BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

CENTRALIZED MANAGEMENT OF CONTINUOUS INTEGRATION
CENTRALIZOVANÉ ŘÍZENÍ KONTINUÁLNÍ INTEGRACE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR   MIKHAIL ABRAMOV
AUTOR PRÁCE

SUPERVISOR   Ing. RADEK BURGET, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2021

Department of Information Systems (DIFS)   Academic year 2020/2021

Bachelor's Thesis Specification

24064

Student: **Abramov Mikhail, Bc.**
Programme: Information Technology
Title: **Continuous Integration Dashboard**
Category: Web
Assignment:

1. Study the existing continuous integration services and their application interfaces. Focus on Jenkins and Travis CI.
2. Study the current tools and platforms for implementing client-server web applications.
3. Design a web application for integrating results from multiple CI services and their presentation in a suitable form.
4. Implement the designed application using a suitable technology. Also implement CI/CD interaction through the dashboard.
5. Test the implemented solution on a suitable group of users.
6. Evaluate the achieved results.

Recommended literature:

- Jean-Marcel Belmont: Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI, Packt Publishing Ltd, 2018

Requirements for the first semester:

- Items 1 to 3

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Burget Radek, Ing., Ph.D.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 12, 2021
Approval date: October 22, 2020

Abstract

This bachelor thesis deals with the development of a web application for integrating CI/CD instruments. The main goal was to study the currently available technologies and prepare an application corresponding to the provided requirements. During system implementation, were learned development aspects such as CI/CD instruments, backend frameworks, frontend frameworks, databases, web server technologies, containerization tools. As a result of bachelor's thesis we prepared an application consisting of three parts: server part prepared with Django REST framework, client part prepared with ReactJS based on TypeScript and PostgreSQL database. This application is prepared to run divided into three parts in containerization tools such as Docker and OpenShift.

Abstrakt

Tato bakalářská práce se zabývá vývojem webové aplikace pro integraci nástrojů CI/CD. Úkolem bylo prostudovat dostupné technologie a připravit aplikaci odpovídající zadaným požadavkům. Během implementace systému byly nastudovány aspekty vývoje, jako jsou nástroje CI/CD, backendové kostry, frontendové kostry, databáze, technologie webových serverů, nástroje pro práci s kontejnery. Výsledkem bakalářské práce je aplikace skládající se ze tří částí: serverová část připravená pomocí rámce Django REST framework, klientská část připravená prostřednictvím rámce ReactJS na bázi TypeScript a PostgreSQL databáze. Tato aplikace je rozdělená do tří částí a připravena ke spuštění v kontejnerizačních nástrojích, jako jsou Docker a OpenShift.

Keywords

CI/CD, web, backend, frontend, database, Django, ReactJS, Docker, NGINX, OpenShift, docker compose, REST, dashboard.

Klíčová slova

CI/CD, web, backend, frontend, databáze, Django, React, Docker, NGINX, OpenShift, docker compose, REST, dashboard.

Reference

ABRAMOV, Mikhail. *Centralized management of continuous integration.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radek Burget, Ph.D.

Rozšířený abstrakt

Cílem této práce je vyvinout webovou aplikaci, která bude integrovat řadu systémů kontinuální integrace. Důležitost takové aplikace, nebo jinými slovy, řídicího panelu kontinuální integrace, byla určena konkrétními okolnostmi. Ve společnosti existuje tým rozdělený na několik dílčích týmů, které používají různé nástroje pro kontinuální integraci pro různé účely. Tyto nástroje pro kontinuální integraci v současné době zahrnují Jenkins a TravisCI, ale integrace nových prostředí, jako jsou CircleCI a GitHub Actions, může být v blízké budoucnosti více než nezbytná. Hlavním problémem je, že dříve používané řešení by vyžadovalo pracné přizpůsobení potenciálního rozšíření. Tým tedy dospěl k závěru, že tento problém lze vyřešit v rámci bakalářské práce, jejímž výsledkem bude fungující prototyp.

Jak jsme se rozhodli tento problém vyřešit? Je třeba analyzovat všechny současné a potenciální nástroje pro nepřetržitou integraci. Měla by být navržena struktura webové aplikace. V důsledku toho je nutné prozkomat současné technologie pro vývoj webových aplikací a vybrat požadovaný framework pro vývoj. Protože aplikace bude v rámci společnosti, je nutné prozkoumat technologii LDAP a potenciál jejího využití. Moderní technologie nasazení aplikací zahrnují práci s kontejnery. Z toho vyplývá, že je nutné nastudovat a použít technologii kontejnerizace, jako jsou Docker a OpenShift.

K řešení definovaných úkolů a otázek byly nastudovány moderní metody vývoje webových aplikací. Bylo rozhodnuto rozdělit aplikaci na tři úrovně: úroveň klienta, úroveň serveru a úroveň databáze. Úroveň klienta nebo jinými slovy frontend bude implementována pomocí rámce ReactJS založeného na jazyce TypeScript, úroveň serveru bude implementována pomocí rámce Django REST framework a PostgreSQL bude použit na úrovni databáze. Pro ověřování uživatelů je možné použít podnikový server LDAP. Ověřování uživatelů bude založeno na relacích. Pro integraci služeb kontinuální integrace bude použito API každé služby. Docker se použije k vytvoření každého kontejneru samostatně. Technologie Docker-compose usnadní automatické vytváření trvalých vazeb mezi kontejnery. Po úspěšném spuštění aplikace lokálně a v kontejnerech je nezbytné pokusit se o nasazení aplikace do cloudového prostředí OpenShift. Během vývoje nesmí být zapomenuta možná budoucí rozšíření.

Během vývoje aplikace jsme dosáhli pozoruhodného úspěchu. Cíl práce byl splněn. Výsledkem je, že tým má webovou aplikaci rozdělenou na tři plánované části, dostupné ke kontejnerizaci jak lokálně prostřednictvím dockeru, tak v cloudovém prostředí přes OpenShift. Byla přijata zpětná vazba od uživatelů, byly předloženy návrhy na vylepšení, prototyp byl přijat a aplikace bude v budoucnu nadále vyvíjena.

Centralized management of continuous integration

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Burget, Ph.D. The supplementary information was provided by Ing. Jelínek. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

....

Mikhail Abramov

May 11, 2021

Acknowledgements

Contents

Chapter 1

Introduction

Nowadays, it is not easy to imagine an actual project that would be implemented and supported for a long time without a modern agile development methodology. Modern development involves the separation of the work areas and the separation of the responsibilities. In turn, this leads to the need for constant and continuous integration of processes, solutions and code bases.

How to ensure this constant and continuous process? To answer this question, we will determine what the CI/CD method is.

As mentioned [1] the abbreviation, CI/CD has several different meanings. CI always refers to Continuous integration, which is an automation process for developers. A successful CI means that new changes to the application code are regularly created, tested, and merged into a shared repository. CI is a solution to developing many application parts simultaneously, which can conflict with each other.

The second part CD means simultaneously Continuous delivery or Continuous deployment. Continuous delivery means an automation uploading to the repository and preparation for deploying in live production. Continuous deployment refers to automation deploying from the repository to production. Further, in work, we will mainly write about CI or Continuous integration.

There are many tools for CI/CD process. However, they are united by terminology, for example, there are Jobs inside each tool, Pipelines, Builds and etc. Most importantly, what is a Pipeline in the CI/CD process? A CI/CD Pipeline is a series of steps that must be performed in order to deliver a new version of the software. Usually, these steps are defined in the form of specific code, but it already depends on the specific tool. A ready-to-use Pipeline and loaded into the CI/CD system is already called Job and/or may consist of many Jobs. This is our main object of interaction. We can launch it, set particular settings for automatic launch, define parameters, and much more. The result of a running Job is a Build that contains logs, and most importantly, the result – successful or unsuccessful.

Now when the basic terms announced, let us imagine the situation. Some team is working on several major products. Within some team, there are several sub-teams. Each sub-team uses its own set of infrastructure to test its part of the product, and, accordingly, each team has its own (sometimes more than one) tool for conducting Continuous integration. Furthermore, of course, it is necessary to consolidate the results of many systems somehow quickly and in an accessible form at any time for control and management purposes.

For me, this is not an imaginary story. I work in such a great team. When I was looking for a topic for my thesis, I received an offer to develop a new system for multiple CI tools

integration or, other words, an Dashboard mainly for consolidation but with the possibility to expand its functionality in case of success in development and positive feedback. I was certainly glad of this proposal. For me it is an opportunity to study new technologies both for the server part of the application (backend) and for the client part of the application (frontend), get acquainted with new for me authentication mechanisms and new tools for CI/CD purposes.

Thus, in this work, I plan to describe the process of planning and developing the new system in as much detail as possible.

First of all, I will conduct a study of the CI/CD tools, existing solutions and technologies available for development, both for the server-side (backend) and for the client-side (frontend). This information will be in the Chapter 2.

Further, in Chapter 3, it will be necessary to prepare a plan for implementing the program. In the process, I will try to answer such questions as: „What tools will be used?", „How to authenticate users?", „How to communicate with the tools?", „What do users want to see?", „How will it be easier for a user to interact with the dashboard?" and other questions.

And finally, in Chapter 4, I will describe the process of implementing the program with examples of code and difficulties that arise in the implementation process. In addition, I will try to collect feedbacks, process them and form proposals for further expanding the prototype.

In conclusion, I would like to say that this work is important to me not only because of the thesis itself and the university's graduation but also, last but not least, important that it will be my first individual project at work. I hope that my application will be assessed positively and I will continue to develop it outside of work on the diploma. I plan that this program will be easy to expand both in functionality and in the number of systems with which it can communicate.

Chapter 2

Related technologies

This chapter is exclusively devoted to the collection and analysis of theoretical information. First of all, what these CI/CD tools are and what tools exist, we will take a closer look at the fundamental aspects that unite many solutions, especially those that will be needed to implement the application. Secondly, we will consider the existing external and internal solutions. At the end of this chapter, we will consider the existing tools for developing a future application and determine the priority tools for future implementation.

2.1 Continuous integration tools

Even though in the introduction, we have already outlined some basic definitions, We want to go through them again in more detail within the framework of this section.

What is Continuous integration? As defined by Jean-Marcel Belmont [2] – Continuous integration (hereinafter referred to as CI) is the essential task where code is both merged and tested on a mainline trunk. A CI task can do any multitude of tasks, including testing software components and deploying software components. Continuous integration is continuous because a developer can be continuously integrating software components while developing software.

As we may have noticed, this definition contradicts the one given earlier in 1, since the CI process does not include deployment, which is already part of the CD inside the CI/CD process. For this work, we will use the definition that CI is a triad: build, test, merge as shown in Figure 2.1.

Figure 2.1: CI/CD process scheme. Figure RedHat [1].

When we have defined what CI is, it is time to define what CI tool is. Nikhil Pathania [3] defined CI tool as an orchestrator, as center of integration system as shown on Figure

2.2. It connected to version control system tool (hereinafter referred to as VCS), build tool, repository tool, testing and production environments, quality analysis tool.
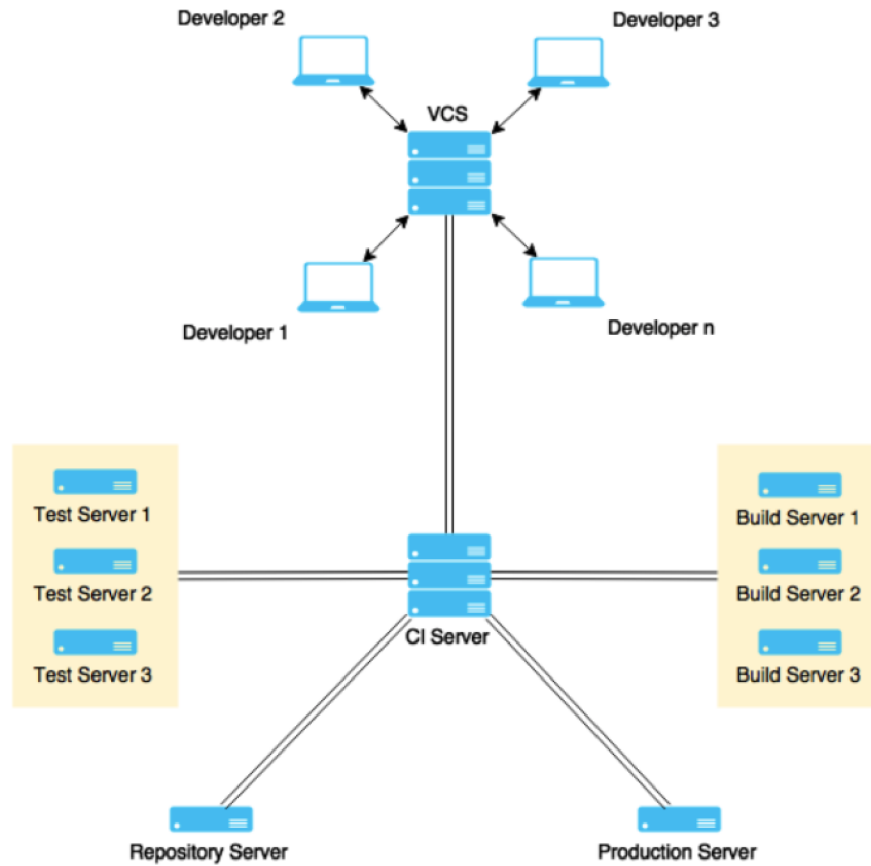

Figure 2.2: CI tool scheme. Figure Nikhil Pathania [3].

Each CI tool consists of various Pipelines. Each Pipeline has its purpose. As we previously mentioned – A ready-to-use Pipeline and loaded into the CI/CD system is already called Job and/or may consist of many Jobs. Meanwhile, Job consists of most minor tasks which run sequentially. It can be some atomic command like a copy or a more complicated sequence: repository cloning, building, testing. And completed with whatever the result, Job is a Build.

Having demonstrated what all tools have in common, we move from general to specific CI tools. We will also carry out the necessary unification of terminology to further integrate the tools.

2.1.1 Jenkins

What is Jenkins? Jenkins is an open-source automation server that can be used to automate all sorts of tasks. It is not limited to CI alone. Jenkins can be used to achieve also Continuous Delivery and Continues Deployment.

Following Jenkins User Documentation[4], Jenkins can be installed through native system packages, Docker, or even run standalone by any machine with a Java Runtime Environment (JRE) installed.

6

What is Jenkins made of? As noted earlier, all systems consist of approximately the same elements set, which for the purposes of that work will be called: Pipeline, Job, Build.

First and foremost, Jenkins Pipeline is present in this system. Nikhil Pathania [3] gives the following definition of pipelines inside Jenkins – a group of multiple Jenkins Jobs that run in sequence or parallel or a combination of both.

How can we create a Jenkins Pipeline? The Pipeline needs to be defined in a particular text file, which will bear the name - Jenkinsfile, which in turn can be committed to a project's source control repository. This approach is called „Pipeline-as-code" and allows pipeline to be versioned and reviewed like any other code. Example of Declarative Jenkins Pipeline:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {}
        }
        stage('Test') {
            steps {}
        }
        stage('Deploy') {
            steps {}
        }
    }
}
```

If Jenkins Pipeline is a set of Jenkins Jobs, then what is Jenkins Jobs? Jenkins Job, in other words – Project, is a user-configured description of work that Jenkins should perform, such as building a piece of software. At this stage, it is necessary to unify the terminology. As we can see, Jenkins defines a Pipeline as a sequence of Jobs and a Job as a specific project with a goal. For the purposes of the thesis, we admit that a pipeline can include from zero (has predefined tasks inside itself) to N jobs. When loaded into the system, the Pipeline is a job (or a project in terms of Jenkins terminology) and can be included as part of another pipeline. In other words, a Jenkins Jobs is a Jenkins Pipeline loaded into the system with which communication is possible.

Last but not least, Jenkins Build. The definition of this term differs from source to source. Nikhil Pathania [3] is more inclined to define that a build is anything from a simple command to an entire script within a project. When the official documentation [4] defines a build as the result of a single execution of a project we will adhere to the documentation definition, as it most closely corresponds to the general definition we gave earlier.

One of the most important things that distinguish Jenkins from other systems is plugins. Plugins allow us to expand the scope of Jenkins incredibly.

For remote work with Jenkins, a powerful API system is presented. Tool provides machine-consumable remote access API to its functionalities. Remote access API is offered in a REST-like style. Currently it comes in three flavors:

- XML

- JSON with JSONP support

- Python

Jenkins Advantages:

- Jenkins is free opensource system

- Has many settings

- Plugin system from git to aws

- Full control over the system

- Powerful REST API

Jenkins Disadvantages:

- Dedicated server, container or cloud container required

- Setting up and maintaining takes experience and time.

In the thesis, Jenkins is the basis for the CI instruments study. We will use the knowledge gained about Jenkins to compare it with other tools, determine the pros and cons for each tool, and unify various systems' terminology.

2.1.2 Travis CI

What is Travis CI? Travis CI is hosted and automated CI solution. Travis CI uses an in-application configuration file in YAML syntax called .travis.yml.

Comparing to Jenkins, Travis CI designed around the principle of opensource development in easy and fast to use way. Travis CI can be set up within minutes of creating a project in the VCS. Although Travis CI is not as customizable as Jenkins CI in this respect, it has the distinct advantage of fast setup and use.

Let us go over the main aspects for the purposes of this thesis: Pipeline, Job, Build.

What is Travis CI Pipeline? The documentation does not define the term Pipeline. However, there is the concept of a workflow or configuration file. Its content is, in essence, an element that can be called a Pipeline.

Example of a minimal workflow file for Travis CI:

```
language: node_js
node_js:
    - "6.14.1"
install:
    - npm install
script:
    - npm test
```

In that example, we defined language and language version, then Pipeline will install dependencies and finally will run tests.

Secondly, Job. Following Travis CI, documentation [5] Travis CI Job is an automated process that clones target repository into a virtual environment and then carries out a series of phrases such as code compiling, tests running. A job fails if the return code of the script phase is non-zero. We can notice that this definition also intersects with the general definition of Build. Travis CI Build is defined as a group of jobs that run in sequence.

Despite some confusion with definitions in Travis CI documentation. For work purposes, we define:

- Workflow defined in configuration file as Pipeline

- As Job we defined workflow recognized by Travis CI
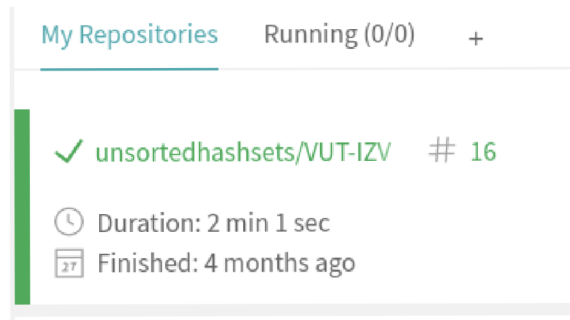

Figure 2.3: Travis CI Job. Figure Mikhail Abramov.

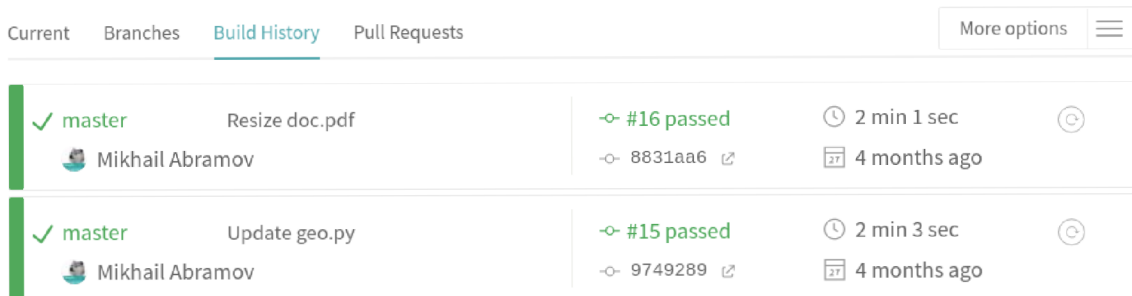- As Build we defined finished Job or group of Jobs

Figure 2.4: Travis CI Builds. Figure Mikhail Abramov.

Travis CI Advantages:

- Fast and easy start

- Easy-to-read YAML configuration files

- Free version exists

- no need for a dedicated server

- Powerful REST API

Travis CI Disadvantages:

- Limited customization options

2.1.3 CircleCI

What is CircleCI? It is a cloud-based system that does not need to set up a separate server and does not need to be administered. This system is very similar to the one already

disassembled, to Travis CI. CircleCI also does not require a separate server and complex configuration. Only the configuration file is needed in the repository.

As described in Circle CI Documentation [6] – „CircleCI believes in the configuration as code". Entire CI is orchestrated through a single file called config.yml. The config.yml file is located in a folder called .circleci at the root of the Project. CircleCI uses the YAML syntax for config. Configurations file example:

```
version: 2.1
# Define the job
jobs:
    build:
        machine:
          image: ubuntu-2004:202010-01
        steps:
          - checkout
          - run: my-build-commands
# Orchestrate job
    workflows:
        build:
            jobs:
                - build
```

This system has a more different structure from the previous one. CircleCI consists of:

- Pipeline – represents the entirety of configuration

- Workflows – responsible for orchestrating multiple jobs

- Jobs – responsible for running a series of steps that perform commands

- Steps – run atomic commands and shell scripts

Thus, for our purposes, we can assume that Job in the general sense is a CircleCI of Workflow. Therefore, Build is a cumulative result of Workflows.

CircleCI Advantages:

- Fast and easy start

- Easy-to-read YAML configuration files

- Free version exists

- no need for a dedicated server

- Good REST API

CircleCI Disadvantages:

- Limited customization options

2.1.4 GitHub Actions

The last but not the least system that we plan to consider is GitHub Actions. Native solution provided from VCS GitHub. Like Travis CI or CircleCI, it is a cloud-based system. However, it is also separately noted in the documentation [7] that GitHub Actions is an „event-driven" system (which does not distinguish the system from the previous two). Schematically, the system's operation looks as shown in the Figure 2.5.
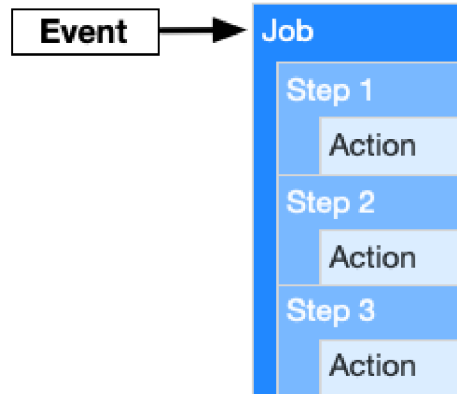

Figure 2.5: GitHub actions work scheme. Figure GitHub Actions Documentation [7].

To start using GitHub Actions, we have to add the configuration file FILENAME.yml in .github/workflows directory. Example of the configuration file:

```
name: 'test pipeline'
  # On action (event-driven)
  on:
    push:
  jobs:
    test-job:
      runs-on: ubuntu-20.04
      steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-node@v1
        name: 'setup node'
        with:
          node-version: '14.x'
      - name: 'install'
        run: npm i
      - name: 'test'
        run: npm run test
```

Thus, for our purposes, we can assume that the Pipeline in the general sense is the GitHub Actions configuration. Job is GitHub Actions Workflow. The Build is the cumulative result of GitHub Actions workflow jobs.

GitHub Actions Advantages:

- Fast and easy start

- Easy-to-read YAML configuration files

- Free version exists

- no need for a dedicated server

- Good REST API

- Possibility to use more than one pipeline

GitHub Actions Disadvantages:

- Limited customization options

2.2 Existing integration solutions for CI tools integration

After exploring the resources that we need to integrate into a single system, existing solutions of similar dashboards should be examined. The goal is to find a similar program or remotely similar to the planned one. Sources – are opensource projects, including projects within the company.

2.2.1 External solutions

The only Project we have found that resembles the idea of integrating with CI tools was dash-ci[1]. Dash-CI is a simple dashboard to show real-time continuous integration tasks status and other project information.
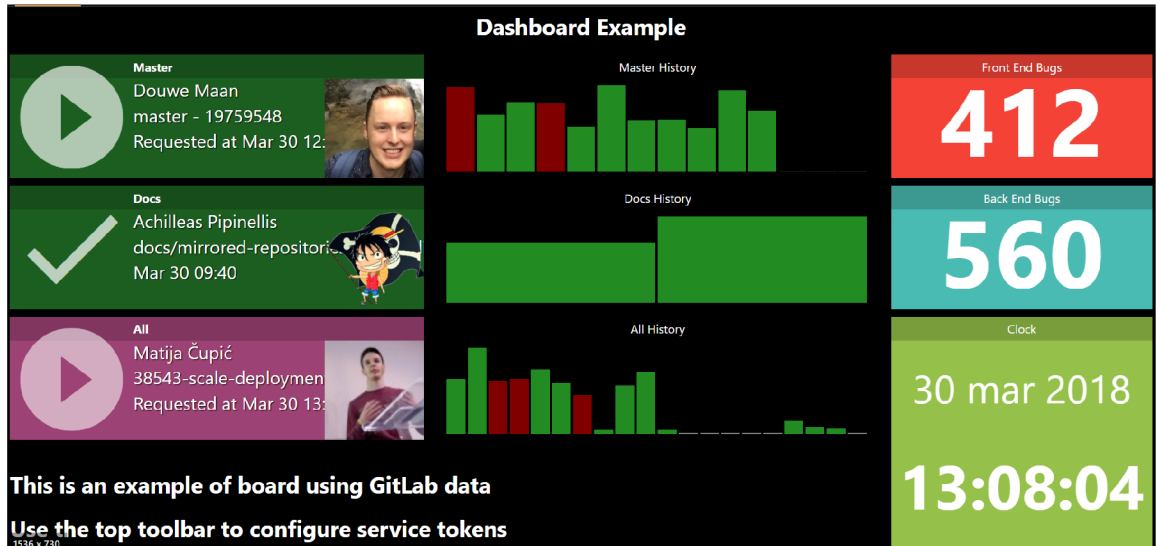

Figure 2.6: Dash-CI page view. Figure Marcos Junior.

This Project is interesting for its unusual and modular structure of the dashboard itself. Powerful customization is available. The implementation includes only the client-side with local storage. The most important feature, in our opinion, is communication with pipelines inside the GitLab VCS system.

---

[1]Dash-CI, Marcos Junior: https://github.com/junalmeida/dash-cid

2.2.2   Internal solutions

Previously, our team used this application – Fuse Tools CI Dashboard[2]. Web application with the client part developed in ReactJS, and the server part implemented with the flask framework.

Figure 2.7: Fuse Tools CI Dashboard. Figure Mikhail Abramov.

Fuse Tools CI Dashboard Advantages:

- Unlike the external solution, the interface is very intuitive.

Fuse Tools CI Dashboard Disadvantages:

- Limited functionality, the possibility of its expansion is limited by architectural solutions:

  - Lack of a token system leads to the impossibility of adding more than one token-requiring CI systems (Travis CI, CircleCI, etc.)
  - Lack of a token system leads to the impossibility of adding additional function, the most commonplace - rebuild, which is token-requiring command in Jenkins too.
  - Lack of account system leads to the inability to use personal tokens
  - The absence of a database implies hardcoded seeds inside the backend code

From our point of view, the existing solution is an ideal example for creating an extended prototype that takes into account the old limitations in the architecture, and most importantly, to create an architecture that can be extended without significant changes in the main codebase.

---

[2]Fuse Tools CI Dashboard, Lars Heinemann: https://github.com/lhein/build-dashboard

2.3 Dashboard implementation tools

In this chapter, we will look at the fundamental technologies needed to implement a dashboard. These technologies include - LDAP for user authentication purposes, a framework for the server-side, a framework for the client-side, containerization and deployment technologies.

2.3.1 LDAP

As we noted earlier, none of the available applications supported the user system. Thus, we wondered how it is possible to verify users with their corporate data within the company, but without having direct access to them within the system. The corporate LDAP server can solve this problem.

What is an LDAP server? The Lightweight Directory Access Protocol (LDAP) [8] is an open industry standard to improve functionality and ease-of-use, and to enable cost-effective administration of distributed applications, information about the services, resources, users, and other objects accessible from the applications needs to be organized in a clear and consistent manner.

LDAP is relatively simple protocol that uses TCP/IP and allows us to perform authentication (bind), search (search), and compare (compare) operations, as well as add, modify or delete records. Typically, the LDAP server accepts incoming connections on port 389 over TCP or UDP. For SSL Encapsulated LDAP sessions, port 636 is typically used.

At its core, the technology uses the principle of Directories. A directory is a information about objects arranged in some order that gives details about each object. In computer terms, a directory is a specialized database, also called a data repository, that stores typed and ordered information about objects. The directory structure and use of attributes are shown in the Figure 2.8.
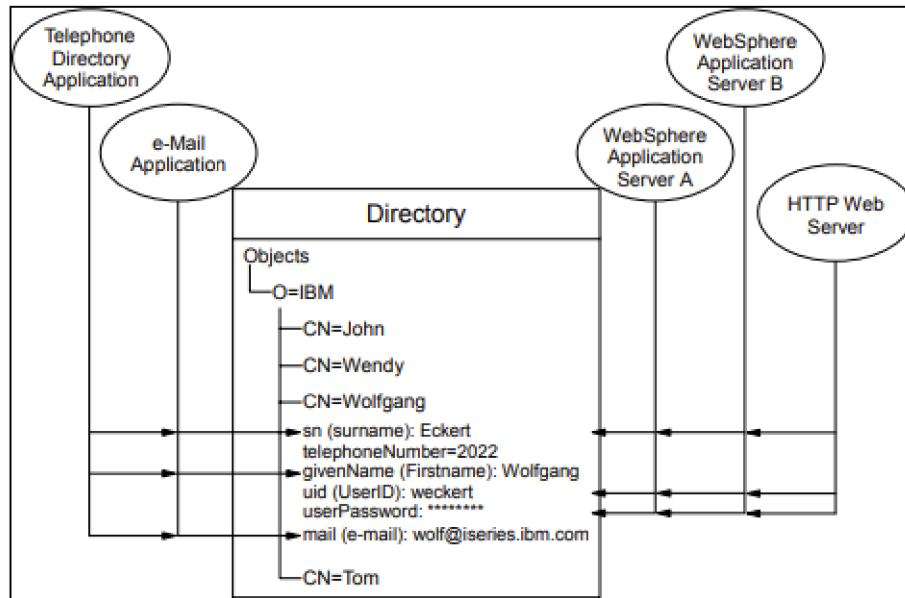

Figure 2.8: Several applications using attributes of the same entry. Figure [8].

Main features of LDAP for implementation purposes:

- Connection (bind) - allows us to associate a client with a specific directory object to control access for all other read/write operations. In order to work with the Directory, the client must authenticate as an entity whose Distinguished Name is in the namespace described by the Directory.

- Disconnect (unbind) - Allows the client to switch to authentication with a new distinguished name within the session with the LDAP server.

In UNIX/Linux systems, directories (and LDAP as a protocol for accessing them) have become widespread for storing system information, such as user accounts and service settings. One of the most common LDAP servers on UNIX/Linux systems is OpenLDAP. On Windows, the LDAP server is built into ActiveDirectory.

Usage example with OpenLDAP [9]:

```
ldapsearch -x -h ldap.corp.redhat.com \
          -b dc=redhat,dc=com -s sub 'uid=mabramov'
```

Part of the LDAP response:

```
...
cn: Mikhail Abramov
l: Brno
co: CZE
...
```

2.3.2   Server-side framework

What is a backend framework? A software framework is a basis on what developers can make applications in a faster and standardized way. There are a wide variety of frameworks for implementing the server-side of the application, or in other words, the backend. They are based both on the most popular and used languages such as PHP and Python, as well as on more rare ones. The most popular frameworks include:

- Django (Python)

- Laravel (PHP)

- Ruby on Rails (Ruby)

- ExpressJS (NodeJS)

- CakePHP (PHP Mapme)

- Asp .NET (C#

- Spring Boot (Java)

- Koa (NodeJS)

- Phoenix (Elixir)

Since the Django framework contains all the elements necessary at first glance, it is widespread and can be extended by many libraries such as, for example, a restframework and LDAP. As well, it is implemented in the Python language widely supported in our team. We decided to use the Django framework.

Django is a leading open-source backend framework based on the Python programming language [10]. It is suitable for the development of sophisticated database-driven websites. Django architecturally based on the model-template-view (MTV) pattern [11]. The view portion typically inspectts the incomming HTTP requests. Usually views defined in view.py file. View layer actively communicate with model layer usually defined with powerful ORM[3] tool in models.py file. To access the view layer, we need to define URLs entities in the urls.py file. The view layer will efficiently and quickly form responses in the JSON format for quick and convenient communication with the client's side.

We do not plan to use the template level actively. In our application, Django will be responsible for the REST[4] API erver with a full range of CRUD[5] operations and responses in the JSON[6] format.

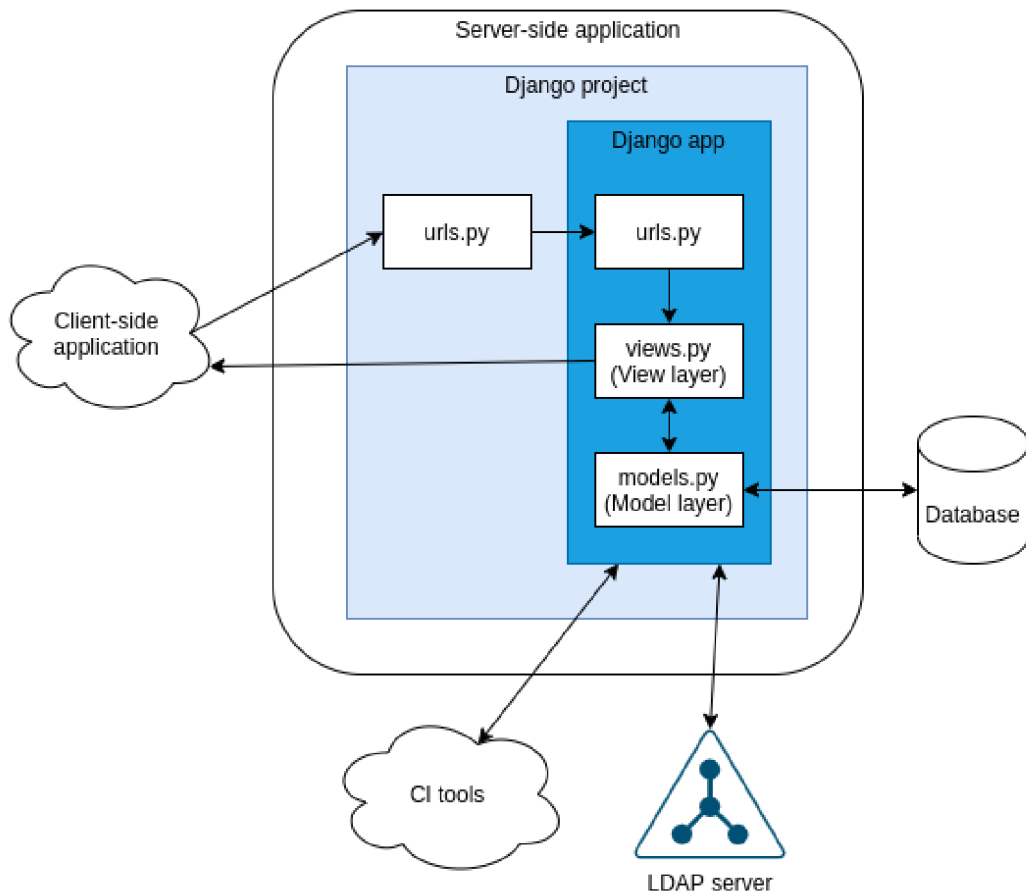Schematically, the work of Django in our application is presented as in the Figure 2.9.

Figure 2.9: Django in application. Figure Mikhail Abramov.

---

[3]An introduction to the Django ORM: https://opensource.com/article/17/11/django-orm

[4]What is a REST API? https://www.redhat.com/en/topics/api/what-is-a-rest-api

[5]What is CRUD? https://www.codecademy.com/articles/what-is-crud

[6]Introducing JSON: https://www.json.org/json-en.html

2.3.3 Client-side framework

What is a frontend framework? As mentioned earlier, a software framework is a way to standardize and speed up production. There are a wide variety of frameworks for implementing the client-side of the application, or in other words, the frontend. They often use different proportions of JavaScript or other language and markup languages as HTML and CSS. The most popular frameworks include:

- ReactJS

- Angular

- Vue.js

- jQuery

- Emberjs

When choosing a framework for the front end, we were guided by the main aspects for us. The most important thing is that we wanted to study TypeScript. Most frameworks and especially the top of the list allow it to be done: ReactJS, Angular and Vue.js. Next, we eliminated Angular due to the hard-weight of the decision, which did not correspond to the planned application. Between the dynamically developing Vue.js framework and the already developed ReactJS framework infrastructure, the choice was pronounced in favour of the latest.

Therefore, the question is - What is ReactJS? React[12] is a JavaScript library that helps build the frontend of an app. It helps the developer manage the components data, components state in a structured fashion. It uses virtual DOM[7] to render frontend efficiently.

TypeScript[13] works great with react and adds the ability to work in an object-oriented style. Allow using interfaces and classes or in React way – Component Props[8], which adds more scope for inheritance and encapsulation.

ReactJS Advantages:

- The reusability of components makes it easy to collaborate and reuse them in other parts of the application

- Stable and flawless performance using the virtual DOM

- Switching between versions is usually very easy

- Gentle learning curve compared to Angular

ReactJS Disadvantages:

- Disordered documentation

- The large selection of tools is baffling

- It takes a long time to master all the nuances

---

[7]ReactDOM: https://reactjs.org/docs/react-dom.html
[8]Props: https://react-typescript-cheatsheet.netlify.app/docs/basic/getting-started/basic_type_example

2.3.4 Database layer technology and integration with server-side

To store information about users and CI tools, we plan to use a standard relational database. A relational database is a type of database[9]. It uses a structure that allows us to identify and access data related to another piece of data in the database.

The most common representatives are:

- MySQL

- PostgreSQL

- Oracle DB

- SQL Server

- SQLite

In principle, each base is approximately the same. We are only interested in support in containers, including Docker and Openshift, and integration with our backend for which we have defined Django.

We decided to study PostgreSQL. Firstly, we determined an engine for working with PostgreSQL in Django, and it must be downloaded and added in the settings in that way:

```
'ENGINE': 'django.db.backends.postgresql_psycopg2'
```

Secondly, we made sure that this database is supported in both Docker[10] and Openshift[11], which we will write about later.

To communicate with the database or, in other words, create database objects: tables and rows. Make queries. We are going to use Django native ORM[12] tool with psycorg2[13] PostgreSQL database driver.

2.3.5 NGINX and WSGI

The WSGI[14] or Web Server Gateway Interface is a universal interface through which a Web application and a Web server communicate. The WSGI server receives requests from the client, forwards them to the application, and finally forwards the application response back to the client.

WSGI HTTP servers include Gunicorn[15] which is supported in Django.

The most reliable way to use the Django app for production is through Gunicorn and NGINX[16] as a reverse proxy. NGINX is a very powerful webserver, and we want to take advantage of its high-performance connection handling mechanisms and its easy-to-implement security features. NGINX was originally developed to solve the C10K problem[17] – that is, to easily support 10,000 or more simultaneous connections. Using NGINX as the web server for our Python app makes our Dashboard application faster.

---

[9]Relational Database: https://www.codecademy.com/articles/what-is-rdbms-sql

[10]postgres: Docker Official Images: https://hub.docker.com/_/postgres

[11]Deploy PostgreSQL in OpenShift backed by OpenShift Container Storage: https://www.openshift.com/blog/deploy-postgresql-in-openshift-backed-by-openshift-container-storage

[12]An introduction to the Django ORM: https://opensource.com/article/17/11/django-orm

[13]Psycopg2 Tutorial: https://wiki.postgresql.org/wiki/Psycopg2_Tutorial

[14]WSGI: introduction: http://wsgi.tutorial.codepoint.net/

[15]Gunicorn - WSGI server: https://docs.gunicorn.org/en/stable/

[16]NGINX: Getting Started: https://www.nginx.com/resources/wiki/

[17]C10K problem: https://www.aosabook.org/en/nginx.html

This solution is well illustrated schematically in the Figure 2.10 by Harshvijaythakkar[18]. This image shows in detail what happens in the process of communication between the client and the server in the Figure 2.9.

Figure 2.10: Django in application. Figure Harshvijaythakkar.

2.3.6 Application deployment

Even though each element individually has an own local development environment. The final part of the implementation should look like three containers, initially in the Docker and then in the OpenShift. Now is the time to describe these tools in more detail.

What is Docker? Docker[14] is open-source software used to develop, test, deliver, and run web applications in containerized environments. It is needed for more efficient use of the system and resources, rapid deployment of ready-made software products, and scaling and porting them to other environments with guaranteed preservation of stable operation. For example, small command:

```
$ docker run --name some-postgres \
    -e POSTGRES_PASSWORD=mysecretpassword -d postgres
```

Allow us to run the container with the PostgreSQL database and work with it.

However, to set up correct communication and access to containers, such as only the backend can access the database, only the frontend can access the backend, and the frontend is open for use from outside this system, an additional tool is needed. It is Docker Compose[15] – a tool for defining and running multi-container Docker applications. This tool will allow us to correctly configure the ports of containers, their availability and communication between specific containers.

The final stage of deployment will be the installation of these containers in the OpenShift application. OpenShift[17] is an open and extensible container application platform that enables to use of Docker and Kubernetes[19]. OpenShift includes Kubernetes for container orchestration and management. Added developer and operations tools here to:

- Develop applications quickly

- Easy to deploy and scale solutions

- Serve the long-term lifecycle of an applications

After unloading containers with application elements into OpenShift, the application will be available to users.

---

[18]Django, Nginx, Gunicorn: https://medium.com/analytics-vidhya/dajngo-with-nginx-gunicorn-aaf8431dc9e0

[19]What is Kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Chapter 3

Architecture design

This chapter is devoted to analysing the team's needs in Dashboard and developing the future application architecture.

3.1 Requirements analysis

Analyzing the team's needs, we divided the main issues for future implementation into two thematic districts: CI tools and actions, interface. The requirements for the future program are very flexible and in general it is necessary to provide a prototype with potential solutions for future expansion.

3.1.1 Required CI tools and actions

After a short briefing, it was clear that at the time of setting the task, it was necessary to provide support for two systems: Jenkins and Travis CI in sections 2.1.1 and 2.1.2 correspondingly. However, it is necessary to envisage expanding the program to more instruments.

In turn, we also drew attention to the shortcomings of the previous solution discussed in the Chapter 2.2.2. Thus, we are convinced that it is necessary to provide a system of unique tokens due to the fact that each instance of Travis CI needs a unique token. In the future, tokens will also be required for advanced functionality. And, consequently, a system of users to ensure a unique constraint of user-token-tool.

In discussing the actions possible through the dashboard, the team decided that the necessary action at the current moment is to request the status of the last Build. However, it is required to provide the possibility of expanding the application's functionality. It is worth noting that even this action in the Travis CI system requires a token, so we again understand the need for a token system.

Required tools:

- Jenkins

- Travis CI

Required actions:

- Last Build status

Necessary architectural solutions:

- Possibility and simplicity of expanding the current set of CI tools

- Possibility and simplicity of expanding the current set of actions

- User system

- Token system

3.1.2 User interface

The requirements for the interface are also not significant. It is necessary to develop a table system, but with the ability to hide part of the toolkit that is not of interest to the user at the moment.

3.2 Schematic solution

In this part, we will consider the main elements of the schematic design of the future application and mockups of the planned interface. At the beginning of this chapter, I would like to draw attention to the prepared deployment diagram in Appendix D. It will perfectly complement the theoretical material presented in Chapter 2.3.6.

3.2.1 Tool Integration Methods

Earlier in the Chapter 2, we considered some CI tools. All these tools provide the user with quite powerful communication tools such as API and sometimes CLI. Thus, it is permissible to use API requests from the backend server to the tool server API, followed by unifying the response and transmission from the backend to the frontend.

Learning the basics of Jenkins API[18], let's try to get the result of the last build using the CURL[20] program (user must be inside the corporate VPN).

```
$ curl --silent https://master-jenkins-csb-fusetools-qe.cloud.paas.psi.\
redhat.com/job/fuse-single/api/json
```

As a result, we received a lengthy JSON response with all information about the Job, including information about the latest builds. But we can find one very important for us element: 'color: „yellow"'. This color indicator in Jenkins indicates build statuses.

Statuses and colors examples:

- SUCCESS – green

- FAILED – red

- UNSTABLE – yellow

- ABORTED – gray

We add jq[21] tool to request to filter JSON response (user must be inside the corporate VPN).

---

[20]CURL man page: https://www.mit.edu/afs.new/sipb/user/ssen/src/curl-7.11.1/docs/curl.html

[21]jq manual: https://stedolan.github.io/jq/manual/

```
$ curl --silent https://master-jenkins-csb-fusetools-qe.cloud.paas.psi.\
redhat.com/job/fuse-single/api/json | jq -r .color
```

Now our response is just „yellow“. This communication method will be easily implemented on the server-side in Python using the Requests library.

The next step, try to request status from Travis CI with the token element. It is possible to learn more about Travis CI API in the official documentation[19].

```
$ curl --silent -H "Travis-API-Version: 3" \
   -H "Authorization: token XXXXXXXXXXXXXXXX" \
   https://api.travis-ci.com/repo/unsortedhashsets%2FVUT-IZV/builds \
   | jq .builds[0].state
```

The output result is „passed“.

Therefore, for the purpose of implementing a solution that can be easily extended, it is necessary to create one essential function for requesting status and mapping responses. In the future, these functions will be supplemented with additional ones for each CI instrument.

The advantages of such an implementation:

- Easily expandable system

- If changes are made to a separate API, other systems will continue to work

The disadvantages of such an implementation :

- It is necessary to study each separate structure of the API

- Each individual tool requires its own mapper

3.2.2   Use cases

We will consider individual elements of the Use Case diagrams. The complete scheme is presented in the Appendix A.
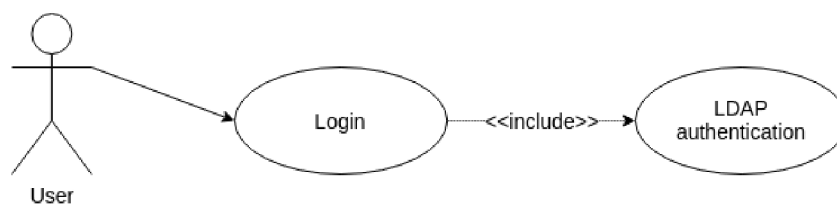

Figure 3.1: Use Case fragment: Authentication. Figure Mikhail Abramov.

The first fragment 3.1 presents an obvious User Case, namely: the user opens the Login window, enter the username and password, confirms his attempt to log in, then a mandatory action takes place - communication with the LDAP server to authenticate the user, and if successful, define him as an existing user.

Similar procedures are provided for administrators, with a slight difference in the verification of roles.

The second fragment 3.2 demonstrates that the user performs several actions with the Jobs of CI tools. Actually, planned action is only a request for the status of the last job,
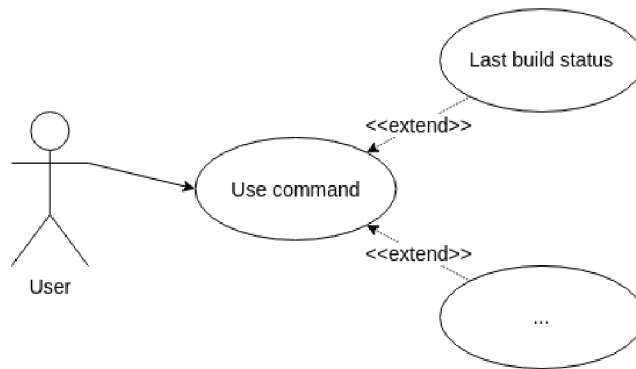
Figure 3.2: Use Case fragment: Actions. Figure Mikhail Abramov.

but the system's functionality could be easily expanded. The user chooses which action to use from the available ones. In the case of the request for the last Job status, a request will be sent to the backend server, which in turn will request the status of the last Build of a specific Job.
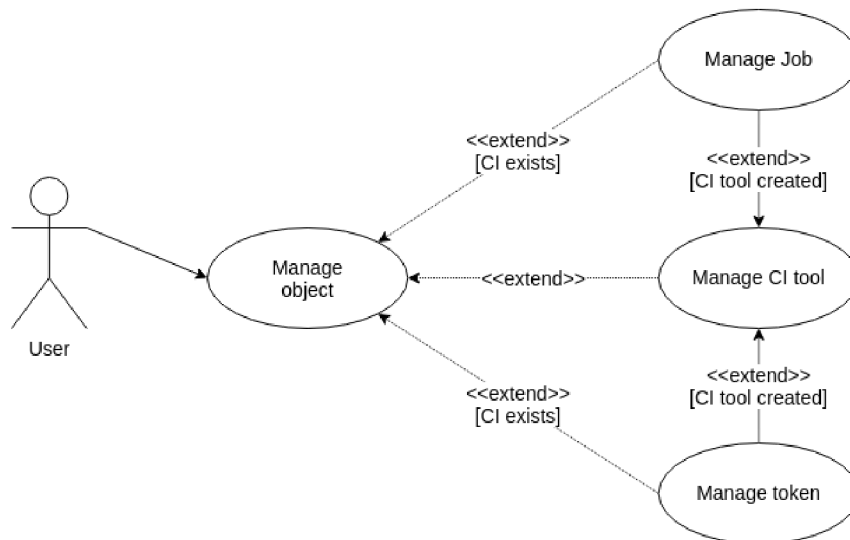

Figure 3.3: Use Case fragment: Objects. Figure Mikhail Abramov.

Third, and in our opinion, the most important, the fragment 3.3 demonstrates the ability for the user to directly manage data about CI tools, their jobs and their tokens to each CI tool.

In square brackets present the condition of use. For example, it is impossible to add/edit a Job and/or a Token if the CI tool for this Job and/or Token has not been created. And when the tool is created, it becomes possible to supplement it with Jobs and Tokens immediately. Example of use: a user creates a CI instrument and immediately adds a Job to it or adds a Token to an existing tool.

The last fragment 3.4 shows the administrator's capabilities. Users who will be marked with a special role for administering the backend server will be able to enter the adminis-
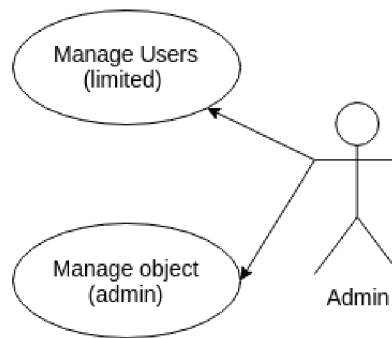
Figure 3.4: Use Case fragment: Admins. Figure Mikhail Abramov.

trator interface and edit the content of the database, with a limited ability to work with users.

3.2.3 Entity Relationship diagram

Analyzing the requirements and tasks, we have identified the following components of the Entity Relationship model:

- User – an essential entity represents a user who has passed the authentication procedure and was added to the database

- CI tool – an entity is a user-added CI tool that can have relationships to the following two entities

- Job – an entity with a more than obvious connection with CI tools - one tool can have many jobs, and one job belongs exclusively to one tool

- Token – an entity that connects the user and a specific tool

What attributes can these entities have? The User certainly has a standard set: name, surname, email, login, password, etc.

To communicate with the CI tool, we need to have the following attributes: CI name, URL. But we are planning to expand these attributes, add additional ones slightly:

- Access – we decided to test the system of public and private instances. For example, a private instance will be available only to its owner (the person who added it)

- Type – we need two options: Travis CI and Jenkins

- Owner – the person who added CI tool

Great, now we have information for communicating with the tools, but what data is needed to work with the Jobs of these tools:

- CI tool – we have to know to what tool certain job belongs

- Job name – essentially the job name

- path – path to job inside CI tool

24

And finally, the Token entity. We need to know what user owns the Token and what CI tool relates with it, so we have two foreign keys. And indeed the Token.

We hope that we have described the entire process of creating the model. The Entity Relationship diagram can be found in the Appendix B.

3.2.4 Database

Since the model prepared in the Entity Relationship diagram does not differ in particular complexity, it just became necessary to supplement it exclusively with some logical constraints.

We have added three constraints:

- CI tool – We decided to add a unique constraint CI_Name-Owner-Access. It allows to avoid redundancy in CI tools and provides to the user, for example, the possibility to make his private copy of existing public instance with other jobs

- Job – Here we have developed the CI_tool-Job_name-Path unique constraint, this will allow us to avoid redundancy of the same jobs, but will allow you to have the same jobs within different instances

- Token – This object has an obvious unique constraint: Owner-CI_tool, which means that one user can have only one token for one tool

The Database diagram can be found in the Appendix C.

3.2.5 User interface mock-up

The three main pages of the layout are presented in the Appendix E. Pop-up windows will deliver all additional elements (forms). This sections will cover the main elements of the interface.
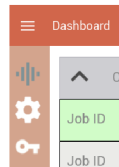

Figure 3.5: Mockup fragment: Menu. Figure Mikhail Abramov.

Firstly, the menu element 3.5 is represented by a vertical line with page icons: dashboard, objects, tokens. On hovering over the icons, they will be highlighted, and a text tooltip will appear. If user clicks on the top bar of the menu, the menu will expand to the right and next to the icons will be written text description, not in the tooltip format.

Figure 3.6: Mockup fragment: Header. Figure Mikhail Abramov.

The second element 3.6 is located in the upper right corner of the application and consists of two functional buttons: switching the mode from day to night and vice versa,

and a login/logout button. When user press the mode button, the colour scale will be changed to the opposite of the actual one, and the icon will be replaced to the contrary. If user clicks on the login, a pop-up window with a login form will open.

Figure 3.7: Mockup fragment: Action table. Figure Mikhail Abramov.

The third 3.7 and fourth 3.8 elements are similar in their basis. The basis of this structure is a table of collapsing tables. In the action table, the rows of the collapsing table are represented by Jobs indicating their status in text and in colour. Data such as ID, job name, build number and status will be presented in text format. There will also be action buttons. User can launch a status update either individually or for all Jobs of certain CI tool.

Figure 3.8: Mockup fragment: Objects table. Figure Mikhail Abramov.

The fourth element 3.8 allows user to communicate with REST API in CRUD way to add, update and delete elements from database. Each button entails opening a pop-up window with the required form. In case of deletion, with the consent form to delete the object.

26

Figure 3.9: Mockup fragment: Tokens table. Figure Mikhail Abramov.

The last element 3.9 involves adding a token to an existing instrument. It is represented by a regular table with function buttons. Interaction with buttons will cause pop-ups like those described earlier.

Chapter 4

System implementation

This chapter will describe the process of implementing the application. The chapter is divided into several thematic areas: implementation of the server-side, implementation of the client-side, deployment (this part includes local installations in containers and in the Openshift), notes and comments from users, plans for further work.

4.1 Server-side implementation

This part will cover the main aspects of the implementation of the server-side of the application. From our perspective, the most exciting aspects of the implementation are the REST framework using, communication with the database, authentication, and work with target CI tools.

4.1.1 REST framework

In our implementation, we used the djangorestframework library [20]. This library significantly expands the capabilities of the Django framework. It allows us to transform Django from an MTV application engine as described in section 2.3.2 into a powerful tool for creating an API server.

   The central and most essential elements of this framework are the introduced elements: routers, serializers, viewsets. The primary part of our application is based on these three elements as well as models and signals.

Routers

The router mechanism allows us to quickly and easily associate views with the final URLs and, most importantly, that will allow to integrate into one line not only, for example, a detail view, but also a list views and allows us to support many HTTP methods

   For these purposes, we used DefaultRouter. This router includes routes for the standard set of list, create, retrieve, update and destroy actions. The viewset can also mark additional methods to be routed, using the @action decorator, but additionally includes a default API root view, that returns a response containing hyperlinks to all the list views.

   These routers are implemented in the file – /backend/ci_dashboardApp/api/urls.py

```
    ...
    router = DefaultRouter()
    ...
```

```
    router.register(r"job", JobViewSet)
    ...
    urlpatterns = [
        path("set-csrf/", set_csrf_token, name='set_csrf_token'),
        ...
        path("", include(router.urls)),
    ]
    ...
```

In this example, we define the router as a DefaultRouter, then register the views for Jobs and form a list of URLs templates in which, in addition to the discussed router, we add particular URLs. Each undescribed element will be explained later in details.

As a result, this fragment allows us to define these routes:

- .../job/ – Jobs list view with GET and POST supported methods

- .../job/{number} – Job detail view with CRUD operations.

- .../job/{number}/status – Particular action predefined with decorator and responsible for last build status retrieve supports only GET method.

- .../set-csrf – Particular route defined in standard Django style, responsible for csrf token retrieve.

Viewsets

Django REST framework allows us to combine the logic for a set of related views in a single class, called a ViewSet. Conceptually similar implementations named 'Resources' or 'Controllers' in other frameworks.

In our implementation, we mainly used the capabilities of ModelViewSet. The actions provided by the ModelViewSet class are .list(), .retrieve(), .create(), .update(), .partial_update(), and .destroy().

This class will be considered on the example of JobViewSet class implemented in /backend/ci_dashboardApp/api/views.py:

```
...
class JobViewSet(ModelViewSet):
    queryset = Job.objects.none()
    serializer_class = JobSerializer

    def get_queryset(self):
        CIObjects = CI.objects.filter(Q(owner=self.request.user.id) |
                                      Q(access="Public")).values_list('id')
        return Job.objects.filter(ci__in=list(CIObjects))

    @action(detail=True, methods=['GET'])
    def status(self, request, pk=None):
        try:
            CIObjects = CI.objects.filter(Q(owner=self.request.user.id) |
                                          Q(access="Public")).values_list('id')
            job = Job.objects.get(Q(pk=pk) & Q(ci__in=list(CIObjects)))
```

```
        except Job.DoesNotExist:
            raise Http404
        try:
            token = Token.objects.get(ci=job.ci.id,
                                 user=self.request.user.id)
        except Token.DoesNotExist:
            token = None
        return Response(ct.processCI(job, token))
...
```

This ModelViewSet assumes working with models, and thus we define a set of objects and use the serializer to bring the object to the python form we need.

In this example, we initially define the set by queryset the database via ORM. That queryset will be used in the list view during get requests and for more complex operations. But it is impossible to implement filters in the default query. We developed the get_queryset method to generate Jobs set with appropriate filters in our example: Jobs would be in the CI tool where the user is owner or instance has Public access.

Also available the status method with @action decorator supporting GET method. In this method, we defined the Job we are interested in and the Token to it and call the function processCI to get the last build status.

Serializers

The serializers in REST framework work very similarly to Django's Form and ModelForm classes. Serializers allow complex data such as querysets and model instances to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types.

Serializers are in the /backend/ci_dashboardApp/api/serializers.py file. Example:

```
...
 class CISerializer(serializers.ModelSerializer):
     jobs = JobSerializer(many=True, read_only=True)
     type = serializers.ChoiceField(choices=['JENKINS', 'TRAVIS'])
     access = serializers.ChoiceField(choices=['Private', 'Public'])

     class Meta:
         model = CI
         fields = "__all__"
...
```

On example is a serializer for CI tools. We add the list of Jobs inside it for better response and predefine values of fields: type and access. Also, the serializer will display all the fields of the model, this is defined by the line: fields = „___all___". Individual attributes for display can be defined here too.

4.1.2 Database and ORM

To work with the database, the standard Django ORM was used, additionally was installed the driver for the PostgreSQL database as described in section 2.3.4.

It is necessary to predefine the data for accessing the database in the Django configuration file. This configuration file is located in /backend/ci_dashboardSite/settings.py.

These settings define the essential parameters for connecting to PostgreSQL, for example: driver, user, password, database name. This list is presented below in the code snippet.

```
...
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': POSTGRES_DB,
        'USER': POSTGRES_USER,
        'PASSWORD': POSTGRES_PASSWORD,
        'HOST': 'dashboard-database',
        'PORT': 5432,
    }
}
...
```

Secondly, necessary to write the models in the Django ORM standard. The Job model is shown below in the code snippet.

```
...
class Job(models.Model):
    job = models.CharField(max_length=60)
    path = models.CharField(max_length=60)
    ci = models.ForeignKey(CI, on_delete=models.CASCADE,
                           related_name="jobs")

    class Meta:
        constraints = [
            UniqueConstraint(fields = ['job', 'ci', 'path'],
                             name = 'oneCI_oneJob'),
        ]

    def __str__(self):
        return self.job
...
```

In this snippet, in code format, we defined the database table as previously presented in section 3.2.4, with CI_tool-Job_name-Path unique constrain. The rest of the work is done by ORM queries and serializers.

4.1.3 Authentication

Previously in the section 2.3.1 was shown that we will use the LDAP technology for the purposes of user authentication. For that we need the django-auth-ldap library. In addition to the LDAP, authentication will also be session-based.

Firstly, it is necessary to make the required settings. In settings.py we added:

```
...
AUTHENTICATION_BACKENDS = (
    'django_auth_ldap.backend.LDAPBackend',
)
```

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.SessionAuthentication',
    ),
}
...
```

In this fragment, we determined that the application will use Sessions and authenticate through the LDAP. Next, needed to configure each of these elements:

```
...
SESSION_EXPIRE_AT_BROWSER_CLOSE = True
SESSION_COOKIE_AGE = 5 * 60
...
```

In this fragment, we determined that after five minutes, the session cookie will expire and after the browser closing too.

We also set up ldap:

```
...
AUTH_LDAP_SERVER_URI = "ldap://ldap.corp.redhat.com:389"
AUTH_LDAP_USER_SEARCH = LDAPSearch("DC=redhat,
                                   DC=com",
                                   ldap.SCOPE_SUBTREE,
                                   "(uid=%(user)s)")
AUTH_LDAP_START_TLS = True
...
```

In this fragment, we determined LDAP server address, the query and necessary TLS settings. All the necessary settings were made, then we needed to write methods for login and logout. These methods are very simple. They use basic orders like login and logout.

The login method additionally uses an authentication order, which allows the user to be authenticated with username and password parameters thought predefined in settings file authentication backend.

```
...
class LDAPLogin(APIView):

def post(self, request):
    user_obj = authenticate(username=request.data['username'],
                            password=request.data['password'])
    login(request, user_obj)
    return Response({'detail': 'User logged in successfully'}, status=200)
...
class LDAPLogout(APIView):
permission_classes = (IsAuthenticated,)

def post(self, request):
    logout(request)
    return Response({'detail': 'User logged out successfully'}, status=200)
...
```

The logout method additionally has permission_classes, that setting allows us to limit the number of users only to those who are already authenticated.

After authentication, the signal for changing the User object will pass. User passwords will be additionally deleted because we do not want to hold any password in our system, and administrators will be assigned.

The result of these processes is shown in the Figure 4.1.


```
Initiating TLS
search_s('DC=redhat,DC=com', 2, '(uid=%(user)s)') returned 1 objects: uid=mabramov,ou=users,dc=redhat,dc=com
Creating Django user mabramov
Populating Django user mabramov
```

Figure 4.1: LDAP authentication logs. Figure Mikhail Abramov.

4.1.4 CI tools requests and responses

Earlier in the section 3.2.1, the main methods of communication with the CI tools were studied, and several experiments were carried out to communicate with them through the CLI instruments.

As planned, the structure of the functions looks like: there is the primary function for selecting the necessary data (assembly of the URL) – processCI. Then, depending on the required CI tool, one of the two currently existing functions (getJenkinsJobStatus, getTravisJobStatus) is called to make requests to these tools. Regardless of the system, the result obtained is sent to the mapper (mapStates), where the results, be they colours or values in Jenkins or Travis CI respectively, replaced with the required designation.

Thus, it is possible to quickly process both group requests and individual ones. And the system has the necessary ability to expand to new CI tools.

4.2 Client-side implementation

This section focuses on client-level implementation. This part is divided into two main areas: user interface and communication between client and server.

4.2.1 Graphical User Interface

To build a graphical user interface were used capabilities of the material-ui [16] framework for ReactJS. Material-ui is a set of React components that Google Material Design implements. These components work in isolation, which means that they are self-supporting and inject only the styles they need to display.

We divided the structure of the frontend project into several components:

- utils – contains several useful constants

- themes – includes styles for light and dark theme and additionally includes modal popup styles

- pages – contains three main pages: CItools, Dashboard and Tokens

- models – includes many interfaces and objects used within the application

33

- config – this part defines routing

- components – the most significant part contains such elements as footer and header, as well as menus and all modals and tables

In our opinion, there are two areas in the GUI that we would like to consider in more detail: Modals and React-router-dom

Modals

The application makes extensive use of modal popup structures. The simplest example of such a popup element is the DeleteModal object shown in Figure 4.2.
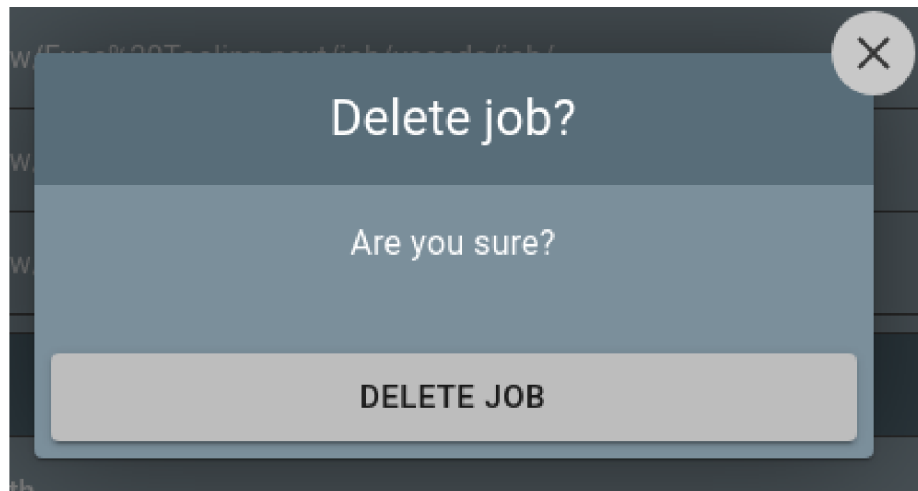

Figure 4.2: Popup window. Figure Mikhail Abramov.

All buttons used for adding, changing, deleting, as well as login and logout use such systems. The structure of the formation of this object originates in the basic definition of what a Modal is.

```
export const Modal: React.FC<ModalProps> = ({
    onBackdropClick,
    children
    }) => {
        return ReactDOM.createPortal(
          <Overlay onClick={onBackdropClick}>
            <div onClick={(e) => e.stopPropagation()}>{children}</div>
          </Overlay>,
            document.getElementById('modal-root')!
        );
};
```

At this moment, we are using DOM routing with portal[22] technology. This technology is used to render children into a DOM node outside the parent component's DOM hierarchy. With the help of styles, modal overlaying the surrounding objects and function onBackdropClick allows us to close the window and return to the current routing branch when we click beyond the modal window.

---

[22]React: Portals: https://reactjs.org/docs/portals.html

The next step in building a popup window is BaseModalWrapper:

```
...
return (
  <Modal onBackdropClick={onBackdropClick}>
    <ContainerComponent>
      <CloseButtonComponent onClick={onBackdropClick}>
        <CloseSign />
      </CloseButtonComponent>
      {content}
    </ContainerComponent>
  </Modal>
);
...
```

At this moment, we are already filling the portal created before with components, or instead with container shapes and such improvements as the close icon. The close sign is at the same time a button that activates the already familiar function onBackdropClick to close the window.

Then in the RWDModal object, we limit the window images. If the screen is too narrow, then at this stage, the window will be hidden.

```
...
<MediaQuery minWidth={580}>
  {(matches) =>
    matches ? (
      <BaseModalWrapper
        CloseButtonComponent={DesktopCloseButton}
        ContainerComponent={DesktopModalContainer}
        {...props}
      ></BaseModalWrapper>
    ) : (
      ''
    )
  }
</MediaQuery>
...
```

This functionality is implemented using the MediaQuery hook. It listens for matches to a CSS media query. It allows the rendering of components based on whether the query matches or not. It is possible to expand it to a mobile version, but we do not see the need to create a mobile version for this application.

This way, we have prepared the base for all modal pop-ups. Next, we simply fill the RWDModal with content in each component:

- CIModal – responsible for the functionality of adding and updating CI tools

- JobModal – responsible for the functionality of adding and updating Jobs

- TokenModal – responsible for the functionality of adding and updating Tokens

- LoginModal – responsible for the Login add Logout functionality

- DeleteModal – responsible for all objects deletion

React-router-dom

What is React-router-dom? In accordance with source[23] it is a tool that allows us to handle routes in a web app, using dynamic routing. Dynamic routing takes place as the app is rendering on user machine, unlike the old routing architecture where the routing is handled in a configuration outside of a running app.

To create the routing architecture, a separate file was created with a list of RouteItem objects, previously created as a model.

```
...
{
    key: 'router-dashboard',
    title: 'Dashboard',
    tooltip: 'Dashboard',
    path: '/',
    enabled: true,
    component: Dashboard,
    icon: DashboardIcon,
    appendDivider: true,
},
...
```

This RouteItem is a route to the base URL „/" and, as a component, contains a page - Dashboard. Further, this list is used during application rendering in the App.ts file. The main part of the system routing is located there.

```
...
<Router>
  <Switch>
    <Layout toggleTheme={toggle} useDefaultTheme={useDefaultTheme}>
      {routes.map((route: RouteItem) =>
        route.subRoutes ? (
          {/* skip the sub-routing part */}
        ) : (
          <Route
            key={`${route.key}`}
            path={`${route.path}`}
            component={route.component || DefaultComponent}
            exact
          />
        )
      )}
    </Layout>
  </Switch>
</Router>
...
```

---

[23]What is a React router? https://www.educative.io/edpresso/what-is-a-react-router

At this stage, for each item from the RouteItem list, we form a route. Swith object is responsible for rendering, renders the first child Route that matches the location.

4.2.2 Frontend-Backend communication

For communication between frontend and backend we use axios[24] tool. Axios is a lightweight HTTP client based on the $http service within Angular.js v1.x and is similar to the native JavaScript Fetch API. Axios is promise-based, which gives the ability to take advantage of async and await for more readable asynchronous code.

When the application is launched, the basic setup of the tool takes place:

```
...
axios.defaults.withCredentials = true;
axios.defaults.xsrfCookieName = 'csrftoken';
axios.defaults.xsrfHeaderName = 'X-CSRFToken';
axios.defaults.headers.common['Authorization'] = 'sessionid';
...
```

With these settings, we define that Axios will always send credentials for all requests. Among the credentials data, there will be CSRF[25] token and a Session token. Similarly, we describe the base URL depending on how the frontend was launched. These tokens saved after launch and authentication are shown in Figure 4.3.

Name	Value
sessionid	v8hoe42ud8lg44m64dkktnhbn0ecfw9e
csrftoken	nzIN1tTN69ndga2xhB35Vc1SNpwxKR4Svq...

Figure 4.3: CSRF and Session tokens. Figure Mikhail Abramov.

To demonstrate communication, we return to the DeleteModal modal window shown earlier. This window is responsible for confirming and sending a request to delete objects. Example of a delete request code:

```
...
axios
  .delete('/api/${aim}/${id}/', {
    withCredentials: true,
  })
  .then(() => {
    onBackdropClick();
    window.location.reload();
  });
...
```

In this code snippet, a DELETE request is sent, then, if successful, the window is closed, and the current position of the route is re-rendered. An example of a request is shown in Figure 4.4. The rest of the requests work similarly. Communication passes from React

---

[24]How To Use Axios with React: https://www.digitalocean.com/community/tutorials/react-axios-react

[25]CSRF tokens: https://portswigger.net/web-security/csrf/tokens

Figure 4.4: DELETE request. Figure Mikhail Abramov.

application through the Nginx layer and then into the WSGI Django system, where the request is processed and response formed.

4.3 Deployment

This section is devoted to deploying an application in containers, using Docker, docker-compose, Nginx and OpenShift.

4.3.1 Docker

The Docker engine was used to form an environment where each part of the application will work autonomously and can potentially be used in a cloud container.

Two Dockerfile config files have been implemented. The first was for the backend and the second for the frontend. The database layer does not require special preparation of the container.

The installation of the container for the backend consists of two parts and the final preparation for the OpenShift. Firstly, we use the python:3 image as a basis and install the RedHat certificates necessary for communication with the LDAP.

```
FROM python:3
MAINTAINER Mikhail Abramov

RUN curl -ks 'https://password.corp.redhat.com/legacy.crt' \
        -o '/usr/local/share/ca-certificates/legacy.crt'
RUN curl -ks 'https://password.corp.redhat.com/RH-IT-Root-CA.crt' \
        -o '/usr/local/share/ca-certificates/RH-IT-Root-CA.crt'
RUN /usr/sbin/update-ca-certificates
```

Then, using the package installers, we install the necessary programs and libraries, copy the contents of the backend directory and use the pip utility to install the required python libraries (this part is not shown). In the final part, we make the necessary settings for users and groups to avoid errors when launching a cloud container.

```
RUN chgrp -R 0 /app && \
    chmod -R g=u /app
```

As a result, we get a ready-made container for working with both docker and OpenShift.

The container for the frontend is more interesting. Consists of two stages and two basic images. The first part is based on the image of node:10 and is responsible for building the React project.

```
FROM node:10 as build
MAINTAINER Mikhail Abramov
```

```
WORKDIR /app
COPY . /app/
RUN yarn install
RUN yarn build
```

The second part is based on the Nginx:stable-alpine image, and in the process, we copy the Nginx settings and the built frontend project from the first stage. Then it is enough to launch Nginx, and the container for the frontend is ready.

```
FROM nginx:stable-alpine as frontend
MAINTAINER Mikhail Abramov

COPY nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=build /app/build /usr/share/nginx/html
CMD ["nginx", "-g", "daemon off;"]
```

The final task is to create the connections between these autonomous containers. This task can be solved by the docker-compose tool. In the root directory of the project, we create a file docker-compose.yml. Its main and most important part for us is services. Here we define three services: dashboard-database, dashboard-frontend, dashboard-backend.

```
dashboard-database:
    image: dashboard-database:latest
    container_name: dashboard-database
    image: postgres
    env_file:
        - .env
```

Using the postgres image and the .env variable file we have previously defined, this part of the code will create a dashboard-database container with an operational database ready for use. Next, we create a frontend container using the Dockerfile we already know.

```
dashboard-frontend:
    image: dashboard-frontend:latest
    container_name: dashboard-frontend
    build:
      context: ./frontend
      dockerfile: Dockerfile
    ports:
      - 80:80
    depends_on:
      - dashboard-backend
```

Build defines the directory and name of the dockerfile to create the container. Ports is command to expose ports HOST:CONTAINER, so frontend container will be accessible from HTTP post. Command depends_on means that the frontend will wait till the backend is deployed. The features of the service for the backend are the commands expose, entrypoint and waiting for the database's launch.

```
    ...
    entrypoint: ./wsgi-entrypoint.sh
    expose:
      - 8000
```

Expose opens ports for other containers, and the entrypoint defines the task when the container is initialized. As a result, we have a system of ready-to-use containers as shown in Figure 4.5, and it is enough to go to localhost in the browser to start working with the system.

```
[mabramov@mabramov VUT-ITT]$ docker ps
CONTAINER ID   IMAGE                     COMMAND                 CREATED        STATUS          PORTS
 NAMES
3b1106f7c1f4   dashboard-frontend:latest  "/docker-entrypoint.…"  10 hours ago   Up 53 minutes   0.0.0.0:80->80/tcp, :::80->80/tcp
 dashboard-frontend
e1cf32511eae   dashboard-backend:latest   "./wsgi-entrypoint.sh"  10 hours ago   Up 53 minutes   8000/tcp
 dashboard-backend
3724fecbe10e   postgres                   "docker-entrypoint.s…"  14 hours ago   Up 53 minutes   5432/tcp
 dashboard-database
```

Figure 4.5: Containerized application. Figure Mikhail Abramov.

4.3.2 Nginx

This section is devoted to the description of the Nginx configuration file. As mentioned earlier, Nginx will make the server more efficient and simplify the processing of requests. Let's take a closer look at the server configuration

```
listen 80;
server_name _;
server_tokens off;
client_max_body_size 20M;
```

Our server will listen on HTTP port – 80. We do not define name for server. We switch off option to show server version on error pages. The last setting is to set maximum request size. This means that requests larger than 20MB will result in error with HTTP 413. Next, we have four location block defined which specify configuration for each URL. Firstly, location of our frontend application:

```
location / {
    root /usr/share/nginx/html;
    index index.html index.htm;
    try_files $uri $uri/ /index.html;
}
```

Secondly we define routes to proxy_api (all request will be redirected to proxy_api):

```
location /api {
    try_files $uri @proxy_api;
}
location /admin {
    try_files $uri @proxy_api;
}
```

And the last one is our backend for redirection:

```
location @proxy_api {
    ...
    proxy_pass http://dashboard-backend:8000;
}
```

This small setting allows us to run a simple Nginx web server.

4.3.3 Openshift

At the first stage, a database was prepared using an application from the OpenShift catalogue. In the installation process, a deployment configurations was created, and the service became available to other containers thought name dashboard-database and port 5432. The container configuration is shown in Figure 4.6.
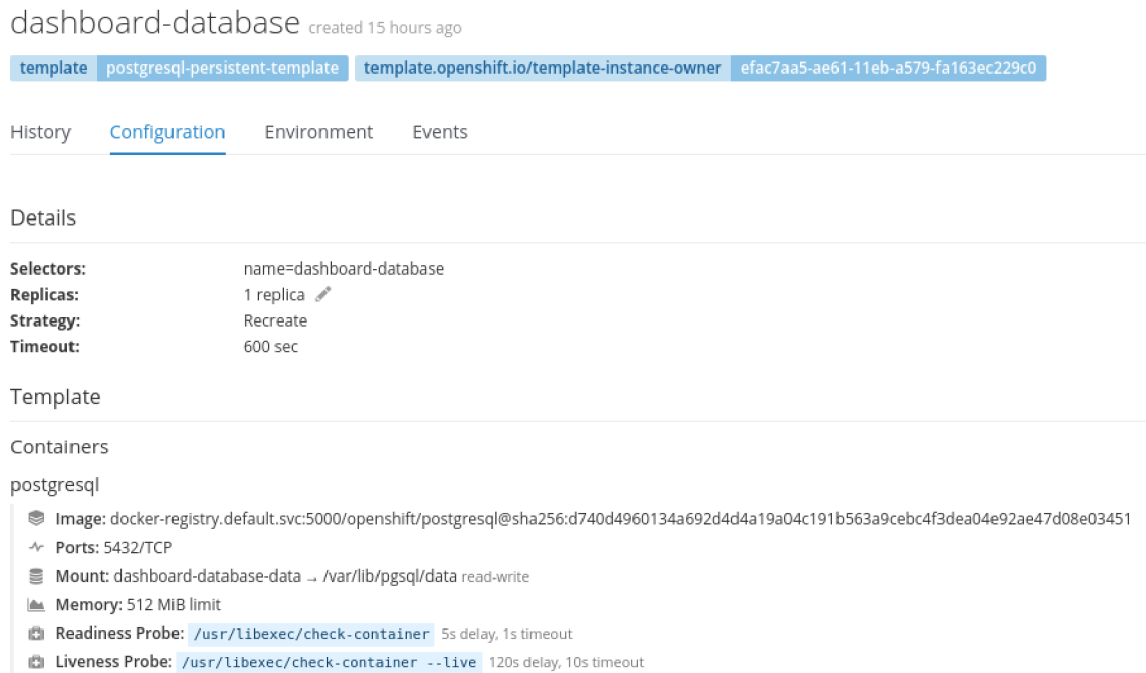
Figure 4.6: Containerized database. Figure Mikhail Abramov.

The next step is to install backend and frontend containers in the OpenShift, four files were generated using the Kompose[26] tool.

- openshift/dashboard-backend-deploymentconfig.yaml

- openshift/dashboard-backend-service.yaml

- openshift/dashboard-frontend-deploymentconfig.yaml

- openshift/dashboard-frontend-service.yaml

The received deployment configurations for the backend and frontend were uploaded to the OpenShift system. Following them, the services configurations were uploaded for the respective containers. The Figure 4.7 shows a backend service with an added route for creating a domain (not necessary).

Installing the frontend container is entirely similar to installing the backend container, but a mandatory element is establishing a route for creating a domain.

---

[26]Kompose (Kubernetes + Compose): https://github.com/kubernetes/kompose

41
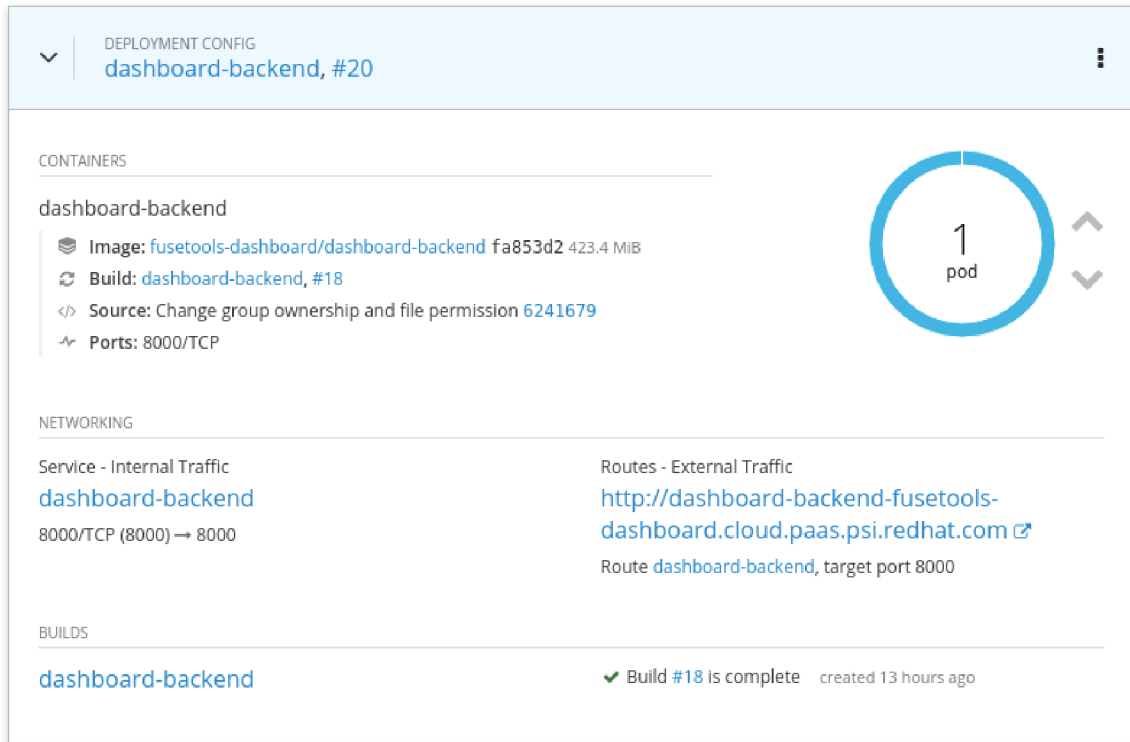
Figure 4.7: Containerized backend. Figure Mikhail Abramov.

4.4   User experience review

For the process of collecting feedback, two groups of users were identified: managers and developers. The solution was demonstrated to representatives of these groups. Then the representatives conducted independent testing without a predefined questionnaire and framework since each of the users possesses the necessary technical skills, and the task itself presented these frameworks. The number of testers was two, one for each group, respectively. Testers represented the team for which this program was developed. Four demos were prepared. Standalone demo with an alternative option to run in a container for testing backend functionality exclusively within the Django REST framework. Standalone demo for frontend for local work with frontend. And a full containerized demo to evaluate the work from the user's point of view.

After demonstrating the application to colleagues and their independent work with the application, comments and recommendations appeared. There were three strong recommendations that we consider as potential extensions to the existing application.
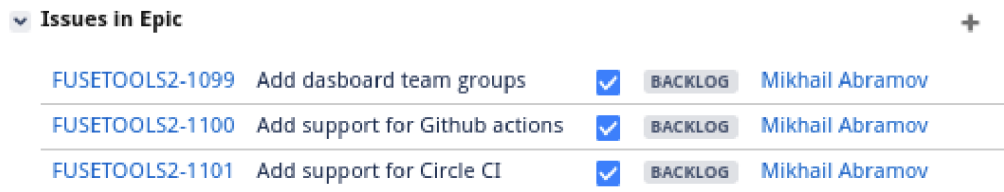
- Add dasboard team groups

- Add support for Github actions

- Add support for Circle CI

Can anyone from RedHat login? Yes, everyone can enter. Therefore, it is possible to change the system of public and private instruments to a system of private and command instruments. Can a person be on several teams? Yes, it means that it is necessary to provide for the ability to view both individual commands and all at once. Is it easy to

42

expand the system? Yes, it is easy to slightly change the database, backend and frontend, but everything architectural is already there. How to work with teams? The easiest option is manual through the administrator, the option is more difficult to integrate the Rover corporate system for managing teams.

As planned, by the time the work on the bachelor's thesis was completed, it became necessary to expand the system by two instruments at once. These tools were considered at a theoretical level, and their addition to the system seems realistic. Thus, it is necessary to add two additional functions for sending requests to the backend and expand the mapper.

Based on the recommendations received and requests for expansion, an epic was created in the Jira system and these tasks were added there as shown in Figure 4.8.

Figure 4.8: Continuing work on the project. Figure Mikhail Abramov.

Also, some inaccuracies in the code will be eliminated. The UNSTABLE status will be added for all CI systems. Styles will be improved as we find non-standard object behaviour.

Chapter 5

Conclusion

To solve defined tasks and questions were studied modern methods of web application development. It was decided to use a strict separation of the application up to three levels: the client level, the server level and the database level. The client level was implemented using the ReactJS framework based on the TypeScript language, the server level was implemented using the Django REST framework, and PostgreSQL was used at the database level. For user authentication we used the corporate LDAP server. Authentication is sessions based. For the integration of continuous integration services, the API of each service was used. Docker used to create each container separately. Docker-compose technology used to facilitate the automatic creation of persistent links between containers. After successfully launching of the application locally and in containers, we made attempt of deployment on OpenShift service. During development, the ideas of the potential future extensions were produced and accepted for further work.

During the development of the application, we have achieved notable success. The goal of the work has been completed. As a result, the team has a web application divided into the planned three parts, accessible to containerization both locally via docker and cloudy via OpenShift. Feedback from users was received, suggestions for improvement were made, the prototype was accepted, and the application will be developed in the future.

As for me, this bachelor thesis was an exciting broad range of technologies used. I learned a new language for me - TypeScript, new containerization methods for me - docker-compose and Openshift, many frameworks and solutions. I am pleased with positive feedbacks and even more with feedbacks that carry recommendations and extensions.

Bibliography

[1] *DevOps: What is CI/CD?* [online]. [cit. 2021-04-22]. Available at:
https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[2] BELMONT, J.-M. *Hands-On Continuous Integration and Delivery.* Packt Publishing
Ltd., 2018. ISBN 9781789130485.

[3] PATHANIA, N. *Learning Continuous Integrations with Jenkins.* Packt Publishing
Ltd., 2016. ISBN 9781785284830.

[4] *Jenkins User Documentation* [online]. [cit. 2021-04-23]. Available at:
https://www.jenkins.io/doc/.

[5] *Travis CI User Documentation* [online]. [cit. 2021-04-23]. Available at:
https://docs.travis-ci.com.

[6] *CircleCI Documentation* [online]. [cit. 2021-04-23]. Available at:
https://circleci.com/docs/.

[7] *Product: GitHub Actions* [online]. [cit. 2021-04-23]. Available at:
https://docs.github.com/en/actions.

[8] TUTTLE, S., EHLENBERGER, A., GORTHI, R., LEISERSON, J., MACBETH, R. et al.
Understanding Ldap - Design And Implementation. IBM Redbooks, 2004. ISBN
073849786X.

[9] *OpenLDAP: community developed LDAP software* [online]. [cit. 2021-04-23]. Available
at: https://www.openldap.org/.

[10] *Django Software Foundation : The web framework for perfectionists with deadlines |
Django* [online]. [cit. 2021-04-23]. Available at: https://www.djangoproject.com/.

[11] ELMAN, J. *Lightweight Django.* O´Reilly media, 2014. ISBN 9781491945940.

[12] RIPPON, C. *Learn React with TypeScript 3.* Packt Publishing Ltd., 2018. ISBN
9781789610253.

[13] *TypeScript: React* [online]. [cit. 2021-04-24]. Available at:
https://www.typescriptlang.org/docs/handbook/react.html.

[14] *Docker docs: Docker Engine* [online]. [cit. 2021-04-25]. Available at:
https://docs.docker.com/engine/.

[15] *Docker docs: Docker Compose* [online]. [cit. 2021-04-25]. Available at:
https://docs.docker.com/compose/.

[16] *Material-UI: A popular React UI framework* [online]. [cit. 2021-04-29]. Available at:
https://material-ui.com/.

[17] *OpenShift Container Platform Documentation* [online]. [cit. 2021-04-25]. Available at:
https://docs.openshift.com/container-platform/.

[18] *Jenkins: Remote Access API* [online]. [cit. 2021-04-25]. Available at:
https://www.jenkins.io/doc/book/using/remote-access-api/.

[19] *Travis CI – API Reference* [online]. [cit. 2021-04-25]. Available at:
https://docs.travis-ci.com/user/developer/.

[20] *Django REST framework* [online]. [cit. 2021-04-28]. Available at:
https://www.django-rest-framework.org/.

Appendix A

Dashboard UC diagram
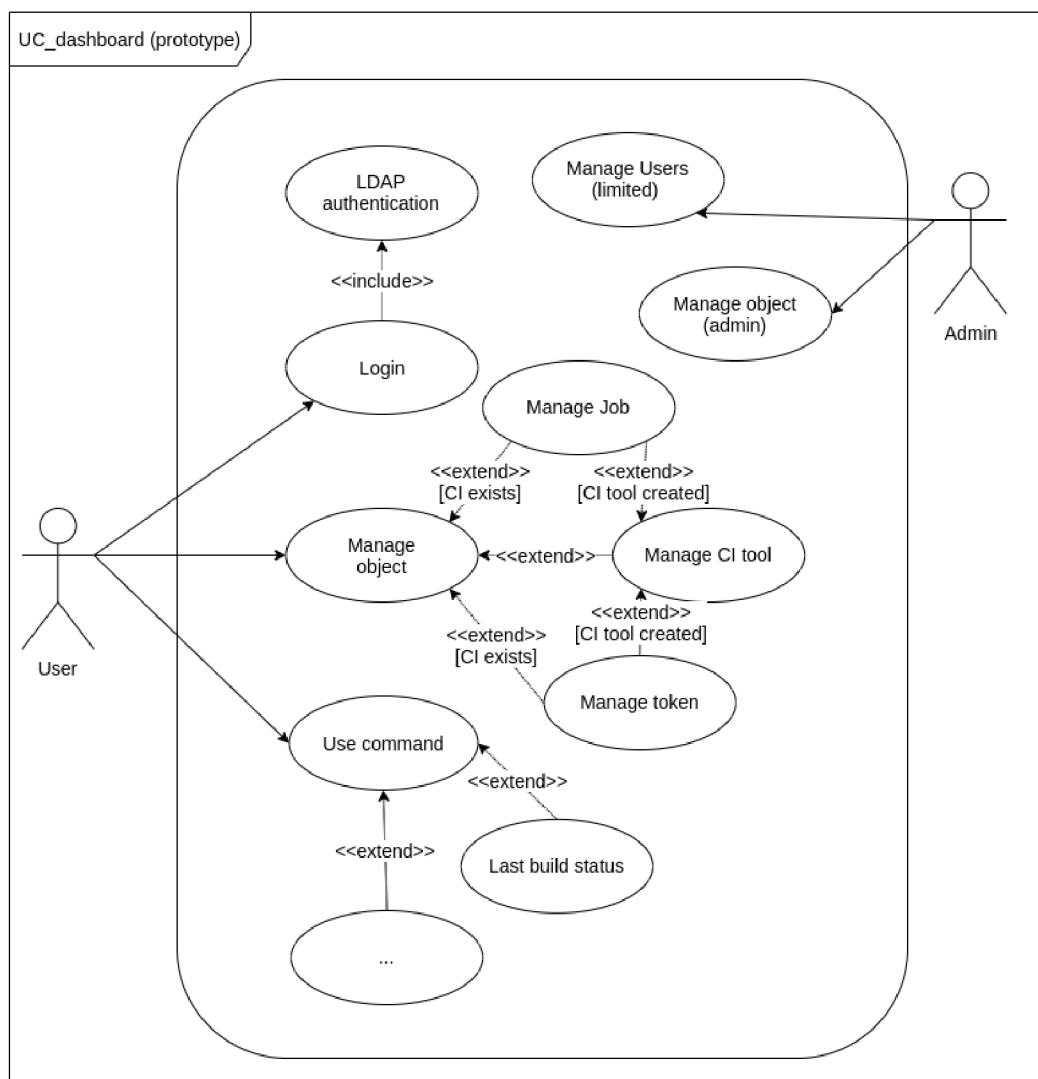
Figure A.1: Dashboard Use Case diagram.

Appendix B

Dashboard ER diagram
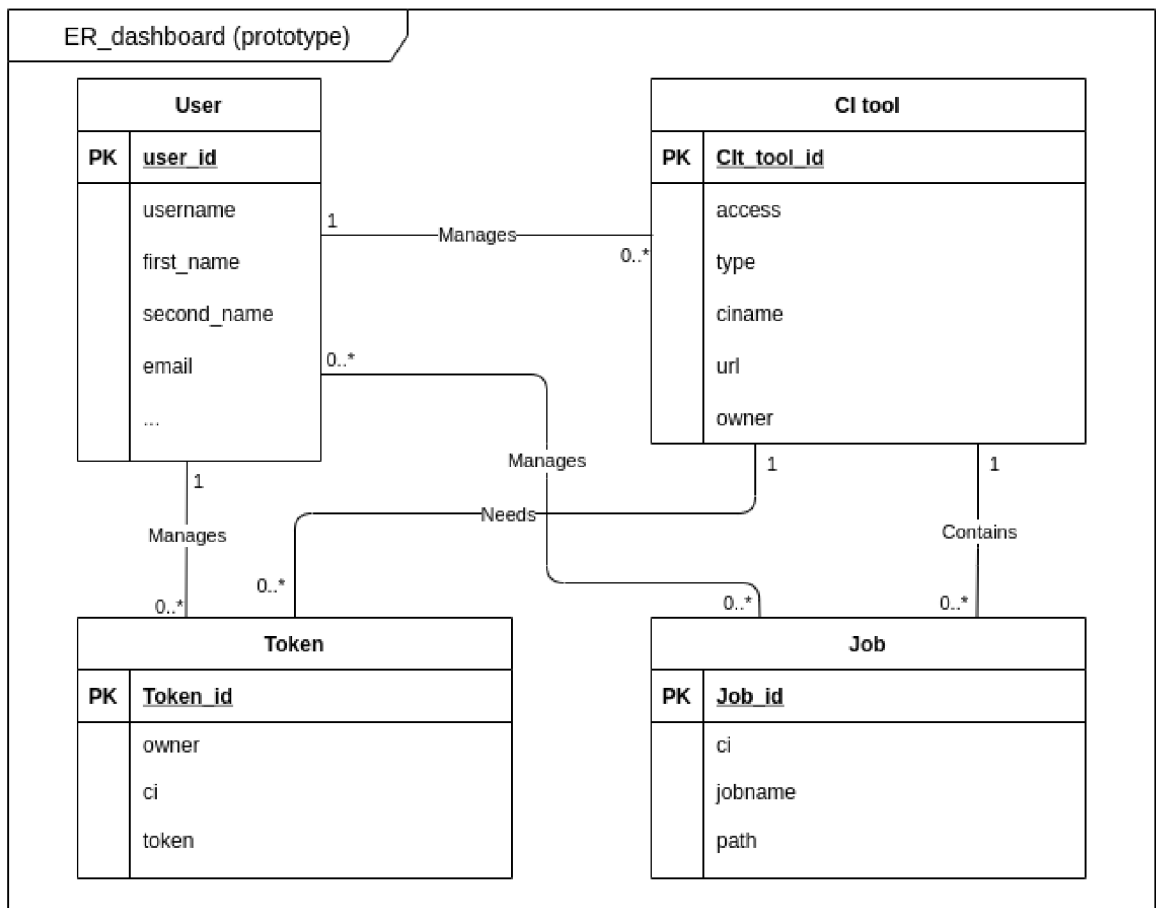
Figure B.1: Dashboard Entity Relationship diagram.

Appendix C

Dashboard Database diagram
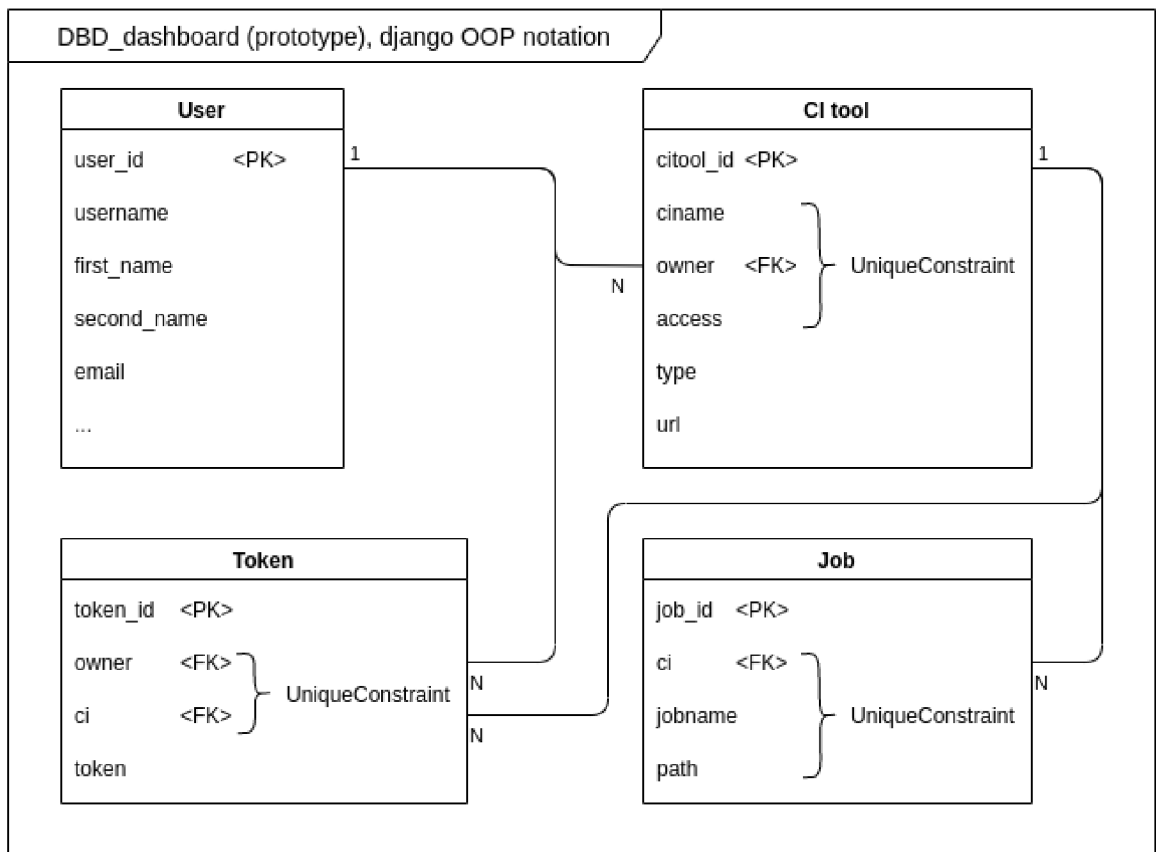
Figure C.1: Dashboard DataBase diagram.

Appendix D

Dashboard Deployment diagram
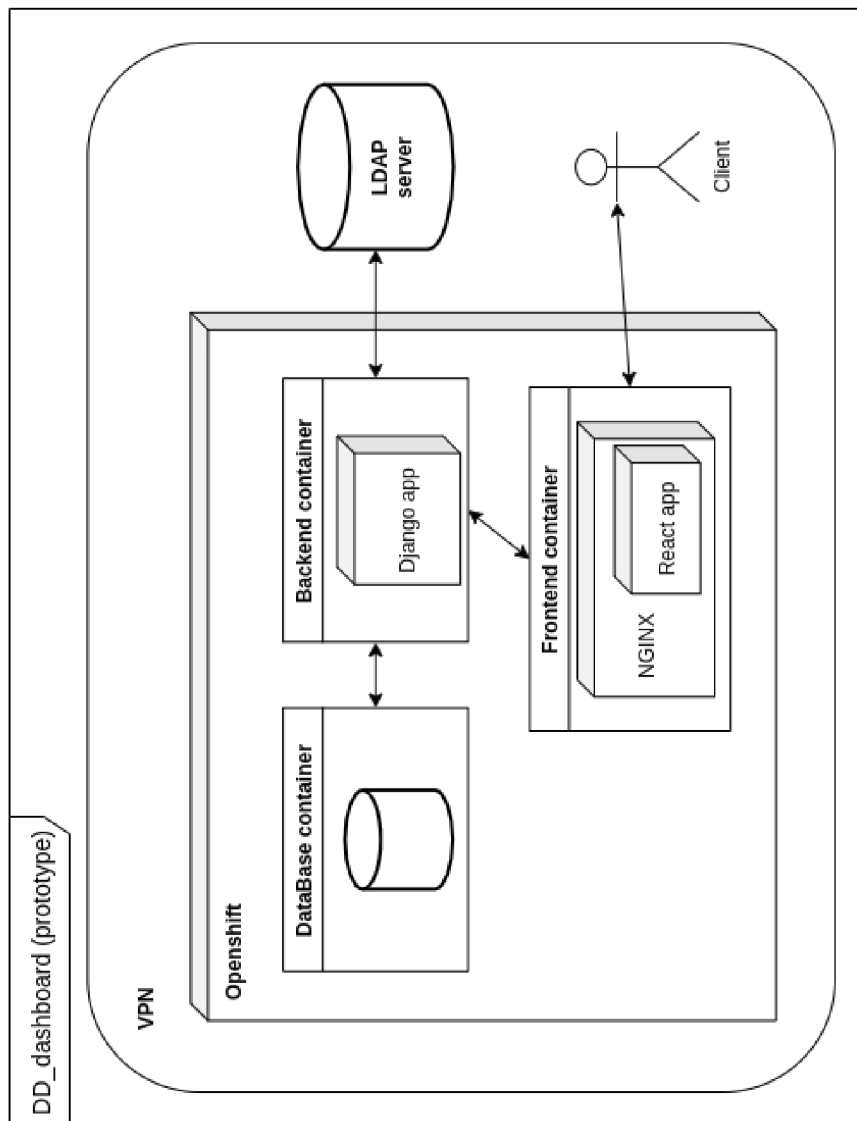
Figure D.1: Dashboard Deployment diagram.

Appendix E

GUI mockup
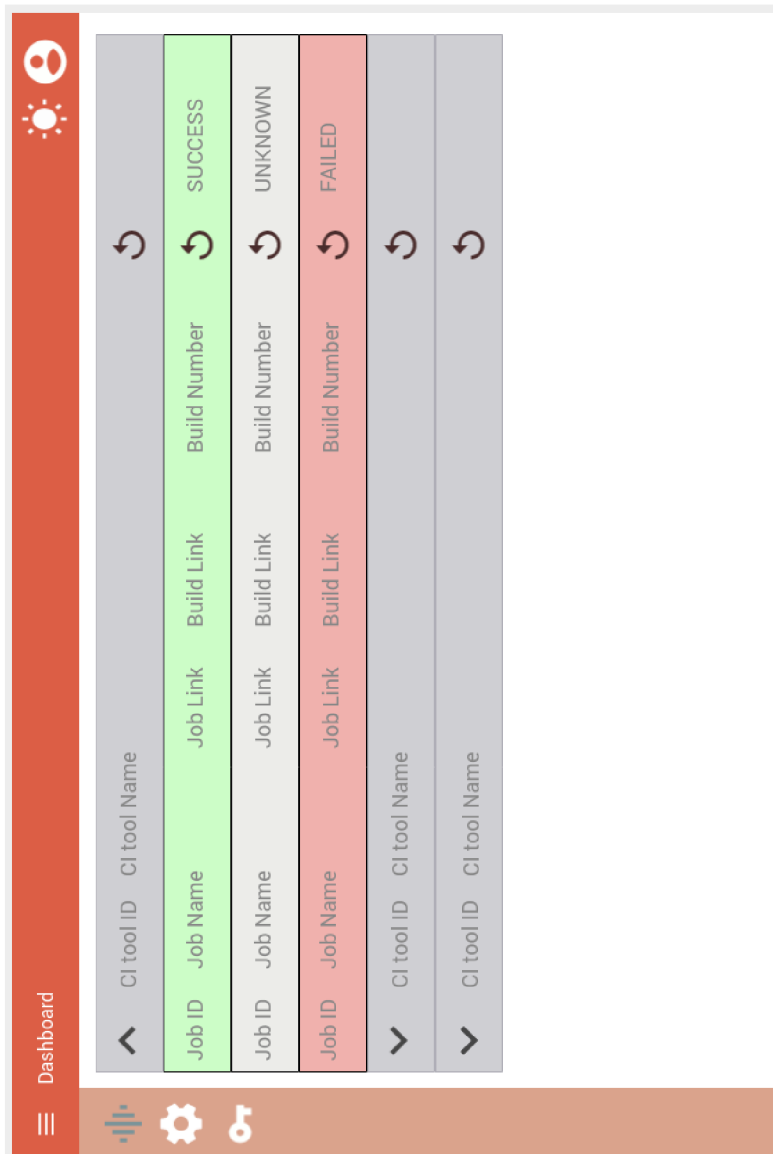
Figure E.1: Application mockup – Dashboard.
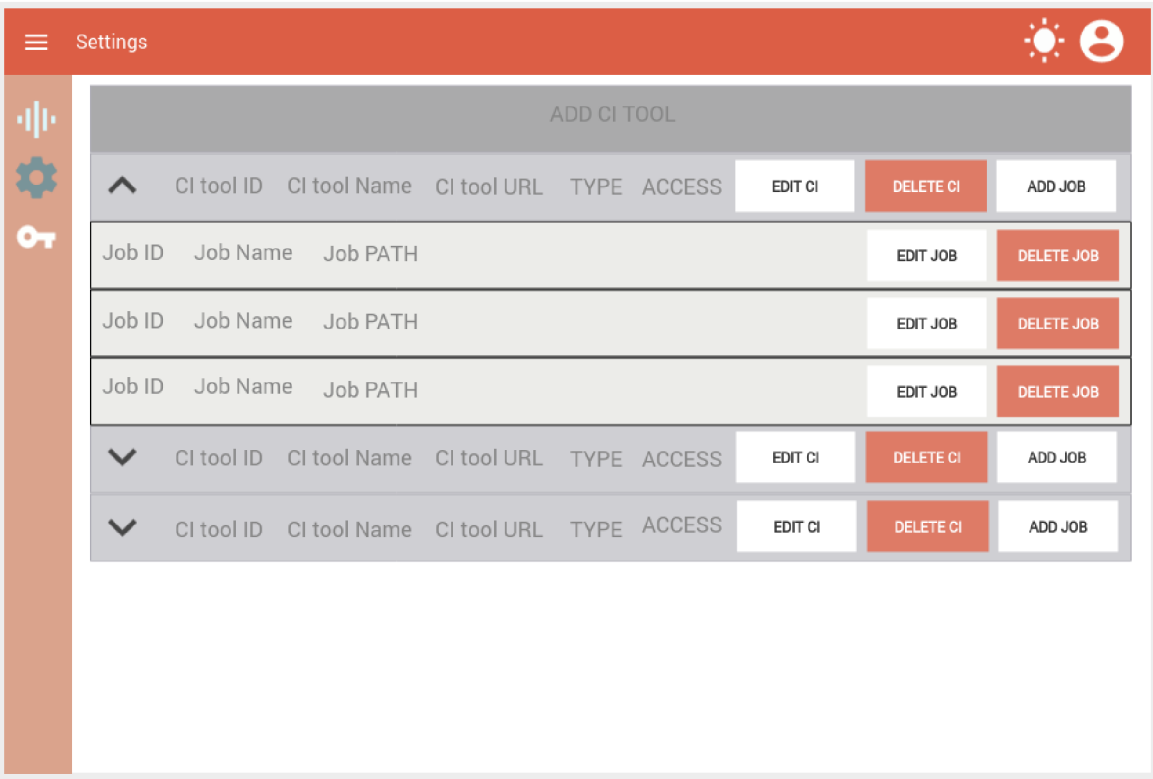
Figure E.2: Application mockup – Settings.
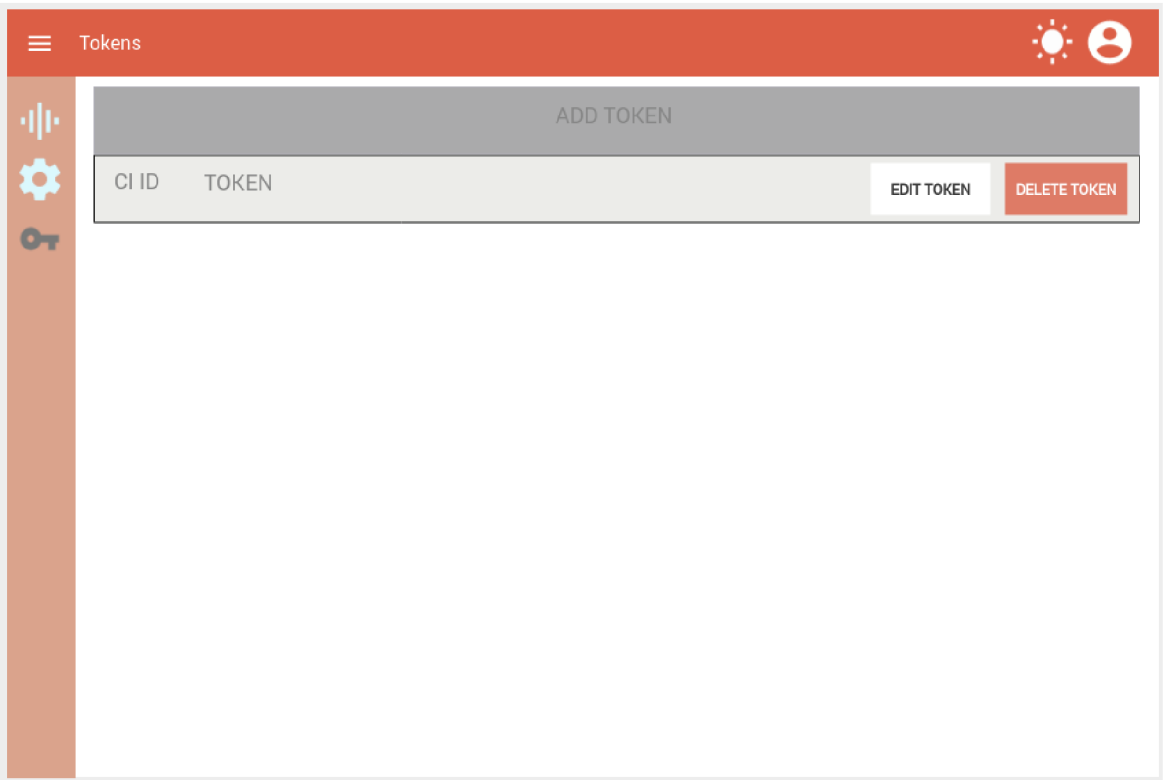
Figure E.3: Application mockup – Tokens.

Appendix F

Application GUI
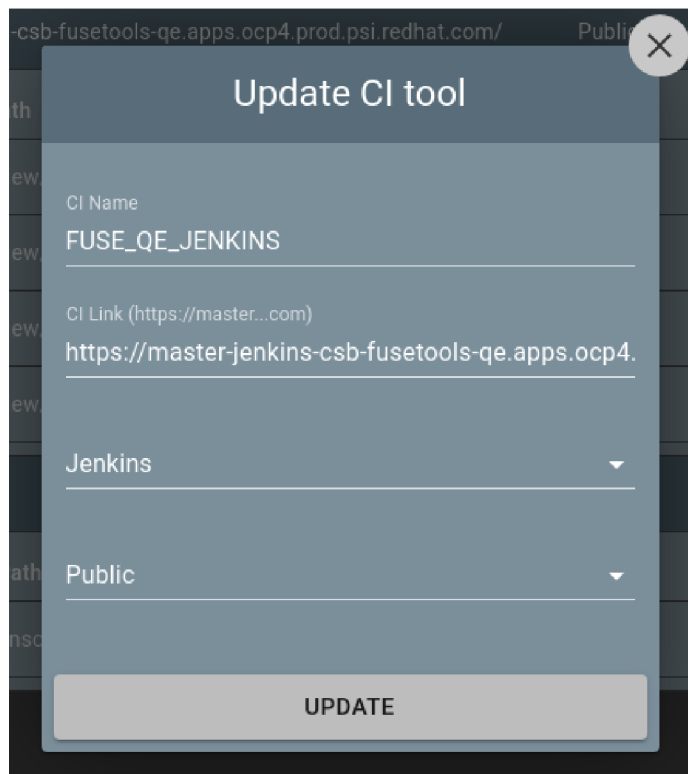
Figure F.1: Application – modal popup window to update CI tool.

Figure F.2: Application – Dashboard page.

Figure F.3: Application – Settings page.