tableName and columnName can be defined to tell the factories where to find the values for the property in the database. Also a load and save function can be defined. The load function gets the result of the database query as input, and should return a promise which resolves into the value for that property. The save function gets the input value for that property as its input. It should return a promise as well which resolves to either a value for its column, or an object containing column-value pairs if multiple values needs to be stored.

defaults (required)
: A list of properties which will be included in the response bodies of GET requests by default.

collectionDefaults
: When defined, this list will be used as the default properties for a GET request to a collection. If this property is not defined, defaults will be used for this purpose.

extensions
: The extensions property should be an object containing rules of how tables are related to each other. The object should contain key-value pairs in which each key is the name of a table and the corresponding value is the foreign key field. These foreign key fields should be defined as {tableName}.{columnName}. If the foreign key is a column from the extension table then the factories assume that this foreign key references the id field of the base table. If the foreign key is a column from any other table, the factories assume that this foreign key references the id field of the extension table.

Example
In this example a promise library called Q is used, which implements the Promises/A+ proposal [14].

```
1  var def = {
2    baseTable          : "User",
3    extensions         : {
4      "Student": "Student.user_id"
5    },
6    properties         : [
7        "User.firstName",
8        "User.lastName",
9        "User.email",
10       "User.nationality",
11       "Student.status",
12       "Student.universityMail",
13       {
14         name       : "password",
15         tableName  : "User",
16         columnName: "password",
17         save       : function (value) {
18           var deferred = Q.defer();
19
20           deferred.resolve(bcrypt(value));
21
22           return deferred.promise;
23         },
24         load       : function (row) {
25           var deferred = Q.defer();
26
27           //Password should of course never been shown
28           deferred.resolve("*****");
29
30           return deferred.promise;
31         }
32       },
33       {
34         name      : "fullName",
35         tableName: "User",
36         save      : function (value) {
37           var deferred = Q.defer();
38
39           //Find the first and lastname
```

```
40            var nameParts = value.split(" ");
41            var lastName = nameParts.pop();
42            var firstName = nameParts.join(" ");
43
44            //Store them seperately
45            deferred.resolve({
46              firstName: firstName,
47              lastName : lastName
48            });
49
50            return deferred.promise;
51          },
52          load      : function (row) {
53            var deferred = Q.defer();
54
55            deferred.resolve(row.firstName + " " + row.lastName)
                ;
56
57            return deferred.promise;
58
59          }
60        }
61    ],
62    defaults           : ["firstName", "lastName", "email"],
63    collectionDefaults: ["firstName", "lastName"]
64  };
```

F Mocha Example

```
1  /*
2   * Begin with a describe, which describes a system component
3   *  or a general section of functionality.
4   * Within this 'describe', one can define subcomponents with
5   *  a nested 'describe', or use 'it' to define functionality.
6   */
7  describe("GET request", function(err, done){
8
9    /*
10    * This nested describe defines a subcomponent of the
11    *  general test for 'GET request'.
12    * It will test the request url ../api/v1/company
13    */
14   describe("/api/v1/company", function(err, done){
15
16     /*
17      * This is is the highest nesting level, where
18      *  functionality is defined in the first argument.
19      * The second argument is a callback function which
20      *  truly implements the test.
21      */
22     it("should respond to the request with a list of all
          companies",
23        function(err, done){
24
25       /*
26        * This is where test functionality is written, using
27        *  for example:
28        *  assert() to test output
29        *  called(), calledOnce(), etc. to test if functions
30        *    are called etc.
31        */
32     }
33   }
34 }
```

G Caching Sequence Diagrams

The sequence diagrams below show a schematic overview of how the cache will be updated on certain types of requests. Some components in the system have been omitted since they are not relevant for these diagrams.

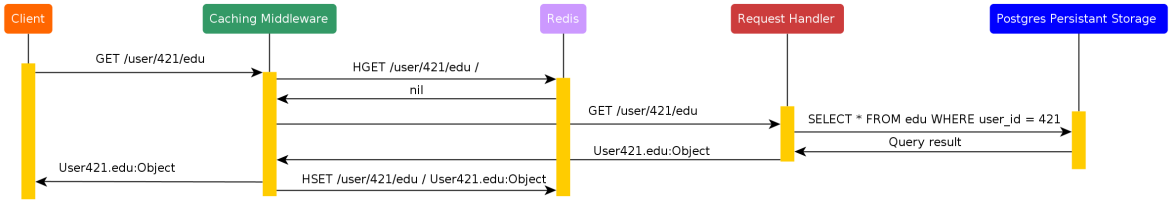


Figure G.1: Empty cache, GET request

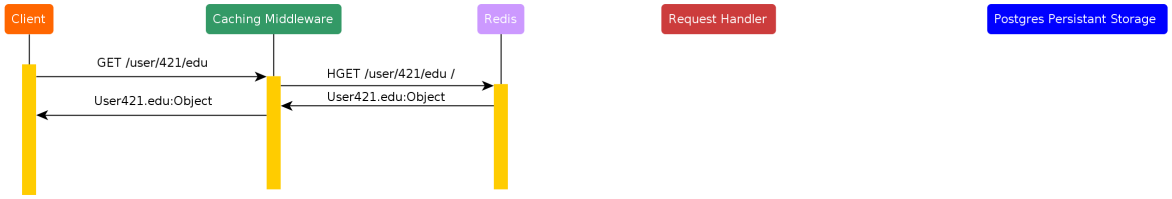


Figure G.2: Non-empty cache, GET request

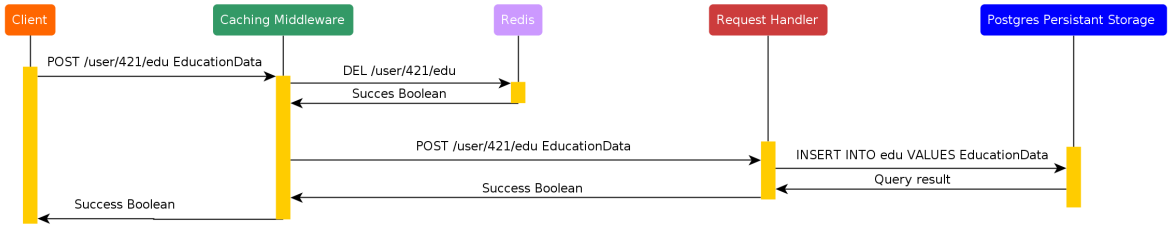


Figure G.3: POST request

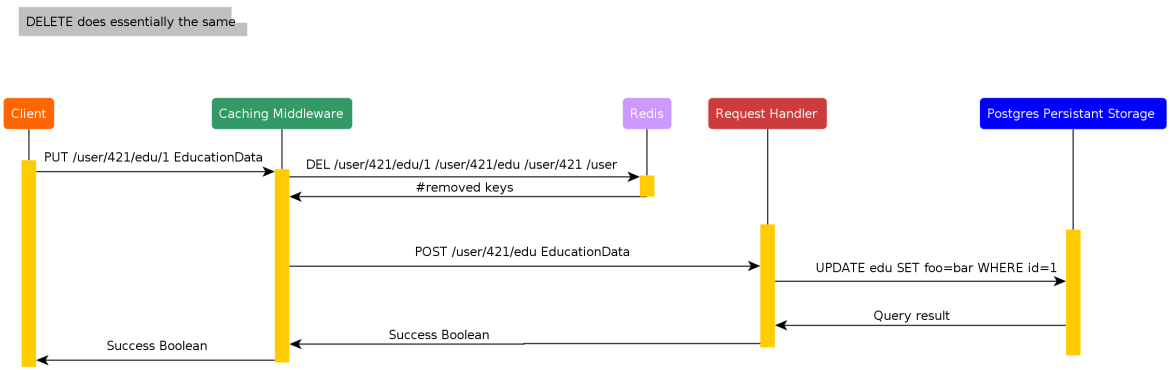


Figure G.4: PUT request

H Additional Tools and Libraries WHOOPS

Alongside the main technologies which are used there are also quite a lot of tools and libraries used to speed up development or to help organizing code better. Below a list of the most important tools can be found.

Mocha
Mocha is the unit testing platform used to test the REST server.

Q
Q is a library which implements the Promises/A+ specification [14], a specification for a clean interface for asynchronous code.

Async
Async is another library to improve asynchronous code. This library provides an interface for executing multiple asynchronous functions in series, parallel, or in some complex order.

Lo-dash
Lo-Dash is an utility library for JavaScript which offers a lot of functions inspired by functional programming for dealing with arrays, collections, objects and more. This library reduces the amount of boilerplate code significantly.

Debug
Debug is a module for Node.js for logging. This module improves the debugging output and it allows the developers to categorize log messages such that log messages can be shown selectively.

I Project Proposal

Project Proposal
BSc Technische Informatica

offered by

Magnet.me

Willem Ruyslaan 225
3063 ER Rotterdam

www.magnet.me
010-7630506
Info@magnet.me

Technische Informatica

Bachelor Project

Faculty of Electrical Engineering, Mathematics and Computer Science, Mekelweg 4, 2628 CD Delft, The Netherlands

BachelorprojectTI-EWI@tudelft.nl
TI3800 (15 ECTS)

Contact and Coordination

Martha A. Larson
M.A.Larson@tudelft.nl
http://homepage.tudelft.nl/q22t4/

Hans-Gerhard Gross
H.G.Gross@tudelft.nl
http://swerl.tudelft.nl/bin/view/GerdGross/WebHome

Company description

Magnet.me is an internet startup involving a site where students fill in their resumé, and companies create selection criteria for their networks. Based on the resumés of students and the selection criteria of the companies, both can be matched; students will be invited automatically to join the networks of these companies based on their profiles, and receive updates from the companies regarding new vacancies, internships and events. Magnet.me is completely turning around the recruitment process; through Magnet.me each company already has a network before a vacancy has even come up.

Project description

The Magnet.me website is currently built on an external backend and Content Management System (CMS) which does not allow for scaling and is not oriented towards constantly developing webapplications, like the Magnet.me site. Therefore, Magnet.me has requested the design and implementation of a back-end and CMS by the internal IT-department.
The development of code for the website can currently only be done from within the CMS. This limits the tools which can be used to improve the development process such as version control or automatic build tools. Therefore, a completely new back-end should be created and a new, custom CMS should be build on top of this back-end to allow the business owners to view and edit information about the entire system. Since the current CMS is not build specifically for Magnet.me, a lot can be improved in terms of usability.

Auxiliary Information

The Magnet.me IT department is very young. The student team will have to operate completely independent, without any technical support from the company.

Project team

Supervision

Name Company Supervisor	Email	Phone
Vincent Karremans	Vincent@magnet.me	06-4516454

Name TUDelft Supervisor	Email	Phone
Alessandre Bozzon	A.Bozzon@tudelft.nl	015-2786346

Student team

Name	Study ID	Email	Phone	Prerequisites* fulfilled signature
Tiddo Langerak	4024346	tiddolangerak@gmail.com	0611755301	FULFILLED
Alex Walterbos	4012917	atw.231@gmail.com	0634089166	FULFILLED

*Prerequisites: by signing this box, as a student, I declare that I fulfill the prerequisites of the TI3800 bachelor project as stated in the digital study guide.

Orientation Report Replacing the Magnet.me IT Infrastructure and Content Management System

Tiddo Langerak & Alex Walterbos

June 12, 2013

1 Introduction

In this document, the orientation phase of the Bachelor Project involving the development of the IT Infrastructure and Content Management System (CMS) will be described. It will consist mostly of the features requested by Magnet.me, the requirements for these features determined in collaboration with Magnet.me, the techniques we have chosen to use to meet these requirements and the resources that we have used to come to these decisions.

2 Problem Description

2.1 Motivation

The motivation for the replacement of the IT Infrastructure within Magnet.me arises from the company's desire to grow. The current system is holding Magnet.me down, both in growth speed and possibilities. Magnet.me intends to expand it's business internationally. The current system won't be able to handle the increase in traffic which can be expected from this expansion. Another reason for the system replacement is the need for an in-house system. With the out-sourced system, Magnet.me's IT-team oftenly finds it self lacking control or access to vital parts of the system. To reduce dependencies, the system must be fully maintainable from within Magnet.me.

Unfortunately, the current IT infrastructure *is* the Caret CMS. This results in a high dependency in the infrastructure, and therefore multiple system components have to be replaced entirely This Bachelors Project is the first step in the process of replacing and upgrading the current IT infrastructure. After this project, the website will be transferred to the newly developed system. The app will be connected later, and in the future third party software will be able to connect with the system as well.

2.2 Description

In this project, the student team (consisting of Alex Walterbos and Tiddo Langerak) will design and implement the web back-end and Content Management System.

The web back-end consists of the Web server that will route all incoming requests, the Database that will store all data, and the Matching System, which can be explained shortly as the Magnet.me business logic that matches students to company's Potential Employee Profile (PEP).

The student team will *not* implement the website. Instead, the live website will be transferred to the new system after the Bachelers Project has been completed.

3 Orientation in Available Content Management Systems

Besides the currently used CMS, developed by Click, there are many other available Content Management Systems that could be used instead. In this section, we briefly explain why these CMSs are not proper replacements for the current CMS, regarding the requirements given by Magnet.me and because of several other perspectives.

Content Management Systems like Typo3 [7], Joomla [3] and Magnolia [4] are general-purpose, standalone content management systems. Such systems would easily suffice for Magnet.me, but it takes a lot of resources to integrate these systems with the Magnet.me database, website and other systems. For this reason, and to ensure that all functionality is optimized for Magnet.me demands, the CMS will be implemented by the student team. With the chosen REST server-client structure, the CMS will be implemented as a thin client operating with a connection to the data provided by the server. Since the design of the thin client is heavily dependant on the server design, the general-purpose systems will not fit in this structure.

4 Current Situation

Before the new requirements and a new design can be created it is important to understand what the current situation is. Therefore the design and usage of the current application has to be investigated.

The current application is build by a third party vendor called Click [8]. Unfortunately, large parts of the current design are not documented, incomplete or are not made available to people outside of Click. Therefore, all information below is acquired by analyzing the system, unless explicitly stated otherwise.

4.1 Design

The core of the current system is the Caret CMS. The Caret CMS is a commercial CMS developed and maintained by Click[1]. It is originally developed by one of the co-founders of Click in the late 1990s and is mainly used by Click to develop websites for customers, including Magnet.me.

The CMS consists of three parts: a framework on top of which websites are build, a web interface for managing the website and the content (the actual CMS), and a PostgreSQL database for storing all information. A top level overview of this design can be found in figure 1. The following sections will describe the design in more detail.

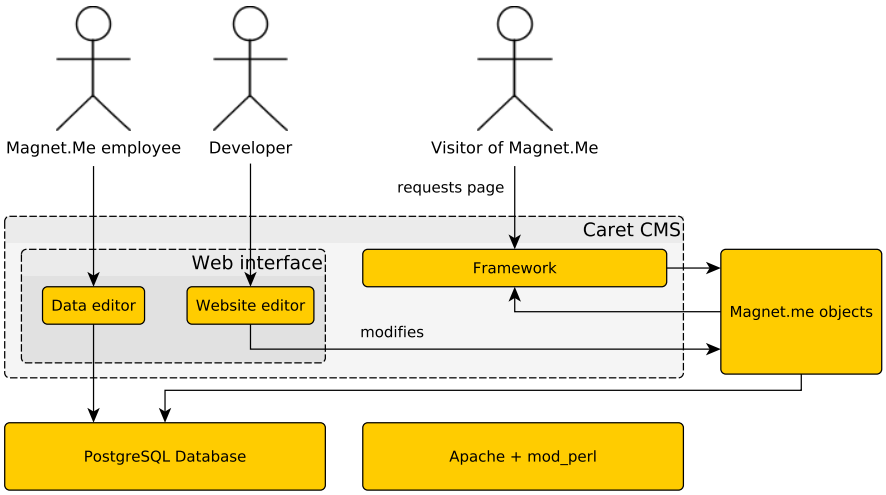


Figure 1: Current situation

4.1.1 Framework

Instead of using regular source files the framework uses a tree of special objects created using the web interface. These objects are very similar to regular files and folders, with a few notable differences. First of all, each object has a predefined type which determines how it is treated by the framework. For example, an object with the type 'DataForm' can be used to create forms whose input will be stored in the database. These objects can also have string properties which can specify some behavior of the object. For example, a DataForm object can have a property called 'DataFormTable' which specifies the table to store

[1]The Caret CMS is officially developed by another company called *Caret web content management*. However, this is only a different company in writing, in practice Click and Caret web content management are one company. To keep things simple, this report will always refer to Click when mentioning one of these companies.

the input in. Most object can also contain text, e.g. HTML, CSS or JavaScript, which will be parsed by the framework. Most of the text will simply be send to the browser, but special keywords can be used which instruct the framework to do some preprocessing such as inserting other objects, repeating blocks of code or inserting data from the database. Finally, objects can have other objects inside them such that the object tree can be constructed.

URLs are directly mapped to objects in the object tree. For example, a request to the fictional URL `https://magnet.me/site/foo/bar` will try to find the bar object inside the foo object and uses that object as a starting point of the application. The object will be processed by the framework and the result will be send to the browser.

The framework does not allow custom server side code. It does allow queries to the database for some type of objects, but other than that only client side code can be written. Only functionality provided by the CMS can be used. Custom server side code can only be written by a few employees of Click, who can upload Perl script as plugins for the CMS.

4.1.2 Web interface

The web interface is the place where the entire application is build, apart from a few custom server side scripts. The web interface provides a few different editors, of which the data editor and website editor are the most important ones.

The website editor is the interface in which the objects as described above can be created and edited. It shows a tree of objects and allows to browse trough these objects as if they represent a file tree. The actual contents of these objects can be edited using simple multiline textfields inside this editor. There is some very basic revision control implemented. Older versions of individual objects can be restored and compared, but merging or branching is not supported. Since the objects are only accessible from inside the framework it is not possible to use external tools such as minifiers, version control or unit testing frameworks without constantly having to copy the content of the objects manually between the CMS and actual files.

The other important part of the web interface is the data editor. This is a basic interface to raw information stored inside the database. This editor simply allows Magnet.me employees to edit any field inside the database. Foreign keys and cross references are automatically resolved by the Caret CMS, such that employees don't manually have to manage these references. Employees can use dropdown menus or checkboxes to create references. The database editor allows to do some very basic filtering, a filter on only one column can be applied, and the editor also allows to sort on any column. The database editor does not provide any other high level functionality, such as updating multiple records simultaneously or displaying the data in a more meaningful way to the Magnet.me employees.

4.1.3 Database

The final part is the database. The database management system used is PostgreSQL, as stated above, and the database is custom made for the Magnet.me website. No real foreign keys are specified in the database, references are handled by the CMS instead. The design of the database for the Magnet.me website is unnecessarily complex and a thorough study of this design goes way beyond the scope of this report. The ERD provided to us can be found in appendix A. A larger version can be provided by Alex or Tiddo upon request.

4.2 Problems

The Caret CMS doesn't provide any straightforward way to synchronize the application with any other computer. Therefore, the only way to make changes to the application is by using the web interface provided by Caret, as mentioned above. This introduces a few problems: the web interface is a simple text area, and therefore it is not an ideal environment to program in. There is no syntax highlighting, code completion, or any other useful functionality which can help in the development process. In practice it works best to move the code between a decent IDE and the CMS by copying and pasting the source code.

Another problem is that you can only have multiple objects open at once by launching the CMS in different tabs in your browser. Creating objects and switching between them takes a very long time, it usually takes more than 2 seconds for the CMS to respond to a request. This makes it very slow to navigate through objects and this slows down the entire development process significantly.

Since it is not possible to synchronize the objects with a file system, it is also not possible to use any form of version control, unit testing framework, or any other type of automatic build tool. Testing has to be done manually, and 'locks' to objects can only be acquired by verbal communication with team members. This also slows down the development process, and limits the number of people which can work on the application at the same time significantly.

Finally the database editor has it's problems as well. First of all, the editor simply shows the tables which exists in the database. For a software developer that's fine, but for the other employees of Magnet.me, the once who use this system most, this isn't ideal. Many changes requires updates in multiple tables, which can be easily forgotten. Also, relationships between tables aren't clear from the CMS, and therefore it is easy to create an invalid state of data. It also provides very limited options for viewing detailed and useful statistics. Finally the database editor is not secure as well. A few security leaks, mostly XSS, have been found already in a few weeks time. It can be expected that more leaks will be found in the near future.

5 Requirements

This section will discuss the requirements of the system. It will be devided in two parts: functional and non functional requirements. These requirements will be prioritized and categorized in four categories: system wide, web back-end, matching system and content management system.

5.1 Non Functional Requirements

Nr.	Description	Must/Could
System wide		
1	Security measurements should be automatically enforced where possible	m
1.1	All data should be sanitized by default before being used as output	m
2	The systems should be robust	m
2.1	Every request should receive a response	m
2.2	An error of a subsystem should never let the entire system crash	m
2.3	All systems should recover from any non fatal error [2]	m
3	The systems should be scalable across multiple servers	m
4	Subsystems should not interfere with eachother	m
5	New data should be visibile to all subsystems within 10 minutes of an update	m
CMS		
1	Requests should be processed within 100ms	c

[2]Only fatal errors from critical dependencies are considered to be fatal errors for a Any. subsystem error which is generated by a subsystem itself is non fatal. Examples of such errors are hardware failures or inaccesible databases.

5.2 Functional Requirements

Nr.	Description	Must/Could
	System wide	
1	Support for multiple languages	m
	Web Back-End	
1	It should provide an interface for the web applications including:	m
1.1	An abstraction for the data layer	m
1.2	An authentication interface	m
2	The following types of software should be able to be build on top of the back-end:	m
2.1	The website	m
2.2	The CMS	m
2.3	A mobile app	m
2.4	3rd party software	c
3	Requests should be authenticated	m
	Matching system	
1	It should match students with companies according to the model provided by Magnet.me (see Appendix B)	m
2	It should always act on the most recent data available	m
3	It should recalculate matches within 30 minutes after a change have been made which could influence the matches	m
	CMS	
1	The CMS should only be accessible for Magnet.me employees	m
2	All data provided by the companies and students can be viewed and edited, with the exception of passwords	m
3	The CMS should always show the most recent data available	m
4	Different employees can have different access rights to parts of the CMS	m
5	Subscriptions for companies can be managed	m
6	Translations can be added for the website	m
7	Statistics can be viewed from the CMS, including:	m
7.1	New subscriptions over time	m
7.2	Accepted and rejected matches per company	m
7.3	Accepted and rejected matches per student	m
7.4	Geographical distribution of companies and students	m
8	Matches can be made manually	m

6 Matching System Analysis

In this section, an analysis of the Matching System is made. The system, as explained in Appendix B, calculated matches between students and companies. To make decisions in the design of this algorithm, and perhaps more importantly the implementation of the algorithm, we analyze the algorithm's runtime, and estimate when and how often it is called on the running system.

6.1 Runtime of the Algorithm

Using a Potential Employee Profile, which belongs to a Company Profile, a match is generated by calculating a set of linear calculations. This linear set of calculations contains various forms of calculations, most of which are simply threshold comparisons. The threshold comparisons are used to apply scores for a student, depending on the range they fall in.

Another form is the so called 'exit'. Exits imply an early decision of a mismatch, as the exit is used to indicate a strict demand in the PEP. If a student does not comply to this demand, a mismatch is returned in the algorithm and further calculations are unnecessary (within that specific match calculation). Because of these exits, the set of linear operations executed will not be of a constant size. Therefore, we estimate that the calculation of a *single* match runs in constant time.

The actual load on the server, generated by the Matching System, is just as dependant on the amount of match calculations that have to be generated. Therefore, we analyse the usage of the algorithm as well.

6.2 Usage Analysis

Matches have to be recalculated whenever new information is entered, or existing information is edited or removed. This means that, whenever a student updates his or her resumé, this new information could result in a new match with a company. The other way around, whenever a company updates the PEP, all students must be tested for a match with the new PEP. One can imagine that the numbers of match calculations will become massive as the userbase and client base (companies) grows. With n students and m companies, every resumé update implies m match calculations, and every PEP update implies n match calculations.

A network request that follows from a made match should be visible within half an hour from the update that inflicted this new match. For example, a student that adds a new language to his resumé should be getting all potential new requests within half an hour of adding the language. This is a generous amount of time given for the calculations, at first sight. However, when many updates are done simultaneously by, suppose, 5000 users, and there are 2000 companies in the system, then $5000 * 2000 = 10,000,000$ matches have to be calculated. Such numbers of simultaneous updates are unlikely to occur, but

when Magnet.me enters the international market, the numbers of student- and company-accounts will likely rise fast.

Another limitation in the calculation process is that the match calculations may not significantly interfere with the request handling process on the server to ensure a smooth user experience.

7 Design Decisions

Before a decision can be made about which tools and platforms are needed it is important to have a basic top level design since the top level design highly influences the tools and platforms needed. In this section only a brief overview of the design decisions will be given. A more detailed overview will be included in the Bachelors Project final report.

7.1 Interfaces

There are a few interfaces which will provide (restricted) access to data inside the system. These interfaces needs to be identified such that the design will suite the needs of all these interfaces.

CMS
: The CMS is a web interface which Magnet.me employees can use to view and edit data inside the system.

Main website
: The main website is the access points which will be used most. This is the main interface for the data in the system and will be used mainly by students and companies.

Mobile app
: In the near future a mobile app will be created for the Magnet.me platform. This app will need to use data from the system as well.

3rd party software
: Magnet.me wants to provide an interface for 3rd party software to use data from the system, like many social networks do. However, currently this is only an informal plan and not a hard requirement. Whenever we find ourselves in a tradeoff involving this 3rd party software functionality we will prioritize the 3rd party software last.

These interfaces can also be seen in Figure 2

7.2 Main Components

In the previous sections a few components are described which will act upon the data. These include the interfaces (CMS, main website, mobile app, 3rd party software) and the matching system. Figure 2 shows a schematic overview

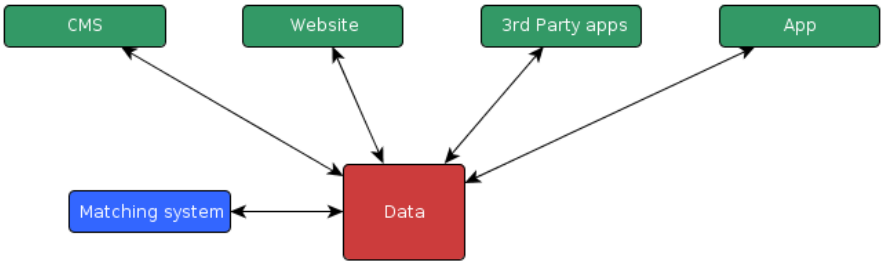


Figure 2: Main components

of these components in which the green blocks provides an interface to the outside world and the blue component only acts on the data. Only the CMS and the matching system will be made as part of this Bachelors Project, but the other components needs to be taken in account as well. A design needs to be constructed which allows all these components to act upon the data in a consistent and robust way.

Not all components act on the data in the same way: the matching system needs to act on the raw data from the database, whereas the other components are mainly interested in a human friendly representation of the data. Also, the matching system is the only main component whose goal is not to provide an interface for the data to users. In fact, it has no (direct) user input at all. Finally not all components have the same origin: the CMS and Main website are generated by the server, the matching system runs in the same network, and the mobile app and 3rd party software are completely external applications.

7.3 Solutions

Since the matching system is used entirely different from the other components, it makes sense to separate it entirely from the other components. By separating the matching system from the interfaces more specific tools can be chosen to suite the needs of these components.

However we have a few more options for the other components, the interfaces, which will be discussed below.

7.3.1 More Separation

One possible solution is to create some more separation. Both the CMS and the main website are served from a server in the same network as the database. Therefore the server can combine layout and data and serve that to the browser, which is what most websites do. However, the mobile app and 3rd party software are only interested in data. Therefore a separation between these two categories can be made.

One application can be build which will generate web pages, and serve those to the browser. This is exactly what most web languages, like PHP, are designed for [9]. Another application can be build which will simply provide an interface for 3rd party software to access data from within the system. A common architectural style which can be used for this is Representational State Transfer (REST) [17].

7.3.2 Client-side Web Applications & REST

Since the invention of Ajax [18] another approach has been made possible as well, and that is creating full client-side web applications which communicate with the server using REST. This approach has become increasingly popular due to the increasing maturity of frameworks such as Backbone.js [2], and in fact Figure 2 already hints that such an architecture might be a good choise. With this approach static pages are served to the client, without any data embedded into the pages. The data will be requested from the server using a REST API. Figure 3 shows which components are required for this architecture.

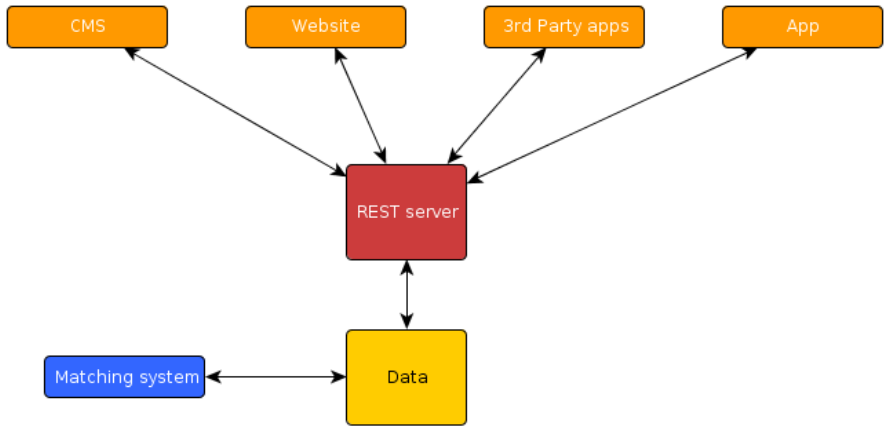


Figure 3: Components for a REST based architecture

Using this technique there is no need for a separate server side application to serve data to the web pages. There can be a single REST API which can be used by all different interfaces. This technique also allows browsers to cache the application for a longer period of time, and thus reducing the amount of data which needs to be transmitted. Finally this technique also helps to create a better seperation of concerns. The web server is no longer responsible for generating web pages, but solely for providing a REST API to the clients.

Due to these advantages this approach will be used to create the web back-end and CMS for Magnet.me.

8 Tools, Techniques and Programming Languages

In our orientation phase, we have come across numerous new aspects regarding our project. We have collected all significant finds that we will likely use during our project in this chapter.

8.1 Server setup

The Magnet.me server is a virtual server, hosted and maintained by DutchCloud. The server is, however, completely controlled from within Magnet.me. During the project, the student team will manage this server.

Since the server is completely configurable, the student team will decide on many aspects of the server management. These aspects include the choice of Operating System, Web Server and installing Tools (which will be discussed in the section Development Process (8.2)).

8.1.1 Operating System

For reasons like stability, maintainability and extended functionality for web development, Linux is an obvious choice for our OS. A widely-used distribution of Linux, on servers, is CentOS [10], developed by Red Hat. Since this Operating System is sufficient for the Magnet.me servers, this is the OS we will work with.

8.1.2 Web Server

In our orientation, we have found that the market leaders in Web Server Software for Linux are Apache [1] and NginX [5]. Apache provides a lot more functionality than NginX, but non of this functionality (that NginX does not offer) is required for the Magnet.me server. An advantage of NginX over Apache is performance[16]. Especially with higher amounts of requests, NginX prevails over Apache with ease. The server will mostly be used as a reverse proxy, and not a lot of extra functionality will be needed. Therefore, performance is more valuable than functionality, and therefore NginX seems to be the better choice.

8.2 Development Process

Our development will be organized with the Agile programming style 'Scrum', using Scrum cycles better known as Sprints to aim for quick results. We will also work with Behaviour Driven Development (BDD)[13]. Shortly explained, this is a design method in which desired behaviour is determined. This behaviour is reflected in the tests, which will then be used in a method similar to Test Driven Development.

8.2.1 Tools

During our project, we will be using several tools, for example to organize our scrum process. These tools have been listed below.

Trac [6]
Trac will allow us to organize all (completed) tasks, organized with Milestones and Tickets.

Agilo for Trac [11]
Agilo for Trac is a tool integrated in Trac, which allows us to use Scrum cycles (Sprints) within our Milestones.

Git[12], hosted with Gitolite on the Magnet.me server [12]
Gitolite allows us to host the Git version control tool privately.

8.3 Languages & Platforms

To chose a programming language a few criteria has to be chosen on which languages can be compared. Chris Britton suggest the following criteria [15]:

1. Ease of learning
2. Ease of understanding
3. Speed of development
4. Help with enforcement of correct code
5. Peformance of compiled code
6. Supported platform environment
7. Portability
8. Fit-for-purpose

We use the same criteria, with the exception of portability. Note however that different scores can be assigned to a language depending on its use case. Therefore the scores assigned to a language for the Matching System might not be the same as the scores assigned to the same language for the REST Server.

8.3.1 Matching System

A few languages have been considered for the matching system: Python, Javascript (NodeJS), C#, C++ and Java. Appendix C shows a comparison table for these languages. It is very important the the matching system is correct and robust, and thus the most important criteria are 'Help with enforcement of correct code' and 'Ease of understanding'. Using all criteria we found that C# and Java are the best choises for the matching system. C# however does not have a complete implementation for the Linux platform, whereas Java does. Therefore Java is the language which will be used for the matching system.

8.3.2 REST Server

For the REST server a few languages have been considered as well. These languages are Python, Javascript (NodeJS), PHP, C# and Java. Note that some of these languages require additional frameworks or libraries to act as a REST server. The scores given in the table do take those frameworks in account. Also, scores can differ from the scores in Appendix C, since the tables compare these languages for entirely different purposes. In Appendix D another table can be find to compare these languages. For the REST server 'Fit-for-purpose' and 'Speed of development' are the most important criteria. It has to be noted that all languages seem very good choises for the REST server. Using all criteria Python and Javascript seems to be the best choises for this platform. Javascript scores a little better, and the students also have more experience with Javascript. Therefore Javascript (with NodeJS) will be the language used for the REST server.

8.3.3 CMS

There is not much left to chose for the CMS. The languages which are going to be used are already predefined (HTML, Javascript, CSS). However, the framework is not. The current Magnet.me website already uses AngularJS in some places for data binding and synchronisation. AngularJS is also a suitable choice for the CMS and there is no good reason to use another platform. Choosing another platform only increases the complexity of the entire system. Therefore, AngularJS will be used as the client side framework as well.

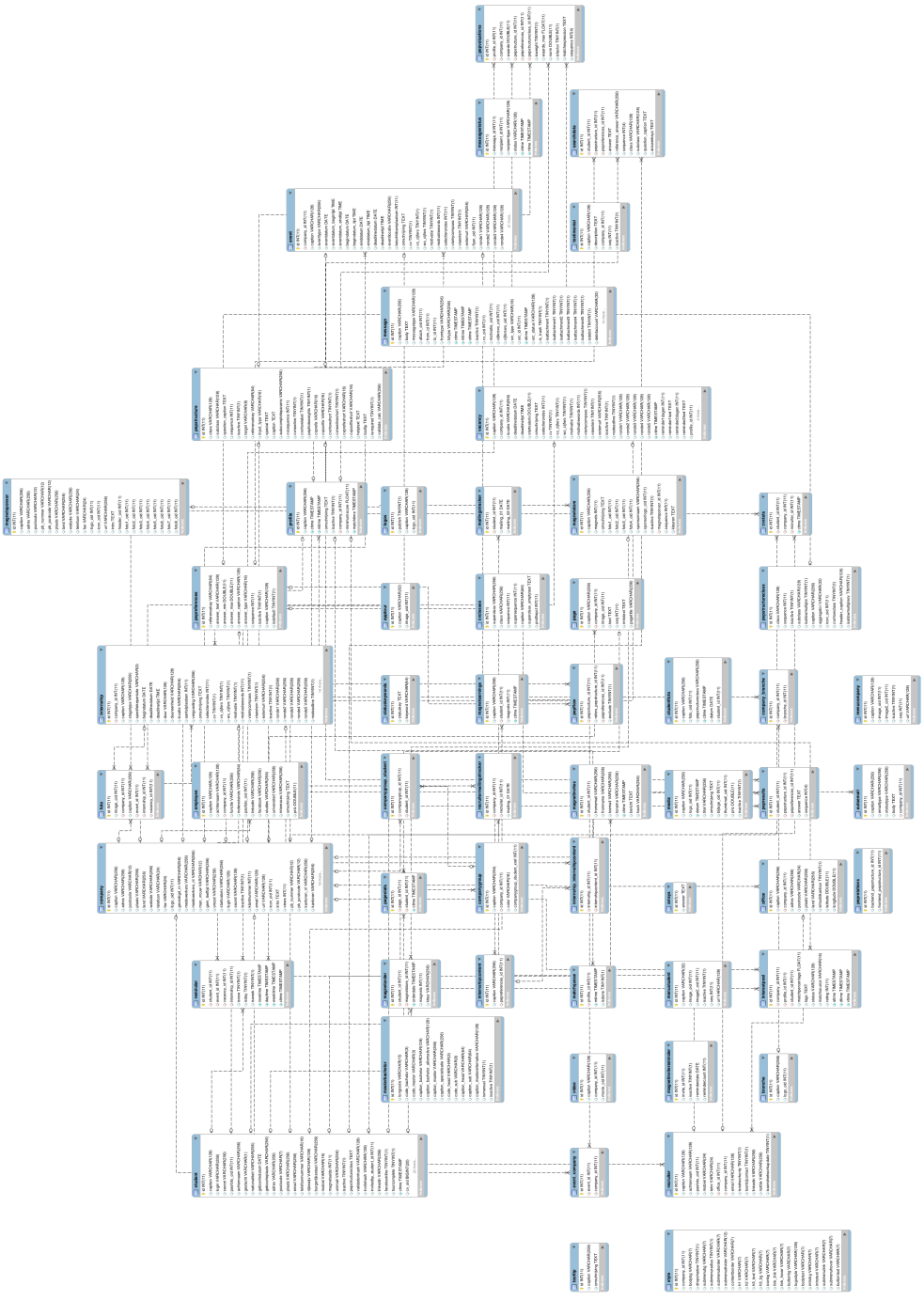
9 Database

One of the most important components of the system is the database. There are many different types of data models to chose from such as relational, key-value, column oriented and document oriented.

Since Magnet.me plans to expand to other countries it is expected that we might have to deal with distributed systems in the future. Therefore we can look at Brewer's Theorem: "When designing distributed web services, there are three properties that are commonly desired: consistency, availability, and partition tolerance. It is impossible to achieve all three." as stated by Gilbert and Lynch [19].

Based on Brewer's Theorem we could exclude some types of databases which can help us in our choice. However, the bachelor's project supervisor, Alessandro Bozzon, advised that most databases can be scaled one way or another across multiple physical locations and he advised the students to chose a database based on previous experience and personal preference [14]. Therefore, a PostgreSQL database will be used to store the data in the system.

A Old database ERD



B Matching System

The matching system of Magnet.me uses a set of linear operations based on matching criteria provided by companies. The data provided by students is categorized. For example, there are categories for education and relevant working experience.

A category can have subcategories. Different categories can have different levels of nesting. Each category has a weighting factor. Inside the deepest subcategories are rules.

Each rule has a condition and a score associated with it. Students profiles can score points in each (sub)category by fulfilling a condition. All points scored inside a (sub)category are summed and the result is multiplied by the weighting factor of the category. The resulting score will be the students score for this category.

By recursively summing scores and multiplying the intermediate scores with the weighting factors of their parent categories a final score will be achieved. If this score is higher than the threshold score provided by a company, a match will be made.

Some rules can specify an early exit: if students do not fulfill those conditions, it will not be matched with that company regardless of its score.

C Comparison Of Languages: Matching System

Python is arbitrarily chosen as the reference language.

Criteria	**Python**	**Javascript**	**C#**	**C++**	**Java**
Ease of learning	0	0	-	–	-
Ease of understanding	0	-	0	–	0
Speed of development	0	0	-	–	-
Help with enforcement of correct code	0	0	+	+	+
Peformance of compiled code	0	0	+	++	+
Supported platform environment	0	0	-	0	0
Fit-for-purpose	0	-	+	-	+

D Comparison Of Languages: REST Server

Python is arbitrarily chosen as the reference language. Note that the scores are all relative to each other and chosen with its intended purpose in mind.

Criteria	Python	Javascript	C#	PHP	Java
Ease of learning	0	0	-	0	-
Ease of understanding	0	0	0	0	0
Speed of development	0	+	-	0	-
Help with enforcement of correct code	0	0	+	0	+
Peformance of compiled code	0	+	+	-	0
Supported platform environment	0	0	-	0	0
Fit-for-purpose	0	+	-	0	-

References

[1] Apache web server. `http://httpd.apache.org/`.

[2] Backbone.js. `http://backbonejs.org`.

[3] Joomla content management systeem. `http://www.joomla.org/`.

[4] Magnolia enterprise cms. `http://www.magnolia-cms.com/magnolia-cms.html`.

[5] Nginx. `http://nginx.com/`.

[6] Trac. `http://www.trac.edgewall.org`.

[7] Typo3, enterprise level open source cms. `http://typo3.org/about/`.

[8] Click. `http://click.nl`, Accessed april 22th 2013.

[9] General information. `http://php.net/manual/en/faq.general.php`, Accessed april 25th 2013.

[10] Centos. `http://www.centos.org/`, Accessed may 10th 2013.

[11] Agilo for trac. `http://www.agilofortrac.com`, Accessed may 15th 2013.

[12] Gitolite git hosting. `http://www.gitolite.com`, Accessed may 15th 2013.

[13] Dave Astels. A new look at test-driven development. *http://blog.daveastels.com/files/BDD_Intro.pdf*, 13(06):2010, 2006.

[14] Alessandro Bozzon. Personal communication, May 2013.

[15] Chris Britton. Choosing a programming language. `http://msdn.microsoft.com/en-us/library/cc168615.aspx`, Accessed may 10th 2013.

[16] S. Dmitrij. Blog: Nginx vs cherokee vs apache vs lighttpd. http://www.whisperdale.net/11-nginx-vs-cherokee-vs-apache-vs-lighttpd.html, Accessed may 10th 2013.

[17] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[18] Jesse James Garrett et al. Ajax: A new approach to web applications. http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications, 2005.

[19] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.