FleXentral

Managing the contingent workforce

Arjo van Ramshorst
Joost Rothweiler

Bachelor's Thesis
Computer Science
Delft University of Technology

FleXentral

Managing the contingent workforce

by

Arjo van Ramshorst Joost Rothweiler

Student numbers: 4226631 4246551
Supervisor: Alessandro Bozzon
Project duration: April 18, 2016 – June 17, 2016

An electronic version of this report is available at http://repository.tudelft.nl/.

TU Delft

Abstract

The market for contingent workers grows every year. Managing this workforce gets more and more challenging because of regulatory compliance, the difficulty of finding matching employees, and keeping track of expenses. FleXentral tries to be a platform that solves these issues. The goal of this project was to build a proof-of-concept of this concept, that can be used to show the potential to customers, but that will also be used as a basis for the final application.

We created a set of requirements together with our client that such a system would require, and built the first version in a span of 10 weeks. This version includes a matching module that matches flexworkers to project-managers based on their competences, a negotiation module that simplifies the process of creating a contract and a reporting module where financial and compliance data can be inspected on an organizational-, project- and personal level.

The final product we delivered during this project has basic versions of these systems implemented as a modern, responsive web-application. It is far from production ready, but it is demo-ready for potential customers, and of good quality to form a strong foundation for creating a production-ready platform.

We recommend that for further development, the focus should be in four directions: the matching algorithm, the compliance module, user testing and non-functional requirements as maintainability and scalability. Each of these directions should be a cycle of continuous improvement.

Preface

As a Bachelor's student in computer science at the TU Delft, we complete our program with a thesis project of 10 weeks. This report is the result of our work, including the assignment, research and development done by our team of two students. Our assignment was issued by FleXentral, a new company based in Amsterdam. We would like to thank Onno Hektor, André Bonvanie and Tony den Doop from FleXentral for the great assignment and support throughout the project. We would also like to thank our supervisor Alessandro Bozzon from the TU Delft for his guidance and feedback.

Arjo van Ramshorst Joost Rothweiler
Delft, June 2016

Contents

1

Introduction

Managing a company its contingent workforce is an ever-increasing challenge for two important reasons. The first, and maybe the most important one, is that for the last couple of years the percentage of contingent workers in companies has grown every year. Secondly, the rules and legislation by governing organizations for flexworkers often change. Currently, most large companies have employees with the sole task of managing the contingent workforce, but this takes up more and more resources and time.

There are three main aspects that make managing the contingent workforce a tedious task. First, it is difficult to find qualified personnel. Secondly, it is difficult to keep track of regulatory compliance and finally, with a contingent workforce it is often impossible to get valuable insight into finance. FleXentral, the company that provided this project, aims to address each of these problems by providing an online software solution, allowing for better contingent workforce management.

1.1. Project assignment

Our project focuses on solving these three problems by creating an online system where both flexworkers and organizations have more insight in daily tasks. We developed a matching module, where flexworkers and organizations can be matched based on their competences and experience, we built a dashboard where flexworkers can take care of their own compliance, and organizations can see on a detailed level where possible problems are. Finally we built a module that tracks expenses for projects, visualizes them on a project and organizational level. This data is gathered by a providing flexworkers the option to track expenses and transactions per project, and also by providing them the option to send invoices through the platform.

In this report we will describe the process we went through and the choices we made during the 10 weeks we worked on this project. The following section explains the rest of this report is structured.

1.2. Report structure

The structure of this report reflects the process we went through while developing the application. We start with a detailed problem analysis in chapter 2, then we list the functional- and non-functional requirements in chapter 3. In chapter 4 we elaborate on how we approached the project, e.g. the process, tools and planning. Chapter 5 goes into detail about the application design and development. Following that comes a chapter about quality assurance (chapter 6), where we explain how we made sure our implementation met certain quality standards. We reflect on our work in chapter 7, where we discuss how well we implemented the requirements, the feedback from SIG and our own reflection of the entire process. Finally in chapter 8 we conclude on the project and give our recommendations for future work.

2

Problem analysis

In this chapter we will describe the starting point of our project, and go into detail about the three problems in Contingent Workforce Management that FleXentral aims to provide a unified solution to.

2.1. Current situation

FleXentral is a start-up with ambitious plans and goals, as well as an interesting concept. Before we started this project, the product was still missing and that is what our project is about: building and designing the first version of the concept from scratch. It is not necessary for this version to have all the features and details implemented, it is more a proof of concept/prototype that can be used for after our project to show potential customers what FleXentral aims to achieve. On the other hand, it is not meant to be a throwaway prototype. The goal is that after this project the application will be continually improved. This will be done by cooperating with potential customers, and using their feedback to constantly improve the application. This means that even though we're building a first version, it should be future proof: maintainability is an important factor in all design decisions we make.

2.2. Problem detail

The goal of FleXentral is to provide a solution for three problems that many companies with a large contingent workforce are currently dealing with. First, it should make it easy to gain insight in the compliance level of contractors and, in the case of problems that need attention, where they are and how they can be addressed. Secondly, it should give companies insight in how their budget for contingent workforce is spend, by providing a platform that handles the transactions, and administration for all projects done by contingent workers. Finally it should reduce the costs of the entire contingent workforce, by intelligently recommending flexworkers to projects. This removes the need of expensive recruiters. In the next sections we will go into detail on these problems, why they are problems, and how FleXentral aims to solve them.

2.2.1. Compliance

Compliance, and more specifically: regulatory compliance, is the term used to describe the goal of being compliant to the laws and regulations imposed by governments. A significant part of these regulations are about Contingent Workforce Management, and are thus important to organizations that employ a relatively large amount of flexworkers.

Problem Recently more and more compliance rules are put into place which makes it very difficult for large organizations to keep track of every contingent worker whether he/she is compliant. Even more so because there is no good standard for compliance. Different countries have different compliance rules. Some rules and laws are only for specific sectors, and

even within companies different contingent workers might need to meet different requirements.

Not being compliant to these rules often leads to fines, so most organizations try hard to follow these rules. However, with this ever increasing complexity, it is difficult for organizations to find potential risks. Most compliance rules are verified by data. This can be a document (An example document is the *VAR-verklaring* in the Netherlands. Incidentally this is also an example for the fact that these rules are ever changing: in may 2016 the *VAR-verklaring* has been replaced by a model agreement provided by the government.) but it can also be a number (a Chambre of Commerce number is an example of this) or a signed agreement between both parties.

Current solutions Currently there are software solutions being developed that organizations can use to keep track of compliance status. The problem with the current solutions is that it is managed by the organization itself, and not by the contingent worker. This means that for example when a document is expired, an employee from the company has to make sure the contingent worker sends a new document that is then stored in the organizations system. This costs a lot of time, and in turn money.

FleXentral's solution The difference between the currently offered solutions and FleXentral is that FleXentral is an online platform, that is not directly bound to an organization. Flexworkers create an account on the platform where they can manage their compliance. The platform offers flexworkers a place where they can store all their relevant data that is necessary for compliance rules. This data can then be selectively shared with organizations they have an agreement with, according to the rules they have to comply with. Whenever this data expires, both the flexworker and the relevant users from a company receive a notification that something must be done about this.

FleXentral also provides insightful dashboards that show where issues are and how they can be fixed. By providing both parties this information, and having a single source for the relevant documents, it will reduce the amount of time (and as a result the cost) that is necessary for being compliant for both the flexworker and the organization.

2.2.2. Transactions

Contingent workers get paid by sending an invoice to the company, this arrives at the financial department of the company, where they need to acquire approval by the relevant project manager before the invoice can be paid.

Problem There are two problems with handling transactions this way, which FleXentral aims to solve. The first one is the amount of work that is required by both parties to send, approve and pay transactions. Multiple parties are involved (sometimes multiple times) in this process, which make it slow and costly. The second problem, which is even more important for most organizations, is that a contingent workforce makes it very difficult to see where the budget of a project is spend on.

The reason why this is a problem, is because there is often no central place where hours made by flexworkers are logged, and it is difficult to link a paid invoice to a certain project. Finding what projects are potential problems financially is a very tedious task. The information has to be retrieved from various places. Often this is also not a trivial task, which leads to an organizations that is unaware of how their budgeting is spend on a contingent workforce.

Current solutions Currently, there is no standard solution for these problems. Every company has its own way of handling these problems. For invoicing there are a lot of tools available to the flexworker, but getting them to the right person is still a time consuming task. For both problems, there is room for improvement.

FleXentral's solution The first problem is solved by providing a way for flexworkers to send automatically generated invoices through the platform. These invoices can be generated by adding the hours worked on a project, and are then combined with the facts in the contract to generate an invoice. This invoice is then showed on the dashboard of the relevant project manager, where he can approve it and send it to the financial department for payment. This simplifies the process of sending invoices, and in turn reduces the time it takes.

The second problem FleXentral aims to tackle is getting insight in budget spend on contingent workers. Because of the way the first problem is approached, all data necessary to generate helpful information is already stored in the system. By providing a dashboard showing information of certain detail for certain users, it can help provide insight, and highlight potential problems. For example, a CFO is more interested in high level reporting that shows budgeting information about certain projects, project managers or even certain entities of the organization in different countries. For project managers it is more important to see information about the projects that they lead, and if a certain flexworker might cost more or less than initially planned.

2.2.3. Matching

The final problem which FleXentral aims to tackle is finding the right person for the job. This often takes up a lot of time and resources or is done by an external recruiter or recruiting company.

Problem A contingent workforce, by definition, is bound to change often. New projects require taking on new employees for a relatively short period of time. This has the advantage of being very flexible. The workforce can be much larger when the economy allows for it, but as it goes down, it is relatively easy to quickly reduce the amount of employees without being bound to contractual obligations. The negative impact of this is the fact that organizations constantly have to be on the look for potential people for a job or project. This results in employees whose only task consist of finding potential employees, but even this is difficult, because doing this effectively means knowing a lot of people, and that is difficult when you are confined to a desk in a company.

Current solutions Most organizations solve this problem by hiring an external party to do this for them. Often in the form of a recruiter or some sort of employment agency. This works, but it is a costly practice, since these recruiters often ask 30% or more of all fees paid. This significantly reduces the attractiveness of hiring flexworkers, since it makes them often at least 30% more expensive than a comparable normal employee. Also, they need more time to be brought up to speed on how a company and project works in comparison to standard employees.

FleXentral's solution FleXentral aims to remove the 'middle man' in transactions between organizations and flexworkers. Currently, the two tasks that are often done by this external party, when setting up a transaction is: Finding the right person for a job, and secondly: Handling the legal documents and the transaction. Our goal is to provide a platform that offers a solution to both these problems.

Finding potential matches will be done by an algorithm that matches projects to potential employees based on certain properties. Examples of these properties include: users' competences, how much experience a user has in a certain field and how well he performed during projects in the past (based on evaluations done by previous project managers).

Whenever a project manager found a possible employee with the help of this algorithm, the second problem comes around, the negotiation of contractual details, like fee and duration of the project, and signing the right documents. The platform should provide a way to guide these negotiations in the right way until a legal agreement is signed by both users, and the transaction is complete. All of this should be done with the least possible input and time of both flexworker and project manager.

2.3. Conclusion

In this section we described the three problems of Contingent Workforce Management that FleXentral aims to solve: Simplifying the process of adhering to regulatory compliance, simplifying and providing insight in budgeting on a project and organization level, and reducing the time and effort required to find the right flexworkers for a job or project. We also described how FleXentral aims to solve each of these problems.

3

Requirements

In order to formulate proper requirements, there has been looked at what is required to build a proof of concept version that shows just enough functionality to reveal future features and convince possible clients. At the same time, Based on meetings and research from the last couple of weeks we have come to the following requirements together with our client.

3.1. Functional requirements

The functional requirements for this first version of the FleXentral platform consist of 7 modules that should be implemented. In this section we will further discuss each one of them.

3.1.1. Registration module

Both companies and flexworkers should be able to register themselves to the platform. Organizations are required to give basic information as well as information concerning possible different entities. Also they should be able to set credentials for employees that fulfill different roles under which Line/Project manager, Controller and CFO.

Flexworkers should be able to sign up requiring only very basic information like name and e-mail address but are encouraged to connect through LinkedIn. After the sign up process is complete, more information is asked for in order to get the status of compliant (see compliance module).

3.1.2. Compliance module

An important module is the compliance module, stating the level of compliance of a flexworker regarding a country, organization and project. The compliance check allows both organizations as well as flexworkers to check the amount of risk an organization takes when hiring a certain flexworker for a specific job in a specific country. This information is based on registration numbers, certificates and diploma's provided by the flexworker.

3.1.3. Marketplace module

The information provided by both flexworkers and companies meet in the marketplace module. Here companies can search for talent and flexworkers can look for matching jobs for which they are compliant.

3.1.4. Search & Match

Apart from the marketplace module, The FleXentral platform should contain a module that automatically searches and matches jobs to flexworkers. This allows for the organization to, whenever a new job is submitted, directly get a shortlist of flexworkers that are suited for the job.

3.1.5. Negotiation

The negotiation module consists of a basic chat functionality where employers and flexworkers can negotiate on future jobs. Base offering, planning and terms and conditions provided by the organization should meet base pay, availability and terms and conditions provided by the flexworker. On agreement, the employer should be able to set up a contract which will then be offered to the flexworker together with a sign off option.

3.1.6. FleXentral transaction

Flexworkers should be able to log their activity in the FleXentral transaction module. In this module the effort is passed as input by the flexworker. This effort is then approved by the line/project manager at the organization and submitted to controller (financial approver) against budget. An invoice should be generated and available to both the organization as well as the flexworker. Both FleXentral as well as the flexworker should get notification of payment by the organization.

3.1.7. Reporting

Reporting shall be divided into different dashboards, available to users based on their roles at a company or as a flexworker. While different organizational roles require insight into effort, budget, planning progress, compliance status and actuals, flexworkers should get a clear insight in accounts receivable, pending and approvals.

3.2. Non-functional requirements

Apart from the functional requirements, The FleXentral platform should also meet equally important non-functional requirements. In this section we discuss the non-functional requirements the platform should meet.

3.2.1. Technological independence

Technological independence means that our application should not be too dependent on existing frameworks and/or libraries. If an application is technologically independent, parts of it can be ported to other languages or frameworks when necessary, without having to rewrite portions of the code that are not rewritten to another language/framework. In essence, this means that most of the code should be framework agnostic.

The goal of FleXentral is to become a platform that is used on a very large scale. Since our experience with, and knowledge of building large scale applications is not very extensive, and the time for the bachelor's project is limited, we have built more of a prototype/proof of concept application. Ultimately, our goal is to build a strong base application that can be further built upon after our project is finished, but chances exist that a part of the system has to change in the future for any possible reason. An example could be that another database will be used to store data for performance or security reasons. Furthermore, at some point our client wants to develop a mobile application. Adapting the application for this should take as little effort as possible.

3.2.2. Robustness

Robustness is often defined by how 'error-proof' an application is. This requirement can be separated into two different topics. The first one is whether or not users on the platform are able to reach pages and data they are not authorized for. Users should not be able to reach pages they are not meant to visit. This is important, especially since the platform will contain valuable private data.

The second topic concerns error handling on the platform. Once a user (both intentionally as well as unintentionally) tries to perform an action that he or she is not meant to perform, the platform should always display errors in a reasonable fashion. Users should receive feedback if something is not working or the system breaks down. Also, it is important that when an error occurs in the back-end, a reasonable response is still returned through the API to the front-end. The front-end will then be able to return this response to the user.

3.2.3. Scalability

The scalability of an application is measured by how well the application handles increasing amounts of data. As mentioned before, the goal of FleXentral is to be a large-scale application. Hosting a calculation intensive application with a large amount of users can cost a lot of resources. Designing the system to reduce the number of requests to the server, and the complexity of these requests, can save money.

3.2.4. Maintainability

Maintainability is difficult to define exactly. Different people find that different things improve or reduce the maintainability of code. Basically, it comes down to 'how easy is it to change or add features without breaking existing code?' Another often-used measurement is: 'If someone sees my code for the first time, how long does it take until that person understands it?'

Writing maintainable code is important for every project that is envisioning to become large-scale. While at the moment this is not yet a big issue, since we write the entire application with just the two of us, and we both work on all aspects of the application, it might seem clear to us (at the moment) what does what, and in what file the one function that handles this specific functionality is located, but what if we look at this code a year from now? Or what if a larger team of developers will continue working on FleXentral? It is important that we already take this into account during the development of our first version.

3.2.5. Usability

Because users are going to have to work a lot with the platform on both the computer as well as on mobile devices, usability is an important requirement for this first version. The platform should perform fast, be easy to use and work well on every device. Preferably the platform would be built in such a way that pages would never have to refresh and the website would feel like a native app when using on mobile phone.

4

Project approach

At the start of this project we have worked to form a well-defined project approach in order to work as effective as possible throughout the project. In this chapter we will discuss the methodologies and tools we have chosen in order to successfully fulfill different aspects of the project assignment.

4.1. Scrum

Scrum is an agile software development methodology that allows for flexibility and effective development through the entire project. We have chosen to implement scrum as our methodology for this project because it ensures a shippable product at the end of every sprint, it allows for clear insight in the work that is still ahead of us and allows for us to focus on only building code that is necessary for the particular sprint.

At its core, Scrum consists of a team of which every member takes part in the planning and splitting of tasks. Roles that are assigned to people include the product owner, being the voice of the client which in our case is assigned to Onno Hektor. Furthermore you would normally further divide the team up into developers and assign the role of scrum master but since we are only a team of two developers, we have decided to collectively take on the role of scrum master.

4.2. Tools and libraries

In this section we will explain what tools we have chosen to use for the development of the application and why.

4.2.1. Back-end

One of the first and main design considerations we have made, is that the front-end and back-end should be completely decoupled in our application. This is due to the fact that at this moment, it is unsure how the back-end will be implemented in the future. It is unclear at this moment whether every company that is going to use the application requires a different server that hosts its own data, or one server will be provided by FleXentral. Right now there are no definite answers to this, and since these are only questions relevant to the back-end of the application, we have chosen to create an application in which the front-end has no need for any knowledge of the back-end, except for the web API routes available.

Even though the back-end likely will not be the final implementation, for testing and developing of a working application, it is necessary that we have something to act as our back-end. This should be an API that the front-end can use, and a way to store, retrieve and update data.

Since almost every programming language can be used to create a server to store data and create an API entry point to retrieve it, the choices for this are endless. Every language has its pro's and con's, so there is no one that is necessarily 'better'. We based our choice mostly

on what language we are most familiar with, so we can develop the back-end as quickly and efficiently as possible, without having to worry as much about scalability. The back-end should only have to handle data and authentication.

We both have the most experience using PHP, so it is logical for us to use PHP as the language for the server. Especially, because since PHP 7 is released, it is for most cases just as fast as most competing languages like Java and C#. PHP is also used by a large chunk of the internet, so there is a lot of support online, and there are a lot of tested and tried frameworks for it. In order to reduce the writing of boilerplate code, and increase development efficiency we decided to use a PHP framework. The framework we have choosen to use for this project is Laravel.

Laravel

Laravel is a relatively new open-source PHP framework. It is based around the Model-View-Controller (MVC) architecture. Since the release of Laravel 5, it is also very stable, has long term support and backwards compatibility for newer versions. Documentation is well-written and extensive, and there is a lot of support from other users. It is used in several big and small applications, and it is developed actively. Besides that, we both have used it with success in previous applications. We don't have to put much effort in learning the framework because of this.

Another useful feature of Laravel is its use of migrations to manage the database state. Since we are working on this application with two persons, both in different local environments, it should be possible to work with a local database that always matches your current environment, even though we are working on different modules with different database requirements. Laravel's database migrations make this a breeze, since the state of the database is managed by the version control. This means that the database is always at the same level as the code base.

Finally, Laravel has an object relational mapping system called Eloquent. Eloquent makes it very easy to define relationships between different models in the database, and make it simple to retrieve related models. Since the application we are building requires a lot of many-to-many relations, this reduces the amount of SQL we have to write, leading to quicker prototyping.

4.2.2. Front-end

The front-end is the most important part of this application. As stated in the requirements, ease of use is a key value in this project. Static websites are outdated and feel slow nowadays. The client wants the application to feel like a modern application, not like a web page. In order to achieve this, JavaScript is necessary. The application will become very large, with a lot of different modules that all have to be updated in real time. Using JQuery or a similar DOM manipulating framework is tedious to use when the application grows.

First of all, the larger an application gets, the slower DOM manipulation becomes. Secondly, as a developer, using JQuery and DOM manipulation to handle the visual aspect of the application is extremely difficult to maintain when new features are added. New features are likely to break older code. It is also very difficult to properly test. Finally, when using DOM manipulation to handle data input, there is never a single source of truth for data values. This can cause synchronization issues, that can lead to very hard-to-identify bugs. Especially with a large application this can cause major development delays.

We decided to use a framework that handles these three points better. After researching the possibilities we decided to go with Facebook's React, in combination with Redux. This combination has a solution to these three problems.

React

React is mostly a solution to the 'view' part of MVC. It has little functionality by itself, other than rendering views. It uses 'components', that are nested, much like the HTML DOM-structure. The only difference is that you can create custom "DOM-elements" that automatically get filled with data.

What sets react apart from other solutions is that react has one-way data binding. Unlike a framework like AngularJS, where updating the value in the view automatically updates the value in the storage and the other way around, which is called two-way data binding. Both one-way and two-way data binding have pros and cons, but the disadvantages two-way data binding brings, (like synchronization issues and no single source of truth) is in large, complicated applications more of a disadvantage than an advantage.

With two-way data binding, when you enter something in a form, the model gets updated. If something else updates the model, the form gets updated. When this happens at the same time, strange behavior can occur, which can be hard to debug. React handles this as follows:

When you enter something in a form in React, an event is fired that updates the model with what you entered, and then finally the view updates with the value that is stored in the model. The model is always right about the value, this reduces the amount of bugs, but increases the boilerplate code needed for forms and other data bindings in the view. In order to reduce that, we use another framework in conjunction with React called Redux.

Redux

Redux is not really a framework on its own, but rather a way to store data. An application that uses Redux has one storage place for data. This is called a 'store'. Every application has exactly one 'store'. The store is basically one large JSON object containing all data the application uses in a treelike [key => value] format. Since this store is the only place where Redux stores data, and the only place where it can be updated, it is the 'single source of truth', as specified before.

What makes Redux a great solution against synchronization faults, is the fact that it has only one way to update the data in the store. Updating the store is done by firing 'actions' after an event happens. An action has a 'type' that describes what happens, and one or more parameters.

An action is handled by a reducer. A reducer gets the current state, and the action (including parameters) as input, and returns a new state. Important to note is that a reducer should always return the same result with the same input. So no random generators, time based functions or AJAX requests are allowed in reducers. Actions are always fired in sequence, so there are no race conditions, and no synchronization issues. Asynchronous calls are handled by middleware, which fires an action before the call, when it's successful and when it failed. Because actions always return the same value, it is easy to unit test every action that can happen in the application.

4.2.3. Other development tools

We use multiple languages and frameworks, and also multiple small helper packages, which simplify stuff like handling date and time, form generation, authentication and more. Keeping track of everything gets harder the more external tools we use. To make managing this easier we use multiple tools, mainly Composer for PHP and Node Package Manager (NPM) for JavaScript. Also, we use Gulp and Browserify for building and minifying JavaScript. Furthermore we use Jenkins for continuous integration.

Composer & Node Package Manager

Composer and NPM are tools that keep track of all dependencies of a project. They can automatically update all tools to the latest version, and they make deploying a project as easy as running one command to install all dependencies.

Gulp & Browserify

We use Gulp to 'package' the JavaScript code, including the dependencies we use into one minified JavaScript file that is stored in the public folder of our application. React and Redux use a lot of ECMAScript 6 & 7 features (ES6/ES7), therefore we have decided to use ES6 and ES7 to write most of our JavaScript code. Since not all modern browsers understand ES6 and ES7 features, we use Browserify with Babelify to compile it to basic JavaScript that every browser understands.

Jenkins
Jenkins is a cross-platform continuous integration and continuous delivery application that has allowed us to develop according to the methodology of Test-Driven Development. Over time, we believe that using continuous integration has increased our productivity enormously and will continue to do so for this project.

4.3. Planning

In this section we define the planning we initially made for this project as well as the actual execution of the different development steps per week.

4.3.1. Initial planning

As compliance and the searching and matching of people to jobs are the most challenging and important aspects of the FleXentral platform, we have chosen to spend most of our time researching and implementing those two parts. In the first two weeks we have spend time researching, setting up the framework and structure for front-end and back-end. For the weeks that follow we have defined the following planning.

Week 3 The focus of the third week is to allow for contingent workers to create a profile. Also after this week, project managers should be able to to post jobs that contain information that can later be matched. The back-end of the platform should be made ready to be used to check basic compliance status of contingent workers.

Week 4 Once flexworkers can sign up and projects can be created, the next step is to start implementing the negotiation module, allowing for contingent workers to communicate and close deals with companies.

Week 5 Flexworkers should be able to find matching projects and project managers should be able to find matching talent. In week 5 we focus on implementing automatic search and match to make this possible. Project managers should retrieve matching talent right after they have set the required competences for a specific job. Flexworkers should see only best matches in their personal dashboard.

Week 6 The focus of week 6 is to implement transaction effort hours and achievement registration for contingent workers. This way flexworkers can log their work and send invoices to companies.

Week 7 Every user would like to have insight in information specific for different roles assigned to this user. The focus of this week is to show reporting for both company users as well as contingent workers.

Week 8 + 9 Further improving search and match module as well as the compliance modules in order to improve automatic matching.

Week 10 Wrapping up.

4.3.2. Actual execution

In the actual execution of the project, tasks for different weeks have changed slightly over time. We found that certain aspects of the platform took a lot longer to implement and have spent a significant amount of extra time implementing the front-end. In the end, using React and Redux might have slowed us down in the first couple of weeks but definitely has lead to a result that better matches the clients needs. Also it has lead to more easily maintainable and extendable code. The actual execution of the development per week has become as followed:

Week 3 We knew from the start that the first 3 weeks would be the most busy ones. Setting up the entire environment and creating the first modules seemed to be more work than we thought. After week 3 we managed to have set up a basic profile page for contingent workers and project managers. Also after this week we were able to show basic project information pages for (at this time automatically seeded instead of created in the interface) projects.

Week 4 This week we have spend a lot of time in further designing and implementing the database and overall back-end. The negotiation module has required a lot of time to implement both in the back-end as well as the front-end. At the end of this week we managed to implement most of the back-end functionality and get a basic chat functionality ready in the front-end, which would later become the negotiation module. At the end of this week we were also able to give a first 'working' demo where flexworkers and project manager could sign in, find each other through project matching and start communication.

Week 5 After the first few weeks we noticed that some of our decisions in implementing the front-end were a little bit naive. The 'store' we use to save UI state, database records etc. in the front-end started having deeply nested relations which made it difficult to extend and maintain. Therefore in this week we have spend time on refactoring some large parts of our application in order to gain more speed in the development in the future.

Week 6 The planning for week 6 was to implement transaction effort and achievement registration for contingent workers. Together with our client we have decided that this would not become the main focus of this first version and we should therefore first spend time on other modules. Therefore we have started discussing how a basic search and match should be implemented in this first version to work on this first. After this week we were able to show decent search and matching.

Week 7 Reporting is an important aspect of the platform. However, it is much more easily developed once all the data is already is in the dashboard. Therefore we have decided to move reporting to the last two weeks to first focus on other modules. During week 7 we have further developed and finished the negotiation module and improved our matching algorithm.

Week 8 + 9 At this point, our application contains a well-structured back-end with almost all basic functionality built in. In the first couple of days of week 8 we did another refactor of the front-end application to bring it up to par with the back-end. After that we have to focus on tying all lose ends together in the front-end to make the platform usable. During week 8 and 9 we will focus on building remaining forms (for data that was previously seeded in the back-end), create basic reporting pages, allow flexworkers to write hours to projects and further wrapping up. Also we will further improve our matching algorithm, compliance module and perform user tests.

Week 10 During week 10 we will focus on wrapping everything up and getting the platform ready for our demo and presentation.

5 Implementation

In this chapter we will go into the implementation details of the project. In section 5.1 we will describe the application structure and design, and in section 5.2 we describe the database design. We will conclude this chapter with details on our matching algorithm in section 5.3.

5.1. Application design

This section is about the architectural design choices we have made. First, we will describe the structure and design of the back-end, then we will go into detail with the front-end and finally we will talk about the coupling of back-end to front-end and the difficulties that brought.

5.1.1. Back-end

The back-end in our application is a stand-alone application, able to handle REST requests. Its main purpose is storing, updating and retrieving data. Also, it serves the opening page of the application, which is just a very simple HTML file, with one JavaScript file attached to it. This JavaScript file contains our entire application bundled and minified.

Design goals of back-end We designed the back-end with two main goals in mind: Providing data and protecting data. It is a stand-alone application that runs on a server, receives requests, and returns the data.

The first goal was achieved by creating a REST API. It has certain end-points that can be called from clients. Based on which route is entered, the server returns either a subset of data from the database, or it stores or update one or more objects in the database. After a store or update action, it returns the updated data to the client.

Protecting the data was the second design goal. The back-end should make sure that the database cannot be corrupted. This means that even though the front-end has bugs or security issues, the back-end would not be affected by it. People with malicious intent (or by accident) should not be allowed to retrieve, store or update data for which they do not have the correct access rights.

To authenticate a user, a unique token is send with each request from the client to the user, the back-end can use this token to retrieve the corresponding user from the database to check access rights. In the following section we will explain what components a request travels through to ensure that a user can only handle data that is intended for said user.

Request data flow

Whenever an API request is made, it is passed through a number of components. In figure 5.1 a basic overview of this flow is given. This flow is explained in more detail in this section.

Router Every request starts at the router. The router is a class that retrieves the URL, and then calls the correct function in a controller class to handle the request. Before the router

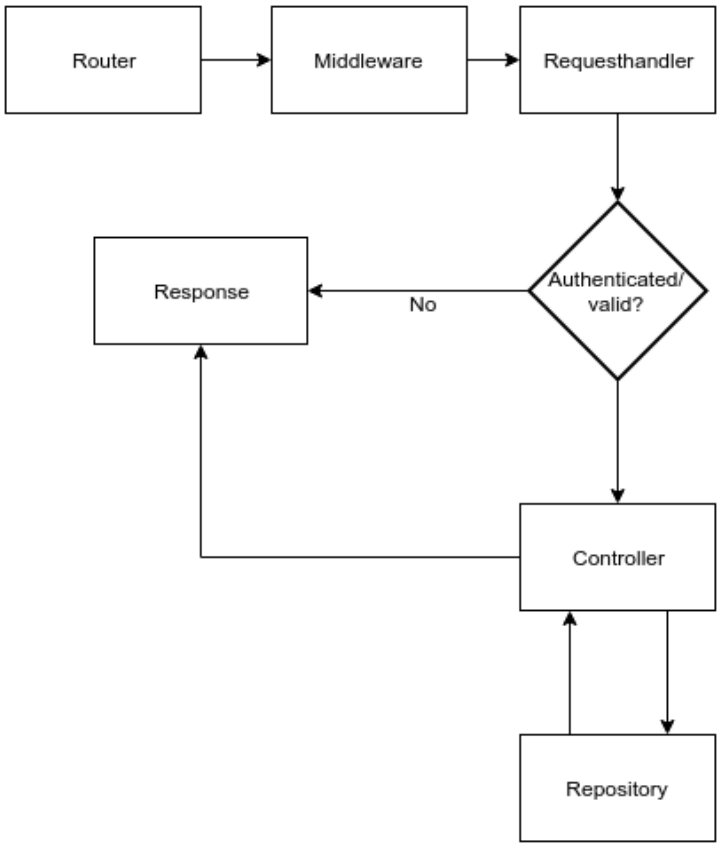


Figure 5.1: The flow of an API request.

class is called though, the request is passed through the middleware classes and the relevant request handlers.

Middleware Before the request enters the controller function, the request is send through the middleware that is registered to the specific route. In our application, we use two different kinds of middleware: basic authentication and model binding.

The authentication middleware just checks whether the user is authenticated. This is called before all API calls are handled. If the user is not logged in, it immediately returns a JSON response with a forbidden (403) status. Otherwise, it calls the next middleware class, which is the one for model binding.

Some of the routes require an action on a model, for instance updating the information on a project. This middleware allows the application to inject the right instance of a model based on an ID that's specified in the URL. In the case of a project, this middleware checks the database for that ID, and injects the relevant model into the request, so the controller can use it directly.

Request handler Before the request arrives at the controller, the relevant request handler is called. Every handler has two functions, namely authentication and validation.

The authentication middleware called before only checks if the user is logged in, but it's the task of the request handler to check if the authenticated user is allowed to execute a specific task. An example of this is updating the information of a project. This can only be done if the user is the owner of the project, otherwise every authenticated user could update all projects by sending the right requests to the server. If the check fails, a 403 response is send immediately.

The second task is input validation. This checks if the request contains the required values, and if these values are of the right form (for example: a string or number, minimum

or maximum length or an email address). It can also handle more complex validation, like checking if an email address is not yet in a certain database table. If one of these validation rules fails, a JSON response is returned, containing an object with all validation errors, including a text description of why that particular validation rule failed. If everything is validated successful the request finally arrives at the controller.

Controller The controllers only function is calling the right repository function(s) and sending a response to the client based on the result of the repository call. If successful, the response can contain the data that the client requested, or just an acknowledgment that a model was successfully updated. If unsuccessful, it returns a response containing an error message saying what went wrong.

Repository The repository is called by the controller, and is the abstraction layer between the database and the application. The repository contains functions that store, update or retrieve objects from the database. This allows the rest of the application to be unaware of the specific database implementation, since every database request is handled by a repository function. On certain 'events', in our case adding or removing a competence to a user is an example of this, the repository fires an event, which is handled by an event handler.

Event handler The event handler handles events received from anywhere in the application. An event is an indication that a side effect must happen. The example mentioned in the previous paragraph is an example of that. The repository handles updating the database, so that adding or removing a competence is stored. For our application though, whenever this happens, the matching scores for that user need to be recalculated. There are two reasons why the repository should not handle this. First, it is not in line with the single responsibility principle, and secondly, this calculation has no influence on the value of the response of this request.

The way we handle events is that an event is just a description of what happened, in our example this means the event is for example: 'UserCompetenceUpdatedEvent'. To handle these events, event handlers can subscribe to certain events. Whenever an event is fired, these handlers are called. Our server has a number of queue workers running that can handle these events in the background. This way both problems are solved, the repository has no knowledge/task of updating the scores, and the user does not have to wait until the side-effect calculations are finished.

5.1.2. Front-end

For the front-end we use a combination of React and Redux, as explained in the previous chapter. The basics are explained there, so we won't go into as much detail about it in this section. We will, however, discuss our implementation using these frameworks. An impression of the front-end interface design that we have developed can be found in appendix B.

The general idea

React uses a top-down approach to property handling. This means that children components are unaware of parent components. Properties are send down from parents to children. This approach makes it possible to distinguish two different kinds of components. Smart components and dumb components.

Dumb components are components that are completely unaware of the rest of the application. A dumb component is in fact just a function that retrieves properties and renders a HTML component. In the previous chapter we talked about reducers, which consist of so-called 'pure' functions, dumb components are the same. They receive properties and based on these properties they render a component. Every time these properties are the same, the resulting render is exactly the same. This makes them easily testable and reusable.

Smart components on the other hand, are aware of the environment. In our case this environment consists of the Redux framework. In our implementation, only the main screen

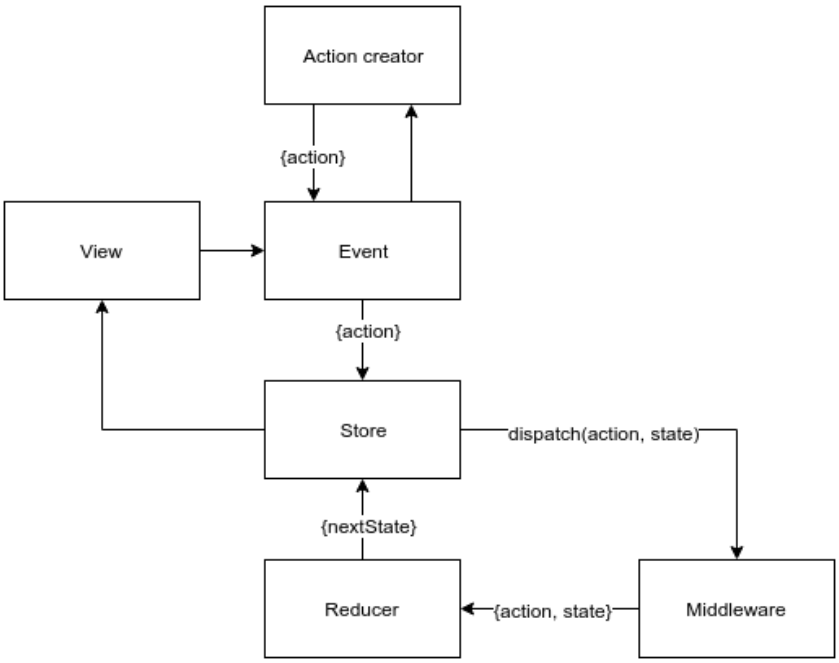


Figure 5.2: Event flow in our application

components are 'smart' components. These are all components directly linked to a route. We call these components 'Scenes'. Scenes select their data required from the store, and pass it to the relevant 'dumb' children, which are responsible for the looks of the application. This approach makes it easy to debug and develop. All data retrieval for each scene is done in its own separate file.

The data retrieval is handled by selectors. Selectors are functions that take input, run a function over that input, and return the result. The input of a selector can consist of multiple things. Two of which are standard: the store and parameters. The third one however, is the power of selectors. They can also have the output of another selector as input. This means you can build incredibly complex selectors out of a tree of simple selectors. What makes this even more powerful is that selectors recalculate intelligently. If the input does not change, it does not have to recalculate the result. So if just one small part of the data, somewhere down in the selector tree, changes, it only has to recalculate the relevant selectors. The ones unaffected can just retrieve the results from memory.

The store is coupled to smart components through these selector functions. Besides the fact that this reduces the complexity and duplication of coupling the Redux store to the view components, it also has another advantage: application performance. React uses smart rendering. This means that a component is only re-rendered when its properties have changed. Since selectors only return a new object when the data is changed, this makes react perform very well, even when there is a large amount of data visible on a certain page.

Application flow

React is updated by events. An example of an event is a user that interacts with the application or an event that is fired by an asynchronous API call that returned. An event passes through several components of the front-end application before the view is updated. This flow is shown in figure 5.2.

Whenever an event fired, an action is created by the relevant action creator. As stated before, an action is just an object with a type, and some relevant data, also called 'payload'. This action is passed to the store, which has a function 'dispatch(action, state)', that passes the action through the registered middleware functions. In our application, we use two kinds of middleware at the moment. The first logs all actions, their previous state and their next state. This makes it easy to debug, since every step the application takes is logged. The

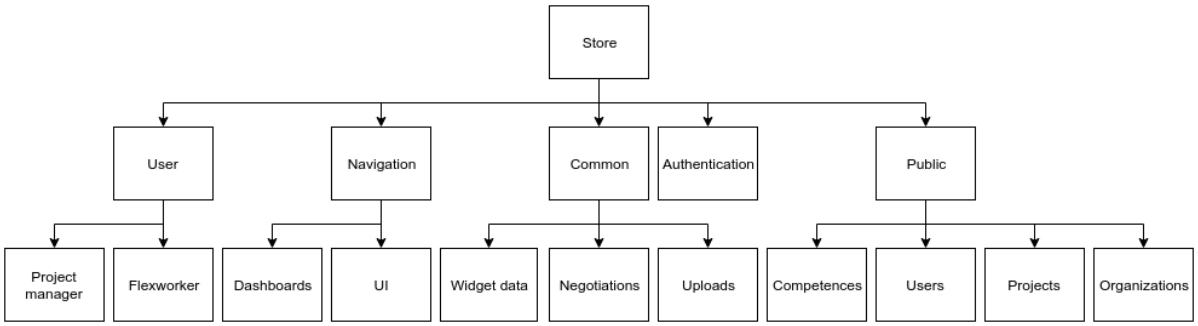


Figure 5.3: The original tree layout of the Redux store

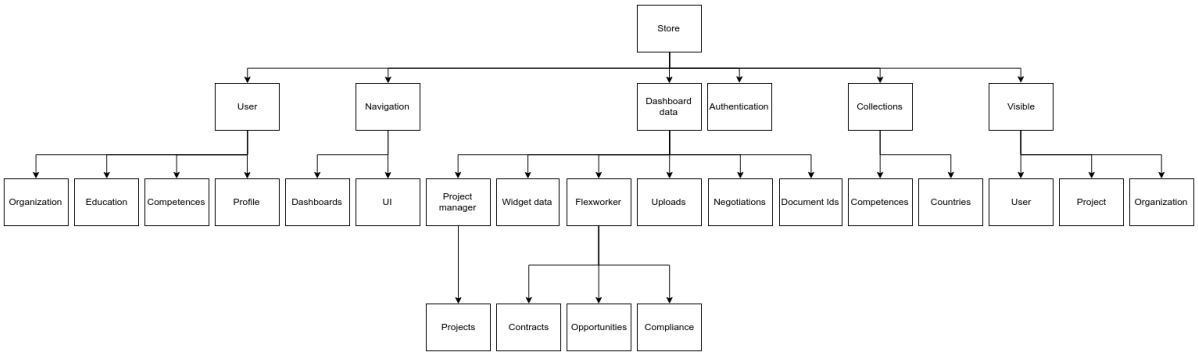


Figure 5.4: The refactored store tree

second one allows us to fire asynchronous actions, like API calls, in a synchronous style, where one asynchronous action is split in three different actions that are dispatched. One before the request is send, one when the request was successful, and one when the request was rejected.

The requests that are passed through the middleware, end up in the reducer. The reducer takes the current state and the action and returns a new state. This new state is send to the store, and completely replaces the previous state. The new state is send back to the store, and the view is notified that the store has updated.

Store design

The front-end of our application requires a place where data can be stored. Redux provides this for us in the form of a store. A Redux store, as explained in the previous chapter, is a tree structure where all application data is stored. In figure 5.3 the initial layout we created for our store is shown.

After using this model for a couple of weeks, we have discovered its limitations. First, there was not a clear distinction for where new features should be built. Some of the components of the store also had multiple responsibilities, which made it hard to implement new features, or to use one store for multiple front-end components. Its components had to be more decoupled. After a brainstorming session we decided to refactor the store to the model seen in figure 5.4.

The newer model has multiple advantages over the old one. First, and most important, the structure is more clear. Every branch has a clear purpose, which makes it easier to know where to add new components. Apart from that, every leaf now holds a subset of the data that has close relations to each other, or often needs to be used together. This new structure made implementing new features a breeze, where in the old model, it was tempting to just 'hack' a few stores together to get the right data for a component. This lead to a lot of regression bugs, which made it sometimes frustrating to implement a new feature. Since we used the new structure, this hasn't happened yet.

5.1.3. Coupling front- and back-end together

While designing the structure of the store and the API routes of the back-end, we had to balance independence on one side and development speed on the other. Too much independence leads to more complexity during development. Too little independence, on the other hand, might make developing new features easier in the beginning but often increases the complexity of the application as it grows. To solve this problem we decided to implement an extra layer in the back-end that functions as a sort of adapter.

This layer consists of two types of components: controllers and decorators. We designed this extra layer on top of the original CRUD (Create, Read, Update, Delete) controllers. How we implemented it, is by giving each leaf in the store tree its own endpoint in the API. This endpoint has basic routes, depending on the specific leaf, but in general they all have a 'list', 'store' and 'update' route which respectively return all objects the user is allowed to see, store a new object and update an existing object. These controller functions only return the object that that specific part of the store focuses on. This worked great, but since most of these stores require also some related objects to show the correct data (like a project has a project owner, a company and one or more flexworkers), we needed to return these objects too. We decided that this was not the task of the controller, so we built decorator classes.

Decorators, in our application, are classes based on an interface with one function: decorate. In essence, this function takes an object and loads the required related models for that decorator. Each route can define which decorator to use for the return of the data. This made sure that every API call gets the correct data, without having to specify the relations necessary in a repository or controller class. This further separates the concerns.

We implemented this system at the same time as the refactoring of the store, and it proved to be a success. Implementing new features cost less time, and introduced fewer bugs.

5.2. Database design

During the process of designing our database we have tried to stick as much as possible to the agile software methodologies. We chose to, instead of having a detailed plan at the start of the project, design only basic aspects up front and try to have the system evolve through various iterations and time. For this reason, our initial design only contained users, organizations, projects, competences, and contracts. Throughout the process of each sprint, we have further developed relations as well as additional tables at the point of which they were also actually being implemented in the rest of the back-end. This way we were always certain that the database only contained models that were currently used by the application.

In order to be able to work together on the same project and database, Laravel has a built in functionality called 'migrations'. This functionality allows developers to run a single command and always have an up-to-date and synchronized version of the database. It acts as a sort of version control for databases. Throughout the process of designing the database, we have always updated our changes in these migrations to make sure that our local environments stay synchronized.

In Figure 5.5, a high-level overview of the current database design is presented. This design contains all the currently implemented system components including matching, negotiation and compliance. Because this is only a proof of concept of the platform, and because the database will be further developed according to the same agile methodology, this design is not final. However, it is likely that the three main systems components will remain and new blocks will be added on top of these.

5.3. Competence Matching

The current implementation of the matching algorithm is based on the competences provided by the user together with those required for a project. A predefined list of competences, obtained from the ESCO classification of European Skills, Competences, Qualifications and Occupations[1] is stored in the database as a flat representation and used for the matching.

Because there are many skills that could be regarded the same or almost the same, we

[1] http://ec.europa.eu/social/main.jsp?catId=1042

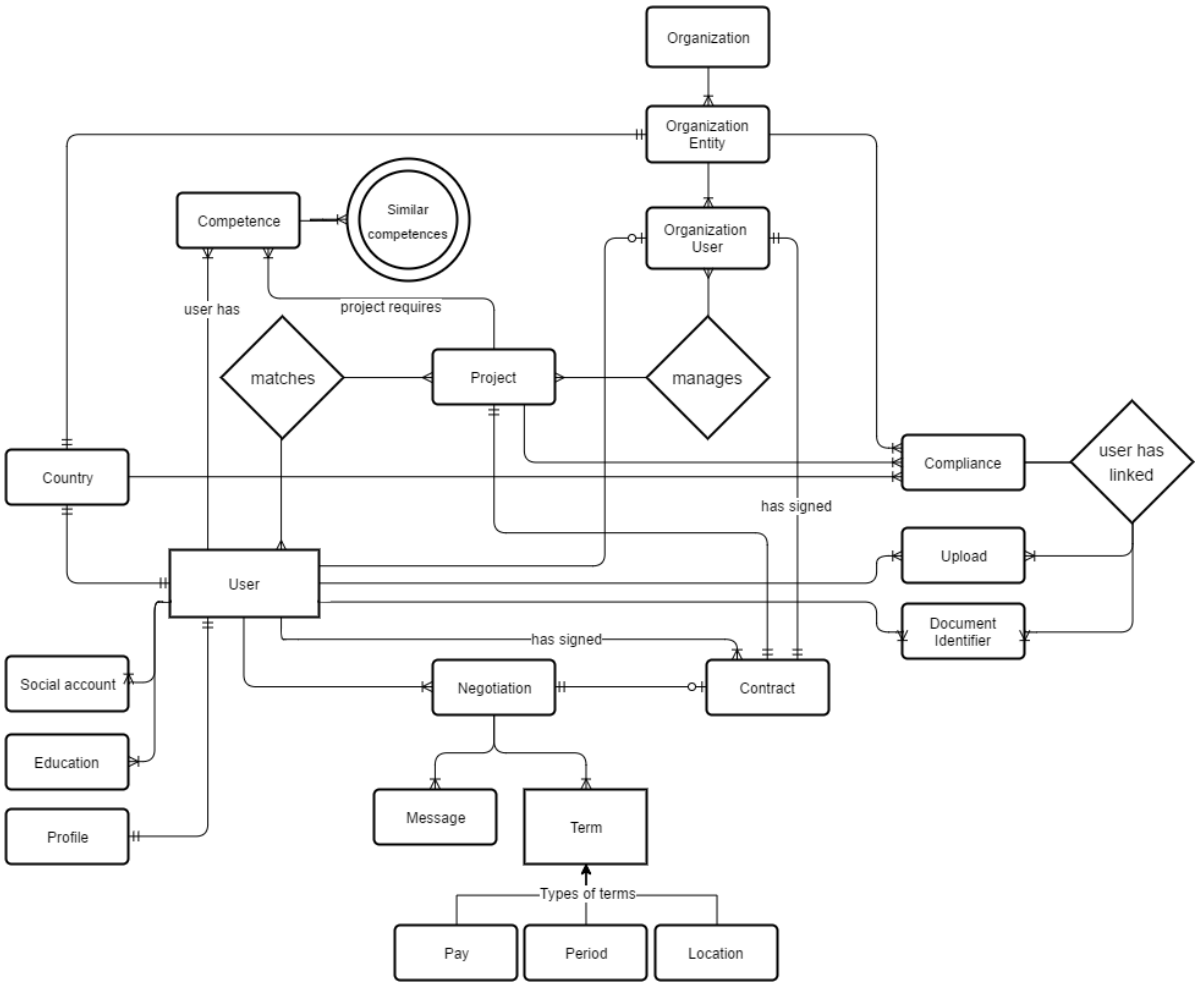


Figure 5.5: High level design of the database

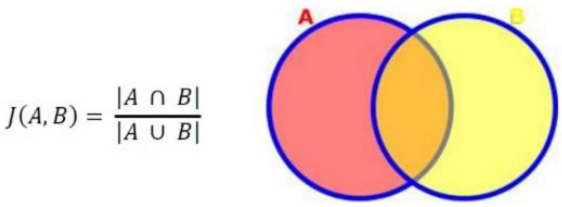


Figure 5.6: Definition Jaccard similarity coefficient.

were required to think about how to guide people in entering certain competences. Also, we had to think of a way so that people entering 'JS Developer' and 'JavaScript programmer' are matched to the same set of jobs. We have achieved this by applying the following methods in the front-end, back-end and algorithm.

Front-end: times used When a flexworker wants to add a competence to his or her profile, he or she has the choice between an (nearly) endless set of competences from the ESCO framework. By showing in the front-end how many other flexworkers have chosen to set the same competence, the flexworker is stimulated to choose competences that are being used most often by other users. It is a natural choice to pick a competence of which you know many people have also chosen the same competence. This way we prevent users from all choosing different competences to describe (almost) the same ones. The same goes when project managers set competences for projects. They also will see how many flexworkers have chosen certain competences and are likely to choose the ones that are used most to get the best match.

Back-end: competence similarity In the back-end, a scheduled script runs to check the similarity between different competences. We have chosen to calculate the similarity between competences as a non-euclidean distance and save this distance in the database. The equation to calculate this distance states that the distance between two competences is equal to the amount of times competence x and y have been used together by a user, divided by the number of times competence x was used plus the number of times x was used without competence y. The result is that of each competence we can say how likely it is that a user possessing this competence, will also possess another competence. This can be used in our matching algorithm to match 'similar' competences.

Jaccard We have chosen to base our algorithm on the Jaccard similarity coefficient. This coefficient is a statistic used for comparing the similarity and diversity of samples sets. It measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample set (see figure 5.6). The reason why we have chosen to use Jaccard is because it prevents the matching of overqualified people (who add a large amount of competences to their profile). We believe that the matching score should not be based on the fact whether someone possess all required competences for the job, but more as to whether someone's entire profile best matches the project's requirements. Jaccard similarity allows us to find this closest match.

Algorithm The competences chosen by flexworkers and for projects together with competence similarity form the basis of our algorithm. For our implementation, we have taken the Jaccard similarity coefficient and added extra rules to account for similar, but not exactly matching competences. The main idea is that first we compute the denominator and divisor (as we would with Jaccard) between user and project competences. After that, a threshold is computed which states the minimum level of similarity between two competences. This threshold is currently calculated as the average amount of similarity but should later be chosen based on findings from interaction on the platform. Then, for each user competence that is not in the intersection of the two competence sets, the top similarity of all project

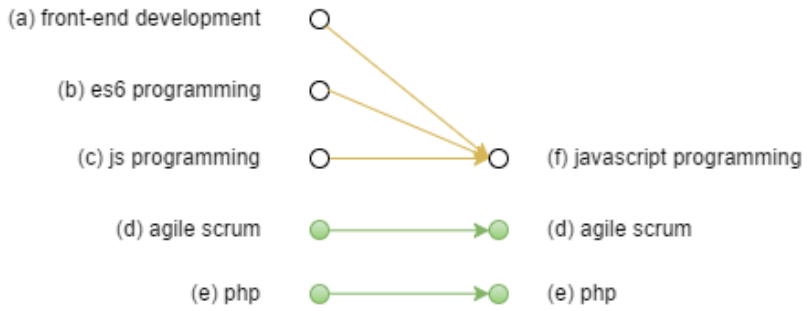


Figure 5.7: Visualization matching algorithm.

competences is looked up in the database. If this similarity value is higher than the previously defined threshold, this similarity value (between 0 and 1) is added to the denominator value. Furthermore, $(1 - similarityvalue)$ is added to the divisor value. This way, As long as a competence from the user is somewhat similar to that the project manager proposed for his project, the overall matching value becomes larger and our algorithm returns a better match.

A visualization of the algorithm is given in figure 5.7. This figure shows 6 different competences of which 2 are set by both the flexworker as well as the project manager. Therefore, the Jaccard similarity coefficient would be equal to $2/6 \approx 0.33$. In this example, we assume that the similarity between competence a and f ($sim(a, f)$), as well as those between b and f, and c and f ($sim(b, f)$ and $sim(c, f)$) is higher than the threshold set in the algorithm. For convenience, we will even assume that the similarity between competences is approximately equal to 1 (100%) since in essence the competences that exceed the similarity threshold, are almost the same. The algorithm will now compute the matching score based on the similarity as followed:

$$score = \frac{2 + sim(a,f) + sim(b,f) + sim(c,f)}{6 + 3 - sim(a,f) - sim(b,f) - sim(c,f)} = \frac{2 + 2,4}{6 + (3 - 2,4)} = \frac{2}{3} \tag{5.1}$$

When we now compare the Jaccard similarity coefficient to our algorithm, we can say that we have successfully improved the the matching based on similar competences. The algorithm has become less sensitive to the naming of competences and has the ability to match users to projects even while they choose to use different descriptions of the same competences.

An overview of the implementation in PHP is given in figure 5.8. In chapter 8 we will further discuss our recommendations regarding the improvement of this algorithm in the future.

```php
/**
 * Matching algorithm returns a Jaccard index taking in mind the similarity
 *
 * @param User $user
 * @param Project $project
 *
 * @return float|int
 */
public function similarityJaccard(User $user, Project $project)
{
    $user_competence_ids = $user->competence_list;
    $project_competence_ids = $project->competence_list;

    $intersection = array_intersect($user_competence_ids, $project_competence_ids);
    $union = array_unique(array_merge($user_competence_ids, $project_competence_ids));

    $denominator = sizeof($intersection);
    $divisor = sizeof($union);
    // If there is no skill in common, return 0
    if($denominator === 0) {
        return 0;
    }
    // Very low threshold for testing
    $threshold = max(0.5, DB::table('competence_similarities')->avg('value'));

    foreach($user->competences()->get() as $user_competence) {
        // if user competence not in intersection find similar
        if(!in_array($user_competence->id, $intersection)) {
            $similar_competence = $user_competence->similarities()
                ->whereIn('similar_id', $project_competence_ids)
                ->orderBy('value', 'desc')->first();

            if($similar_competence->pivot->value > $threshold) {
                $value = $similar_competence->pivot->value;
                $denominator = $denominator + $value;
                $divisor = $divisor + ( 1-$value );
            }
        }
    }
    return $divisor === 0 ? 0 : ($denominator / $divisor) * 100;
}
```

Figure 5.8: Matching algorithm with jaccard and similarity

6

Quality assurance

In order to assure a high quality product, different measures have been taken in order to assure that quality code is built and retained. In this chapter we first discuss the development framework we have used for the overall development process. After that we discuss how we have tested our code throughout the development process.

6.1. Agile

During this project we have applied the agile software development framework Scrum. By having goals and demos set for every sprint, we and the client have had the chance to step in and change direction on time at any stage of development. This way we have ensured that we were always on the same page and changes could be made in time. The project management tool we have used is called Team Foundation Server, for which a license was provided to us by the client. This tool has given us insight in what still had to be done every sprint and allowed us to contact the client in time if this threatened to be too much.

6.2. Testing

Apart from the flexible approach, we wrote unit tests for different aspects of our code in order to ensure that every part works as intended. And just as important, to detect regression bugs as early as possible, so it keeps working as intended.

6.2.1. Unit testing

Unit tests have been written for both front-end and back-end. For the developing of the back-end we have implemented the methodology of Test-Driven Development. For the front-end, it has been more difficult to do the same since code structure has changed dramatically over time and we would spend too much time writing tests for code that would then have to be refactored again. Therefore, we have started testing our front-end at a later stage in the process. To reduce the amount of bugs in the front-end anyway, we used a couple of developing tools build for React and Redux to inspect the workings of the front-end application by hand. While this is not the best approach when an application grows a lot in size, for the first couple of weeks this was enough. In a later stadium, when the front-end structure and design was robust and likely to remain the same, we started writing tests for the front-end.

To make sure that the master repository was always working and bug free, we used continuous integration to run the tests and build the pull requests before we merged. This made sure that all tests in the master branch would always run, and the application would always build correctly. After this we could deploy it to the server with a simple script.

Back-end In order to test the back-end we used PHPUnit, a programmer-oriented testing framework for PHP. Because almost all interaction between user and back-end is done through API calls, we have focused on testing the routes and the controllers that are called

```
describe('widgets-reducer', () => {
    it('handles RETRIEVE_WIDGET_DATA_SUCCESS', () => {
        const initialState = {}
        const action = {type: 'RETRIEVE_WIDGET_DATA_SUCCESS', widget_data: {
            unread_messages: 3,
            pending_terms: 1,
            open_negotiations: 4
        }}
        const nextState = widgetReducer(initialState, action)
        expect(nextState).to.deep.equal({
            unread_messages: 3,
            pending_terms: 1,
            open_negotiations: 4
        })
        expect(initialState).to.deep.equal({})
    })
})
```

Figure 6.1: The basis for a reducer test

by the routes. By applying the methodology of Test-Driven Development, we have been able to make sure that every time an update is pushed, it can immediately be verified that all API routes still work as supposed to, and a newly added route also works as intended.

Because almost all of the PHP code that was written by us is contained in the App folder of our project, we have strived to develop a high percentage in coverage over this folder through unit tests. The overall coverage in this folder is currently over 75%. Of course coverage does not tell us too much about the quality of our tests but we believe that it gives a decent indication of the number of tests developed for the back-end.

In the current state, all middleware, custom request handlers, controllers, repositories and decorators have been tested, allowing us to confidently update our back-end code and extend functionality without having to worry about unknowingly breaking existing code.

Front-end As earlier mentioned, we have not completely been able to apply the methodology of Test-Driven development to the front-end as well. In the final three weeks of this project, when we became quite content with the structure of our React and Redux code we have developed most of our JavaScript unit tests. These tests have been developed using the Mocha framework. Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing a lot more simple.

Most of the front-end logic that needs to be unit tested can be found in three types of components: actions, reducers and selectors. Figures 6.2, 6.1 and 6.3 show basic implementations for these types of tests. In our application, most of the tests have additional tests to assert behavior specific for that reducer, action or selector. These were often developed after a certain bug happened to make sure it wouldn't happen again.

As we became more proficient in writing tests during the late phases of our project, we started developing some features with writing tests before implementing. While not every part of the front-end lends itself for test-driven development, actions, reducers and selectors are very suitable for it.

6.2.2. User testing

Apart from writing Unit tests, we have also designed user tests or usability tests in order to assure quality in user experience as well. Because most of what has been implemented in the platform so far is in the flexworker dashboard, and because organization user dashboards do not yet contain much more functionality than that of the flexworker, we have chosen to write and execute user tests for flexworkers.

The setup for the flexworker user test can be found in Appendix D. This test has been

```
describe('widgets-action', () => {
    it('creates RETRIEVE_WIDGET_DATA_SUCCESS', () => {
        const expectedAction = {type: 'RETRIEVE_WIDGET_DATA_SUCCESS', widget_data: {
            unread_messages: 3,
            pending_terms: 1,
            open_negotiations: 4
        }}
        const jsonResponse = { unread_messages: 3, pending_terms: 1, open_negotiations: 4 }
        const action = retrieveWidgetDataSuccess(jsonResponse)
        expect(action).to.deep.equal(expectedAction)
    })
})
```

Figure 6.2: The basis for an action test

performed by a total of 5 people. This number does not give us enough data to draw truly grounded conclusions as to whether results count for every user but it does give us insight into the flaws or difficulties people meet when using the platform. The people that have executed this test are either flexworkers themselves or are familiar with their way of working. In the rest of this section we will discuss the most interesting results found during the execution of this test.

Create profile Most of the testers chose to sign up using LinkedIn. The reason for this might be because just like LinkedIn, FleXentral is a professional platform that requires the same type of information and is used partially for the same purposes. Two of the users signed up using the registration page and found no struggles in doing so.

First impression Once users signed in, it was not always immediately clear what they could use the dashboard for. People who signed in using LinkedIn immediately recognized it to be their own profile page with their LinkedIn avatar set. Users who had just created an account, however, seemed more 'searching' for overview. The overall opinion about the layout was that it looked fresh and well-designed but it could contain more structured content. People who would scroll down found that the top navigation tabs would disappear and the content was not 'guiding' enough.

Education & Competences The first action that was asked of the users, was to add educational information and competences to the user profile. The 'edit' button in the top right corner of both panels seemed clear indication of where to click. However, a small text on the empty panels could improve the user's call to action more. Right now, users did not feel the need to add competences while this is one of the main aspects of the user to project matching for flexworkers. Therefore, this could stand out more to make the user more inclined to update the profile page from the beginning.

Projects Once users had their personal information set up, it was clear where to click in order to find opportunities or projects. With the 'start negotiation' button fairly present, users could easily find their way to the negotiation module.

The negotiation module worked alright for the users but it required some time to get to know the basic actions. We believe that this was partially due to some small bugs that were still present in this module at the time of testing. This lead to users trying to propose terms while these were already opened by the project manager, which lead to an invalid response. One thing we should work on in this module is the real-time refreshing of messages, terms and contract signing. Users that propose a contract should immediately receive feedback, stating that they have sent a proposal, otherwise it is a natural response to keep clicking the button expecting something to happen. Overall the negotiations module could contain more feedback to the user in order to improve the user experience.

```
describe('competences-selector', () => {
    it('gets a component list', () => {
        const state = {
            user: {
                competences: expectedResult.response
            }
        }
        const expected = [ { id: 8,
            name: 'Engineering',
            times_used: 25,
            created_at: '2016-06-08 11:11:56',
            updated_at: '2016-06-10 13:29:36' },
            { id: 8,
                name: 'Engineering',
                times_used: 25,
                created_at: '2016-06-08 11:11:56',
                updated_at: '2016-06-10 13:29:36' },
            { id: 9,
                name: 'Hospitality',
                times_used: 19,
                created_at: '2016-06-08 11:11:56',
                updated_at: '2016-06-10 13:29:45' },
            { id: 10,
                name: 'Insurance',
                times_used: 20,
                created_at: '2016-06-08 11:11:56',
                updated_at: '2016-06-10 13:47:11' } ]
        const result = getCompetencesList(state)
        expect(result).to.deep.equal(expected)
    })

    it('gets an empty list when there are no competences', () => {
        const state = {
            user: {
                competences: {}
            }
        }
        const expected = []
        const result = getCompetencesList(state)
        expect(result).to.deep.equal(expected)
    })
})
```

Figure 6.3: The basis for a selector test

Compliance At the moment of user testing the compliance module was not yet fully functional. Therefore we chose to tell the testers to skip this step and go on to the next one.

Effort The users found the module for effort writing very basic. It was clear that, using a simple form, hours could be added to certain projects. There was, however, not yet enough insight into the hours written and therefore this module also still felt unfinished. The module for effort is not yet anywhere near a final implementation and should therefore be tested more in the future to gain more valuable feedback.

Conclusion All in all we can say that the users tests have been a great success in order to gain insight in our users their needs. What we should focus on most in the future is the guidance of our users through the product. We should work more with messaging and live feedback in order to fully gain the advantage that we have with our current JavaScript front-end implementation.

7

Evaluation & Reflection

In this chapter we evaluate our work and reflect on the quality of the work that we did during this project. We will first evaluate our work based on project requirements. After that, we discuss feedback and improvements proposed by the Software Improvement Group (SIG). Lastly, we evaluate our work on a personal level stating what we have learned and in which way.

7.1. Requirements

We start off by discussing how we measure the quality of our delivery based on requirements. We split this section up into two parts, of which in the first part we will discuss how well we built the application based on the functional requirements we initially described. For every requirement we will discuss what we implemented and where the implementation possibly lacks in functionality. Secondly, we explain the non-functional requirements that were partially requested by our client, and partially imposed on us by ourselves.

7.1.1. Functional

The functional requirements are listed in chapter 3. In this section we will elaborate on what we implemented, what the difficulties were and what still needs to be done. At the start of this project, the requirements were globally divided into the following seven modules. Even though we discovered during our sprints that some of these modules were so close together in (partial) functionality, that the distinction between some of them in the final delivery is small.

Registration The registration module should allow for both flexworker registration as well as organization registration. We have currently implemented both, although the registration for organizations is not yet a final implementation. We believe that organizations should not register their entire organization in a single form (as they do right now) but should be able to structure their organization in a step by step manner inside the dashboard. The reason why we have not yet implemented this is because it was not really considered the focus of this project and because there is still some debate about the best way to implement the user experience in this. Apart from flexworker and organization registration, one of the requirements was that users should be able to connect using LinkedIn. We have implemented this functionality and allow each type of user to log in using their LinkedIn account.

Compliance The compliance module is a module of which its final implementation is going to be very dependent on feedback from first clients. Right now the entire back-end is setup to meet the current requirements, allowing flexworkers to see the rules they are obliged to follow in order to be compliant. It also allows for users to link registered document identifiers and files to the rules in order to check them and get the status of being 'compliant'. Because the details regarding this module are so much dependent on first clients, it is important to

get the requirements from these clients clear as soon as possible. In future implementations, registered document identifiers listed by flexworkers should be checked automatically using government API's. Although we have designed the database in such a way that the type of a document could easily be linked to an API call, we have not yet implemented this in the front-end to allow for automatic checking. The reason why we have not yet implemented this, is because such functionality is still not as interesting at this stage of the product.

Marketplace It is clear that a large focus on the project is on the matching of projects and talent. In our initial requirements, the marketplace where flexworkers can find jobs and vice versa was mentioned completely separate from the search & match module. However, during the project and based on discussions we have had with our client, we have decided that an important aspect of the marketplace is the matching algorithm. Therefore, we have chosen to focus on this algorithm and combine the two requirements into one single module for this version. The result is a list of project opportunities for each flexworker, from which he can start a negotiation. For the project manager, a list of 'top talent' is generated for each project within the project overview page. The marketplace is therefore now limited to only the by us generated top matches. In the future this might change in a way that project managers can search more accurately using filters and not only based on the matching score.

Search & match We have had some long discussions with our client and between the two of us in order to choose a right first way to implement a matching algorithm. The way we have chosen to go with is an algorithm that checks the similarity between the set of competences set by flexworkers and those set by project managers for their projects. Because we do not have any data yet and because there is no open set available that we could use as a ground truth to test our algorithm, we have decided to stick with this basic implementation and wait for traffic before further improvements are implemented. In the future, we plan on using data that is gathered on the platform in a more clever way in order to improve the matching algorithm. Luckily by then we will also be able to test different algorithms based on contracts signed by users in the past.

Negotiation The negotiation module is where flexworkers and project managers can communicate location, period and pay. All of the required functionality has been implemented into this module and it is considered to be finished, except for minor user experience adjustments. Since the negotiation module is the module where both flexworkers and project managers could potentially spend a lot of time and because it is one of the core functionality of the FleXentral platform, we are glad to say that this module has been implemented in a very similar way to what we had planned.

Transaction Currently we use a simple drop-down menu to allow for flexworkers to write hours to projects. In the future, however, we would like to create a more fancy transaction module with better user experience in time writing. Also in the future, flexworkers should be able to automatically load hours from a negotiated contract if the terms allow for this. An example of this is that when someone would sign a full-time contract, these hours could already be loaded to the transaction module and would only require acceptance instead of manual writing. Because this module really stands on its own, it has not yet had major importance for the rest of the platform and is implemented in a very basic manner.

Reporting The reporting module has always been the last one on the planning. The reason for this is because it requires data in order to show value. At this moment we have though of some basic reporting for each type of user on the platform. In the weeks between the final uploads of this report and the presentation we will focus on making those reports visible in the front-end.

7.1.2. Non-functional

The second part of our evaluation focuses more on the technical side of the project. In this section, for each non-functional requirement we will discuss how we implemented the different requirements and in which way this satisfies the client's needs.

Technological independence Since the scope of our project was quite large, and our time was limited, we decided to use multiple frameworks to reduce our workload. These frameworks are described and elaborated upon in chapter 4.

One way we have achieved technological independence was to create the front- and back-end of our application in a way that they were both completely stand-alone applications. For the back-end we used the Laravel[1] framework to create a REST API. For the front-end we decided to use Facebook's React. The front-end communicates with the back-end using asynchronous calls to the API. The front-end stores this data in a normalized form, so the visual layer is not depending on the structure of the API responses in any way.

While the Laravel framework has quite a few extremely convenient helper functions and classes that can be easily abused to create an application that is completely dependent on the framework, we have tried to limit their usage to encapsulating adapter classes. One example of this are the repository classes. Laravel has an Object-relational mapping interface, which, although very easy to use, can not be easily ported to another framework. To make this dependency less problematic, we encapsulated the retrieval and storage of data from the database in repository classes based on the repository pattern. If in the future the database/framework needs to be replaced, only these classes have to be rewritten.

Robustness As for robustness we can draw different conclusions for the front-end and back-end. Both have been developed as separate applications and are also tested separately.

When looking at the back-end, we can confidently say that data is well protected from unauthorized users and data corruption is highly unlikely. Because we are using different layers in our implementation in order to assure robustness like middleware and request handlers, every action is thoroughly authorized and validated before it is passed to the controllers and repositories. Furthermore, almost every component in the request data flow is well tested and can therefore be assumed to work as expected while extending the platform. Also, all errors in the back-end are caught and returned in JSON format, so the front-end can always handle this in a user-friendly fashion.

The front-end, on the other hand, is not yet as robust as we would like it to be. Due to recent refactoring it has become a lot more robust though. Splitting the data components in smaller, less coupled components with a single responsibility removed most of the errors and bugs that showed up. However, more feedback about the status of the application to the user should be implemented in future versions, and also better error handling. Most of the errors that occur in the front-end are not breaking, but since they aren't handled in a proper way, they might confuse the user.

Although the front-end will never be able to corrupt data since requests will always pass through the back-end, the application will not feel robust as long as feedback from actions is not received. These problems however, could also be considered as usability problems instead of robustness problems. In the current state, bugs in the front-end sometimes happen, but in the worst-case scenario only a page-refresh is required in order to continue. In the future, more tests for front-end and feedback from users can contribute to a more robust front-end.

Scalability We have tackled the problem of scalability in our application in two ways: by reducing the amount of complex calculations, and making the client handle the data filtering calculations.

The matching part of our application is, in regards to computational complexity, the most resource intensive aspect. Of course this calculation doesn't need to be executed every time a user wants to view matching projects or flexworkers. We tackled this by having a command that recalculates the scores for a certain flexworker or project. A listener that listens for

[1] www.laravel.com

events that affect the matching scores for a flexworker or project can fire that command when these scores need to be recalculated. When finished, the results are stored in a database, which can then easily be queried without the need to recalculate heavy calculations.

Secondly, filtering and combining data is done by the client. The server is responsible for providing the data a user is allowed to see, but we use selectors to 'query' the data in the front-end, using the computing power from the client. This reduces the strain on the server, since it has to handle much less API calls. These selectors are built in a treelike fashion, and only recalculate when data changes.

Because of this, our application only has to load the data on the first page load, then it is stored in the clients memory. After that, at most one API call per page change is done, and often no call is done. Storing and updating values still require the server, but there is no possible way around that.

Right now, with the amount of test data that is in the database, the first full load of the application including logging in takes 450ms, and just loading the data 150 ms. After this is ready, page changes happen immediately, because an API call to the server is usually not required. When the application has real users, and is used more, this can of course change, but we believe that this is a good start.

Maintainability As explained in chapter 5, our back-end is implemented by following a defined flow, where each request is sent through the separated components in a standard order. Since each type of component has a specific function (e.g. request handlers for authentication and validation, controllers for collecting the correct data and repositories for the link between the database and the application) it is easy to find what class is responsible for which event. Every type of class has a distinct single purpose. In our opinion, the structure of the back-end is clear, easy to debug, and easy to understand. There are minimal side-effects going on, and we are actively busy removing the last side-effects until everything in the back-end has a single, well-defined purpose.

Designing the front-end for maintainability turned out more difficult. As we will elaborate on in section 7.3.4, we have had two moments during the project where we refactored a large part of the front-end. The second refactor was mostly done to improve the maintainability of the application. The structure is now clearer and better defined, and we split up a lot of files with multiple responsibilities into multiple files with a single responsibility. After this overhaul, we are fairly satisfied about the structure of the front-end, but improvement is possible.

All in all, the project as it currently is, in our opinion is fairly maintainable. We have focused most on following the single responsibility principle and on decoupling as much as possible. It is far from perfect, but writing maintainable code is a constant, never-ending process of leaving code better and more clear than the way you found it.

Usability In chapter 3 we defined usability as fast, easy to use and available on every device. Also, it would be preferable if the pages would never have to refresh. Currently, two of these requirements are met: it is fast, and web pages never have to refresh.

As mentioned in the section about scalability, a full initial load takes 450 ms, after which every page change loads instantly, thanks to our implementation of React. We think that this is fast enough. The other two requirements are more difficult to define. In order to evaluate whether the application is easy to use, we can look at the user test results (chapter 6). These results show that there is a lot of improvement possible in the ease-of-use department. However, the feedback we got from the tests gave us very valuable insights, which we will use in order to improve the application.

Finally, although the application is fairly responsive, and therefore available on (at least the latest) mobile devices, it is far from perfect, and a large part of the application is not yet optimized for looks on smartphones.

We feel that there is still a lot of improvement possible for usability, but at the moment, the application is good enough to functino as a starting point. Periodically receiving feedback from user tests can help the application become a full-fledged, user friendly application.

7.2. SIG feedback

The Software Improvement Group (SIG) has designed a method that empirically determines metric thresholds from measurement data. This measurement data for different software systems are pooled and aggregated after which thresholds are selected that both bring out the metric's variability between systems as well as help focus on a reasonable percentage of the source code volume. Their method respects the distributions and scales of source code metrics, and it is resilient against outliers in metric values or system size.[2]

SIG has applied their method to a benchmark of 100 object-oriented software systems, both proprietary and open-source, to derive thresholds for metrics included in the SIG maintainability model. SIG has also made an analysis of this project and in this chapter we will first discuss the results of the test and after that note on improvements we plan on implementing during the last 2 weeks of the project.

7.2.1. Results

After SIG ran our code through their model at the 1st of June 2016, it received an overall score of 3.8 with 13,000 lines of code tested in both the back as well as the front-end together. In this section we discuss each of the metrics that was tested and its results.

Volume (5.2) Because our project code is compared to different software systems that are usually the size of 20 to 500 times the size of our code so far, the volume metric is not a metric that gives us a lot of insight in the quality of the code written so far. Our reviewer noted that we have written slightly more code than average Bachelor thesis projects but that not much other valuable information could be retrieved from this.

Duplication (3.2) Some of the classes in PHP (Laravel) and certain views in the front-end still contain some duplicated code of which we are aware. The duplicate code in the back-end is mainly present in the request authorization for creating negotiation terms because this logic is the same for different requests. This will be refactored to a single request class for creating terms that is then extended by the different requests that currently contain the duplicate code. In the front-end, the negotiation page contains some duplicate code in order to display contract details. This shall be refactored to be built up out of components instead of duplicate code.

After the duplicate code in these two parts of the project have been removed, SIG will be informed by this update and duplication should be sufficiently improved.

Unit size & Unit Complexity (4.1) What must be noted when looking at the results is these metrics, is that SIG could not yet cope with all of the Ecmascript 6 (ES6) syntax that we use in our front-end together with some best-practices from React. Therefore, Unit size and Unit Complexity were rated slightly lower. The fact that this was due to ES6 and React was noted by the reviewer from SIG at the time of review. Some of the back-end methods including the ones for database migrations could still be improved but the overall size and complexity in the back-end scored quite good.

Unit interfacing (5.3) Both our back-end as well as front-end methods do not contain methods where there is a clear problem in unit interfacing. The number of parameters passed usually does not exceed 3 or 4 and therefore the rating on unit interfacing quite good as well.

Module Coupling (5.4) What was striking when we looked at module coupling, it became clear that no class came anywhere near the danger zone defined by the SIG metric. One class, NegotiationRepository was referred 10 times, being the most referred class in our system but still in the safe zone. Throughout the project we have focused on decoupling all modules as much as possible so this is a good confirmation we have done this right.

[2] https://www.sig.eu/en/about-sig/publications/deriving-metric-thresholds-benchmark-data/

Component balance (1.9) Because our front-end contains a lot more code than our back-end and because the React components can become quite large, the component balance in our project does not score as good as we would like to within the SIG model. This result, however, is very much influenced by the parameters set to measure the project. Depending on what folders you choose to measure, this metric could give an entirely different outcome and the reviewer agreed with us that this value did not correspond to any flaw in our code that we should work on.

Component independence (4.1) The components in our project are sufficiently independent and no strange relations could be found during the analysis. Just like Module Decoupling, this has been a focus of ours throughout the project so we are happy to see we have scored so well.

7.2.2. Improvements

Of course there are some improvements to be made. In this section we discuss the two improvements our reviewer proposed to us.

Duplication An important aspect to improve on is the duplication in our code. Both back-end and front-end still contain obvious duplicated code that could be reduced and isolated into a single class or component. We have refactored this, and a 'before and after' can be viewed in Appendix C.

Front-end testing What also came out of the review, and what we were aware of is that the front-end was not sufficiently tested at the time of review. Because of the constant changes still being made to the front-end, this had been a low priority. After the review we started writing tests for the different kind of components our project has. As explained in chapter 6, we have tested mostly actions, reducers, and selectors. Because of the limited amount of time, not all of the components are thoroughly tested, but the most important ones are. An impression of current front-end tests can be found in appendix C.

7.3. Reflection

In this chapter we will reflect on how we think the project went. We do this by focusing on the following four parts of developing an application like this: Understanding of the problem, (high-level) design of the application, the development process and implementation decisions. Finally we will talk about what we learned and we conclude with how satisfied we are with the results.

7.3.1. Understanding of the problem

Since we are both students, the problem FleXentral aims to solve was not one we were familiar with when we started this project. We had little knowledge about the process of managing a contingent workforce and were unaware of regulatory compliance. During the first couple of weeks we had extensive contact with our clients to get more knowledge about the domain and the details of the problem. We started with high level descriptions of the components necessary to provide both parties, the organization and the flexworkers, the features to solve the problems. From that, we worked our way on to designing the data structure and the workflow of users in the application. This process continued through every sprint to make sure that the system we were building was on the right track.

7.3.2. Design

When we started the project, the client had a thorough problem description, but the details of how it should be solved were not yet thought out. At the start of the project we tried to combine our (limited) knowledge of building web applications and the clients knowledge about the problem to create a high level design of the system. This resulted in the modules described in section 7.1.1. Before we could design a system around these modules, we also had to figure out what types of users the application has, and what each user wants to see.

We put together an initial design during the first couple of meetings with the client, and used that as a basis to start developing from. After the first sprint however, it was already clear that this needed refinement. After that, we started a constant feedback process for the high-level design of the modules together with the client. This happened during our scrum meetings, where we would pitch the ideas we had, and then together with the client reformed these ideas until we were both satisfied with them.

The most important question we want to answer in this paragraph is: Does the application do what the client wants and does it solve the problem the client specified? Right now the answer to this is: partially. As we said earlier in this report, the goal of this project for our client was not to have a full-fledged, end-user ready product. The goal was to develop an application that could show the potential of such a system: like a proof-of-concept. With this in mind, 'partially' can be better explained. The system in its current form does show what problems it will fix. Before it can be used to actually solve these problems in a real-world environment, with real-world organizations and flexworkers as users, a lot more work must be put into it. In short: the application does what the client wants, but it does not yet solve the problem.

7.3.3. Process

From the start, we decided together with the client that we were going to use the scrum method during the project. Before we started, we had a meeting to set up a global planning for each sprint. This planning can be found in section 4.3.1. At the beginning of each sprint we divided the global goals for that sprint into smaller, more detailed stories and added an approximation of the time each story would cost. In the first two sprints this was relatively successful, even though our approximations were often too high or too low. The total approximation of what we were able to do in the sprint was close to what we approximated. In the following sprints this became harder. This was due to the fact that a significant amount of time had to be spend on refactoring (see section 7.3.4) and fixing bugs.

During the sprint, the board would become much fuller because of what we discovered while developing. This were not only bugs, but also smaller features with a higher priority than what was already on the board. This led multiple times to finishing a sprint with stories originally added unimplemented, and stories, bugs and small features added during the sprint finished instead. Because we developed this application in a team of two, this was not a major issue, but if we had a bigger team, this would not be very efficient. After this happened in two sprints, we decided that we would reduce the amount of work added to the board initially to leave room for unforeseen issues. After we started doing this, clearing a board during a sprint became attainable.

Even though we had a small team, we wanted to approach the project as professionally as possible. To achieve this we used Team Foundation Server for managing the backlog and the sprints, and git (together with Jenkins continuous integration), for version control and quality checks. At the start of the project we decided that we would only merge code with the master repository through pull requests, and only if that pull request passed the build check done by Jenkins. Most of the time we succeeded in this, but there were a couple of moments where a quick bug-fix was pushed directly to the master repository. Because just the two of us were developing, and since we were together most of the time, this was not a really big problem, but when working in a bigger team, this would definitely lead to complications

All in all, with a few minor exceptions, we think the process overall went okay. In the future, more effort is needed to reduce these exceptions even more, but, as a first large, real-world project, we think this project has definitely pushed us in the right way.

7.3.4. Implementation decisions

In this section we will reflect on two decisions we made during the implementation of the application. First our choice of frameworks, and secondly the fact that we decided to do a large refactor of the entire application, twice.

At the start of the project we decided to use two frameworks: React and Laravel. While we both had some experience with Laravel, React (and front-end developing in general) were quite new to us. The decision to go with React was based on multiple (technological) reasons,

as explained in chapter 4. Another, maybe even more important, reason for us to use it, was that we wanted to try and learn something new and different. From a learning point of view, this was a big success, and with the knowledge we have now the project is almost finished, we would do it exactly the same way. From a project perspective, it might not have been the best idea to start using a new, relatively young and very different (compared to other front-end frameworks) frameworks.

The first reason was that setting up the application structure and getting used to the framework and language took us a lot longer than if we had used a framework we had experience in. While this is no problem if time is not an issue, in a relatively short span of 10 weeks every day counts.

A bigger problem surfaced in our third week of implementing. We discovered that the application structure we decided to use at the start, had scaling problems. The more features we added, the more technical debt we found ourselves in. This was really demotivating, so something had to be done. After careful considerations we decided to completely refactor the front-end application. Using the knowledge we had gained during the designing and implementation of the first front-end application structure, and the problems it brought, we completely redesigned the structure. Refactoring to the new structure took us more than a day, but it was definitely worth it. Developing new features became easier, and much quicker.

In the late phases of the project, we did another refactor (although much less extreme than the first). This time we divided most of the components we used in to smaller, more decoupled components, which managed their own state. The reason for this reason was, that we felt that we had to change to much existing files to add functionality, which is never a good sign.

Even though both refactors were a success in the long run, it took us in total at least three days. Both would have been prevented if we had experience with the framework beforehand. All in all, the whole process gave us a lot of experience in the challenges of designing and developing an application from scratch.

7.3.5. What we learned

Apart from the whole experience: analyzing the problem, setting up requirements, creating an initial design for the structure of the application and the data and implementing such a system from scratch, we also learned some more specific things during the project, in this section we will elaborate on each of them.

Planning is hard Creating a sprint planning that has just the right amount of work planned is difficult, even with just two team members. Luckily, over the course of the project we became better at this. To improve this even more, we learned that it is useful to compare the initial time set for a story/feature/bug with the actual time spent. Doing this made us better at guessing the time required for stories.

Refactor early In the previous section we discussed how we refactored the application twice. One of the reasons these refactors cost so much time, was that we put off refactoring for a while. If we incrementally did that during developing, the impact would have been less, and in the long run it would have cost us less time. After the second refactor we decided to follow the boy-scout rule: leave the place cleaner than you found it. We applied this rule to our programming: whenever you are editing a file, and see something that could have been done better or cleaner, do it immediately. This includes writing comments for functions without them, splitting code in different files and renaming obscure variables or method names.

Implement something only when required by the application This is something we also learned during refactoring: Both times, we discovered a lot of unused code: mostly methods that were written at some point with the thought of: we will probably require that sometime in the future. What we learned is that these methods are almost always *not* used in the future. They are just forgotten. This taught us to only write code exactly when you need it and never assume that *someone might need this method at some point.*

A client's view on progress differs from developers' One thing which, as students, was new for us, was having a client without programming experience. As a computer science student almost everyone around you is aware of the process of developing an application like this. Especially the fact that progress is made even though it is not always directly visible. Often, for real-world clients, this is not the case and clear communication of this fact, and the progress in general, is very important. This is important for both parties, since we, as developers, sometimes had the tendency to spend less time on how something looks and feels, and more on the internal logic. During the meetings with the client however, more valuable feedback would be given when things were more visual. From this we learned to implement visual representations of features we were going to add as quickly as possible. This made the feedback loop quicker, and saved us time in the long run. Not only that, creating a basic visual implementation before fully implementing all logic necessary for a feature also shows potential flaws in the design early.

Don't use a framework in which both developers have zero experience for a production application with a deadline As already explained in the previous section, this was in hindsight not the best decision, and although it turned out (more than) fine in our situation, it is not something we would likely do again. Learning a new framework is much easier in a 'sandbox' environment, where you can just take your time to build stuff, break stuff, and build it again to figure out what happens and what the best way is to do things. Using it directly on a system as complex as this, with time pressure, made the start of our project unnecessary difficult.

7.3.6. Conclusion

In the end, we are very satisfied with what we achieved in this short time. We had some hiccups along the way, but that happens in every project. Finding solutions to, and overcoming these problems is what made this a very valuable experience for us. For us both, this is the first large, complex, real-world system we had to design and build from scratch and we learned a lot from it.

8

Conclusion & Recommendations

During the last couple of months we have gained insight in the problems that people working both as- and with contingent workers run into. We have developed a first version of a platform that could potentially solve many of the problems, but it still requires a lot of work before it can be used at a large scale. In this chapter we discuss our recommendations for future work on the FleXentral platform.

8.1. Continuous improvement cycle

There are four main aspects to the recommendations we have. We have summarized these aspects in a continuous improvement cycle. Four aspects that form the basis of our application are the matching module, compliance module, the processing of feedback based on user testing and the different non-functional requirements. We believe that these four aspects will never be finished and can always be improved. We have visualized this process as a cycle, as seen in figure 8.1. The idea of the cycle is that in the process of further developing the platform, each of the different aspects should always be worked on in order to create and keep a high quality product: As the Dutch always like to say: 'Standing still equals going backwards'. In this section we will discuss each of these aspects and explain why and how they could be further improved.

8.1.1. Matching

The implementation of our matching algorithm is still very basic. It currently uses the set of competences set by a user and a project manager to generate a score. This works and returns a good representation of whether of not someone is capable of finishing the job. However, the matching algorithm does not yet take in mind personal preferences, availability, location

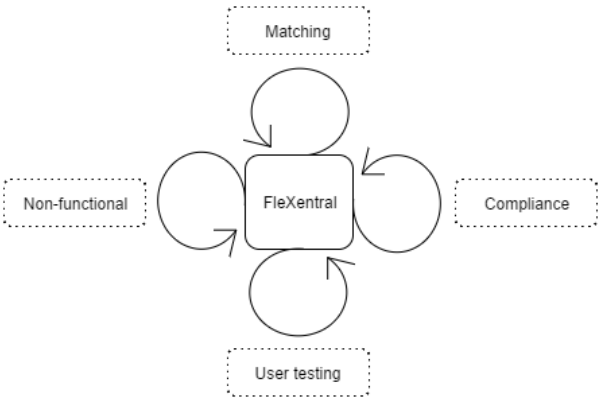


Figure 8.1: Visualization of the FleXentral improvement cycle

etc. One of the problems in developing a system like this is the cold-start problem: We did not have any real-world data to test the algorithm and to see what works and what didn't.

We recommend that FleXentral starts testing the product with possible customers as soon as possible, so that real world data can be generated. When this data is available, structured ways of improving and testing the algorithm can be devised. At some point, with enough data, the algorithm could even become self-improving and/or deep-learning to better predict matches between jobs and flexworkers.

8.1.2. Compliance

Because compliance is such a broad concept, it is very difficult to implement a truly functional module that covers all of the different aspects in such a limited time. We have implemented a basic module in which the most well-known rules on compliance for contingent workers can be checked. However, during pilot versions and while attracting new customers, it is inevitable that this module will be extended in the future. The final goal is to have a state of the art compliance module which covers everything. Creating this requires time and feedback from real-world users who have real-world desires and walk into real-world problems. To improve a system like this continually, user tests have to be performed.

8.1.3. User testing

Constant feedback from users is crucial for online platforms like FleXentral. During development, input from both flexworkers as well as organization users should be highly valuated in order to create a quality product. We would recommend that a structured feedback cycle is used in this process. A project like this should be continuously improving and this only works if there are ways to measure the perceived quality of the product.

8.1.4. Non-functional

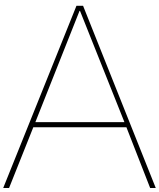
It is important to keep the non-functional requirements in mind while implementing new features to the platform. During this project we have spent a lot of time and effort working on technological independence, robustness, scalability and maintainability. In our eyes, these aspects will form the basis where FleXentral can be built upon. While there is always room for improvement, we are definitely on the right track. If every future developer getting his hands on FleXentral leaves the code just a bit cleaner, better or faster than the way he found it, this will only improve over time.

8.2. Final conclusion

The goal of this project was to create a proof of concept version of the online Contingent Workforce Management solution called FleXentral. During the first two weeks we have set the requirements for 7 different modules and multiple non-functional requirements. In the process of developing the system we ran into different problems that led to certain implementations taking us a lot longer than initially expected. In addition to this, the use of a framework we were not familiar with made the start of the project more difficult and in general led to a slower process. During the project however, our knowledge about the systems we used has increased a lot, and in the end we both learned a lot by doing it this way.

All in all we believe that it is safe to say that we delivered a high quality product to our client. Although it is far from ready as a first production version, a strong foundation has been built and the product has come to a point where it already has value to show to possible customers in the form of a demo. As our client would say, the house was built, now we need to start decorating the rooms.

This project has been our first experience working directly and full-time with- and for a client to deliver an end product from start to finish. The experience was definitely worth the effort we put into this. We learned a lot of valuable knowledge, and are grateful for the fun and challenging opportunity we had at FleXentral. While there is still a lot of work to do before our application is production ready, we believe that in the near future, FleXentral has the potential to become *the* standard for Contingent Workforce Management.



Research report

FlexCherry: Research Paper

Arjo van Ramshorst, Joost Rothweiler

02-05-2016

Contents

Contingent workforce management

A contingent workforce is a group of workers who work for a company on a non-permanent basis. This group is also known as independent professionals, contract workers, independent contractors or flexworkers.
Before, the majority of a business' work was completed by traditional full-time equivalents (FTEs) with the contingent workforce supporting only a fraction of the business projects. Today, however, we see more and more independent contractors supporting and completing an increasing amount of larger and to the company more critical projects.
A company's strategic approach towards the managing of the contingent workforce is called contingent workforce management (CWM) and it is used by companies mainly to decrease both the costs in management of the contingent workforce as well as the risk in hiring them.
The State of Contingent Workforce Management research study finds that nearly 35% of today's total workforce is made up out of non-employee workers and that the impact of this group of workers can be felt across business of all sizes, regions and industries.
It also states that in 2015, the contingent workforce has reached its tipping point, no longer remaining a minor spend category but changing the business environment in such a way that its influence will be irreversible. [1]

Challenges

Due to the speed at which CWM has grown and the point that it has reached today, it has become extremely difficult to design good CWM programs. Challenges that play an important role in how executives structure their CWM programs include visibility and intelligence, compliance risks, reduction of costs and the need to improve the way in which talent is engaged. Each of these challenges will be further discussed in the section.

Compliance

Compliance of independent contractors is becoming a more important thread and less ignored as the number of contingent workers is growing. Contingent workers must adhere to all applicable compliance rules set by governments. Hiring contingent workers that are not compliant leads to great risks regarding unwanted costs through for example federal audits resulting in reclassification.
When signing a contract between a company and a contingent worker, the company does not have to make contributions to social security, unemployment and other compensations, saving itself expenses of administrative tasks and protecting itself from the laws that contribute to employer's social security.
In 1989, a GAO study showed that 38% of the employers in the U.S. were misclassified as contingent workers, causing the government to lose billions of dollars in taxes as a result of underpayment in taxes.
In the 1980's, Microsoft employed around 1000 contingent workers that were later, after an audit conducted by the Internal Revenue Services (IRS), classified as normal employees. Microsoft decided to comply with the IRS, paid the necessary taxes and hired some of the workers as employees. However, few of the contingent workers demanded the benefits they could have received for the period they were classified as independent contractors. This resulted in Microsoft settling the suit in 2007 for an amount of $97 million.
As we learn from this example, non-compliance and misclassification could be an expensive addition to hiring contingent workers, making compliance one of the most important factors to influence CWM programs.

Visibility and intelligence

Another challenge is visibility and intelligence and mainly the lack of it. Understanding the quality of the labor done by the contingent workforce as well as insight into costs and risks through compliance can lead to greater progress in business projects. Lack of visibility is a problem for different stakeholders involved including procurement, HR and finance.

Reducing costs

What has always been an important factor in workforce management is the reduction of costs. In order to work, CWM programs in the future must ensure the reduction of costs and improve savings across the entire contingent workforce.

Talent engagement

In the future, talent should be engaged real-time and on-demand from sources in which project requirements and the necessary skill are matched with the worker's skill set. This will be an enormous challenge in CWM programs.

Matching jobs to flexworkers

One of the main research goals for FleXentral is the matching of projects to contingent workers. Matching in this case is a specific form of recommendation, namely, recommending contingent workers to projects, and projects to workers. In this section we will elaborate on why matching is necessary for this application, what the difference is between the matching problem and the recommendation problem, and we research how this problem is tackled in existing literature.

Goal of the matching

The main purpose of FleXentral is removing the middle man in transactions between organizations and the contingent workforce. This also includes head-hunters or external recruiters. In addition to this it should also reduce the number of unsuccessful matches (for either the employer and the employee). Automated matching, or recommending of suitable projects/workers, can save a lot of time and resources for both the company and the worker.

Difference between recommendation and matching

Most recommendation system research has been done in one-way recommendation, for example: movie/book recommendation or item recommendation in online shops. Since movies, books and items don't have any preference for who watches, reads or buys them, these systems only have to use the users' preferences to recommend something.
Matching people to jobs, or recruiters to applicants is a two way recommendation. Even though a recruiter thinks someone is a very good match, the applicant can think otherwise. Both should be fine with the matching, otherwise it will not be a successful one. This can be translated into two requirements.
First, projects should be matched to contingent workers that have the skills and abilities that are required for a certain project. Compliance is also important for this requirement.
Secondly, contingent workers should be matched to projects based on their own preferences. This includes for example location of work, work hours, preference for a specific company or project type and duration of the project.
If we want an efficient and useful matching algorithm for FleXentral, we have to find a solution that satisfies both these requirements. In the next two sections we discuss two methods from different research papers to match applicants to jobs. Since these two papers are focused on normal jobs in contrast to the independent contractors as in our application, we will omit some features from the papers and add some of our own.

Skill, ability and requirements matching

The first way to match people to jobs is matching requirements to abilities[2]. This is based on giving a certain score to every person-job matching possible. In literature, this is called the person-job fitness (P-J

fitness). In P-J fitness, there is a distinction between two different ways of calculating the score.
First, there is the needs-supplies fitness. This is based on the needs and preferences of an individual, and the environmental supplies. For example: what is the location of a project, how much does it pay, what perks does the job have, how much do I like this company, how much do I like the job specification? This part is mostly focused on the employer. The more this matches the employers' preferences, the higher the needs-supplies fitness score.
Secondly, there is the demands-abilities fitness. This is mostly from the viewpoint of the recruiter: Does a certain employer have the abilities necessary for this project? The better a employers' abilities match a projects' requirements, the higher the demands-abilities fitness score.
In order to create a strong match, both these scores need to be maximized. This form of two-way matching is also called 'Bilateral matching'.

Communication-based matching

A second approach to matching people to jobs is communication-based matching[3]. This approach uses a popularity score based on the communication between recruiter and applicant. Since the negotiation and communication will happen on the FleXentral platform, this is also a viable option to find suitable matches for projects and contingent workers. This form of matching sees all forms of communication as a positive or negative action. Positive actions improve the popularity score, and negative actions decrease the popularity score. An example of a positive action is hiring someone, but also checking out someones profile, or sending them an offer for a job. A negative action is for example declining a job offer (for the project) or declining an application (for the worker).
Using the information from both sides, a popularity score can be calculated, much like the Pagerank recommendation algorithm by Google for finding relevant websites. In this case, positive actions are like hyperlinks from website A to website B. The higher score website A has, the more impact on score a referral from A to B has. Similarly, a positive action from a high scoring company A to a person B, has more impact on the popularity score of person B than from a company with a low popularity score.
Also, after a completed job, contingent workers and companies can give each other an evaluation, which can then be used as an initial popularity value. Just as with Pagerank, there should be a constant amount of 'popularity' at any time. An evaluation function is necessary to compute the popularity score of all users based on their initial scores and their recent actions.

Data required for matching efficiently

Requirements matching requires data from flexworkers and projects to work. The following lists contain the data we will need to successfully match jobs to flexworkers. First the data we require from flexworkers:

- Demographic data (e.g. age, gender)
- Educational data
- Language skills
- IT skills
- Job experience
- Compliance information
- Availability (e.g. location, preferred work-hours, duration)
- Fee (normalized to hourly fee)

Project managers and recruiters use the same categories of data to set up requirements. They can also add weights to requirements so that certain requirements can be more important than other requirements. Also, for example for salary, they can add an indication of what they are willing to pay for a certain job.

Storing the required data

Storing this data in a way that a matching algorithm can handle is difficult, since the number of skills is too large to define in advance for all sorts of jobs, adding skills dynamically is a must. Unfortunately multiple people could describe the same skill in different ways, (i.e. 'javascript programmer' vs 'js programmer'). In order to handle this problem we propose two methods.
The first method to tackle this problem is a system based on tags. You add a skill, and while typing, similar tags are proposed with a number of how many other people have used that tag to describe a similar skill. This should motivate people to use the same tag as other users, since recruiters are also most likely to use the most-used tag to describe a requirement for a job.
Secondly, FleXentral admins should be able to link tags, making them in essence the same tag, so searching for or requiring a specific tag should find all tags that essentially mean the same thing.
Also, since some skills are similar but can be used in a more broad sense than others, we can take a tree-based ordering of tags. If a recruiter is looking for someone with 'front-end developing' skills, and a contingent worker has 'javascript' listed as skill, it should still be seen as (somewhat) of a match in skills. The tags mentioned above could have a 'parent' tag, which links to a tag that is less specific than it. Close tags in this tree could also get a (smaller) similarity score. The method proposed by Mine used a similar ranking of skills which proved successful[3].

Algorithm

In order to develop an algorithm for this problem, we are going to develop a combination of the two methods proposed in the previous section. Using the requirements matching to give a certain rating for how well a user and a project match. This is used to filter the entire list of contingent workers or projects down to a list with only (at least partially) fitting results. To order this list, we're going to implement an evaluation function to calculate the popularity score of users and projects in (semi) real-time.

Finding the perfect match

When a recruiter is interested in a listed flexworker, or the other way around, they can send a proposition to the other party. Both parties get in a 'negotiation', where they can reach consensus over things like price, duration and work-hours. After they both agree, a match is made. All these actions, if successful, are positive. If the negotiations don't work out, the action is negative. Either way, the popularity scores of both the user and the project are adjusted appropriately for a next project. After a project, both users are asked to enter an evaluation score for the other party. This can be used to adjust the popularity score, but also for future ways of improving the algorithm by adding similarity scores for example.

Planning

As both compliance as well as searching and matching people to jobs are the most challenging and important aspects of the FleXentral platform, we have chosen to spend most of our time researching and implementing those two parts. In the last two weeks we have spend time researching, setting up the framework and structure for front-end and back-end for the FlexCherry project and working on the database design. In the coming weeks we plan spending our time as followed:

Week 3 Allowing for contingent workers to create a profile as well as project managers to post jobs that contain information that can later be matched. After this week, the same information should also be ready to be used to check basic compliance status of contingent workers.

Week 4 Implementing the negotiation module allowing for contingent workers to communicate and close deals with companies.

Week 5 Automatic search and match that allows workers to find matching jobs on registration and allows for project managers to find matching talent after creating a new project.

Week 6 Implementing transaction effort hours and achievement registration for contingent workers to log their work and send invoices to companies.

Week 7 Reporting for both company users as well as contingent workers to get insight in their works.

Week 8 + 9 Further improving search and match module as well as the compliance modules in order to improve automatic matching.

Week 10 Wrapping up.

Appendix 1: Requirements

Based on meetings and research from the last couple of weeks we have come to the following requirements together with our client.

Functional requirements

The functional requirements for FlexCherry consist of 7 modules that should be implemented.

Registration module

Both companies and flex workers should be able to register themselves to the platform. Organizations are required to give basic information as well as information concerning possible different entities across different countries. Also they should be able to set credentials for employees that fulfill different roles under which Line/Project manager, Controller and CFO.
Flex workers should be able to sign up requiring only very basic information like name and e-mail address but are encouraged to connect through linked in. After the sign up process is complete, more information is asked for in order to get the status of compliant (see compliance module).

Compliance module

An important module in the FlexCherry project is the compliance module, stating the level of compliance of a flex worker regarding a country, organization and project. The compliance check allows both organizations as well as flex workers to check the amount of risk an organization takes when hiring a certain flex worker for a specific job in a specific country. This information is based on registration numbers, certificates and diploma's provided by the flex worker.

Marketplace module

The information provided by both flex workers and companies meet in the marketplace module. Here companies can search for talent and flex workers can look for matching jobs for which they are compliant.

Search & Match

Apart from the marketplace module, FlexCherry will contain a module that automatically searches and matches jobs to flex workers. This allows for the organization to, whenever a new job is submitted, directly get a shortlist of flex workers that are suited for the job.

Negotiation

The negotiation module consists of a basic chat functionality where employers and flex workers can negotiate on future jobs. Base offering, planning and terms and conditions provided by the organization should meet base pay, availability and terms and conditions provided by the flex worker. On agreement, the employer should be able to set up a contract which will then be offered to the flex worker together with a sign off option.

FleXentral transaction

Flex workers should be able to log their activity in the FleXentral transaction module. In this module the effort is passed as input by the flex worker. This effort is then approved by the line/project manager at the organization and submitted to controller (financial approver) against budget. An invoice should be generated and available to both the organization as well as the flex worker. Both FleXentral as well as the flex worker should get notification of payment by the organization.

Reporting

Reporting shall be divided into different dashboards, available to users based on their roles at a company or as a flex worker. While different organizational roles require insight into effort, budget, planning progress, compliance status and actuals, flex workers should get a clear insight in accounts receivable, pending approvals and P&L.

Non-functional requirements

Apart from the functional requirements, FlexCherry should also meet equally important non-functional requirements.

Usability

Because users are going to have to work a lot with the platform on both the computer as well as on mobile phones, usability is an important requirement for this first version. The platform should perform fast, be easy to use and work well on every device. Preferably the platform would be built in such a way that pages would never have to refresh and the website would feel like a native app when using on mobile phone.

Manageability

The architecture of FlexCherry must have the ability to be managed in order to ensure continued health with respect to scalability, reliability, availability, performance and security. It should also allow for dynamic system configuration.

Data integrity

Access to features or data should be restricted to certain users, protecting the privacy of data entered into the platform.

Security

Data should be kept private and well secured from parties both on and off the platform. Since a lot of private and valuable data will be stored, security is an important aspect of the implementation.

Scalability

For this first version scalability is not yet the most important non-functional requirement since it is going to be a proof of concept. We do, however, need to keep scalability in mind for future versions.

Capacity

Just like scalability, capacity is a non-functional requirement that should be kept in mind for future versions but is not yet important for this first version.

Appendix 2: Tools and libraries

In this section we will explain what tools we are using for the development of the FleXentral app and why. The main design choice we think is necessary for this application, is that the front-end and back-end need to be completely decoupled. Especially because at this moment, it is unsure how the back-end will be implemented in the future. Different paths to this achieve, maybe for every company a different server that hosts his own data, or one server provided by FleXentral for every client of FleXentral? Right now there are no definite answers to this, and since this are only questions relevant to the back-end of the application, we should create an application in which the front-end has no need for any knowledge of the back-end, except for the api-routes available.

Back-end

Even though, as explained above, the back-end likely won't be the final implementation, for testing and developing of a working application, it is necessary that we have something to act as back-end. This should be an api that the front-end can use, and a way to store, retrieve and update data.
Since almost every programming language can be used to create a server to store data and create an api entry point to retrieve it, the choices for this are endless. Every language has its pro's and con's, so there is no one that is necessarily 'better'. We based our choice mostly on what language we are most familiar with, so we can develop the back-end (which is not the most important part of this application challenge) as quickly and efficiently as possible, without having to worry as much about scalability. The back-end should only have to handle data and authentication.
We both have the most experience using PHP, so it is logical for us to use PHP as the language for the server. Especially, because since PHP 7 is released, it is for most cases just as fast as most competing languages like Java and C#. PHP is also used by a large chunk of the internet, so there is a lot of support online, and there are a lot of tested and tried frameworks for it.
To reduce the writing of boilerplate code, and increase development efficiency we decided to use a framework. The framework we ended up choosing was Laravel.

Laravel (PHP)

Laravel is a relatively new open-source PHP framework. It is based around the MVC (Model-View-Controller) architecture. Since the release of Laravel 5, it is also very stable, has long term support and backwards compatibility for newer versions. Documentation is well-written and extensive, and there is a lot of support from other users. It is used in several big and small applications, and it is developed actively.

Besides that, we both have used it with success in previous applications. We don't have to put much effort in learning the framework because of this.
Another useful feature of Laravel is its use of migrations to manage the database state. Since we are working on this application with two persons, both in different local environments, it should be possible to work with a local database that always matches your current environment, even though we are working on different modules with different database requirements. Laravel's database migrations make this a breeze, since the state of the database is managed by the version control. This means that the database is always at the same level as the code base.
Finally, Laravel has an object relational mapping system called Eloquent. Eloquent makes it very easy to define relationships between different models in the database, and make it simple to retrieve related models. Since the application we are building requires a lot of many-to-many relations, this reduces the amount of sql we have to write, leading to quicker prototyping.

Front-end

The front-end is the most important part of this application. As stated in the requirements, ease of use is a key value in this project. Static websites are outdated and feel 'slow' nowadays. The client wants the application to feel like a modern application, not like a webpage. To achieve this, Javascript is necessary.
The application will become very large, with a lot of different modules that all have to be updated in real time. Using JQuery or a similar DOM manipulating framework is tedious to use when the application grows. First of all, the larger an application gets, the slower DOM manipulation gets. Secondly, as a developer, using JQuery and DOM manipulation to handle the visual aspect of the application is extremely difficult to maintain when new features are added. New features are likely to break older code. It is also very difficult to properly test.
Finally, when using DOM manipulation to handle data input, there is never a single source of truth for data values. This can cause synchronization issues, that can lead to very hard-to-identify bugs. Especially with a large application this can cause major development delays.
We decided to use a framework that handles these three points better. After researching the possibilities we decided to go with Facebook.com's React, in combination with Redux. This combination has a solution for these three problems.

React

React is mostly a solution to the 'view' part of MVC. It has little functionality by itself, other than rendering views. It uses 'components', that are nested, much like the HTML DOM-structure. The only difference is that you can create custom "DOM-elements" that automatically get filled with data.
What sets react apart from other solutions is that react has one-way databinding. Unlike a framework like angular, where updating the value in the view automatically updates the value in the storage and the other way around, which is called two-way databinding. Both one-way and two-way databinding have pros and cons, but the disadvantages two-way databinding brings, (like synchronization issues and no single source of truth) is in large, complicated applications more of a disadvantage than an advantage.
With two-way databinding, when you enter something in a form, the model gets updated. If something else updates the model, the form gets updated. When this happens at the same time, strange behavior can occur, which can be hard to debug. React handles this as follows:
When you enter something in a form in React, an event is fired that updates the model with what you entered, and then finally the view updates with the value that is stored in the model. The model is always right about the value, this reduces the amount of bugs, but increases the boilerplate code needed for forms and other databindings in the view. To reduce that, we used another framework in conjunction with React: Redux.

Redux

Redux is not really a framework on its own, but rather a way to store data. An application that uses Redux has one storage place for data. This is called a 'store'. Every application has exactly one 'store'. The store is pretty much one big json object containing all data the application uses in a treelike key =¿ value format. Since this store is the only place where Redux stores data, and the only place where it can be updated, it is the 'single source of truth', as specified before.
What makes Redux a great solution against synchronization faults, is the fact that it has only one way to update the data in the store. Updating the store is done by firing 'actions' after an event happens. An action has a 'type' that describes what happens, and one or more parameters.
An action is handled by a reducer. A reducer gets the current state, and the action (including parameters) as input, and returns a new state. Important to note is that a reducer should always return the same result with the same input. So no random generators, time based functions or AJAX requests are allowed in reducers.
Actions are always fired in sequence, so there are no race conditions, and no synchronization issues. Asynchronous calls are handled by middleware, which fires an action before the call, when it's succesful and when it failed. Because actions always return the same value, it is easy to unit test every action that can happen in the application.

Other development tools

We use multiple languages and frameworks, and also multiple small helper packages, which simplify stuff like handling date and time, form generation, authentication and more. Keeping track of everything gets harder the more external tools we use. To make managing this easier we use multiple tools, mainly Composer for PHP and Node Package Manager (npm) for javascript. Also, we use Gulp and Browserify for building and minifying javascript.

Composer & Node Package Manager

Composer and NPM are tools that keep track of all dependencies of a project. They can automatically update all tools to the latest version, and they make deploying a project as easy as running one command to install all dependencies.

Gulp & Browserify

We use Gulp to 'package' the javascript code, including the dependencies we use into one 'minified' javascript file that is stored in the public folder of our application.
Since React and Redux use a lot of ECMAScript 6 & 7 features, we decided to make use of them too. Since not all modern browsers understand ES6 and ES7 features, we use Browserify to compile it to basic javascript that every browser understands.

References

[1] Christopher J. Dwyer. The state of contingent workforce management report. oct 2015.

[2] Jochen Malinowski, Tobias Keim, Oliver Wendt, and Tim Weitzel. Matching people and jobs: A bilateral recommendation approach. In *System Sciences, 2006. HICSS'06. Proceedings of the 39th Annual Hawaii International Conference on*, volume 6, pages 137c–137c. IEEE, 2006.

[3] Tsunenori Mine, Tomoyuki Kakuta, and Atsushi Ono. Reciprocal recommendation for job matching with bidirectional feedback. In *Advanced Applied Informatics (IIAIAAI), 2013 IIAI International Conference on*, pages 39–44. IEEE, 2013.

B

Dashboard design

In this chapter we give a brief impression of the dashboard design and components implemented. We have included screenshots of entire dashboards as well as smaller components. Each figure includes a brief description of what can be seen in the image.

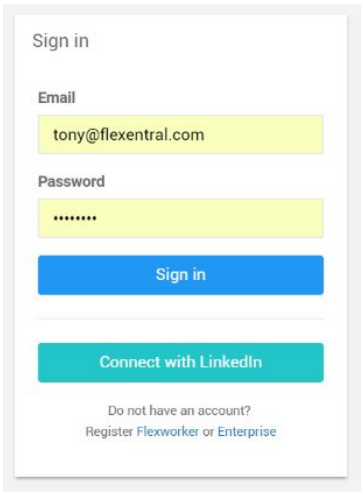


Figure B.1: A screenshot of the login form, used for both flexworkers as well as organization users. By clicking the 'connect to linkedin' button, both users can connect existing accounts to their linkedin accounts or create a flexworker account using linkedin.

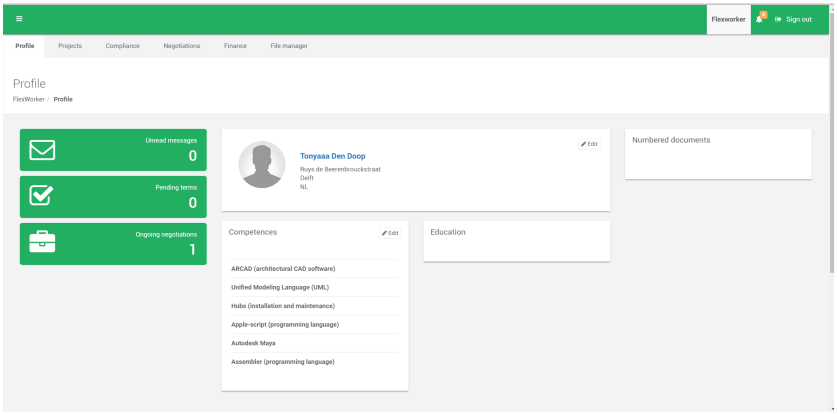


Figure B.2: A screenshot of the flexworker dashboard welcome page. It contains different personal information cards that can be edited using in-page forms.

Figure B.3: The widgets which are real-time and tell the user in a quick overview what is new and what requires attention.

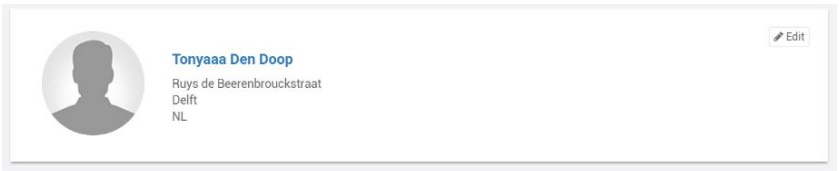


Figure B.4: The user profile card, which can be edited using the 'edit' button in the upper right corner.

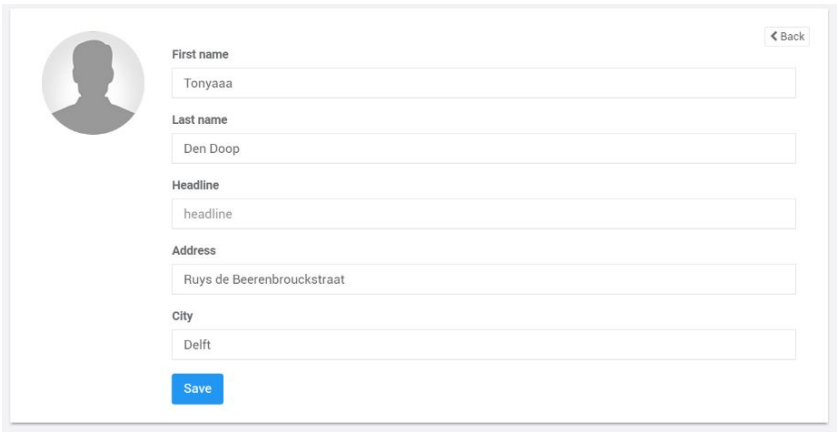


Figure B.5: The in-page edit profile form which is displayed and saved without the need to refresh the web page.

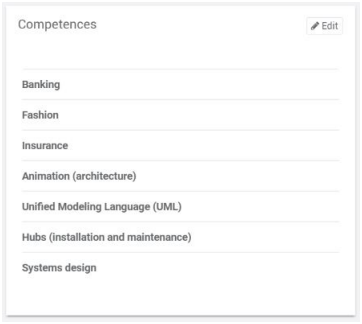


Figure B.6: The user competences card, which can be edited using the 'edit' button in the upper right corner.

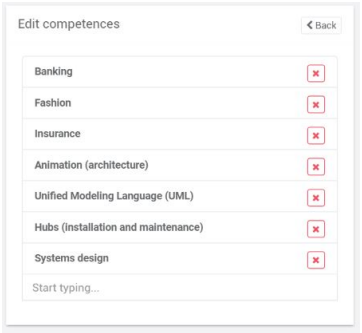


Figure B.7: The in-page edit competences form which is displayed and saved without the need to refresh the web page.

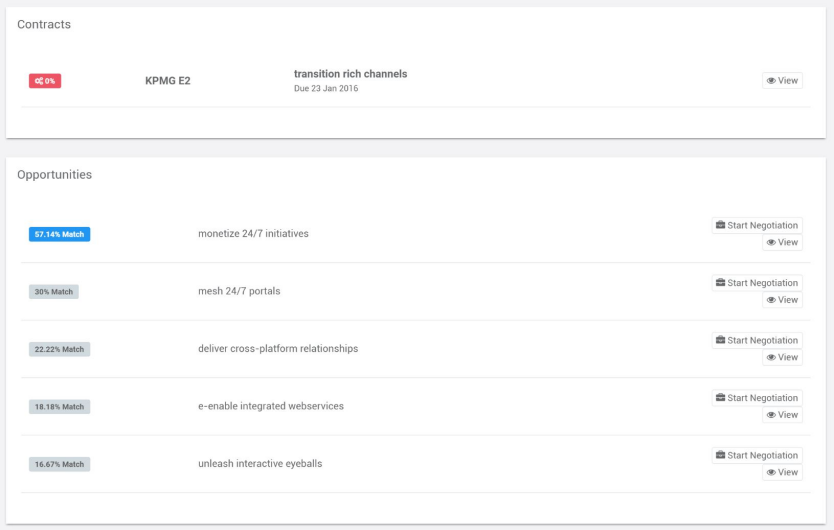


Figure B.8: A table containing rows with current contracts signed by the flexworkers and another table with opportunities based on matching scores.

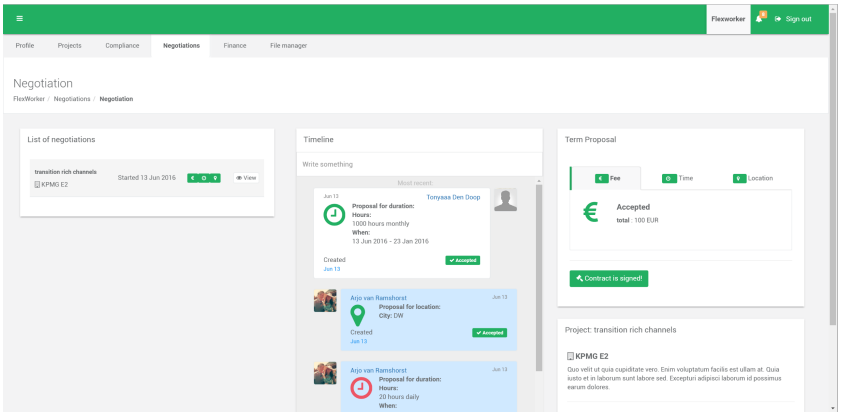


Figure B.9: An overview of the flexworker to project-manager negotiation page.

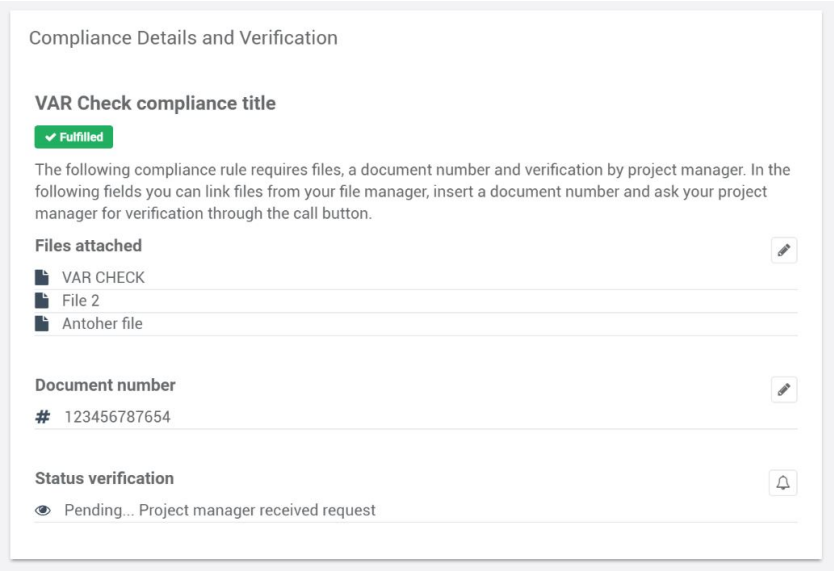


Figure B.10: Compliance rule overview card. A flexworker is linked to compliance rules through countries, organizations and projects. Each rule can be managed and fulfilled using this card by linking document identifiers and files.

C

SIG Feedback Refactors

Before our first SIG feedback, our back-end code still contained some duplicate code that had to be fixed. The reason this duplicate code was existed in the first place was because multiple requests in our negotiations module required the same type of authorization. While testing these authorization rules we have simply placed the same rules in different files but this of course does not contribute to a maintainable codebase. Therefore we have extracted the authorization method into a single request file which is now extended by the different request files that used to contain the duplicate code. Figure C.1, C.2, C.3 and C.4 show the 'before' with the duplicate code in each of the 'authorize' methods. Figure C.5 shows the new class 'TermRequest' which is now extended by each of the previous classes. 'TermRequest' is now the only class that contains this piece of logic in the authorization method.

27 app/Http/Requests/NegotiationLocationTermRequest.php View

```
@@ -2,34 +2,11 @@

 namespace App\Http\Requests;

-use App\Http\Requests\Request;
-use Illuminate\Support\Facades\Auth;
-use App\Negotiation;
+use App\Http\Requests\TermRequest;

-class NegotiationLocationTermRequest extends Request
+class NegotiationLocationTermRequest extends TermRequest
 {
        /**
-        * Determine if the user is authorized to make this request.
-        *
-        * @return bool
-        */
-       public function authorize()
-       {
-               if (!$negotiation = Negotiation::find($this->negotiation_id)) {
-                       return false;
-               }
-
-               $user = Auth::user();
-               if ($user->negotiations()->find($this->negotiation_id)) {
-                       return true;
-               }
-               if ($user->organizationUser()->first() && $project_managers = $negotiation->project()->first()->organizationUsers())
-                       return $project_managers->find($user->organizationUser()->first()->id) ? true : false;
-               }
-               return false;
-       }
-
-       /**
         * Get the validation rules that apply to the request.
         *
         * @return array
```

Figure C.1: Difference refactored 'NegotiationLocationTermRequest'

27 app/Http/Requests/NegotiationMessageRequest.php View

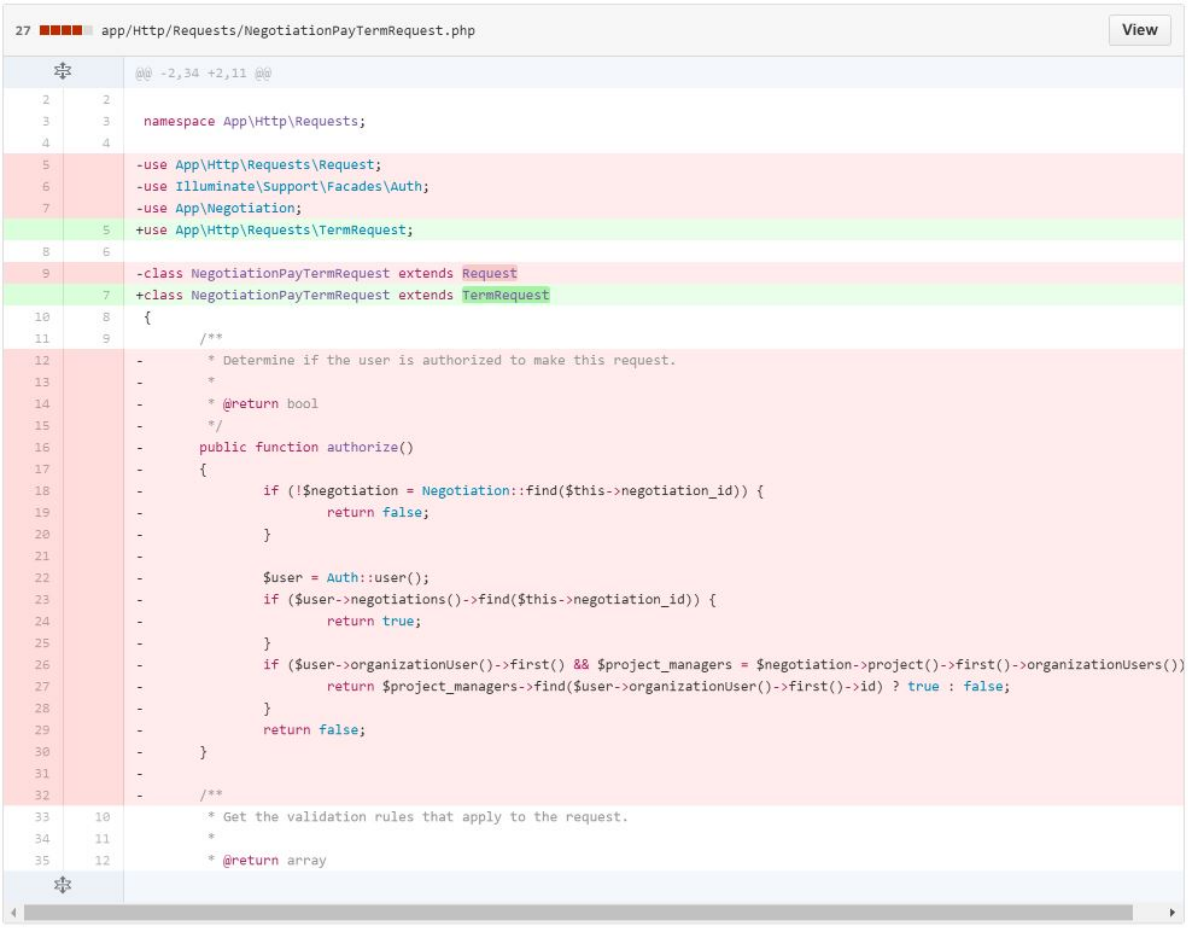
```
@@ -2,34 +2,11 @@

 namespace App\Http\Requests;

-use App\Http\Requests\Request;
-use Illuminate\Support\Facades\Auth;
-use App\Negotiation;
+use App\Http\Requests\TermRequest;

-class NegotiationMessageRequest extends Request
+class NegotiationMessageRequest extends TermRequest
 {
        /**
-        * Determine if the user is authorized to make this request.
-        *
-        * @return bool
-        */
-       public function authorize()
-       {
-               if (!$negotiation = Negotiation::find($this->negotiation_id)) {
-                       return false;
-               }
-
-               $user = Auth::user();
-               if ($user->negotiations()->find($this->negotiation_id)) {
-                       return true;
-               }
-               if ($user->organizationUser()->first() && $project_managers = $negotiation->project()->first()->organizationUsers())
-                       return $project_managers->find($user->organizationUser()->first()->id) ? true : false;
-               }
-               return false;
-       }
-
-       /**
         * Get the validation rules that apply to the request.
         *
         * @return array
```

Figure C.2: Difference refactored ’NegotiationMessageTermRequest’



27 app/Http/Requests/NegotiationPayTermRequest.php View

```
@@ -2,34 +2,11 @@

 namespace App\Http\Requests;

-use App\Http\Requests\Request;
-use Illuminate\Support\Facades\Auth;
-use App\Negotiation;
+use App\Http\Requests\TermRequest;

-class NegotiationPayTermRequest extends Request
+class NegotiationPayTermRequest extends TermRequest
 {
        /**
-        * Determine if the user is authorized to make this request.
-        *
-        * @return bool
-        */
-       public function authorize()
-       {
-               if (!$negotiation = Negotiation::find($this->negotiation_id)) {
-                       return false;
-               }
-
-               $user = Auth::user();
-               if ($user->negotiations()->find($this->negotiation_id)) {
-                       return true;
-               }
-               if ($user->organizationUser()->first() && $project_managers = $negotiation->project()->first()->organizationUsers())
-                       return $project_managers->find($user->organizationUser()->first()->id) ? true : false;
-               }
-               return false;
-       }
-
-       /**
         * Get the validation rules that apply to the request.
         *
         * @return array
```

Figure C.3: Difference refactored 'NegotiationPayTermRequest'

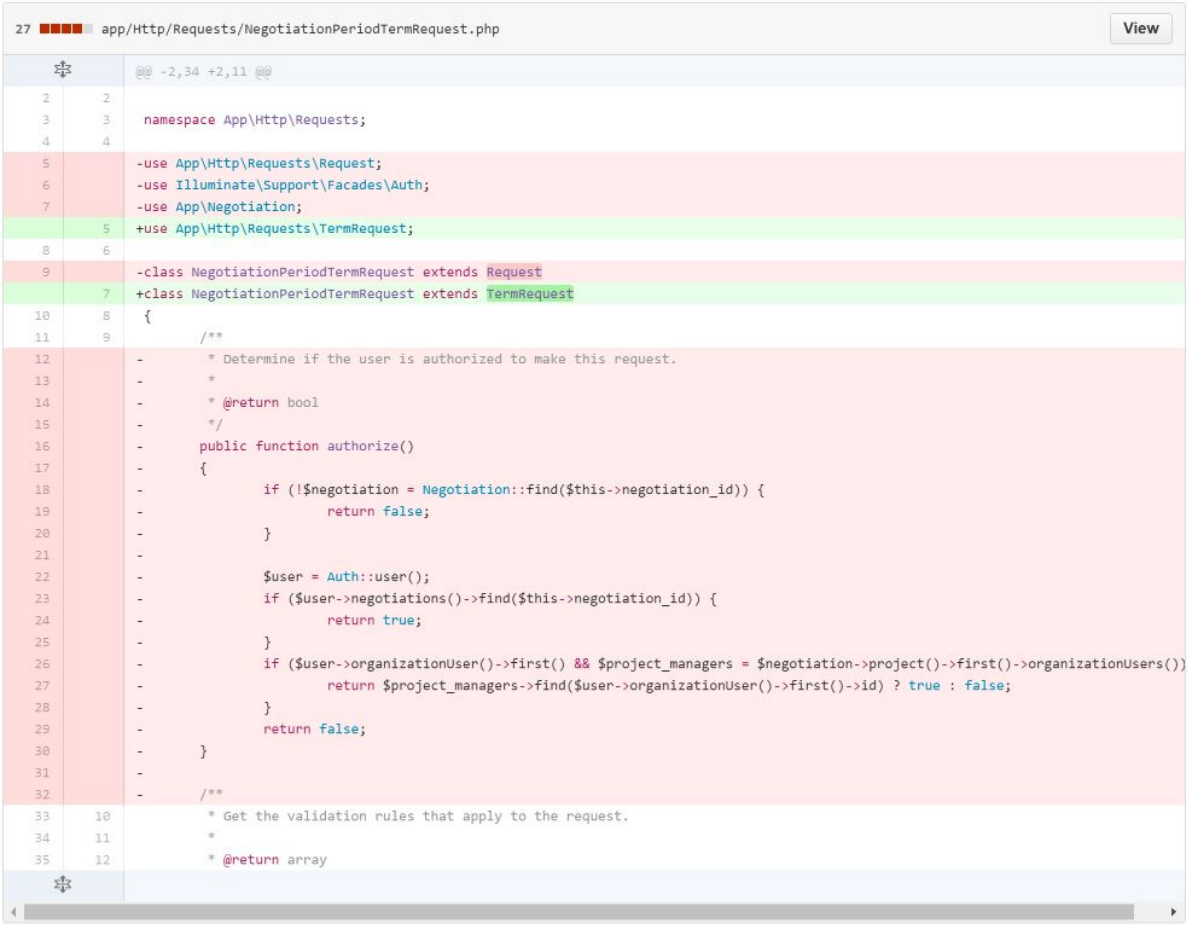


Figure C.4: Difference refactored ’NegotiationPeriodTermRequest’

31 app/Http/Requests/TermRequest.php View

```
@@ -0,0 +1,31 @@
+<?php
+
+namespace App\Http\Requests;
+
+use App\Http\Requests\Request;
+use Illuminate\Support\Facades\Auth;
+use App\Negotiation;
+
+class TermRequest extends Request
+{
+       /**
+        * Determine if the user is authorized to make this request.
+        *
+        * @return bool
+        */
+       public function authorize()
+       {
+               if (!$negotiation = Negotiation::find($this->negotiation_id)) {
+                       return false;
+               }
+
+               $user = Auth::user();
+               if ($user->negotiations()->find($this->negotiation_id)) {
+                       return true;
+               }
+               if ($user->organizationUser()->first() && $project_managers = $negotiation->project()->first()->organizationUsers())
+                       return $project_managers->find($user->organizationUser()->first()->id) ? true : false;
+               }
+               return false;
+       }
+}
```

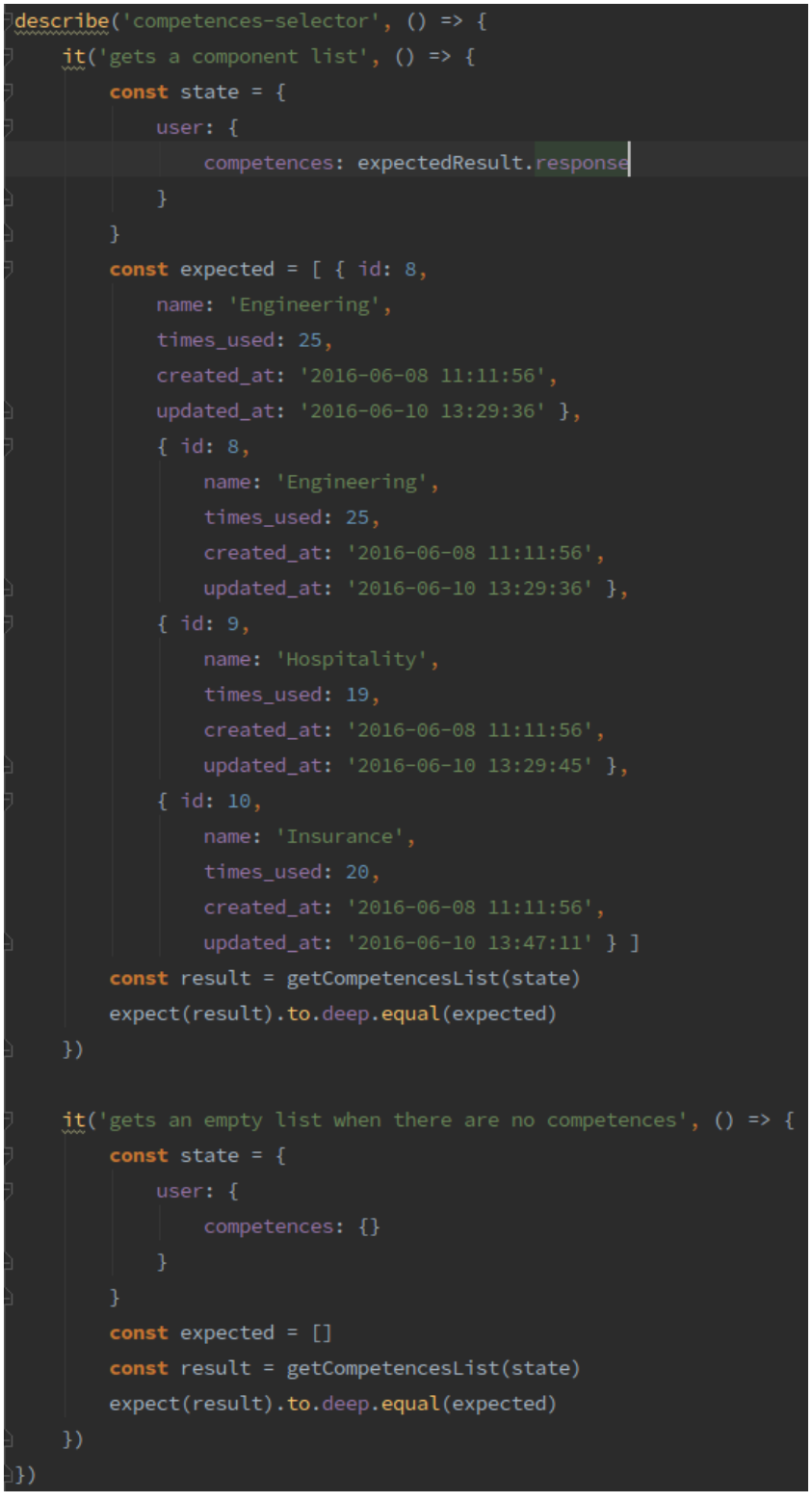
Figure C.5: Extracted authorize method in new 'TermRequest'

```
describe('widgets-reducer', () => {
    it('handles RETRIEVE_WIDGET_DATA_SUCCESS', () => {
        const initialState = {}
        const action = {type: 'RETRIEVE_WIDGET_DATA_SUCCESS', widget_data: {
            unread_messages: 3,
            pending_terms: 1,
            open_negotiations: 4
        }}
        const nextState = widgetReducer(initialState, action)
        expect(nextState).to.deep.equal({
            unread_messages: 3,
            pending_terms: 1,
            open_negotiations: 4
        })
        expect(initialState).to.deep.equal({})
    })
})
```

Figure C.6: A basis for a reducer test

```
describe('widgets-action', () => {
    it('creates RETRIEVE_WIDGET_DATA_SUCCESS', () => {
        const expectedAction = {type: 'RETRIEVE_WIDGET_DATA_SUCCESS', widget_data: {
            unread_messages: 3,
            pending_terms: 1,
            open_negotiations: 4
        }}
        const jsonResponse = { unread_messages: 3, pending_terms: 1, open_negotiations: 4 }
        const action = retrieveWidgetDataSuccess(jsonResponse)
        expect(action).to.deep.equal(expectedAction)
    })
})
```

Figure C.7: A basis for an action test



```
describe('competences-selector', () => {
    it('gets a component list', () => {
        const state = {
            user: {
                competences: expectedResult.response
            }
        }
        const expected = [ { id: 8,
            name: 'Engineering',
            times_used: 25,
            created_at: '2016-06-08 11:11:56',
            updated_at: '2016-06-10 13:29:36' },
            { id: 8,
                name: 'Engineering',
                times_used: 25,
                created_at: '2016-06-08 11:11:56',
                updated_at: '2016-06-10 13:29:36' },
            { id: 9,
                name: 'Hospitality',
                times_used: 19,
                created_at: '2016-06-08 11:11:56',
                updated_at: '2016-06-10 13:29:45' },
            { id: 10,
                name: 'Insurance',
                times_used: 20,
                created_at: '2016-06-08 11:11:56',
                updated_at: '2016-06-10 13:47:11' } ]
        const result = getCompetencesList(state)
        expect(result).to.deep.equal(expected)
    })

    it('gets an empty list when there are no competences', () => {
        const state = {
            user: {
                competences: {}
            }
        }
        const expected = []
        const result = getCompetencesList(state)
        expect(result).to.deep.equal(expected)
    })
})
```

Figure C.8: A basis for a selector test

D

User test script

You probably have a good idea of why we asked you here, but let us explain again briefly. We are asking people to try our platform and see whether it works as intended. This sessions should only take around 10 minutes. Important to know is that we're testing the platform, not you. You can't do anything wrong here. As you use the platform, I am going to ask you as much as possible to try to think out loud. Just say what you are looking at, what you are trying to do and what you are thinking. If you have any questions, just ask them. Also, please do not worry about hurting anyone's feelings as we are doing this to improve our platform and we needs to hear honest reactions.

So let's start The following paragraphs each describe a task that we would like you to perform. You should be in front of your computer right now and see the login screen. You may start by reading the first task now.

And again, it will help us if you can try to think out loud as much as possible as you go along.

Create profile In order to gain access to our platform, you will need an account. Please choose the for you most appealing way to sign up.

First impression After you're logged in, you should see the homepage of your dashboard. Please tell us what you think. Does it look like what you expected? What do you like? And what not?

Education Part of your profile is public, make sure that when people visit your profile, that the information about your current living, previous education etc. is accurate.

Competences As a person, you have certain skills that you would like to inform organizations about as they are searching for people. Please make sure that this part is also up to date.

Projects Start negotiation on a project you like and complete this negotiation.

Compliance Congratulations, you now have a contract signed! Make sure you are compliant with this contract. Certain rules might come with requirements, there are files and document identifiers ready for this on your desktop.

Effort Once you have completed some work for the project, you want to write your effort. Please write some hours for the time you spend helping us today.

E

Info sheet

FleXentral v 1.0

Joost Rothweiler
Arjo van Ramshorst

Project Description

Managing an organization's contingent workforce has become an increasingly difficult challenge over the last couple of years. The number of contingent workers in companies keeps growing, and legislation by governing organizations often changes. FleXentral aims to provide an online software solution solving the three main problems that make managing the contingent workforce difficult. These three problems include the finding of qualified personnel, keeping track of regulatory compliance and insight in finance. Our project focuses on solving this by creating a system used by both flexworkers and organizations that addresses all three of these problems: We have developed a dashboard containing a matching module, compliance module and reporting pages to gain insight in expenses, visualizing everything on a personal, project and organizational level.

Members project team

This project was performed by two team members, Joost Rothweiler and Arjo van Ramshorst. Both have a particular interest in web information systems. Functioning as a team of two members, decisions were almost always made together. We worked together on the project almost every day over the last 10 weeks, therefore we have never had problems with communication. Because the product is designed in a way such that front-end and back-end are completely decoupled, we have decided to draw a line of responsibility between the two of us. Even though we made every decision together, during the project Arjo was responsible for designing and maintaining the React store in the front-end, while Joost was responsible for designing and implementing most of the database. In the end, we have both spent our time integrating the two and developing the visual part of the application.

Client

Company: FleXentral B.V.
Contact: Onno Hektor

TU Coach

Name: Alessandro Bozzon
Department: Web Information Systems

Contacts

Joost Rothweiler: joostrothweiler@gmail.com
Arjo van Ramshorst: arjovanramshorst@gmail.com
Onno Hektor: onno.hektor@flexentral.com

The final report for this project can be found at: `http://repository.tudelft.nl`

F

Original Project Description

Project description

The project team will be assigned to build version 1.0 of a startup software. FleXentral is an enterprise software solution for companies with a substantial flexible workforce. FleXentral's immediate ROI is a cost reduction of 10%-30% per year by cutting out the "middle man". At the same time, it ensures compliance, controls budgets, and provides insights on the effectiveness of your contractors. All in all; complete management of your flexible workforce. At this moment the product has a functional description, the team will have to decide with a lead developer what can be achieved in a version 1.0. Nothing has been set.

Company description

FleXentral is a Dutch startup financed from angel capital and led by two software veterans, André Bonvanie and Onno Hektor. The ambition of FleXentral is to become the "über" of the flexible workforce market. The Netherlands will be pilot country. The project team, which is already selected: Arjo van Ramshorst and Joost Rothweiler, have the option after their Bachelors to continue working on the project.

Auxiliary information

Students will work from a weekly scrum every monday morning in Amsterdam and will be guided by an experienced lead developer assigned to this project. Upto the discretion of the students and the lead developer if working from a home base is needed / required.