Stash Backend

Bachelor Project in Web Development PBA

Written by
Gábor Pintér

Supervised by
Dany Kallas

Københavns Erhvervsakademi
December 18, 2017

Table of contents

1. Introduction

1.1 Abstract

This is a final report of the bachelor project called Stash Backend made for the Web Development PBA program at Copenhagen School of Design and Technology. Stash Backend is a web application targeting front-end and mobile developers by providing the most commonly demanded server-side operations and features such as database management and user authentication in a convenient, easy-to-use manner.

The report documents the design and development process of the product, backed up by both theoretical knowledge and the detailed description of my decisions and motivation. Its purpose is to provide a comprehensive overview of the Representational State Transfer (REST) architecture by introducing the necessary methodologies and by bringing theories into practice.

1.2 About the project

The purpose of the project is to introduce and demonstrate the implementation of a fully functioning RESTful API[1] alongside with its design principles and advantages in practice.

The document covers the historical background of web services, which is necessary to understand the relevance and importance of the REST structure. Starting by analyzing the problem area around web services, the report guides the reader through several scenarios, in which REST might be the most optimal solution for developing web services. This introduction is followed by the description of the Backend-as-a-Service (BaaS) software model which recently became a remarkably popular type of service, and which is nevertheless a perfect candidate for demonstrating the characteristics of REST.

The objectives of the report also include reflecting on developing a general purpose back-end in the shape of Stash Backend, which provides the following features: database management, user authentication and file storage. This presents a perfect opportunity to demonstrate the methodologies and expertise I have obtained throughout the Web Development PBA modules, such as Development Environments, Databases and Web Security. When choosing a problem area for my bachelor project, I seeked for a subject that was a great balance between my acquired skills and unknown technologies. This let me to utilize the proficiency I have acquired through my education which resulted in a relatively fast pace of development, but in the meantime I also had the opportunity to discover new areas of web development and face new challenges, which gave the project a great, exciting flavour.

[1] Application Program Interface

1.3 Reading guide

The report consists of 7 chapters, each describing different states of the development process.

The 1. Introduction includes a quick overview of the project and introduces chapters of the report. This chapter is followed by the 2. Project formulation, in which I describe the sources of my motivation and explain what inspired me to build Stash Backend. This chapter also encompasses a problem formulation and the project goals. After defining the project objectives, the scope and the tasks of the project are described alongside with their limitations.

In the chapter 3. State-of-the-art and trends I describe relevant background knowledge about web services and the Software-as-a-Service models gathered from both studied modules and external sources. The purpose of this section is to discover and compare various approaches and solutions in order to pick the one that suits the project the best.

In the following chapter named 4. Solution design I start focusing on the implementation of the solution: a detailed software design plan is delineated. In this plan I describe the languages, tools and environments and I have chosen for the project and explain how these components are interconnected to form a whole. This chapter also covers the design principles I have applied on the project such as Object-Oriented Programming and the Model-View-Controller pattern.

The solution design is followed by the chapter 5. Solution implementation, where I demonstrate various programming practices backed up by examples from the source code of the application. In this section I bring examples from each component of the application, illustrating how I solved various challenges in the fields of REST resource management, database management and web security.

The next chapter named 6. Reflection includes my final conclusion on the project. Here I also mention a number of future improvements and introduce the future perspectives of the project.

In the last chapter 7. Appendices the reader can find a reference of the application's source code, a quick installation guide and the list of references mentioned in the report.

2. Project definition

The purpose of this chapter is to put the project into perspective by introducing my motivation, defining the problem area and then by composing a problem formulation in the shape of three focused questions.

2.1 Motivation

My personal motivation of choosing REST and BaaS as my subjects derives from choosing backend development as my primary expertise. Throughout the Web Development PBA program I have specialized myself in the field of front-end development; thanks to my deep interest in the Web Development and Mobile Development modules. This however came into change when at the last semester of the program I have started to seek for internship opportunities. I came into the realization, that in order to achieve my ultimate goal, which is to become a full-stack web developer, I have to start deepening my knowledge in server-side development as well.

During the internship program I managed to get familiar with various backend methodologies and structures such as SOAP[2] and REST[3]. I have obtained knowledge which reached beyond understanding how to use web services from a client perspective; I have gained practical knowledge about how to implement my own services. I found it fascinating how various services made by different companies and organizations can be connected under the surface through the web in order to provide the broadest functionality and best user experience possible. This made me realize and appreciate the importance of web services, which by today became an indispensable knowledge for every web developer.

2.2 Problem area

The demand for building web services arose in the early years of the World Wide Web. Initially, software only had a human-to-machine communication mechanism, where the machine relied on internal resources in order to produce the desired output. Developers however wanted to reach beyond building isolated software, and started developing systems which could transfer and exchange data with other systems on the World Wide Web. Hence the term "*web services*" was born.

With web services, developers can easily integrate third-party services and data within their own applications. It presents the developers the opportunity to integrate real-world location data to their applications using Google Maps or authenticate their users using Facebook Login.

[2] Simple Object Access Protocol
[3] Representational State Transfer

REST is one of the most commonly implemented architecture for web services. Numerous industry leading companies provide RESTful APIs as part of their services including Google (e.g.: Google Fit **[2.2.a]**), Microsoft (e.g.: Live SDK REST API **[2.2.b]**) and Apple (e.g.: Apple News API **[2.2.c]**). Thanks to the architecture's flexibility and standardized interaction model, REST quickly became an industry standard and therefore a must-to-know structure for every web developer. Beyond adapting to the industry standards which is nevertheless always a necessary action to take, REST also helps to:

- ð Develop well organized and structured backends for our own applications
- ð Understand how to integrate with third party APIs
- ð Develop and deploy our own web services

2.3 Problem formulation

I will put the project into frame by establishing a problem formulation in a shape of three key questions which are meant to be covered in the following chapters.

1. What are methodologies and constraints behind a REST architecture and what are their purposes?
2. How to apply these principles in practice to the most important web operations including user authentication, database interaction and file transfer?
3. How to secure a RESTful API?

2.4 Project objectives

It is important to put the project into perspective by defining its objectives. Meanwhile the ultimate goal of the project is to learn how to implement a functioning RESTful API is, the only way of evaluating its efficiency is to build an actual product for people that attempts to solve a real-world issue.

2.4.1 Purpose of the project

The purpose of the project is to obtain a comprehensive knowledge about RESTful APIs by building a fully functioning Backend-as-a-Service product, that serves as a backbone for applications built by mobile and front-end developers. Therefore, this general purpose backend called Stash Backend has to be able to handle the most basic backend operations such as user authentication and database management, and also provide a RESTful API for interaction.

2.4.2 Desired effect

Ultimately, the desired effect is to build a Backend-as-a-Service application which makes it easier and faster for mobile and front-end developers to build full-stack applications. The evaluation will therefore involve user tests made by developers, where besides evaluating the application's functioning, the product's usability and convenience will also play a significant role.

2.5 Scope of the project

Besides formulating the purpose and the key objectives of a project, it is extremely important to also take limitations into consideration, which have the following two main factors: time constraints and knowledge.

The project aims to deliver a fully functional prototype that demonstrates the main characteristics of REST by addressing and solving real web development challenges. The application should be able to run locally and solve real web development case scenarios. However, the final product of this iteration is not intended to be used in production, as some of the features might only serve demonstrational purposes, and the field of software deployment will not be in the scope of the project. Therefore from a web security perspective, the application is ought to demonstrate various implementations of security principles, but full security coverage is not expected.

2.5.1 Limitations

There are several limitations that have to be addressed in advance, in order to keep the project within feasible frames:

- **Time constraints** might compel the implementation to take shortcuts and leave some of the features unfinished. In this case, the end user should be informed about this either through this report or through the application itself.
- **Research limitations** in the market of Software-as-a-Services might result in the implementation of a rather less unique end product. As the purpose of this project is to introduce REST, providing a unique, business-wise sustainable product will not be in focus.
- **Security limitations** will also be introduced, as the final application is only intended to be used for test purposes by running locally.
- **Visual design limitations** are also expected, as the main interaction between the end-user (developers) and the software will mainly take place through the API. Responsiveness, brand identity and user interaction design is not expected to be part of the project.
- **Report length constraints** limits the depth and detailness of the project description. Certain fragments of the implementation might not be described in details or might be ignored.

2.6 Planning and organization

The project implementation started with planning and organization. Stash Backend was a one-person project which had its own advantages and pitfalls, which had to be taken into consideration.

2.6.1 Source code version control

The development was based on using today's primarily dominant version control system, Git **[2.6.1.a]**, which proved to be a good choice due to its richness in features and numberless documentation and resources provided online. The code was backed up and hosted by GitHub **[2.6.1.b]**, which is a great option for open-source projects, as it provides clear visualization of the project's codebase and the project's tasks.

2.6.2 Project management

One of the advantages of a one-person project is that there is no need for communication channels; however a well-planned project management system remains necessary. I have decided to follow the agile development methodology **[2.6.2.a]**, where I divided the development into weekly sprints, each one containing tasks assigned for that given week. At the end of the weeks the tasks were evaluated, which helped to introduce new changes in structure or pace in the upcoming sprints.

During the implementation user tests have been done iteratively. In these user tests I have asked developers to perform a series steps in order to achieve a certain goal. Meanwhile the developers tried to accomplish their objectives, I have observed how they approached to solve the challenges, took notes and asked them to rate the convenience of the application. The results of the user tests have been documented in the attachment **Stash Backend - User tests [2.6.2.b]**.

As the development of the project proceeded, new ideas raised which did not fall into the priory category; in this case the issue was recorded in the backlog of the project as a *"nice-to-have"* feature, ready to be implemented as soon the MVP[4] is in place. For project management visualisation I have used GitHub Issues **[2.6.2.c]**, as this service was already provided along with a GitHub Repository, and its features happened to cover all my needs in the shape of labels, milestones, detailed issue descriptions and deadlines.

The application implementation was preceded by project planning and research, where ideas and learnings had to be documented. For this purpose I have chosen Google Drive **[2.6.2.d]** to work with, as it offers great, rich-in-features applications out of the box.

[4] Minimum Viable Product

3. State-of-the-art and trends

The goal of this chapter is to provide professional background knowledge and to familiarize the reader with the terms I am going to use throughout the report. This includes the history of web services, the most commonly used web service structures and the introduction of the Software-as-a-Service model.

3.1 Web services

In order to choose the right structure for our implementation we need to truly understand what web services are and how they work. W3C (World Wide Web Consortium) defines the term “web services” the following way: “*Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks.”* **[3.1.a]**. Simply put, a web service is a function that can and intended to be accessed by other programs over the web. It is targeted towards other programs and not humans.

3.1.1 Benefits of web services

Implementing and using web services have several benefits.

1. One of the advantages is that it can reduce the development time and expenses of a product for a given company.
2. Another benefit is integrity; with these services we have the opportunity to integrate with other applications, making our product more attractive for our users.
3. Web services can also provide solutions for content management, simply by providing data; whether it is data about the user itself, or data retrieved from a third party.

A quick, real-life example is the Facebook Login **[3.1.2.a]** service, which can be used to provide our users the possibility of registering themselves in our application using Facebook. This solution makes our application more appealing by making the sign up procedure easier and more familiar for the user. Furthermore, it presents us the opportunity to retrieve information about the users, such as their locations, friends and photos, which we would have not received otherwise.

3.1.2 Web service types

There are various ways of building a web service. However, according to W3C we can differentiate between two major classes of web services:

1. *"REST-compliant web services, in which the primary purpose of the service is to manipulate XML representations of Web resources using a* ***uniform set of "stateless" operations****; and*
2. *arbitrary web services, in which* ***the service may expose an arbitrary set of operations****."* **[3.1.3.a]**

Without going into too much detail, let's have a quick comparison between these two types. To understand the difference between these classes, let's first look at the common characteristics of these type of services:

- In both systems data (objects) are referred as *resources* and are identified using URIs (Uniform Resource Identifiers).
- Resources are exchanged in widely-known representational formats (XML, HTML, PNG etc.)

However, there are some key differences between these approaches: **arbitrary web services** (often referred as SOAP[5] services) use application-specific interfaces to describe the behaviour of the service. In other words; each SOAP service had its own arbitrary way of manipulating resources, which made the protocol rather verbose. Due to this, the slow parsing of its only messaging format XML and the lack of standardized interaction model, REST architecture began to dominate the field of web services.

REST (Representational State Transfer) is an architectural style for web services most commonly using HTTP. Unlike SOAP, REST provides uniform interface semantics in the shape of HTTP methods (POST, GET PUT, DELETE etc.) which are essentially equivalent to the basic CRUD operations (create, read, update, delete). REST operations are stateless, which means that all the necessary information about a given message is contained within the message itself, and the result (output) does not depend on the state of the communication. REST quickly became an industry standard thanks to its relatively easy implementation and standardized interaction model. On the other hand it is important to point out that popularity does not automatically implies that REST should be used over SOAP in every cases, as SOAP also have its own benefits including (but not limited to):

- It can work on any communication protocol (such as FTP, TCP or HTTP)
- It includes security and authorization as part of the protocol (see WS-Security **[3.1.3.b]**)
- It provides a protocol that guarantees SOAP message delivery with successful/retry logic built in (see WS-ReliableMessaging **[3.1.3.c]**)

[5] Simple Object Access Protocol

Due to these advantages over REST, SOAP is an ideal choice and commonly used solution for financial services (e.g.: as PayPal SOAP API **[3.1.3.d]** or Salesforce SOAP API **[3.1.3.e]**) and payment gateways.

3.2 Software-as-a-Service

Today there is an increasing demand for Software-as-a-Service distribution models because of the many benefits they offer to companies of all sizes and types. By using a SaaS, any given company can easily lower the IT costs of hardware, software and people needed to manage these all.

According to W3C, *"SaaS is software that can be accessed over the web but is hosted and supplied by a third party vendor"* **[3.2.a]**. Meanwhile this definition might sound a bit vague, SaaS is really just a general term for a business model or software licensing which monetizes web services. It is licensed on a subscription basis and has various subcategories; for example Infrastructure-as-a-Service (e.g.: AWS **[3.2.b]**), Platform-as-a-Service (e.g.: Heroku **[3.2.c]**) or Backend-as-a-Service (e.g.: Firebase **[3.2.d]**).

I have chosen to build a Backend-as-a-Service because of three main reasons:

1. The **volume of a simple BaaS** is ideal for this project.
2. As a BaaS is indented to perform **general backend operations initiated by a wide range of developers**, it is a perfect candidate for demonstrating REST.
3. Convenience and ease are very important factors of a BaaS. Targeting these values makes it **easy to evaluate** the success of the project.

4. Solution design

The goal of this chapter is to formulate a detailed solution design which describes the required tools and the list of features of the end application. In order to achieve this, two former steps have to be taken: conducting a target audience analysis which will then afterwards help to formulate a list of functional and nonfunctional requirements. Based on these specifications, I will be able to pick the right development environments, languages and frameworks. By the end of the chapter we will have a clear vision about what features and services the application is ought to offer in order to evaluate the project successful.

4.1 Target audience

Stash Backend is intended to be used by software developers, more specifically mobile and front-end developers. The purpose of the application is to provide a tool for client-side developers, which they can use to build full-stack applications by only focusing on the client-side development of the application, and leaving the rest for Stash Backend to handle. This way their development pace speeds up dramatically, leaving them more resources on focusing what really matters for them.

4.1.1 Establishing the need

In order to target real-world issues and make the evaluation of the solution easier, I have decided to work closely with a small number of developers who acted as the target audience of the project and performed the evaluation.

1. An entrant front-end developer who lacks of professional experience
2. An entrant front-end developer with web development education background
3. A professional front-end developer with a couple of years of industrial experience

Receiving feedback and insights from these developers helped to identify the most demanded needs and to evaluate the efficiency and usability of the end-product from various perspectives.

4.1.2 The intended user experience

The desired user experience is a user interaction as smooth, easy and responsive as possible. From the user point of view, the metric the application should be measured against is easiness; the simpler it is to interact with the system, the quicker developers can prototype their applications.

4.2 Specifications

In this section I will define the specifications by formulating a list of functional and non-functional requirements, which describe what the application is meant to accomplish.

4.2.1 Functional requirements

Functional requirements define what brings value to the user. To define these requirements from the user point of view, I am going to use the user stories format **[4.2.1.a]**, which is a highly effective way of layouting requirements. I have formulated the user stories based on both my personal experiences and on the user survey **[4.2.1.b]** I have conducted in cooperation with the developers:

- As a developer, I want to be able to create a backend for my client application (app), so I can store the data of my application safely.
- As a developer, I want to be able to create separate backends for my apps, so I can use one system for managing all my apps.
- As a developer, I want to have an graphical overview of the backend of my app, so I can see all the data stored on the server.
- As a developer, I want to have a RESTful API to communicate with, so that I can easily interact with the backend from my front-end app.
- As a developer, I want to have endpoints for creating, getting, updating, deleting and authenticating users, so I can implement Registration and Login functionality to my app.
- As a developer, I want to have endpoints for creating, getting, updating and deleting JSON documents, so I can store and retrieve unstructured data sent from my app.
- As a developer, I want to be able to retrieve JSON documents based on a provided key-value pair, so I can retrieve a list of documents (e.g.: posts, books, cars with four seats etc.).
- As a developer, I want to have endpoints for uploading, downloading and deleting files, so I can implement file uploading functionality to my app.
- As a developer, **I want to be able to call these endpoints without any kind of security constraints, so I can build my prototype swiftly**.
- As a developer, I want to be able to set the endpoints to require app authentication, so I can ensure that unauthorized requests will be rejected.
- As a developer, I want to be able to set the endpoints to require user authentication, so I can ensure that specific actions will be tied to specific users (e.g.: users can only update their own profiles).
- As a developer, I want to be able to set the endpoints to require master authentications (app administrator), so I can ensure that only an authenticated master can perform specific, sensitive operations (e.g.: update app name, delete app).
- As a developer, **I want to have a configuration file containing key-value pairs for the settings of Stash Backend, so I can easily customize the behaviour of my backend, without doing any backend-development** (e.g.: turn authentications on/off, set token expiry time, set database credentials etc.).

4.2.2 Non-functional requirements

Based on the functional requirements I will now define non-functional requirements which are essentially a list of requirements formulated from the Stash Backend developer point of view. These specifications describe how the application should be structured and how the application should accomplish the functional requirements.

- The application should **include a web-server**, to which the users (developers) can send HTTP requests. This web-server should be embedded within the application itself, so the users don't have to install one (e.g.: Apache **[4.2.2.a]**) separately.
- From the requirements above, we can see that the application needs two interfaces: both a **web GUI (front-end), and a RESTful API**.
  - Client applications (called apps) will communicate with Stash Backend through the API. This interface will provide a machine-to-machine communication channel, and it will be responsible for handling the application's resources.
  - The application however will also provide a human-to-machine communication channel in the shape of a web interface representing a *Dashboard*, which will visualize the all the data stored on the server, similarly as phpMyAdmin **[4.2.2b]** does. This way developers will be able to see and analyze the resources of their applications. As interface design and front-end development is not in the focus of this project, I have decided not to use JavaScript frameworks, instead, a simple templating-engine will be used to serve the front-end.
- Stash Backend needs to be able to handle **both relational and nonrelational datasets.** The resources within Stash Backend (apps, users, documents) are strongly relational (a document belongs to a user who belongs to an app), therefore an SQL database is preferred over NoSQL. However, as the users of Stash Backend apps need to be able to store unstructured JSON documents, NoSQL datasets also have to be supported. Furthermore, as the application has to provide the possibility of filtering (indexing) those documents, it won't be enough to simply store the documents as Strings in a relational database; a real NoSQL solution is required.
- The application also needs to be able to **authenticate requests,** as certain operations should be only accessible for certain requesters. Reading through the functional requirements leads us to define three types of authentication:
  - **App Authentication** ensuring that the request is coming from an authenticated app.
  - **User Authentication** ensuring that the request is coming from an authenticated user.
  - **Master Authentication**, ensuring that the request is coming from an authenticated master (app administrator).
- The app administrators should be able to **turn authentication off** using the configuration file. As a matter of fact, it should be turned off by default for the sake of swift development.

4.3 Design patterns

As the lists of requirements are in place, we are ready to design the solution on technical level by choosing the right programming concepts, development stack, tools, and we can lay out the project structure.

4.3.1 Design patterns: Object Oriented Programming

Object Oriented Programming, also known as OOP is the most popular programming paradigm in web development. I have chosen to apply OOP in my project because I had significant experience working with this paradigm, thanks to the Web Development PBA program and my internship, and also because this methodology fits perfectly with the REST architecture. Object Oriented Programming is huge topic of discussion which reaches beyond the scope of this project, however I will point out the fundamental concepts which I am going to target during the implementation:

- **Abstraction** is a concept of taking a real-world object and representing it using programming terms, while ignoring the unnecessary details and only focusing on the characteristics that are relevant in the given context. By removing as much clutter as possible and focusing on the most important details we can easily simplify the implementation which helps to write highly maintainable code.
- **Encapsulation** is the process of combining relevant data and functions into a single unit which then hides everything about itself except that is absolutely necessary to expose. This prevents unauthorized components from accessing and/or mutating sensitive data, keeping the data and codebase safe.
- **Inheritance** is a mechanism by which a unit can acquire the characteristics (fields and methods) of another unit, providing the opportunity of building hierarchical architectures. This methodology promotes code reusability, as classes that derive from other classes have to define only their unique attributes and methods.
- **Polymorphism** is an ability of the objects to change their public interfaces based on how they are being used. In other words, polymorphism is the ability to appear in many forms. In practice this means that a method may have several implementations for certain classes, providing different functionality for different situations. This lets us customize the functionality of derived classes meanwhile keeping the hierarchical structure.

In chapter 5. Solution implementation I will describe a number of examples bringing these principles into practice.

4.3.2 Design patterns: Model-View-Controller

Model-View-Controller, also known as MVC is a widely popular design pattern for structuring web applications. In MVC there are three separate but interconnected layers being responsible for three different but clearly defined concerns:

- The **Model** contains and represents the data the users can work with. It also contains rules and restrictions regarding how the data can be manipulated and obtained.
- The **Controller** is the bridge between the *Model* and the *View*. It receives data from the *Model*, processes it if necessary, and sends it to the *View*. It also contains logic for handling user interaction, typically using HTTP requests.
- The **View** presents the data received from the *Controller* to the user, and exposes the options and operations the users can take.

Stash Backend will provide a web GUI interface called the Stash Dashboard, which will apply the MVC design pattern. This means that the concerns of these 3 components will have to be thoroughly designed and separated from each other.

4.3.3 Design patterns: RESTful architecture

Another design pattern I am going to apply is of course REST. This architecture was first introduced in Roy Fielding's dissertation in 2000 **[4.3.3.a]**. The purpose of this pattern is to create a structure that provides consistency and high reliability across multiple services and clients. REST describes six constraints that have to be addressed and implemented:

1. **Client-server**: REST architecture differentiates two parties which results in a client that doesn't have to be concerned about data storage and in a server that is not concerned about the user interface and user state. This improves the portability and scalability of both sides.
2. **Cacheable**: as clients can cache responses, these responses has to explicitly state whether they are cacheable or not. This approach potentially saves interaction between the client and the server which improves scalability and performance.
3. **Layered system:** a REST based system can consist of multiple architectural layers in which layers cannot see past the next layer. These layers are often referred as middleware components, and as a result the client doesn't know how many middlewares are involved.
4. **Code-on-demand:** this optional constraint allows to update and extend the client's functionality from the server in the form of plugins or applets.
5. **Stateless:** the communication between the client and the server should not depend on the state of the communication; everything that is important about a message (request) should be included within the message itself.
6. **Uniform interface**: this constraint requires all the participants in a RESTful architecture to use a uniformed interface for communication, which generally includes the methods and media types provided by HTTP and URIs that identify the resources.

I will demonstrate how we can apply these constraints in practice in 5. Solution implementation.

4.4 Development stack

When choosing the right programming language, I have relied on the amount of my experience with each potential languages. As I am developing the server side of the client-server relation of REST, the programming language I have chosen is Java; a general-purpose, object-oriented language that has been around long enough to provide all the necessary tools and frameworks I will need.

In order to save time on developing solutions for universal issues such as a web server, an http client or document deserialization, I have chosen a framework called Dropwizard **[4.4.a]** that happens to contain all these three, and even more. Among many great features, it also provides an intuitive Configuration **[4.4.b]** interface for using a simple configuration file, with which the application can be initiated, which comes really handy as this is has been defined as a requirement. Dropwizard also supports Views **[4.4.c]**, on which I can easily apply the MVC design pattern to build Stash Dashboard. By using a framework my goal is to speed up the development process by focusing only on those parts of the implementation which are relevant from the project's perspective. To build and compile the solution I have used Gradle **[4.4.d]** build automation system. Using this tool I was able to build my Java files, write build scripts, and handle package management easily. With Gradle I could also merge the entire application into one, single file, that is called a *fatjar*. Releasing the end application using this single file makes the setup process for the users even easier.

In order to manage both relational and nonrelational databases, I have decided to use PostgreSQL **[4.4.e]**, which is a free, open-source relational database that provides NoSQL-style JSON processing as well. Postgres proved to be a good choice thanks to its JSON functions and operators **[4.4.f]** which let me index JSON objects easily. Dropwizard also had great support for Postgres integration in the shape of Dropwizard JDBI **[4.4.g]** which is a flexible library for interacting with relational databases using SQL.

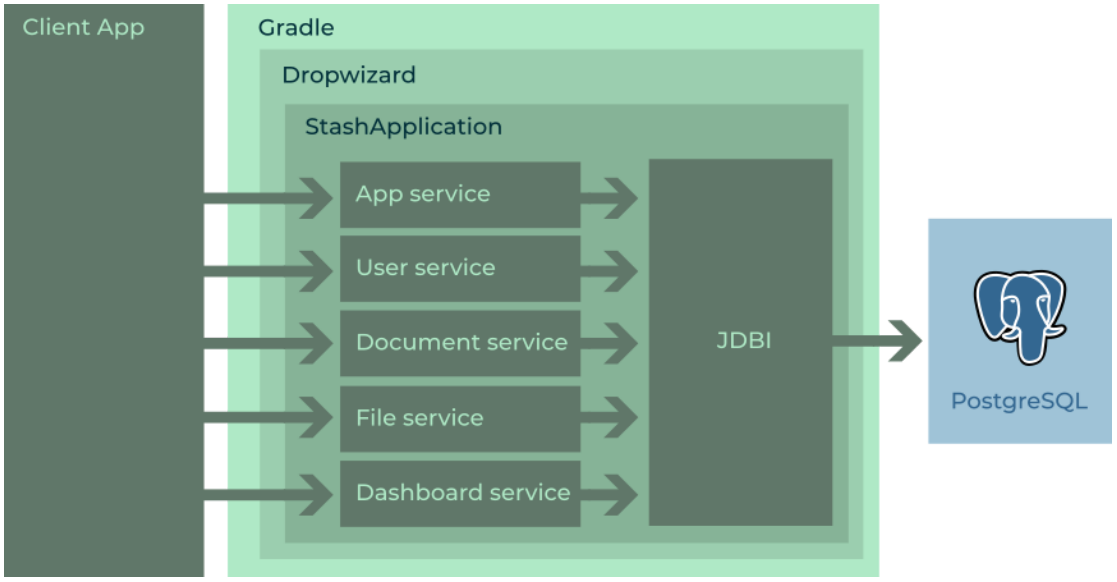


Figure 4.4.a - Stash Backend Architecture

4.5 System architecture

In this section I would like to demonstrate how the application was structured using the formerly selected core technologies; Java, Gradle, Dropwizard and PostgreSQL. In this quick overview I would like present each component's responsibility and the way they are interconnected.

4.5.1 Java and Gradle

Gradle is a great build and automation tool for Java, because it lets us write custom scripts for building, testing or even deployment and also handles package management with which we can easily fetch and utilize external packages. A Gradle project is defined within a file called *build.gradle*, which contains the list of dependencies (external packages), the list of repositories where these dependencies can be found, and the definitions of runnable scripts. This means that the only tools we need in order to compile the source code are Java and Gradle.

```
1   // Gradle project
2   group 'com.gaboratorium'
3   version '0.1.0-SNAPSHOT'
4
5   // Gradle plugins
6   apply plugin: 'java'
7   apply plugin: 'application'
8   apply plugin: 'com.github.johnrer
9
10  // Project properties
11  mainClassName = "com.gaboratorium
12  sourceCompatibility = 1.8
13
14  ext {
15      dropwizardVersion = '1.2.0'
16      configPath = "$rootProject.pr
17  }
18
19  // Repositories & Dependencies
20  repositories {···
22  }
23
24  dependencies {···
53  }
54
55  // Scripts
56  buildscript {···
63  }
64
65  shadowJar {···
69  }
70
71  run {···
73  }
```

Stash Backend utilizes three Gradle plugins: **Gradle Java**, which is required to compile Java files, **Gradle Application**, which is used for running the codebase as an application from the command line and **Gradle Shadow** which is used to compile and encompass the entire application into one, single fatjar. In order to run the solution we need to specify an entry-point of the application, which is done by specifying the main class name ("*mainClassName*").

It is a common practice to specify a section called *ext* (which stands for Extra Properties) in which we can declare a set of variables (such as version numbers and paths), that we can then reuse in our dependency listings and scripts.

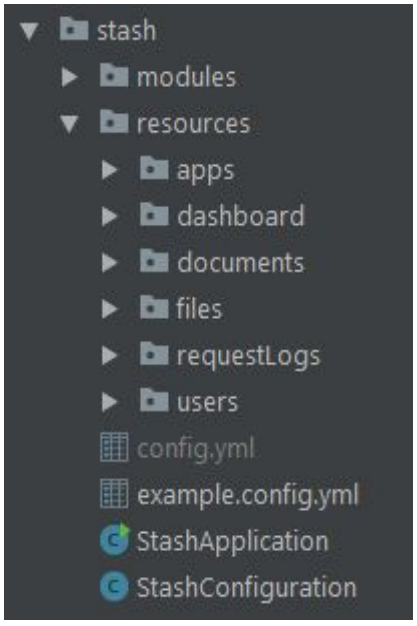
Figure 4.5.1.a - Gradle project structure (build.gradle)

4.5.2 Dropwizard

Dropwizard is a Java framework for building RESTful web services. Among many great features and tools, it includes Jetty **[4.4.2.a]**, a lightweight HTTP server which we can embed into our application directly, Jersey **[4.4.2.b]**, a library for mapping HTTP requests into Java objects and Jackson **[4.4.2.c]**, a library for JSON deserialization. These are all widely used and appreciated tools that fit perfectly into my project. Setting up a Dropwizard application with Gradle is easy. All we need to do is to add the required Dropwizard modules into our dependency list.

```
24  dependencies {
25      // Dropwizard modules
26      implementation "io.dropwizard:dropwizard-core:${dropwizardVersion}"
27      implementation "io.dropwizard:dropwizard-db:${dropwizardVersion}"
28      implementation "io.dropwizard:dropwizard-jdbi:${dropwizardVersion}"
29      implementation "io.dropwizard:dropwizard-auth:${dropwizardVersion}"
30      implementation "io.dropwizard:dropwizard-migrations:${dropwizardVersion}"
31      implementation "io.dropwizard:dropwizard-views-mustache:${dropwizardVersion}"
32      implementation "io.dropwizard:dropwizard-forms:${dropwizardVersion}"
33      compile group: 'io.dropwizard', name: 'dropwizard-assets', version: '1.2.0'
34
35      // ...
```

Figure 4.5.2.a - Dropwizard in Gradle (build.gradle)



Once the dependencies have been successfully fetched by Gradle, we can create our main class called *StashApplication*, which is the entry point of Stash Backend. To run this instance, we also need to provide a configuration object. I will call this configuration object *StashConfiguration*, and it will be created based on a main configuration file called *config.yml*, which we will provide as an argument whenever we run the application.

Inside *StashApplication*, we can define a series of commands we would like to execute every time we launch the application. I will utilize this feature to run database migrations for example. After the *StashApplication* has been successfully initialised based on *StashConfiguration*, it will use *Resources* to access and manipulate data in the database. These *Resources* will share functionalities in the shape of *Modules*, which will provide non-domain specific operations (more about Modules in section 5.3 Modules).

Figure. 4.5.2.b - Dropwizard structure

4.5.3 PostgreSQL

In order to store data we need to implement a database connection. Dropwizard provides a helpful module called Dropwizard JDBI **[4.5.3.a]** with which we can easily establish that. To establish a connection in Dropwizard and perform a database operation the following steps are required:

1. **Create a DBI (Database Interface) object** based on the credentials provided by the configuration object (*StashConfiguration*). This entity will represent the connection to the PostgreSQL server.
2. **Create a DAO (Data Access Object) for each resource,** based on this connection**.** A DAO will encompass SQL operations around one given resource.
3. **Database migrations will be used to set up the initial schema,** and manage upcoming changes in the future. For this purpose I will use Liquibase **[4.5.3.b]**.
4. The connection and the migrations will be established on each application initialisation.

I will describe the database implementation in detail in section 5.2 Database.

```
// Database
final DataSourceFactory dataSourceFactory = configuration.getDatabase();
final DBI dbi = getDBI(environment, dataSourceFactory);

// Daos
final AppDao appDao = dbi.onDemand(AppDao.class);
final MasterDao masterDao = dbi.onDemand(MasterDao.class);
final UserDao userDao = dbi.onDemand(UserDao.class);
final DocumentDao documentDao = dbi.onDemand(DocumentDao.class);
final FileDao fileDao = dbi.onDemand(FileDao.class);
final RequestLogDao requestLogDao = dbi.onDemand(RequestLogDao.class);
```

Figure 4.5.3.a - Creating DAOs based on an established database connection (StashApplication.java)

4.6 Database design

It is important to have a thoroughly designed database model before I start the implementation. The database design consists of the following steps: first I formulate the mini-world, which describes what the application does and how it behaves. To do this, I will rely on the formerly defined user stories. Based on this, I can then define the database entities, which represent independent units (resources) about which the application is going to store data. The next step will be to define the relationships between the entities which might be one-to-one, one-to-many or many-to-many.

Based on the requirements specified above I have defined the following entities: **app, master, document, file** and **user**. To apply the uniform interface principle of REST, these entities will eventually act as the resources of our application, which will be available through their respective URIs. To differentiate between the CRUD operations on the entities, we will obtain these resources using corresponding HTTP methods. Each entity will have they own primary key called an *id*, which will have to be included in the URI, plus a Timestamp representing the entity's creation date and time.

4.6.1 App entity

Apps are the most important entities in Stash Backend. They represent different backends created for different client applications. Each app will have the following mandatory properties: *id*, *app_name* and *app_secret*. The ID and the Secret will be used to authenticate requests which require App Authentication.

4.6.2 Master entity

A master represents a human who administers an app therefore is authorized to perform any kind of requests towards his/her app. Some endpoints will be protected and require Master Authentication (this might be however turned off in the configuration file). A master entity will have a property called *app_id*, which will contain a **foreign key** referencing an app entity. This leads to a **one-to-many relation** where several masters can administer a given app.

4.6.3 Document entity

A document represents any kind of unstructured data. The way this data will be stored is the following: it will of course have a field for it's *id*, a field for the data itself called *document_content* (stored in a JSON format), a Timestamp field for creation date called *created_at*, one called *app_id* and another called *owner_id*. The last two ones will contain a **foreign key**, which will reference an app and a user entity, leading to two other one-to-many relations. These relations represent which app and user the document belongs to. However the *owner_id* field will be optional, therefore documents might be created without providing an owner.

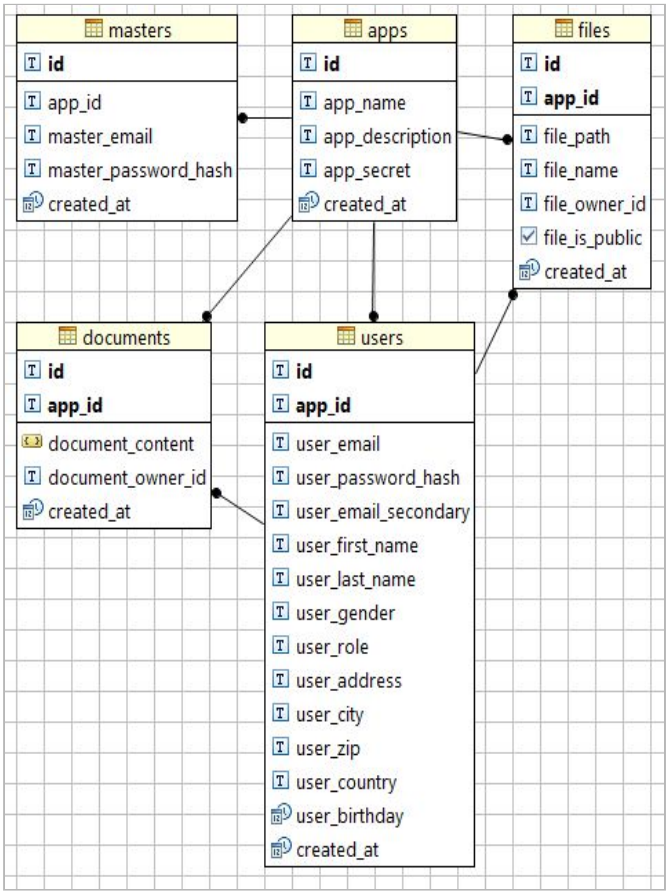
4.6.4 File entity

A file will contain metadata about a given uploaded file. Similarly to a document, a file will have two fields called *app_id* and *user_id* referencing the app and user the file belongs to. Furthermore, the entity will have two fields representing the location of the file on the server: *file_path* and *file_name*. Another field will be added, called *file_is_public* which will be type of bool. This property will play an important role when evaluating the requested file's accessibility. Accessing files that are not public will require User Authentication and User Authorization.

4.6.5 User entity

The user entity will represent a user of an app. It will contain basic information about the user, such as his/her name, gender, address or birthday. Most of these fields however will be optional, the required attributes when creating a new user will be the following: *id, app_id, user_email and user_password_hash*.

4.6.6 Entity Relation Model



Based on the entities I have formulated the following entity relationship model. Each entity contains a foreign key referencing an app, establishing a one-to-many relationship.

It might have been an option to create a **many-to-many** relation between masters and apps; in which case a **cross-reference** table would have been created, containing 3 fields: an entity *id*, a *master_id* and an *app_id*. For the sake of simplicity, this has been not implemented resulting in a master entity model that is tied to one single app entity.

Figure 4.6.6.a - Entity Relation Model

5. Solution implementation

The goal of this chapter is to describe the implementation in details backed up by practical examples.

During the implementation I have introduced a new term, which I am going to reference further down in the report: *services*. The reason for this is to differentiate between the terms of Dropwizard Resources and REST resources. Meanwhile the former is a container of various logic used to get and process data (REST resources), the latter is used to express the representations of the database entities. Ideally, these services should be independent Gradle projects with their respective dependencies; this way we could easily implement, test, deploy and utilize them independently from each other wich would fulfill the constraint of REST about layered systems. Due to the time constraints of the project however I have decided not to prioritize this implementation.

5.1 Stash Application

The app initialisation is defined in the *StashApplication* class, where all of components get registered. This is where the database connection is established using JDBI and where DAOs are created based on that connection. After this, Services are registered using their respective DAOs, and finally Filters (Authenticators) and Modules are registered here as well. The database migrations are also initiated from this class.

5.2 Services

There are 5 services included in the current iteration of Stash Backend, namely: App-, User-, Document-, File- and Dashboard service. Apart from the Dashboard service, they all provide methods for the CRUD operations to be applied on one given resource. These resources are identified by their respective URIs, using the RESTful naming conventions. To apply the REST conventions properly, these operations are not differentiated by the URIs, but by the HTTP methods.

As an example, here are the endpoints provided by the App service (*AppResource.java*):

Method	URI	Description
POST	/apps	Create app
GET	/apps/{appId}	Get app
DELETE	/apps/{appId}	Delete app
POST	/apps/{appId}/authenticate	Authenticate app

We might notice that the POST requests slightly differ in behaviour; meanwhile the first one does what is expected from a CRUD perspective ("create a new resource identified by this URI"), the latter is what the book RESTful Web Services calls an *Overloaded POST* **[5.2.a]**, which means a POST request that does not actually create a new resource. This is however completely legal, as according to the HTTP specification a POST request means "*providing a block of data, such as the result of submitting a form, to a data-handling process*" **[5.2.b]**. In this case, the client provides the corresponding app Secret in the request body, and if it gets evaluated correct, it will get an App Authentication Token in exchange.

Let's have a closer look how we can implement such an endpoint in Java with Dropwizard. Here is the implementation of the *Delete app* method in the App service (*AppResource.java*):

```
@Path("/apps")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@RequiredArgsConstructor
public class AppResource {
```

Figure 5.2.a - AppResource annotations (AppResource.java)

```
@DELETE
@Path("/{appId}")
@AppAuthenticationRequired
public Response deleteApp(
    @NotNull @PathParam("appId") final String appId,
    @HeaderParam(AppAuthenticationHeaders.APP_ID) final Optional<String> appIdHeader
) throws Exception {

    if (!appRequestGuard.isRequestAuthorized(appIdHeader, appId)) {
        return StashResponse.forbidden();
    } else {
        appDao.delete(appId);
        return StashResponse.noContent();
    }
}
```

Figure. 5.2.b - Delete app endpoint (AppResource.java)

As we can see, both the class and its method utilizes the annotations (*@Path("/{appId}"), @DELETE etc.*) provided by Jersey. Specifying a path on the class we can define a root URI, to which the paths of the methods will be appended, which in this case means that this method will be available at "*/apps/{appId}*". I also define what media types this resource can accept and produce. These attributes will be inherited by the methods unless specified otherwise.

On the method I specify the HTTP method using the @DELETE annotation. Below that, I am using an annotation I have registered myself in *StashApplication*; the *@AppAuthenticationRequired* filter will only let this method execute if a certain condition is met. This condition is that the request has to

include two request headers including a valid App Authentication Token and an app ID (more about tokens in 5.3 Security).

The method expects two parameters: a target app ID specified in the URI and the formerly mentioned header including an app ID (which is the requester). The method has one simple logic implemented: it simply does an authorization-check on the requester. If the requester app is authorized to perform the required action, the app will be deleted using the DAO, and a response *204 No Content* will be returned, otherwise a response *403 Forbidden* will be returned.

This is an idempotent method **[5.2.c]**, which is also part of the REST specification. This means that the side-effects of several requests are the same as one, single request. The result will always be independent from the number of requests we sent (e.g.: consider the command "*i = 5*" instead of "*i++*"). Idempotency helps to build fault-tolerant APIs, as the client can keep sending the same requests until it receives a satisfying response without worrying about unintended mutations.

5.3 Modules

There are a couple of components which are shared across services. I called these components modules. These modules are initiated in the *StashApplication* based on the *StashConfiguration*, and they are passed to the services as parameters.

Let's take *StashTokenStore* as an example. I am using Json Web Tokens (JWT) **[5.3.a]** for authenticating the requests, and in order to handle it uniformly across services, I have created a module called *StashTokenStore*. As this module is initiated on an application level in *StashApplication*, some of its attributes (such as token expiry time) can be configured right from the configuration file.

```
@RequiredArgsConstructor
public class StashTokenStore {

    final private String key;
    final private Integer appAuthTokenExpiryTimeInMinutes;
    final private Integer userAuthTokenExpiryTimeInMinutes;
    final private Integer masterAuthTokenExpiryTimeInMinutes;
```

Figure 5.3.a - StashTokenStore constructors (StashTokenStore.java)

```
// Modules
final StashTokenStore stashTokenStore = new StashTokenStore(
    configuration.getAppsTokenStoreKey(),
    configuration.getAppAuthTokenExpiryTimeInMinutes(),
    configuration.getUserAuthTokenExpiryTimeInMinutes(),
    configuration.getMasterAuthTokenExpiryTimeInMinutes()
);
```

Figure 5.3.b - Initializing StashTokenStore module with the StashConfiguration settings (StashApplication.java)

Note that the accessibility of the properties of this class is set to private. This is how the encapsulation constraint of OOP is applied; these properties are intended to be used internally, within the class only and should not be accessed and mutated externally. This practice leads to a safe and easy-to-maintain code.

Another example for a module is the *StashResponse*, which I have created in order to create a RESTful uniform interface for communication. This unified response builder helps me to build response objects that fulfill the REST constraints in one, centralized place.

```
// Ok
public static Response ok() { return build(HttpStatus.OK_200); }
public static Response ok(String message) { return build(HttpStatus.OK_200, message); }
public static Response ok(Object responseObject) { return build(HttpStatus.OK_200, responseObject); }
```

Figure 5.3.c - Polymorphic method for response building (StashResponse.java)

In Figure 5.3.c we can see how we can apply polymorphism in OOP; by changing a method's signature we can alter its functionality, which in this case means that we can build the response with and without a message and a response object.

5.4 Database

Stash Backend depends on a PostgreSQL server connection which is established whenever the application is launched. This connection is being set up in *StashApplication.java*, where the initial migration is executed as well. This migration is defined in XML format and located at "*src/main/resources/migrations/db.changelog.0001.initial-schema.xml*". This file describes the initial tables to be created and their fields. Using Liquibase, we can easily introduce changes in the database schema without losing currently stored data by creating a new migration; to do so we just have to create another migration file with the updated schema and an increased sequence number called *changeSet*.

The application communicates with the database through the methods of DAOs (Data Access Objects). Using Dropwizard JDBI annotations, these interfaces will describe how the methods of DAOs should be translated into SQL scripts. We can also utilize the *@Bind* annotation to bind the method parameters to SQL values.

```java
public interface AppDao {

    @SqlQuery("select * from stash.apps where id = :appId;")
    @Mapper(AppMapper.class)
    App findById(
        @Bind("appId") String appId
    );

    @SqlUpdate("delete from stash.apps where id = :appId;")
    void delete(
        @Bind("appId") String appId
    );
```

Figure 5.4.a - A DAO example (AppDao.java)

Stash Backend DAOs will depend on two main annotation: *@SqlQuery* to execute SQL script and expect something in return and *@SqlUpdate* to execute a script without expecting anything back.

The queried dataset returned from the database has to be parsed which happens automatically if the returned data type is recognizable by the JDBI library (these are mostly strings, numbers, booleans, dates). However, if the returned dataset is unique, a mapper has to be provided which describes how the returned data should be parsed into a Java object. To achieve this, we can use the *@Mapper* annotation.

```java
public class AppMapper implements ResultSetMapper<App> {
    public App map(int index, ResultSet resultSet, StatementContext statementContext) throws SQLException {
        return new App(
            resultSet.getString( columnLabel: "id"),
            resultSet.getString( columnLabel: "app_name"),
            resultSet.getString( columnLabel: "app_description"),
            resultSet.getString( columnLabel: "app_secret")
        );
    }
}
```

Figure 5.4.b - A DAO Mapper example (AppMapper.java)

If we look closely, we might notice that the creation date (*created_at*) of the app is not mapped, even though it is clearly part of the resource. This is completely legal; the reason for this is because by definition a REST application is not supposed to return the actual resource itself, but a *representation* of a resource, which is described by RESTful Web APIs the following way: *"When a client issues a GET request for a resource, the server should serve a document that captures the resource in a useful way. That's a representation."* **[5.4.a].** In this particular case I found it irrelevant to include the creation date in this object, so for the sake of clear abstraction (OOP constraint: remember to remove the clutter whenever possible) I simply ignored it.

5.5 Views

Stash Application provides a web interface which the app administrators can use to visualize the data of their apps. This is provided by the Dashboard service which sends Views to the requester.

```
@Path("/")
@Produces(MediaType.TEXT_HTML)
@RequiredArgsConstructor
public class DashboardResource {

    final private AppDao appDao;
    final private UserDao userDao;
    final private MasterDao masterDao;
    final private DocumentDao documentDao;
    final private FileDao fileDao;
    final private RequestLogDao requestLogDao;

    private final StashTokenStore stashTokenStore;
```

Figure 5.5.a - Dashboard service constructors (DashboardResource.java)

One of the main differences between the Dashboard and other services is the returned media types; while all the other services return representations in JSON format, Dashboard replies in HTML, as these resources are views which are intended to be consumed by browsers. As the REST HATEOAS (Hypermedia As The Engine of Application State) **[5.5.a]** specification says, this has to be indicated in the response headers, so I used the Jersey annotation @Produces to specify it.

```
@GET
@MasterAuthenticationRequired
@Path("dashboard/app")
public Response getAppView(
    @CookieParam("X-Auth-Master-Id") final String masterId
) {
    final Master master = masterDao.findById(masterId);
    final App app = appDao.findById(master.getAppId());
    final String appToken = stashTokenStore.create(app.getAppId(), stashTokenStore.getAppAuthTokenExpiryTime());
    final String fkey = UUID.randomUUID().toString();
    final AppViewModel model = new AppViewModel(
        app,
        master,
        appToken,
        fkey
    );

    final View view = new AppView(model);
    return getDashboardViewResponse(view, fkey);
}
```

Figure 5.5.b - Dashboard: Get App View (DashboardResource.java)

Similarly to the other services, Dashboard service also defines a list of endpoints identified by their respective URIs and HTTP methods. However, instead of receiving the authentication credentials as request headers, we receive them through cookies (*@CookieParam*). The reason for this is that in

this environment the interaction happens through the browser. Once the administrator has logged in, his/her authentication token is stored in a cookie in the browser. Inside the method we can find the business logic; fetching and processing data from the database, preparing for presentation; this serves as the Controller part of the MVC methodology. We already know how DAOs work which play the Model part; serving the data. Finally the View layer of the MVC methodology is represented by a Dropwizard View. The distinction between the Model and the View is even more spectacular if we have a look at how Dropwizard Views are constructed. One of their properties is a *templateName,* which points to the actual template file; which in this case is a mustache file. The other one is a ViewModel, which holds the data that was formerly obtained from the Model and processed by the Controller.

```
public class AppView extends View {

    @Getter
    private final DashboardViewModel vm;

    public AppView(AppViewModel vm) {
        super( templateName: "app.mustache");
        this.vm = vm;
    }
}
```

Figure 5.5.c - Dashboard: App View (AppView.java)

In this particular example we can also see how inheritance in OOP creates hierarchy. Our custom view extends a Dropwizard View by which it can inherit fields and the constructor. That's why I didn't have to specify a *templateName* by myself; it has been already done in a Dropwizard View.

```
{{> partials/head }}
<div class="container-fluid o-page">
    <div class="row o-page">
        {{> partials/sidebar }}

        <!-- Content -->
        <div class="col o-main">
            <div class="col-content-1 o-main--text">
                <section class="c-section--bordered col-content-1">

                    <!-- App logo -->
                    <div class="row justify-content-md-center">
                        <div class="c-avatar">
                            <i class="fa fa-flag c-avatar__logo" aria-hidden="true"></i>
                        </div>
                    </div>

                    <!-- App name-->
                    <h1 class="c-avatar__title">
                        {{vm.app.appName}}
                    </h1>
                    <hr>
```

Figure 5.5.d - Dashboard: App View Template (app.mustache)

Stash Backend utilizes a library called Mustache **[5.5.b]** to define its templates. Using this templating engine I could easily insert partials (e.g.: HTML snippet for a sidebar) and display data stored in the View Model.

5.6 Security

Security is a critical aspect of every given web application. In the following section I would like to describe various kinds of potential attacks and come up with examples how they can be prevented.

5.6.1 SQL injection

SQL injection is type of attack where the attacker attempts to figure out how the application communicates with the database and based on this knowledge, he/she tries to communicate directly with the database by injecting SQL scripts. In this scenario the attacker makes an assumption that the values he/she provides in the URL/HTML form/Request Body will be directly inserted in the SQL code that will be sent to the database server.

A quick example can be trying to look up a user with the following user name: "*0; DROP TABLE users;*" Unless the application sanitizes this value, if we make an assumption that the query follows: "*SELECT * FROM USERS WHERE user_name = :userName*", then using the user name we provided will result in the following query: "*SELECT * FROM USERS WHERE user_name = 0; DROP TABLE users;*". This script would delete our table which is unacceptable.

To prevent these kind of attacks is fairly easy and often times already implemented in modern frameworks providing database communication. However according to OWASP (Open Web Application Security Project) SQL injection is still the world's most critical web application risk in 2017 **[5.6.1.a]**, therefore its prevention is fundamental which means we cannot rely on assumptions. We have to make sure that the tools we are using are prepared for these attacks and if they are not; we have to implement the prevention ourselves.

Reading through the Five Minute Introduction of JDBI **[5.6.1.b]** we can confirm that JDBI utilizes prepared statements, which is the primer defense mechanism suggested by OWASP **[5.6.1.c]**. Diving into the codebase of the JDBI library, we can double-check this by tracking down the argument binders.

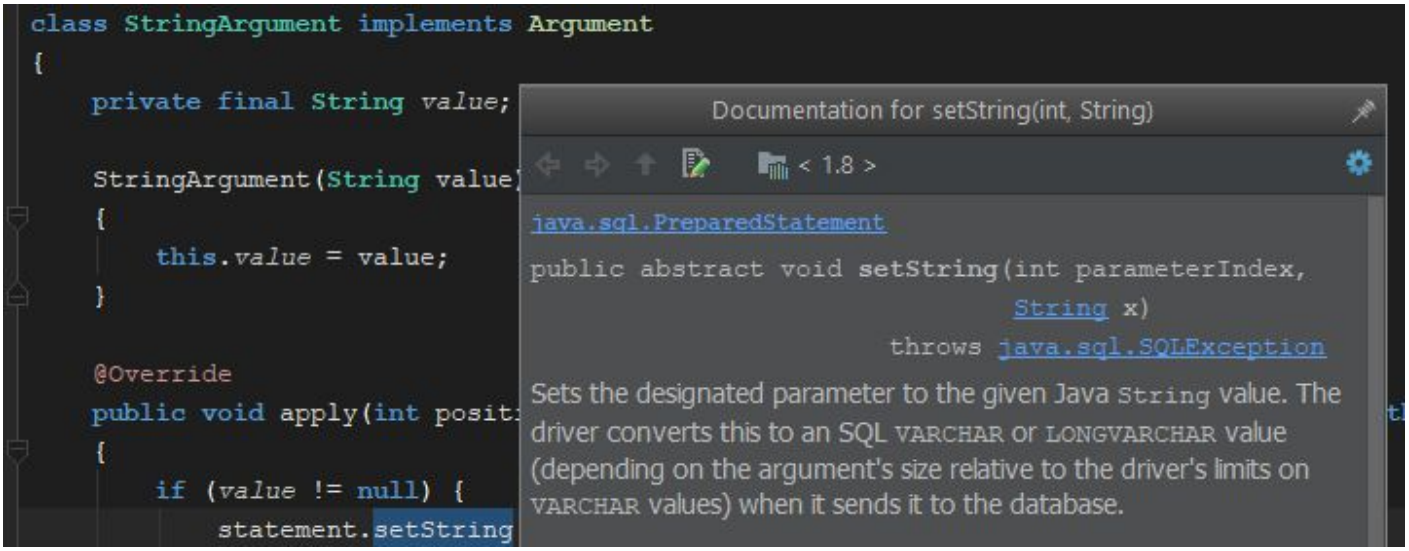


Figure 5.6.1.a - String Argument binder in JDBI using Prepared Statement (StringArgument.java)

5.6.2 Cross-Site Scripting (XSS)

Cross-Site Scripting is the seventh most critical web application risk according to OWASP TOP 10 - 2017. In this type of attack the attacker injects a script into the system which is not executed by the database server, but by an innocent user's browser. To demonstrate this, let's image a user whose name is "<*img src='www.mysite.com/attack*'>". If this data is not serialized correctly, visiting this user's profile where his/her name is visible will end up making our browser a request to the specified URL. By expanding this code we can easily send our cookies within the request, potentially handing our session over to the attacker.

Fortunately, JDBI is prepared to serialize these kind of inputs as well. By inserting malicious code and retrieving this data in the browser we can confirm that both HTML and JavaScript snippets will be serialized, therefore will not be executed.

```
<div class="c-document">
    <div class="c-document__json col-content">
        {&quot;name&quot;:&quot;cat&quot;,&quot;color&quot;:&quot;red&quot;,&quot;script&quot;:&quot;&lt;script&gt;alert(&#39;h
    </div>
    <div class="c-document__meta">
        <ul class="c-document__meta-list">
            <li class="c-document__meta-list-elem"><span class="o-text--bold">DocumentID:</span> 4518b887-24e2-4349-8fa3-74089b
            <li class="c-document__meta-list-elem"><span class="o-text--bold">Created at:</span> 2017-12-13 14:14:49.46105</li>
            <li class="c-document__meta-list-elem"><span class="o-text--bold">Owner ID:</span> gabor</li>
        </ul>
    </div>
</div>
<div class="c-document">
    <div class="c-document__json col-content">
        {&quot;name&quot;:&quot;cat&quot;,&quot;color&quot;:&quot;red&quot;,&quot;script&quot;:&quot;&lt;img src&#61;&#39;https
s-Uses-2016.jpg&#39;&gt;&quot;}
    </div>
```

Figure 5.6.2.a - Malicious code has been serialized before printing (Dashboard: Documents View)

It is also worth to mention that this type of serialization only happens (and supposed to happen) when printing the data; not when we insert it in the database. The reason for this is that we either

should forbid this kind of data in the first place on the client (because it is not a valid user name for example) or keep the original state of the input (because it is intended to be displayed as script).

id	app_id	document_content	docur
d09f0969-7c75-4047-84a2-eb0388dabd1a	test_app	{"name": "cat", "color": "red"}	gabor
4518b887-24e2-4349-8fa3-74089b76cd27	test_app	{"name": "cat", "color": "red", "script": "<script>alert('hello');</script>"}	gabor
dd943ca9-d441-45c1-b05d-3d0ddcbbe8a0	test_app	{"name": "cat", "color": "red", "script": "<img src='https://beebom-redkapmedia.netdı	gabor
0e891314-b648-4aa3-ae73-85cc002922a0	test_app	{"userComment": ""}	gabor
cedbf6e5-dee7-493c-90a4-955add7171d6	test_app	{"userComment": "Nice article!"}	gabor

Figure 5.6.2.b - All data in its original form in the database

5.6.3 Cross-Site Request Forgery (CSRF)

Cross Site Request Forgery is an attack which involves a bit of social engineering as well. In this scenario the attacker attempts to ride the victim's session by tricking them into making requests to a domain for which the victim has access to. This approach is based on the assumption that the victim will use same browser for opening the malicious code as he/she uses for the targeted site and the access credentials are currently stored in that browser.

A quick example could be sending an email to the victim in which we place an "*<img src='www.socialsite.com/me/friends'>*" element. Even if this endpoint requires user authentication, we can perform a successful request, because as the victim opens the e-mail this request will be succesfully made; however only if our assumption was right about the user who uses the same browser for opening the email and he is currently logged in. Note, that this particular request assumably does not make any harm, as GET requests are supposed to be safe requests, but it demonstrates how we could get around the authentication evaluation.

There are several ways of making it harder to forge such a request (e.g.: using POST requests, so image tags in HTML cannot be used, checking the Referer header), however most of these does not provide 100% protection. One of the most effective implementation of security against CSRF is to require a key in the request which is has been formerly provided to the client but has not been stored by the browser. To demonstrate this I have implemented CSRF protection against the logout functionality in the Dashboard. Before this implementation, by simply tricking a client into calling the corresponding endpoint we could simply log the victim out of the system without him/her knowing about it.

```
<li class="nav-item c-sidebar__menu-item">
    <form action="/dashboard/logout" method="post">
        <input id="fkey" name="fkey" type="hidden" value="36f93dd1-92db-4cd2-ab59-e2b3455a5df7" />
        <button class="c-sidebar__menu-link--button">
            <i class="fa fa-sign-out fa-fw" aria-hidden="true"></i>
            Log out
        </button>
    </form>
</li>
```

Figure 5.6.3.a - Protection against Cross Site Request Forgery

The logout button in the sidebar of the Dashboard is actually an HTML form which sends a POST request to the server. Within this POST request a hidden field holds a value called *fkey*, that has been generated by the server when we obtained this page. This value is expected on the server site when the request has been received, and is not stored in the browser. Therefore even sending the right request (POST) to the right endpoint from the right browser with the right cookies will be rejected if this key is not included in the request.

5.6.4 JSON Web Tokens (JWT)

In order to authenticate the requests I am using Json Web Tokens. With this approach I can easily encrypt and decrypt messages using a secret that is defined in the configuration file and never leaves the server. These encrypted messages are called tokens and they contain information about the token owners. These token are also required as part of the requests in order to perform certain actions. As the client-server REST constraint and the separation-of-concerns principle stand, it's the client's responsibility to maintain these tokens offline and safely save them. The tokens can be acquired using the authentication endpoints by providing the claimed identity ID and secret.

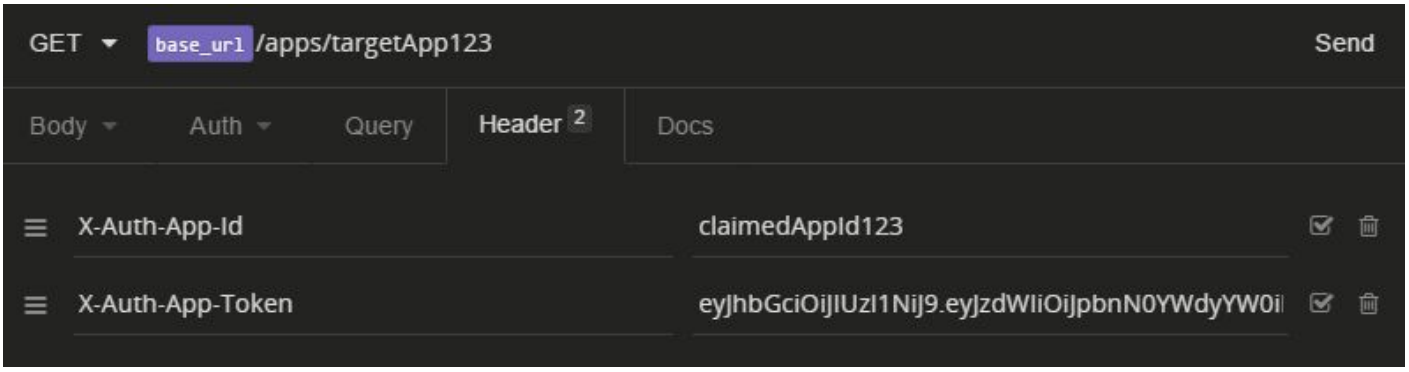


Figure 5.6.4.a - Simple GET request containing all the necessary information for evaluation (Insomnia)

This is where the Stateless constraint of REST has been applied and came into practice; using this approach we can contain all the necessary information required for evaluation within one request: we include the ID of the requested resource in the URI and our credentials in the request headers which consists of two parts: a claim (*X-Auth-App-Id*) and a proof (*X-Auth-App-Token*). No session management is required.

These authentication tokens have an expiry date which can be set in the configuration file, and they contain an entity ID. If this ID and the claimed ID match, the authentication has been considered successful. However there are two other ways an authentication can succeed; if this feature is turned off in the configuration file, or if a successful master authentication overwrites it.

```
public class AppAuthenticationRequiredFilter implements ContainerRequestFilter {

    private final StashTokenStore stashTokenStore;
    private final boolean isAppAuthenticationRequired;

    @Override
    public void filter(ContainerRequestContext requestContext) throws IOException {

        if (isAppAuthenticationRequired) {
            final String token = requestContext.getHeaderString(AppAuthenticationHeaders.APP_TOKEN);
            final String appId = requestContext.getHeaderString(AppAuthenticationHeaders.APP_ID);
            final boolean isParamListProvided = StringUtils.isEmpty(token) || StringUtils.isEmpty(appId);
            final String errorMsg =
                "App authentication failed because auth token was either not provided, corrupted or expired.";

            final boolean isMasterAuthenticated = isMasterAuthenticated(requestContext);

            if (isParamListProvided && !isMasterAuthenticated) {
                requestContext.abortWith(StashResponse.forbidden(errorMsg));
            } else if (!stashTokenStore.isValid(token, appId) && !isMasterAuthenticated) {
                requestContext.abortWith(StashResponse.forbidden(errorMsg));
            }
        }
    }
```

Figure 5.6.4.b - App Authentication fails if it is turned on and master is not authenticated and token is not present or not valid (AppAuthenticationRequiredFilter.java)

I have also implemented request authorization which we can see if we have a look at Figure 5.2.b. The reason for this is that a request might be authenticated correctly and yet not authorized to perform a given operation. However, I have decided the authorization to be part of the business logic; the evaluation process of a given authorization really depends on the context, therefore I have placed this implementation right inside the service method.

5.7 Learnings

During the implementation I have faced various challenges which I had to overcome. In this section I would like to name a few, which I found to be instructive.

In the initial implementation I have not required the app ID in some of the resource URIs, because the app ID could be easily retrieved from the authentication token as well. However, when I started to implement the option of using the endpoints without authentication I realized that I have to separate the domain of the requested resource from the authentication identity.

This solution however immediately introduced a new terminology. I had to come to the realization that being authenticated and authorized is not the same. An application might be authenticated and yet not authorized for a given operation at the same time. This required an implementation of authorization, which introduced another question: where should the authorization logic be placed? I have decided to place it inside the service as its “business logic” might change independently from service to service, furthermore it should operate independently from the authentication.

Another learning was obtained from the user tests. As it turned out from the first tests, setting up the application not was straight-forward enough, as it required to install PostgreSQL server and to create a database with a specific name. The overcome this issue and simplify the setup process, I have implemented automatic database creation, which means that whenever Stash Backend successfully connects to a PostgreSQL server, it will look for a Stash database and automatically create it if necessary.

6. Reflection

In this final chapter I will write about the possible improvements that could be done and also touch on the future perspectives of the project. At the end I will conclude the report and the project by formulating an evaluation and a final conclusion.

6.1 Improvements

Even though the initial goal has been successfully reached which was to develop a working prototype of Stash Backend, there are several possible improvements that can be done in order to improve the functioning of the application and to enhance the quality of the code.

1. One of the most eye-catcher possible improvements from my point of view is the separated authenticators, which could be easily merged. Instead of maintaining 3 separate authenticators using the same JWT technology but different business logic could be easily merged into one module containing submodules. This would increase the maintainability of the code and potentially remove code duplicates.
2. The second most obvious issue is the lack of cache-control. Due to the limitation of the project's time constraint, I have considered it not being essential from the perspective of functionality. In the future however, it might be a good task to handle for the *StashResponse* module.
3. Instead of producing only JSON type responses, the client could request a specific type of file format in the request header, such as XML. This feature would increase the application's flexibility.
4. To apply an extended HATEOAS (Hypermedia As The Engine Of Application State) **[6.1.a]** principle, hypermedia links could be provided in the response messages, providing guidance for the client about how it can communicate with the server.
5. The Document service offers an endpoint for performing a filtered search on our JSON documents. However this endpoint is rather limited, as it only offers two key-value pairs to be provided. This could be extended by using some kind of descriptive query language (similarly to OData **[6.1.c]**) which would then be passed as one single argument.
6. To improve on convenience CORS settings could be added to the configuration file. In the current iteration of the application all requests coming from any domains are allowed. A property in the configuration file could be added accepting a list of domains from which requests would be accepted exclusively.

6.2 Future perspectives

The first functioning prototype has been released, which is ready to be used locally, however there is still room for expanding the project, and not just on a technical level.

One of the future perspectives of the project I always had in mind was to deploy the software remotely on a server making the application a real Backend-as-a-Service. In this case the users would not need to install and configure the application and its dependencies manually on their machines, but instead they could simply register a *Stash Cloud* account, which then would provide them a bunch of endpoints; ready to be used from anywhere.

Even with Stash Cloud, Stash Backend would of course remain open-source. This would lead to a business model that is practiced by several successful companies such as GitLab [6.2.a] or JetBrains [6.2.b]; a business model that offers a community-lead open-source project and a premium service based on a subscription model.

Another future perspective could be to introduce sample backends for certain types of client applications; templates that contains a preconfigured backend for chat apps, note taking apps, social apps etc. This way when creating a new app the user could decide whether he/she wants to create a blank backend or wants to use a template. If implemented right, this would of course increase the convenience of the application even more.

6.3 Evaluation and final remarks

The final version of the Stash Backend at the time this report is being written is versioned v0.1.4 and is available on GitHub. This iteration fulfills all the functional and non-functional requirements specified formerly.

I consider the project successful, as the user tests proved the application to be continuously improving and at the end, my target users could easily utilize the features Stash Backend offers. The tests covered everything from downloading the application to interpreting the documentation and developing the user's own front end applications using Stash Backend services. Through the implementation I managed to obtain a comprehensive overview of REST of which principles I could apply in the implementation of the application alongside with a set of OOP and MVC constraints. REST is a topic with a significant volume of which details could not have been covered in this documentation entirely; I consider this deficit to be due to the limitation of the time and report constraints.

I hope you found this paper interesting and informative and enjoyed trying out the application. As for me, I am satisfied with the results of the project and I am looking forward to keep developing Stash Backend.

7. Appendices

7.1 Source code

The source code of the final product and the standalone application (versioned as v0.1.4) are available for download on GitHub at the following links:

- Source code: https://github.com/gaboratorium/stash
- Standalone application: https://github.com/gaboratorium/stash/releases/tag/v0.1.4

7.2 Installation guide

1. **Download the most recent release (standalone application) from GitHub**
2. **Download and install Java 8**[6]
3. **Download and install PostgreSQL**[7] **server.** During installation, leave the default port number (5432) and provide a password for your *postgres* user. If you were not prompted to set these two, you might have to set them after installation manually.
4. **Set up your credentials in the configuration file** (config.yml). Make sure that the right port number and password is set for the Postgres connection.
5. Navigate to the Stash Backend folder, open a terminal and run: "*java -jar stash-YOUR-VERSION-NUMBER-SNAPSHOT-all.jar server config.yml*"
6. Visit *http://localhost:8080*.

7.3 User survey & user tests

[4.2.1.b] - I have conducted a user survey which contains the potential features of a BaaS. In this survey I have asked a couple of developers to prioritize these features according to how important they find them. The survey can be found in the attached file named **Stash Backend - User survey.pdf**.

[2.6.2.b] - Throughout the implementation user tests have been made which have been documented. The documented user tests can be found in the attached file named **Stash Backend - User tests.pdf**.

[6] Java 8: http://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html - Retrieved 11 December 2017
[7] PostgreSQL downloads: https://www.postgresql.org/download/ - Retrieved 11 December 2017

7.4 Literature

1. Leonard Richardson, Mike Amundsen and Sam Ruby: RESTful Web APIs (2013), ISBN13: 9781449358068
2. Todd Fredrich: RESTful Service Best Practices: Recommendations for Creating Web Services, https://github.com/tfredrich/RestApiTutorial.com/raw/master/media/RESTful%20Best%20Practices-v1_2.pdf
3. Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures - Representational State Transfer, https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

7.5 References

- [2.2.a] Google Fit REST API - https://developers.google.com/fit/rest/ - (Accessed on 10.11.2017)
- [2.2.b] Live SDK REST API - https://msdn.microsoft.com/en-us/library/office/dn631844.aspx - (Accessed on 10.11.2017)
- [2.2.c] Apple News API - https://developer.apple.com/library/content/documentation/General/Conceptual/News_API_Ref/index.html (Accessed on 10.11.2017)
- [2.6.1.a] Git - https://git-scm.com/ (Accessed on 16.11.2017)
- [2.6.1.b] GitHub - https://github.com/ (Accessed on 16.11.2017)
- [2.6.2.a] Agile Software Development: http://www.agile-process.org/ - (Accessed on 17.11.2017)
- [2.6.2.c] GitHub Issues: https://guides.github.com/features/issues/ - (Accessed on 17.11.2017)
- [2.6.2.d] Google Drive: https://www.google.com/drive/ - (Accessed on 17.11.2017)
- [3.1.a]: Web Services Architecture § Relationship to the World Wide Web and REST Architectures - https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest (Accessed on 17.11.2017)
- [3.1.2.a] Facebook Login: https://developers.facebook.com/docs/facebook-login/ - (Accessed on 18.11.2017)
- [3.1.3.a] Web Services Architecture § Relationship to the World Wide Web and REST Architectures: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest - (Accessed on 17.11.2017)
- [3.1.3.b] OASIS Web Services Security: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss - (Accessed on 17.11.2017)
- [3.1.3.c] OASIS Web Services Reliable Messaging: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm - (Accessed on 17.11.2017)
- [3.1.3.d] PayPal SOAP API: https://developer.paypal.com/docs/classic/api/PayPalSOAPAPIArchitecture/ - (Accessed on 17.11.2017)
- [3.1.3.e] Salesforce SOAP API: https://developer.salesforce.com/docs/atlas.en-us.api.meta/api/sforce_api_quickstart_intro.htm - (Accessed on 17.11.2017)
- [3.2.a] Cloud Computing Accessibility: https://www.w3.org/WAI/RD/wiki/Cloud_Computing_Accessibility - (Accessed on 17.11.2017)
- [3.2.b] Amazon Web Services: https://aws.amazon.com/ - (Accessed on 17.11.2017)
- [3.2.c] Heroku: https://www.heroku.com/ - (Accessed on 17.11.2017)
- [3.2.d] Firebase: https://firebase.google.com/ - (Accessed on 17.11.2017)

- [4.2.1.a] User stories: http://www.agilemodeling.com/artifacts/userStory.htm - (Accessed on (18.11.2017)
- [4.2.2.a] Apache: https://httpd.apache.org/ - (Accessed on 18.11.2017)
- [4.2.2.b] phpMyAdmin: https://www.phpmyadmin.net/ - (Accessed on 18.11.2017)
- [4.3.3.a] Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures - Representational State Transfer, https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm - (Accessed on 18.11.2017)
- [4.4.a] Dropwizard: http://www.dropwizard.io - (Accessed on 19.11.2017)
- [4.4.b] Dropwizard Configuration: http://www.dropwizard.io/1.2.0/docs/manual/configuration.html - (Accessed on 19.11.2017)
- [4.4.c] Dropwizard Views: http://www.dropwizard.io/1.2.0/docs/manual/views.html - (Accessed on 19.11.2017)
- [4.4.d] Gradle: https://gradle.org/ - (Accessed on 19.11.2017)
- [4.4.e] PostgreSQL: https://www.postgresql.org/ - (Accessed on 19.11.2017)
- [4.4.f] PostgreSQL JSON Functions and Operations: https://www.postgresql.org/docs/current/static/functions-json.html - (Accessed on 19.11.2017)
- [4.4.g] Dropwizard JDBI: http://www.dropwizard.io/1.2.0/docs/manual/jdbi.html#man-jdbi - (Accessed on 19.11.2017)
- [4.5.2.a] Jetty: https://www.eclipse.org/jetty/ - (Accessed on 23.11.2017)
- [4.5.2.b] Jersey: https://jersey.github.io/ - (Accessed on 23.11.2017)
- [4.5.2.c] Jackson: https://github.com/FasterXML/jackson - (Accessed on 23.11.2017)
- [4.5.3.a] Dropwizard JDBI: http://www.dropwizard.io/1.2.0/docs/manual/jdbi.html - (Accessed on 23.11.2017)
- [4.5.3.b] Liquibase: http://www.liquibase.org/ - (Accessed on 23.11.2017)
- [5.2.a] O'REILLY's RESTful Web APIs: http://shop.oreilly.com/product/0636920028468.do
- [5.2.b] HTTP/1.1: Method Definitions: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html - (Accessed on 23.11.2017)
- [5.2.c] HTTP/1.1 Method Definitions: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html - (Accessed on 23.11.2017)
- [5.3.a] JWT: https://jwt.io/ - (Accessed on 24.11.2017)
- [5.4.a] Leonard Richardson, Mike Amundsen and Sam Ruby: RESTful Web APIs (2013), ISBN13: 9781449358068
- [5.5.a] Todd Fredrich: RESTful Service Best Practices: Recommendations for Creating Web Services, https://github.com/tfredrich/RestApiTutorial.com/raw/master/media/RESTful%20Best%20Practices-v1_2.pdf - (Accessed on 24.11.2017)
- [5.5.b] Mustache: https://mustache.github.io/ - (Accessed on 24.11.2017)
- [5.6.1.a] OWASP TOP 10 - 2017: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf - (Accessed on 24.11.2017)
- [5.6.1.b] JDBI: Convenient SQL for Java: http://jdbi.org/jdbi2/ - (Accessed on 24.11.2017)
- [5.6.1.c] SQL Injection Prevention Cheat Sheet - OWASP: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet -(Accessed on 24.11.2017)