The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Mantas Sakalauskas

ID Number: 4226412

Supervisor: Professor Graham Hutton

Module Code: G53IDS

2017/03

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Issue Reporting and Tracking System For Browser Applications

Mantas Sakalauskas
Student ID 4226412
psyms9@nottingham.ac.uk
mxs04u@cs.nott.ac.uk
School of Computer Science
University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature: ____________________

Date: ____/____/____

Abstract

Software quality is an extremely important issue in today's world. As the World Wide Web grows and mobile is taking over desktop usage, companies and individuals begin running into various kinds of issues while developing complex new software or using existing software. These problems, usually referenced to as 'bugs', are usually the most time-consuming distraction that teams face during application development, testing and maintenance processes.

In most of these cases, the developer teams use tools which allow them to track issues. However, most of the work to describe and log issues is done manually by managers, quality engineers or other people responsible for quality of the product. Usually, after a problem is discovered, the user must manually enter issue related data as plain text to the existing tracking tools or forward it to people responsible in any inconvenient way. To fix the problems reported, developers must recreate the environments that were in place when the issues occurred. However, this becomes a very difficult and tedious work in modern browser-based web applications because of their interactivity, complexity and variety of events happening in the background.

In this dissertation, a complete design and an implementation of a software bundle that attempts to solve this issue by providing multiple interfaces to report and manage issues will be presented. The resulting application is then evaluated against the business needs, research and the requirements for the software set initially.

The project goes beyond standard software engineering projects because of the extensive research needed, modern API usage and browser-based development techniques not commonly used to achieve the required results.

A beta version of the system can be used in a real environment using the instructions found in Appendix 6.

Acknowledgements

I would like to extend my sincerest thanks to those people who helped me to plan, design, implement and further develop this project. I would like to extend my gratitude to all the academic staff in the School of Computer Science at the University of Nottingham.

I would like to recognize Professor Graham Hutton, my final year project supervisor, for continuous support and guidance when developing the project. I am very thankful for his swift and helpful responses throughout the entire year.

A special thanks is also extended to the entire team of Assimil8 Ltd. for sponsoring this project externally and providing me with feedback and support to keep the development process on the right path.

Table of Contents

1. Introduction

1.1. Motivation

1.1.1. Overview

As global internet usage grows every day, almost 50% of the entire population [1] is now connected to the World Wide Web. Such great internet adoption rates ultimately lead to a never-before seen growth in the web development community – with new frameworks for web applications appearing every few months [2], best practices changing monthly and the demand for developers growing every day.

Unfortunately, with new software being developed at such a rapid pace come new problems. However well tested and designed modern such applications may be, issues (also known as ‘bugs’) do arise from time to time, as the applications are orders of magnitude more complex than any from the previous century. These bugs cause end users stress and inconvenience, while developers have much trouble when attempting to track the issues and develop solutions to them. As a result, there is a great opportunity in the market for a software package that would attempt to involve both end users and developers of applications to report issues conveniently and improve communication while developing solutions.

1.1.2. Use Cases and Business Problem Examples

In enterprise business intelligence [3] software like IBM Cognos [4], Business Objects [5], Qlik [6], final solutions do not enable the user to comment on business data when running interactive HTML reports and/or dashboards that are context aware (with filters or different paths taken). An example of such a case would be a company’s Chief Financial Officer running a liquidity dashboard (a report that displays liquidity related data) - it would be filtered by date, divisions and account types, then, if an issue in the business that relates to the revenue pie chart is found, the software would not allow to comment or highlight the problem for other users of the system. With Commentary enabled, a comment could be made so that anyone else running the dashboard and applying the same filters would see the CFO comment and be able to add their own comments.

Another use case is having the ability to record data from an end user. If an end user has an issue when using a Cognos report, i.e. data is incorrect, web page errors, performance issues, alignment issues with dashboards, there would be an ability to take a screenshot, comment and record the issues with the report in real-time. All information about the user session, occurred events would be recorded so that central help desk team could consider the issue. This could be an extremely important feature in large enterprises with thousands of employees worldwide. As an addition, such a use case could apply to any type of application other than enterprise business intelligence systems.

1.2. Aims and Objectives

The aim. This project attempts to create a tool that would allow users and developers of any web application to easily log, track and resolve issues. It would accomplish this by tracking runtime application data as users interact with it, then logging it to the system for further analysis if a problem gets reported. The data tracked includes user input, network, console events, screenshot of the state and other relevant environment information.

The objectives. An initial version of the product would consist of three components, which combined would help reach the aim of the project. The objectives of this project are as follows:

1. Develop a software package, which includes: a plugin tool to report issues, a web interface for administration and reviewing issues, a backend service to manage communication and persist data.
2. Allow clients to report issues with various types of supporting data:
    a. Screenshot of current application state
    b. Drag and drop tool to add comments on any part of the visible application
    c. Draw shapes onto the application to better indicate issues
    d. Network events along with all their data
    e. Console events
    f. Runtime error data
    g. Manual user input sequences, such as keystrokes and mouse clicks.
    h. Browser and operating system version, screen resolution.
3. Allow developers to replay client input sequences which do not modify any data
4. Implement a security model which would allow administrators to manage access for users in local installations of the software
5. Implement an authentication mechanism to use in server–client communication. The mechanism must ensure secure transfer and storage of sensitive data.
6. Implement smart safeguards which would detect sensitive data transferred and would encode it to prevent possible exposure of data.
7. The entire software package should be thoroughly tested by unit and integration tests. Overall test coverage is aimed to be at least 80%.
8. The plugin tool must support at least two latest versions of any modern desktop browser, however is aimed to support any browser newer than Internet Explorer 11.
9. The plugin tool developed should be able to work in any environment and not interfere/alter other running JavaScript frameworks or libraries, CSS styles or HTML mark-up code.

2. Related work and Research

2.1. Existing tools

Overview. Currently, there exist very few tools which help developers counter the issues described in the project brief. Even though there is a small number tools which do address these specific problems that this project is trying to solve, they have their own major limitations and weaknesses.

During research, I have managed to find a total of three tools having a very similar end goal as this project, however all of them are approaching it differently. I have examined each one of the tools and considered their strengths and weaknesses when planning this project.

Analysis. First of the tools is called TrackDuck [7]. It is being used as a feedback tool for end users.

Strengths:

1. Can be linked to many popular issue tracking applications.

Weaknesses:

1. It is only being sold as a service, thus having a monthly fee instead of a single purchase for use.
2. It does not support local installations and private data storage solutions, which is a very important factor for corporate use.
3. The tool is more oriented towards end users in production environments, therefore the amount of information tracked is very limited.

Second tool that I have found during research is named UserSnap [8]. It has a very similar goal to this project, however lacks many key features for team collaboration and installation flexibility.

Strengths:

1. Has browser extensions for cases where a plugin tool cannot be used.

Weaknesses:

1. Very large ad-hoc plugin tool, which is inconvenient for end users to load.
2. Is only sold as a service with a very high monthly and maintenance cost.
3. Does not have logging of most events happening in the background.
4. Does not support logging in, only reporting issues as guest.

The final tool discovered is BugMuncher [9]. It has a very similar feature set to UserSnap, however has more drawbacks.

Weaknesses:

1. Has tiers of pricing, which is very extensive and paid monthly. Users with lowest tiers do not get access to most of the offered features.
2. Screenshots are not consistent in many cases, because they are taken without user's interaction and rendered from gathered assets on server machines.

2.2. Research Methodology

The research methods. After considering various research methods, a decision was made that it is very worth investigating online developer communities. Examples of such communities are StackOverflow [10] and Reddit [11], the latter being the most popular result per Google Trends [12]. The analysis of data found in such communities was done by grouping data to relevant topics, types and keywords [13]. Another type of research is the summarisation of personal knowledge that has been collected by working with web developers. Their experiences and insights helped me form a basis for the project requirements.

The developer communities. Since developer communities online are the most common method of sharing knowledge and finding solutions to problems nowadays, it is a good place to do research. After analysing various StackOverflow discussions based on topics related to 'issue', I have found that most web developers use tracking tools such as Jira [14] or GitHub [15] which only support manually added data. Most people that participated in the discussions were looking for an automated solution which would allow them to track issues right away, but could not find one that was lightweight and would be simple to integrate [16].

The web developers. The knowledge I have collected from working together with web developers has also proved to be very good input for this project's research part. It has helped to understand the problem to very specific details and led to the initial structure and design of the application.

3. System Requirements

3.1. Overview

In this section, the requirements for the whole system will be discussed. All of them are split to functional and non-functional requirements, following the most common categorisation patterns [17]. All the requirements are based on research results. The non-functional requirements apply to all parts of the system, no matter whether they will be running on client or server machines, unless specified otherwise.

Each of the functional requirements listed is a simple action that can be easily tested with both unit and integration tests. On the contrary, non-functional requirements are only a basis that provides strict guidelines for designing the system, to ensure that the resulting product will perform well, be safe to use and be maintainable. They can only be evaluated by acceptance testing or end user testing in a real or simulated working environment.

These requirements have been used to design the complete system (with slight adjustments to the requirements during the development process) and to evaluate the actual system implementation's capabilities and functionality.

3.2. Functional Requirements

The plugin tool. Since the plugin tool is most technologically challenging and innovative, it has a very specific set of functional requirements. It is critically important that as much as possible of these functions are supported by the first production release of the project. The tool must have:

1. An intuitive but not intrusive way of opening the editor view of the plugin
2. An editor view, where a user would be able to input information regarding the problem

3. A screenshot feature for the editor, where the user would be prompted to take a screenshot and attach it to the 'Issue' report.
4. A shape drawing feature, where the user would be able to draw simple shapes on top of the page
5. A commenting feature, where users would be able to drag and drop new comment containers on the page and edit the text.
6. A communication mechanism to the server, allowing user to insert (save) the new issue.
7. Guidance for every part of the editor view (such as help text on hover, action request prompts)
8. An authentication user interface, which would allow reporting users to identify themselves

The administration portal. The administration tool will be the main portal for developers and other users to review active issues and discuss them. Therefore, it must be both intuitive and modern in a way that would make users enjoy using the tool. The administration application must support:

1. A login (authentication) view and mechanism to personalize all the content inside of the application.
2. A home page, where a list of available actions and the latest issues would be displayed
3. A settings page with personalized preferences for each user
4. A user management page to allow management of users (only if authorized to do so) – it must allow modification of data and assigned groups for each user
5. An application management page, where authorized users would be able to add new applications for issue tracking and generate the special plugin tool code
6. A user interface for reviewing application's issues
7. Closing, reopening, commenting on each of the issues raised.
8. An interface for reviewing screenshot in full size

The backend server. This service will be a central point of communication for both the plugin tool and the administration portal. Therefore, a very specific set of requirements can be deduced:

1. The server must handle all data persistence related tasks, such as storing to the database and retrieving from it
2. Must implement a stateless authentication algorithm to support authentication of users on an unlimited fleet of service instances
3. Must store and retrieve data entities and multiples of them, such as Issue, Dialogue (set of comments), Comment, User, Application, Screenshot, Network Event, Console Event data
4. Runtime configuration, such as database connection settings, maximum concurrent user count must be configurable in properties files without redeployment.

3.3. Non-Functional Requirements

Compatibility. The whole system must be very compatible in terms of hardware and software, since it will be used in very different environments across various types of machines. The compatibility requirements can be split into two, one for the frontend components and other for backend components.

The frontend components will be only executed on client machines, therefore full compatibility across a suite of browsers must be ensured. This includes at least two latest major versions of Internet Explorer, Microsoft Edge, Apple Safari, Google Chrome and Mozilla Firefox. A basic functionality should still be available on older versions of these browsers starting from Internet Explorer 10 and later, but slight problems may occur and it would not be considered a critical issue. The operating systems supported must include Apple macOS, Microsoft Windows and any Linux platforms that support any of the aforementioned internet browsers.

The backend server must be able to run on any machine that is capable of running at least Java SE 8 with a Tomcat servlet container of version 7 and upwards. The server must be able to communicate with most modern relational databases (such as PostgreSQL, MySQL, Microsoft SQL Server) which have a public JDBC [18] driver available. Any less known relational databases should still be supported, however specific configuration for the databases might have to be performed and a stable version of a JDBC driver must be available.

Scalability. The entire system must be able to scale and support an indefinite number of users attempting to report issues simultaneously. This requirement must be achieved by implementing a token-based authentication mechanism, which would greatly improve scalability since no cookies would be used [19]. It must also support SQL server instances running on any machine accessible over network connection. As well as being stateless in the backend, the frontend parts must support storage on content delivery networks, which greatly improve load time by distributing content to cache servers located close to users [20].

Performance. The application must be able to perform well under heavy load conditions, such as when multiple users attempt to report issues at the same time. The server must support concurrent connections and handle database access in sequence. The administration portal must cache code and image files between deployments on client's or content delivery network's computers to reduce perceived load time for users [21]. The plugin tool must be very lightweight (less than 20 kilobytes after applying a compression algorithm) and use the least possible number of libraries to reduce impact on load time and performance.

Security. The Commentary application handles a lot of user data, thus there is a high risk that transmitted data might be sensitive. Problems with sensitive data may occur at any point of using the application: data collection, transmission, validation, system logging, storage.

Therefore, to prevent any misuse of the data ant compliance with the Data Protection Act [22], each of the steps must be carefully considered.

Accidental sensitive data collection may occur when collecting data in the background. To minimise the risk of collecting such data, the collection algorithms must specifically filter out any input from secured data input fields, password fields and fields with auto-completion disabled.

The entire software package must ensure secure transmission of the data between the client and the server, as well as the server and the database. The first requirement can be met by only allowing Commentary components to communicate via HTTPS protocol with SSL/TLS encryption enabled [23]. This must be ensured during deployment of the application, as the machines which will serve Commentary must have a trusted SSL certificate installed and standard HTTP access must be disabled on all of them.

After the data is transmitted from the client to the server, the server must perform compulsory access control checks and validate the input before proceeding to persist it in the database. In order to overcome this issue, an access control mechanism that restricts data access only to users with required permissions must be implemented.

It is crucial to minimize the risk of sensitive data appearing in any type of logging, therefore logs should never print any possible sensitive data fields, such as actual network, console or user input events.

Lastly, safe storage of data must be ensured. To achieve this, the database must be set up in a restricted network access environment (usually, the internal network without internet access or behind a firewall), with authentication details only provided to the Commentary Server instances that are permitted to access it.

4. Design

4.1. System Design

4.1.1. High Level Overview

The system must be carefully designed to support all the functional and non-functional requirements. It must be considered that the development is done by taking cues from the Agile methodology [24] and design may change with each iteration of software, however core principles and high level design should always apply.

The Commentary system consists of three main components – the Plugin Tool, Administration portal and Commentary Server. Each of them runs on entirely different machines and environments, since isolated components are used to perform specific subsets of system features. The Administration Portal and Plugin Tool components are executed on client machines, as they are both user interfaces for Commentary. The server runs on a hosted machine, either local or in the cloud.

The Administration portal and Plugin Tool both rely on the Commentary Server component. Plugin Tool uses it for storing new issues, while the Administration portal accesses existing issues and modifies their data. Both components' code is hosted and served to clients by the Server as well. The only dependency that Commentary Server itself has is a cluster of nodes with a relational database engine running (cluster size depends on overall system load). This exact system structure can be easily shown as a diagram, seen in the Figure 1 below.

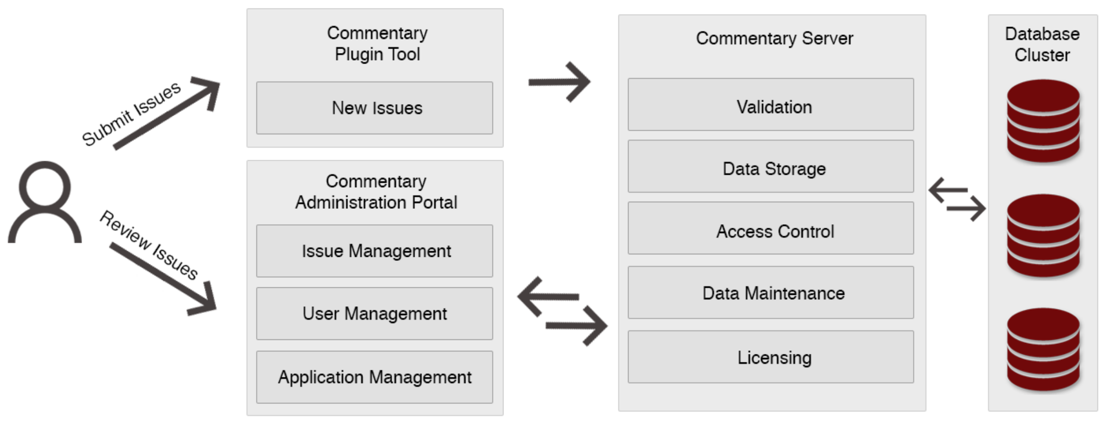

Figure 1

Any Commentary user can either submit a new issue via the Plugin Tool (which should be embedded on any running web application in advance) or review and communicate on existing issues using the Administration portal. As mentioned above, both the Plugin Tool and the Administration portal retrieve and persist data through the Commentary Server, which ultimately results in high load to the server itself. To allow high concurrent user count and large data transfers, Commentary is designed with stateless design in mind for every aspect of it. Provided that all the authentication, authorization and other business logic is designed and developed using stateless patterns (such as tokens, which have all context and required data embedded), the instances can be scaled horizontally (increase the number of actual Server instances which serve clients) arbitrarily, thus increasing response time and, ultimately, end user experience [25].

4.1.2. The Plugin Tool

As mentioned before, the plugin tool is the user interface that should be used when new issues need to be reported. It provides the user with a convenient toolkit to add comments, take a screenshot, draw shapes and record events of a web application. Since the tool will be embedded inside of existing applications, there are two critical requirements – it must be lightweight and not interfere with existing code of the application.

To keep the overall size of the plugin tool small, a framework cannot be used. Therefore, the entire codebase for it must be written using native code. Ultimately, this allows fine-tuning the code to be as lightweight and efficient as possible, but increases the development time by an order of magnitude. Since there is no framework or toolkit that is of acceptable size to embed inside of the plugin tool, a design decision was made to create a small reusable toolkit inside of the plugin tool. Such decision allows to reduce code repetition, increase readability of the code and still allow a very lightweight script to be developed.

To reduce the likeliness of interfering with existing application code, the Plugin Tool has name-spaced stylesheets and DOM objects, which all follow a specific pattern of prefixing element names. As a result, the deployment and maintenance of the plugin tool is minimized, since deployment on most (even complex) applications should be as straightforward as adding a <script> tag without any modifications to the plugin tool code itself.

The actual Plugin Tool script is served by the Commentary Server component. This allows customization of the tool's parameters, such as being hidden by default (invoked with a command), various plugin types (allows further development of plugin tools oriented towards specific applications) and the application that it is being deployed on (used for licensing). As a result, deployment of the script is much simpler, since no actual files have to be transferred to the application's bundle – all that needs to be done is a script tag inserted with source referring to the Commentary Server. An example of such a tag can be seen below.

```
<script src="https://comentary:8080/gen/plugin/appId=7a89ef2a&hidden=false&type=standard">
```

Code 1: Embeddable Plugin Tool Script

Before submitting an issue, the user is requested to authorized via a simple login prompt. Authorization is a critical part of the issue submission process, since it reduces chances of invalid input and system misuse, as only authorized users are allowed to use the tool. As an addition, it also gives developers and administrators the opportunity to contact the issue creator for further details, as every issue then has a creator assigned automatically.

After an issue is submitted, the Plugin Tool automatically closes and the application returns to its normal working state.

4.1.3. The Administration Portal

It is the primary portal to access issue data, manage applications, add or remove users, change permissions and generate plugin tool scripts. Since such a huge feature subset is supported by the administration portal, it is the primarily used interface for Commentary. As a result, it must be very performant and reliable while both enjoyable and easy to use.

To support such a large set of requirements, a good software architecture design had to be developed. A decision was made to use a modern frontend application framework and develop the tool using best practices, while always striving to have a maximum test coverage.

Several core architecture decisions were made in advance of developing the software itself. All the design patterns that are used in the Administration Portal were chosen based on strong arguments that prove why the system would hugely benefit from using the pattern. Below is a list of specific architectural decisions chosen and their benefits:

- Usage of a state container to store persistent application state that is shared across components
    - Ensures consistent behaviour across components by ensuring that there is no redundant data
    - Allows separation of data logic from visual components
    - Allows high precision control over the application state with many good functional programming concepts, such as immutability, function composition and pure functions
    - Allows implementing a unidirectional data flow, where some actions or events cause data to change in a specific way, thus increasing predictability
- Separate all Commentary Server related side-effects (such as storing or refreshing data) from components' logic using a single pattern
    - Provides a common interface for extending the application
    - Allows simpler testing
    - Further increases predictability by controlling chains of events that happen separately from user interface logic.
- Follow Test-Driven Development practices for all state related components
    - Allows detection of problems in the foundation of the application early in the development process
    - Easier to detect bad design decisions by continuous testing
- Split all components that perform any operations on state (referenced to as containers) and components that are only created for display purposes (referenced as components)
    - Reduces the number of components which may be a cause of an unexpected change in the application state. As a result, time is saved while debugging and developing
- Use a strongly-typed programming language
    - Reduces the risk of accidental data manipulation or incorrect arguments. Therefore, usage of types (models) across the entire Administration Portal. application is required.
- Attempt to create shared services for common tasks that do not modify application's business-related state
    - Reduces code repetition and improves testability

4.1.4. The Backend Server

The overview. The highest load will always occur on the backend server, since it is responsible for both serving the front-end components and computing the data (validating, modifying and persisting). Therefore, it is crucially important to design the component in such a way that it would be easy to maintain, reliable and very scalable. Since it is very probable that the application could become large, a modern, well-maintained object-oriented language should be used. All key design decisions and the reasons behind them are such because an object-oriented language is to be used (most of these design decisions are not applicable to any other type of language).

Layered design. The entire application structure is designed in layers. This allows high concern separation [26], therefore reducing the amount of code in each class. Because of concern separation, each class only has code needed for some specific task, which allows execution of tasks by delegation to other layers and results in highly testable code. The general layer structure for the Commentary Server is as in the figure shown below. The resources layer is responsible for receiving requests, parsing them and forwarding to compute further to the service layer. The service layer is the primary point where user input data is computed and being modified to be compatible with all data storage mechanisms, as well as performing some task-specific logic. After the service layer finishes executing (or, in some cases during execution), in most cases it calls the DAO (data access object) layer, which is only used for data persistence and retrieval for specific entities. Once all the layers have completed execution, the call stack should return to the resource layer and it would compute a response to the user, as well as execute some generic post-request tasks if needed.

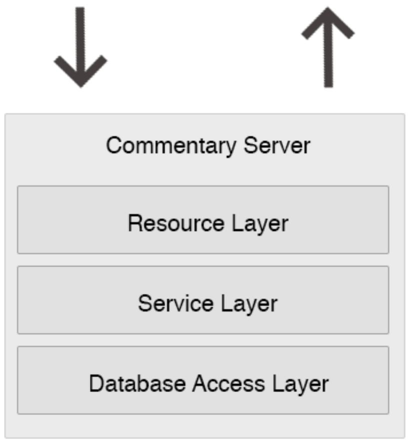

Figure 2: Layered Server Structure

Dependency Injection. One of the key benefits of using a modern object-oriented language is the abundance of dependency injection frameworks and tools. Dependency injection is one of the core concepts/techniques strictly followed throughout development of the entire backend server. This mechanism, where single instances of objects can provide dependencies to other objects, results in code which is much more predictable, since no additional instances of objects are being created when needed, instead the same instance is injected throughout the application. For example, only single instances of the service layer classes should exist, since it usually contains stateless methods which produce output based on input. Another great benefit of dependency injection is the given possibility to configure the application to much greater extent than would normally be possible. As a result, it is possible to change critical application parameters before runtime, instead of compile time, thus the entire application can completely change its behaviour without recompilation or redeployment.

Stateless programming. The aforementioned design decisions do not have any direct impact on the applications' scalability. According to best RESTful API practices [27], it is crucial that the server does not store any client's session data to retain high scalability. Instead, the required context and data should be passed with every request from the client and made possible to recognize by any instance of the server. In the case of Commentary Backend Service, such technique is used throughout the authentication and authorization processes. The client is issued an access token with a pre-set validity period, encrypted and signed with a shared server private key, which can be verified by any server instance that has access to the same private key, but cannot be modified by the client. This eliminates the need to store any client related data, even though authentication and authorization is performed on a per-request basis.

Aspect Oriented Programming (AOP). Even though layered architecture is implemented, there would still be lots of duplicated code for common tasks, such as access control (e.g. permission to perform a task checks). A programming paradigm called AOP is one of the cleanest and practical solutions to this problem. The general idea of this paradigm is that any code which is performed on a standard basis across multiple parts of the applications can be modularized and separated. It can be achieved by marking the code which needs to perform some specific logic before or after execution instead of modifying the code itself. The result is code, which is cleaner and has all the cross-cutting concerns separated from the code itself. A code snippet seen in the figure below briefly shows how the Commentary access control and cross origin management mechanisms are separated from the response layer's code, at the same time making it clear to developers that such security features are in use.

```java
@POST
@Path("/new")
@CrossOrigin
@AccessControl(ADD_NEW_ISSUE)
public Response registerIssue(Issue issue) {
    logger.debug("Registering a new issue: {}.", issue);
    return Response
            .ok()
            .entity(service.save(issue))
             .build();
11 }
```

Code 2: Sample Resource Layer Method

Security Model. Commentary has a custom authorization mechanism, which is based on usergroups and permissions. A usergroup can have users, applications and permissions assigned to it. Each user that is part of a specific group has access to all the applications and all the operations requiring the permissions that the group has. If a user is a member of multiple usergroups, all the available permissions and applications are combined.

Such a security models allows fine-grained control over what specific types of users can do, at the same time minimizing the effort needed to administer them. A future improvement to the system will be support for overrides or additional permissions to specific users, rather than groups.

Error Handling. Commentary is designed to be reliable by making assumptions that things will go wrong and errors can happen in any part of the application. Because of such a defensive strategy (more commonly known as defensive programming), the server side architecture has an application-wide exception handling mechanism, which is hierarchical (implemented using inheritance, see Figure 3 below). Such a design potentially gives the ability to distinguish errors to very fine detail, therefore presenting specific errors and reasons to the client of the application. The exceptions on the very high level are classified as client or server type. Some examples of client type errors are: invalid input, unauthorized access, or anything else that can be retried. Server type errors could be any of: general persistence errors, execution errors, configuration errors or any other errors not directly related to user input. Both error types are further distinguished (using inheritance) to very specific errors, such as authorization, persistence or parsing. As a result, user-friendly descriptions along with the exception data can be provided to the user.

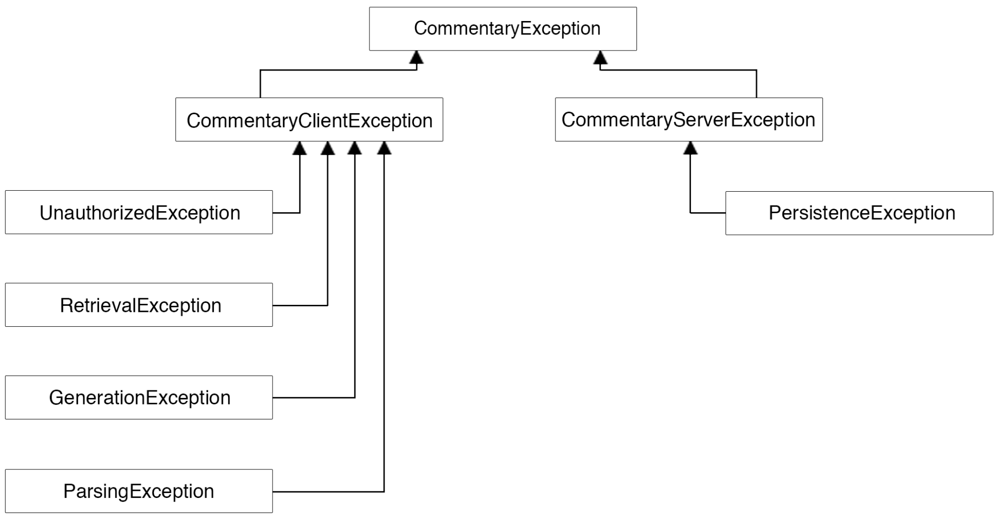

Figure 3: Exception Hierarchy

Access Control. As explained before, Commentary follows a defensive design strategy, under the assumption that users will attempt to compromise or misuse the system from time to time. To prevent such misuse, a robust access control mechanism is required. Commentary has a custom security module inbuilt which is responsible for managing all authentication and authorization before any incoming client request is accepted. The security module components get automatically invoked between requests by taking advantage of the aforementioned AOP design principles. Inside of the module, specific authentication (user identification, password validation) and authorization (capability to perform specific operations) checks are performed. In case any mismatch is found, an authorization error is returned to the user, rejecting the request immediately. The flow chart below briefly shows the process when a mismatch is found and an error is thrown.

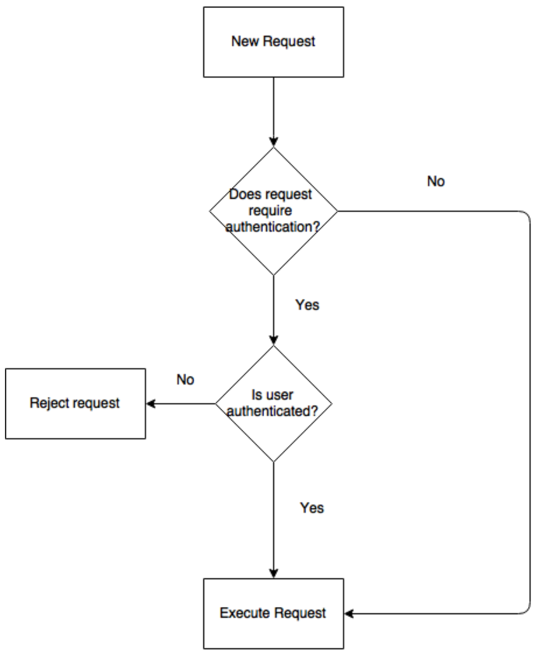

Figure 4: Authentication Execution Flow

Package structure. The Commentary Backend Server package structure is following a strict pattern. All the packages are split by modules, or, in other words, aspects (as defined by Gregor Kiczales in his conference paper about AOP [28]) that each represent a different cross-cutting concern (such as core, security, exceptions, resources, services). Such modularity results in time saved during development process, since related classes are found nearby. Another outcome is that such modular structure inherently compels developers to reduce code coupling between different classes, since it is very difficult to distinguish execution flows between different packages. Appendix 2 includes the entire package structure with brief explanations of every aspect.

Inheritance and Abstraction. Following some of the best object-oriented programming practices from the book Clean Code [29], Commentary backend code implements inheritance and abstraction as much as possible. For example, all the service and DAO layer classes must extend from specific abstract classes, which provide base methods that are commonly used.

In the case that the specific implementation needs to override the methods, it is easily possible to do so. Such approach reduces code redundancy even further and makes testing much easier, since only a single class must be tested, instead of multiple. In the cases where a method is being overridden, only unique tests for that specific method are enough to cover the entire class fully.

Verbose logging. As part of the defensive programming, Commentary implements a reliable logging mechanism. It is a design requirement that each of the steps in the request processing chain logs at least a single message with the entities that are being worked with and the operations that are being executed. This is an important part of the overall application design, since it provides a way to track down and reproduce issues, add monitoring and system failure alerts in real production environments.

Documentation. As the application codebase is growing all the time, it is important to keep track of features and couplings between classes. When implementations and dependencies may change, the task (or procedure) that a method or a class is required to perform will most likely not change. To keep track of the features that each part of the code performs, all the code is documented and supporting documentation is shipped with each release of the software. The documentation includes a brief explanation of the operation, links to relevant operations (sometimes with explanation of difference between the operations), required parameters, return values and a list of possible errors.

Programming practices. Lots of programming practices are design cues are taken from Robert Martin's book Clean Code [30]. The entire Commentary Backend Web Service codebase is designed to have small functions that are small and only do a single task, have descriptive names and attempt to have the least number of side-effects next to atomic calculations. The code is written with low vertical density (empty lines between statements) and a single pattern of indentation for readability purposes.

4.2. User Interface Design

The design language. Commentary is designed with a specific design language and rules in mind. The basis for it is taken from Google's Material Design [31] and its guidelines. Material Design is described as "a visual language for users that synthesizes the classic principles of good design with the innovation and possibility of technology and science" [32]. The Commentary user interface design extends Material Design, and the combination results in a harmonised user interface that is both easy and comfortable to use and adapts to any device. These are the key patterns that are reflected throughout the Commentary user interfaces:

Colours. A single bright primary colour throughout the application is used to mark important areas. Combined with lighter and darker shades of the colour, it helps to guide the users about interactive elements on the screen.

For example, the primary colour (or '500' in Material Design) is used to show the primary components of the application, such as the Commentary Administration Portal action bar.

Darker shades of the primary colour can be used to indicate other very important areas of the application, while lighter shades should be used to present less important details, for example expanded descriptions of issues or details about a specific user.

An accent colour should be used to indicate anything that the user can interact with, including "elements, such as text fields and cursors, text selection, progress bars" [33].

Commentary extends these ideas by incorporating a dark shade of red as the primary colour and a very light shade of grey for the accent colour. This allows to make the application both look modern and professional at the same time, since the colours are a common combination for business-oriented applications. The resulting colour scheme can be seen in the screenshot of the Administration Portal.

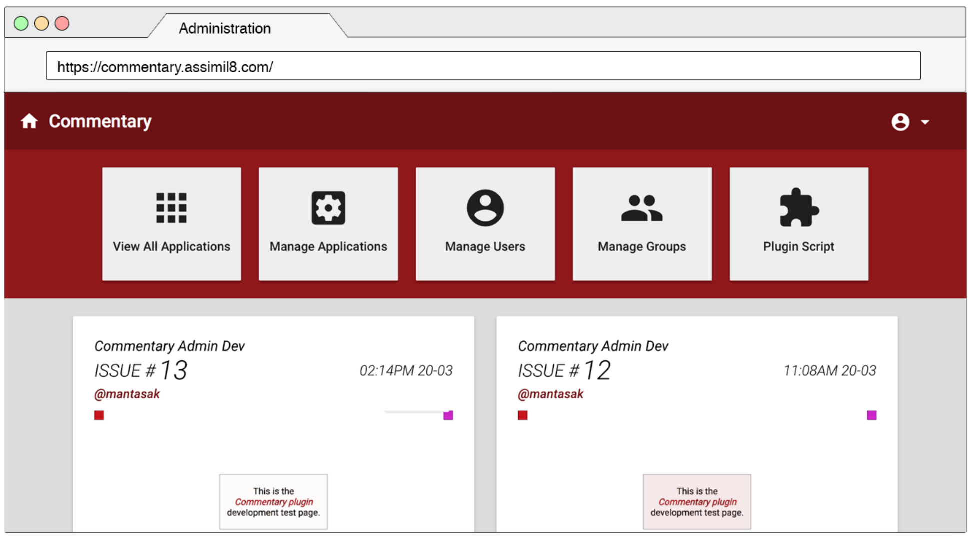

Figure 5: Administration Portal in a simulated environment

Animations. Motion is crucial for any interface, whether it is an input field or a container with text inside. Complexity of interfaces usually causes extra confusion for the end users, therefore animations can be used to minimize the stress.

Animations should always start from the origin – usually that is the interaction point, for example the location at which the button was pressed.

They should always be intentional – to guide the user about changes on the screen – e.g. show which objects on the screen are new and which are being removed.

According to Material Design guidelines, "Animations must be responsive. When you take an action, feedback is immediate" [34]. This is a key requirement for all the animations, since long animations may even reduce the user experience.

Elevation. All elements have shadows based on their elevation, just like objects in real world.

Elevation and shadows help users to distinguish objects based on their relevance and recognize which objects can be interacted with. As an addition, object transformations (such as scaling) always reflect in shadows and elevation as well. Commentary follows general Material guidelines for element elevations, having a total of two elevation types– resting and dynamic [35].

Resting elevation is the standard state for any given element, which is based on the element's relevance and importance to the user. Dynamic elevation is an enhanced elevation, described as "the goal elevation that a component moves towards, relative to its resting state" [35]. The latter usually shows a bigger and more spread shadow, indicating that an element is being resized or interacted with.

5. Implementation

5.1. The Plugin Tool

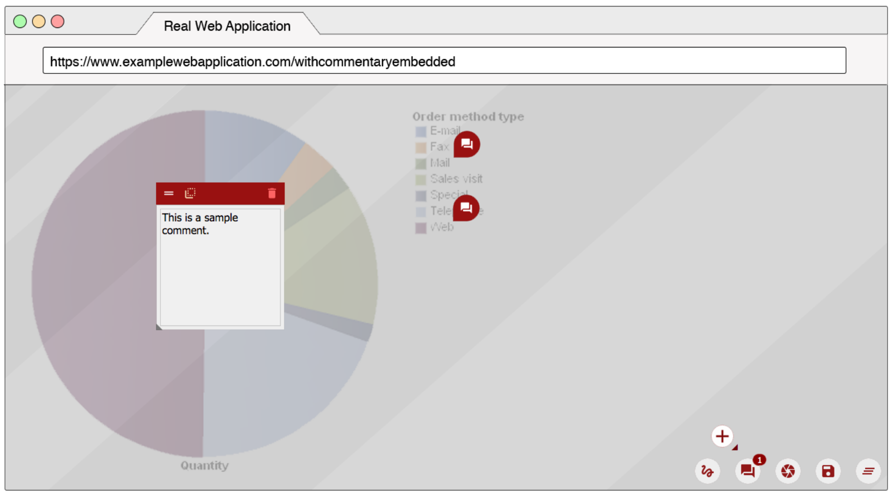
Real Web Application
https://www.examplewebapplication.com/withcommentaryembedded
Order method type
E-mail
Fax
Mail
Sales visit
Special
Tele
Web
This is a sample comment.
Quantity

Figure 6: The Plugin Tool in a simulated environment

Technologies Used. The Plugin Tool is implemented using native JavaScript (ECMAScript 5, as defined in the original specification) and CSS3, both of which are natively supported by all modern browsers. ECMAScript 5 support ensures backwards compatibility with most browsers starting with Internet Explorer 6. As a result, users would be able to submit issues even for legacy applications that run on old machines.

Lightweight Implementation. The first challenge during development of this tool's initial version was making code both modular and lightweight. The current implementation achieves a compressed size of 13.5 kilobytes (using GZIP compression algorithm, which is the standard algorithm in HTTP). This combined size includes both the JavaScript code of the plugin tool and the CSS stylesheets required. However, this result is achieved without any code minification, which would allow saving an extra 30-50% size on average.

Internal Toolkit. After a few attempts trying to separate concerns, a decision was made to split all the code into several JavaScript objects – listeners, utilities and configuration. Inside each of these objects, the code is further split into subcomponents of the plugin tool (comment related functions, screenshot related functions, user event related functions).

The global *listeners* object defines (does not register them) the functions which are invoked on certain events throughout the DOM. The *config* object is modified on the Web Service side before each request with specific settings required. It contains all the customizable settings, such as plugin type, hidden by default and assigned application ID. The *utils* object contains a minimal library, which can add, modify and remove new DOM objects, add specific plugin tool components to the screen, take screenshots, bootstrap the application or submit the issue to the server.

```
var state = {
      //... The state is stored here
};

var listeners = {
      //... The event listener functions are defined here
};

var config = {
      //... The configuration for plugin tool is stored here
}

var utils = {
      //... The inbuilt minimal utility library
};

//Bootstrap / Append sidebar
(function () {
      utils.common.overrideConsole(window.console);
      utils.common.overrideHttpRequests();
      utils.common.overrideEvents();
      utils.common.insertCss('/commentary.css');

      utils.init();
})();
```

Code 3: Plugin Tool code structure

Unique Namespace. One of the core requirements for the Plugin Tool is its ability to be used throughout various applications without any direct impact on the application's behaviour, including load time. As explained in the design documents, name-spacing components is the primary solution to the problem. All the DOM objects created by the Plugin Tool are prefixed with '*commentary*' in the CSS selectors. Such an implementation drastically reduces the chances of Plugin Tool's styles interfering with any existing DOM objects.

Screenshot Capability. One of the major problems encountered during development was the inability to render screenshots of applications because of browser security restrictions. After researching the possible ways to capture what the user sees, the most accurate and user-friendly way to take screenshots was chosen – requesting user to take a screenshot using the operating system's default screen capture mechanism and pasting it from the clipboard to the plugin tool. It is the safest method to ensure that screenshot is identical to what the user sees (since most other alternative ways of taking a screenshot involves rendering it on the server side with a simulated environment).

However, the implementation has difficulties, since each operating system does screen capture in a unique way. To address these problems, other runtime parameters are taken when taking the screenshot. The parameters include current screen resolution, number of screens, current browser position on the screens and the browser window size. After collecting this data, the screenshot image is provided together with it, as it is cropped on the server side to contain only the browser window.

Freeform Drawing. As well as adding comments and taking a screenshot, users can draw free form drawings on top of the screen. Once an issue is submitted, a rendered image with the freeform drawings is saved to the database along with the provided screenshot.

The implementation uses an HTML5 standard *canvas* element. The current iteration of Commentary Plugin Tool only supports drawing using a single colour, however it can be successfully used to achieve any form or shape.

5.2. Commentary Administration Portal

Overview of Languages and Frameworks. After considering all the design requirements, a choice had to be made regarding the framework to be used. Angular 2 [36] and React [37] were considered initially, however Angular 2 was chosen to be a better tool for this component of Commentary. The main reasons behind the choice were great support for testing tools, active developer community and the fact that the framework is very new and is likely to be updated for many following years. These factors are very important for a user interface that exists in a constantly evolving environment (in this case, a web browser). A TypeScript [38] version of Angular 2 is being used for development. This allows even more concern separation, predictability and better overall quality of software.

State Management and Models. The Administration Portal's state is managed in a predictable state container [39] named ngrx [40]. It is a relatively new concept in web development, but has proved to be very beneficial in complex web applications. The whole application state is stored in a single object, called the *Store*. The *Store* contains many different *State*s, or sub-states, each of which represents a single persistent entity, such as a *User*, *Issue* or an *Application.*

Such a mechanism ensures that duplicate storage of the same entities would never occur across different components of the system.

Since the state structure is always identical, it is a very common practice to limit component access to only specific substates [41]. This increases reliability as components cannot accidentally update or read state which they are not required to read. Such an implementation also exists in Commentary, using helper accessor functions which are declared next to *State* definition.

```
export const getIssues = (state: State) => state.issues;
export const getLastIssues = (state: State) => state.lastIssues;
```

Code 4: State Accessor function example

Actions. To prevent accidental mutations and mismatches in the state, it is read-only. The only way to change any part of the state is to create and emit an *Action*, which is a plain JavaScript object with an *ActionType* (description of what needs to be done) and a payload (the new data given).

This allows centralization of the mutations to the state, therefore side-effects and any changes originating from the application components will always be performed in a single location, thus increasing predictability by eliminating race conditions.

Mutation using an action pattern is also beneficial because *Action*s are always executed in sequence. Knowing that each of the actions describes an intent to change part of the state, problems can easily be traced back by knowing the exact sequence of actions that have happened.

Reducers. An action describes what change is required to the current state, but does not describe specifically how the state should change. To do this job, the Commentary application has several pure functions [42] – one for each sub-state, called *Reducer*s.

A reducer is clearly explained within Redux (another state container for JavaScript) documentation as “a pure function that takes the previous state and an action, and returns the next state.” [43] An example of a Commentary reducer can be seen below:

```
export function reducer(state: State = initialState, action: actions.Actions) : State {
    switch (action.type) {
        case actions.ActionTypes.SUCCESS_LOADING_APPLICATIONS:
            return assign({}, state, { applications: action.payload });
        case actions.ActionTypes.SUCCESS_ADDING_NEW_APPLICATION:
            return assign({}, state, { applications: unionBy(state.applications, [
               action.payload ], (a) => a.id ) });
        default:
            return state;
    }
};
```

Code 5: An example of a Reducer

As explained before, Commentary state consists of multiple reducers. During application launch, the entire set of reducers are then combined to one single reducer which accepts all types of actions that the application supports.

Effects. Commentary takes predictability one step further by separating side effects from any other code. It does so by using an extension library for ngrx, called ngrx/effects [44]. The classes which hold the side-effect code are called *Effect*s. An effect is a simple concept that does only one task – it maps an *Action* that intents to execute some side-effect code to other actions, based on the outcome of the side-effect. The figure below shows an effect for a user registration action that maps to a success action or logs an error, depending on the server response.

```
@Effect()
register$: Observable<Action> = this.actions$
    .ofType(actions.ActionTypes.TRIGGER_REGISTER)
    .switchMap(action => this.service.register(action.payload.username,
          action.payload.password)
      .map(res => ({ type: actions.ActionTypes.REGISTER_SUCCESS }))
      .catch((e) => this.handleErrorFn(e, 'Could not register!', action))
    );
```

Code 6: Side-Effect Handler

Services. The robustness and usefulness of the state management mechanism dramatically reduces the number of services required to support other components. Since all the state management is done elsewhere, the only services that are required are for making server calls, handling errors and adding convenience methods to third party libraries.

All the server related services are located separately from other services, deeper in the folder structure. The *RestService* class has a basic implementation of all the network request types needed, including authentication header creation logic. All other services are split per entity, for example: *IssueService*, *DialogueService*, *UserGroupService*. Each of them are dependent on the *RestService* and include the prefix and specific endpoints needed for each provided API call. This leads to highly testable methods, as all of them only append the endpoint and forward the network processing job to the *RestService*.

```
public getForApp(id: String): Observable<any> {
    return this.rest.get(`${this.prefix}/app/${id}`);
}
```

Code 7: A side-effect method, dedicating the call to RestService

The *ErrorService* is responsible for handling and displaying all errors in the application. It does so by subscribing to the sub-state of the application, which stores error-related data. Once a new error is added to the list of errors, it is immediately shown to the user with a brief description.

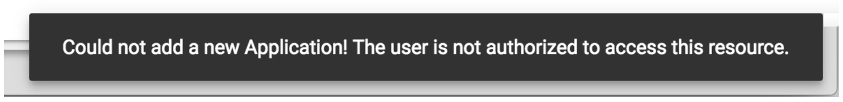
Could not add a new Application! The user is not authorized to access this resource.

Figure 7: A sample error message

Containers and Components. The visual components of the application are separated into two distinct groups – containers and components.

Components are visual elements which only handle internal user interface based state (not relevant to other components). As a result, they are very simple and less error-prone as state is managed internally and is usually very small.

Containers are the higher level visual elements that do interact with the application state. All the user input based *Actions* originate from containers. Usually, containers store multiple *Components* inside, to whom required data is passed through. The example below shows the *UserManager* container, which contains a *UserSearch* component inside. The *UserSearch* component gets the list of users passed through from the *UserManager*, and emits events based on specific user inputs. Such event propagation technique allows to limit state alterations and reads to the container only.

```
<!-- User Manager Container -->
<div class="content">
    <user-search
        [users]="users$ | async"
        (userSelected)="selectUser($event)"
        (newUserClick)="addNewUser($event)"
        (refreshUsersClick)="refreshUsers($event)"
    ></user-search>
</div>
```

Code 8: Example template of a Container

Directives. All the code which is used to alter an existing element or add additional functionality which can be reused is extracted to *directives*.

A *directive* component can be used in the Angular template to indicate an element, which should have extra functionality introduced by the directive. Most commonly, this functionality is visual only, as the directives are encapsulated components of the system.

5.3. Backend Server

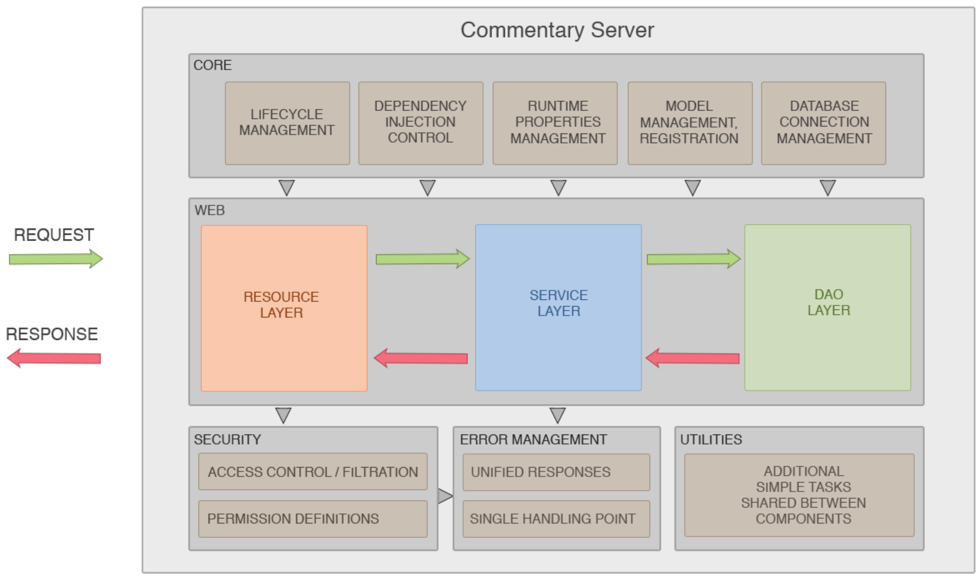
Commentary Server
CORE
LIFECYCLE MANAGEMENT
DEPENDENCY INJECTION CONTROL
RUNTIME PROPERTIES MANAGEMENT
MODEL MANAGEMENT, REGISTRATION
DATABASE CONNECTION MANAGEMENT
WEB
REQUEST
RESPONSE
RESOURCE LAYER
SERVICE LAYER
DAO LAYER
SECURITY
ACCESS CONTROL / FILTRATION
PERMISSION DEFINITIONS
ERROR MANAGEMENT
UNIFIED RESPONSES
SINGLE HANDLING POINT
UTILITIES
ADDITIONAL SIMPLE TASKS SHARED BETWEEN COMPONENTS

Figure 8: Backend Web Service architecture

Overview. As described in the design section, the Backend Server is the most complex and important component of the system. Because of the relatively large size of the codebase, the implementation details below are split by modules and specific concerns.

Languages and Tools. The programming language for the Backend Server was chosen to be Java 8 SE because of the vast number of tools available to make development, deployments, scaling and performance analytics simple. In the case that project would be successful and widely used, all these factors would be very important when extending the system and maintaining it.

Furthermore, to have a layered structure in a Java application, multiple dependency injection and RESTful API framework options were considered. At the very end, Google Guice [45] was chosen as the primary dependency injection framework and Jersey as the servlet container. The benefits of such a combination are that the deployment takes much less time;

the overall package size is minimal (when compared to other options, such as Spring [46]) and the server can be run on any Java Servlet 3.0 container and upwards [47].

Other application dependencies include:
- Jackson [48], a JSON object parsing toolkit
- Hibernate [49], an object-relational model persistence toolkit for relational databases
- Javaslang [50], a functional programming library for Java
- Log4j2 [51], a logging library
- Auth0 JSON Web Tokens [52] (JWT), a JWT-compliant [53] access token creation and validation toolkit
- JUnit [54] and Mockito [55], testing frameworks for Java

All the dependency management for the server is managed using Gradle [56]. It allows great flexibility in dependency version control, compilation and building preferences while keeping the configuration overhead minimal. As an addition, Gradle is becoming the de-facto standard for Java web application dependency management because of great compatibility with Maven [57], simplicity and support in most modern development environments.

Dependency Injection. When the Commentary Backend application starts up, a Guice framework component, called *Injector* is created (see code snippet below).

```
@Override
protected Injector getInjector() {
    Injector injector = Guice.createInjector(
        new JerseyGuiceModule("__HK2_Generated_0"),
        new PropertiesModule(),
        new HibernateModule(),
        new ServletModule(),
        new InterceptorsModule(),
        new TemplatingModule()
    );

    //Installs ServiceLocatorGenerator for HK2 override.
    JerseyGuiceUtils.install(injector);

    return injector;
}
```

Code 9: Application Bootstrap (Guice Injector Creation)

There is only one instance of an injector in Commentary, which bootstraps the entire application by calling the *configure()* method in each of the provided modules. Each of the *Modules* that the injector contains have specific bindings for dependency injection. The code snippet below shows a single binding, which binds the *Session* class to a specific factory inside of the *HibernateModule*. The factory creates a new *Session* object every time it is called. From the point where the binding is registered, any component that requests a *Session* to be injected will have its *Session* dependency set to the returned value from the bound factory.

```
protected void configure() {
    //... Configuration code before
    SessionFactory factory = new HibernatePersistenceProvider()
            .createContainerEntityManagerFactory(new PersistenceUnitInfoImpl(
                    persistenceUnitName, getEntityClassNames(), properties), null)
            .unwrap(SessionFactory.class);

    bind(Session.class).toProvider(factory::openSession);
    //...Configuration code after
}
```

Code 10: Hibernate Module Configuration

The *ServletModule* is a unique type of *Module*, since it does not register any bindings. Instead, it registers a new Java *Servlet* that instructs Jersey (the RESTful API Framework used in Commentary) to start listening for requests and delegate the object creation to Guice after a request has been received and matched.

Aspect Oriented Programming (AOP). As described in the design documentation of the Backend Server, AOP is one of the most important aspects of the application which helps reduce code coupling and split cross-cutting concerns. Java has a great set of features to support AOP development, such as annotations, which can be bound to fields, methods or classes.

Commentary AOP is implemented with help from the Guice dependency injection framework. The general concept is rather simple – each *Annotation* has a provider or an interceptor bound to it. This allows Guice to understand which methods must be intercepted with some additional logic.

However, there are multiple ways to register these providers. Commentary implements several methods to do it, most common ways being an annotation added to the provider class or a specific binding in the module. Both cases can be seen in the figures below.

```
@Override
protected void configure() {
    //Single session interceptor
    TransactionalMethodInterceptor ssmi = new TransactionalMethodInterceptor();
    requestInjection(ssmi);
    bindInterceptor(any(), annotatedWith(Transactional.class), ssmi);
}
```

Code 11: Interceptor Module Configuration

In the above figure, a new method interceptor is created and then bound to a *@Transactional* annotation. This technique of intercepting methods allows executing custom logic before and after the method execution (see Code 12). Each time a method annotated with *@Transactional* will get called, invoke will be called on the *MethodInterceptor* that is bound to the annotation.

See Appendix 3 for detailed explanation of all existing AOP annotations.

```
public Object invoke(MethodInvocation invocation) throws Throwable {
    // ... Some logic before execution of the method
    Object result =  invocation.proceed();
    // ... Some logic after execution
    return result;
}
```

Code 12: AOP Method Interceptor Flow

A much simpler approach is shown below. Instead of binding annotations to specific interceptors, a Jersey provider is created (using the *@Provider* annotation, which Jersey scans for during servlet start-up). As an addition, an annotation (*@Secured*) is attached to the provider. In this specific case, each Resources layer method that has an *@Secured* annotation will have to pass the specific filtration logic in advance of request execution (see Code 13).

```
@Secured
@Provider
public class AuthenticationFilter implements ContainerRequestFilter {
   // ... Inner method and field declarations omitted
}
```

Code 13: An annotation bound Provider

Core. As described in the design documentation, the Commentary package structure follows a strict pattern. The core aspect of the application is the most important since it contains all of the start-up and configuration logic. The Core aspect is responsible for starting up the application, configuring dependency injection, managing database access, data serialization and deserialization, request and response management, creation and destruction of other application components.

The Commentary Core *Application* class implements a *ServletContextListener*, which gets notified about web-apps start-up and shutdown states. Ultimately, this means that the application is designed and implemented to run on a *Java Servlet* engine, rather than a standalone application. This allows to have fast redeployments and robust and predictable server code which is external.

The Core contains a total of five different Guice modules which are configured during startup:

- *HibernateModule* is responsible for database access coordination and initialization
- *InterceptorsModule* is responsible for registering method interceptors to annotations (mostly used for methods which are not in the resource layer classes).
- *PropertiesModule* configures a convenient way of injecting properties files to dependent classes during start-up. This allows externalisation of any configuration that is needed for the application. As a result, application behaviour can easily be modified with only a simple restart of the application.
- *ServletModule* starts up a new servlet container (a Jersey *ServletContainer* instance with properties taken from a configuration file). This is a critical step in the application start-up as once the servlet is running, the application is capable to receive network requests and process them.
- *TemplatingModule* is used to configure an external dependency for template processing. Templates are used for dynamic Plugin Tool generation.

Model. As applicable to any object-oriented programming language, the model (objects) are one of the fundamental building blocks behind any application. Therefore, to make it clear what data the application processes, a separate package containing all the models has been created. All the models are plain old java objects [58] with custom annotations from *Hibernate* defining relationships between them and guidance for conversion to a relational database model. An example of such a model can be seen in the code snippet below.

```java
@Entity
@Table(name = "issues")
public class Issue {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @ManyToOne
    private Application application;

    @ManyToOne
    private User creator;

    // ... Other field declarations omitted.
}
```

Code 14: Issue Persistent Entity

Resources. The resources package stores all the classes that are part of the application's resource layer.

One key difference for these classes when compared to most other classes in the application is that all of them are request scoped. As a result, it is not the dependency injection framework's task to create instances of the classes, instead, Jersey creates an instance of the class for each request. After the instance is created, several service layer objects (which are managed by Guice) are injected to the resource class. The key advantage of such an implementation is that it allows a much greater parallelism.

It is aimed that the resource layer classes only validate the input, log it and dedicate the processing work to services and form a response after processing is done. An example of such case can be seen in the figure below. The *registerIssue* method executes compulsory validation with AOP annotations *@CrossOrigin* and *@AccessControl*, then proceeds to log the input and dedicate the saving task to an injected service and forms a response afterwards.

```
@POST
@Path("/new")
@CrossOrigin
@AccessControl(ADD_NEW_ISSUE)
public Response registerIssue(Issue issue) {
    logger.debug("Registering a new issue: {}.", issue);
    return Response
            .ok()
            .entity(service.save(issue))
            .build();
}
```

Code 15: Resources Layer Method

Security and Permissions. Security is a critical aspect of the application; therefore, a separate package is created for all security related logic. It has two main sub-packages – filters and interceptors. The filters are responsible for rejecting requests which do not pass security requirements or modifying outgoing requests which must have cross origin access.

The most important class security-wise is the *AuthenticationFilter*. It is a request filter, which is invoked only when the user accessing a resource must be authenticated. The logic inspects request's headers and then parses the *Authentication* header. In the case that the parse was successful and a user has been recognized, the request can pass, otherwise it is rejected with an exception.

The second important component is the *AccessControlInterceptor*, which performs required authorization checks. The interceptor checks if the user has all the required permissions for an operation (based on *UserGroup*s, which are explained in the design documentation). Commentary permissions are classified into two groups – static and dynamic, static being constant permissions which allow permanent access to a specific operation. Dynamic permissions are validated for certain methods when a specific resource is being accessed or modified. An example of a dynamic permission is the ability to add a comment on a specific issue. To do that, the *User* must be part of a group which has access to the *Application* that the *Issue* belongs to.

Error Handling. Custom error handling is extremely important to produce great end user experience. To solve this, Commentary implements a custom exception mapping mechanism.

All of the exceptions originated from Commentary code are subclasses of the *CommentaryException* abstract class. This allows the creation a custom exception mapper that is capable of mapping exception to a persistent response that contains user friendly explanation of the problem and internal error codes to help debugging the problem.

Any other exceptions thrown are also caught by an exception mapper that immediately logs the error as fatal to Commentary logs for future review.

Each of the *CommentaryException*s is either of type *CommentaryClientException* or *CommentaryServerException*, which, as described in the design documentation, distinguish input errors from implementation errors. All these errors are further broken down into specific classes. The entire error handling and exception hierarchy is explained in Appendix 4.

Unified Responses. To keep the API experience consistent, a unified response mechanism is implemented. Each of the resource layer classes that is annotated with *@WrappedResponse* annotation invokes an interceptor that wraps each response inside of a JSON object with a result property.

Such implementation ensures that both front-end developers and other API clients always are sure which part of the response contains actual payload.

Services. The services package is the primary package for data processing. All of the data computing and transformation tasks, input verification tasks are done in the service layer.

All the service classes extend *AbstractService,* which provides common methods for invoking DAO layer operations. As explained in design documentation, if a method has to be overridden, it is simply rewritten in the extending class. The *AbstractService* class uses Java Generics [59] to enable support for various entities and injection of object instances from the DAO layer. Dependencies for the service layer classes are injected by Guice automatically on application start-up.

Database Access. The database access layer classes have a very similar structure to the service layer classes. All of them extend the *AbstractDao*, which provides most common CRUD operation implementations. For specific use cases, the implementations can be overridden or new operations created.

Each of the operations should be marked with a *@Transactional* annotation. This annotates that a new *Session* must be created and established with the database before the execution and committed afterwards.

Utilities. This is a package where helper tools and utilities are stored. Most of the classes found in this package are either too large to keep in the standard codebase where used or accessed from multiple classes. An example of the latter case are *PermissionUtils*, which are used both when serializing and deserializing the *Permission*s.

Configuration. As explained before, the application is configurable at runtime through Java Properties [60] files. The current iteration of software supports database access configuration (username, password, address and connection details), logger configuration, access token generation secret configuration and servlet start-up properties. Full documentation of the supported configuration properties can be found in Appendix 1.

Logging. The Log4j2 logger implementation provides a *LogManager* class with static methods to create new loggers whenever needed. All the core Commentary Backend modules have their own loggers preconfigured. Each of the loggers can be specifically tailored to a different logging level, as seen in the code snippet below.

```
loggers = hibernate, jboss, reflections, bridge, connectionpool, modules, core, properties,
security, persistence, resources, errors

logger.core.name = Commentary Core
logger.core.level = debug

logger.security.name = Security
logger.security.level = debug
```

Code 16: Logger Configuration

Logging in Commentary exists in every step in the layered structure, as described in the design documents. All of the logging statements attempt to explain the current step in the overall request flow as detailed as possible, including the entities that are being processed at the given moment. An example of a debug level logging call is displayed below:

```
logger.debug("Attempting to authenticate user: {}.", user.getUsername());
```

Code 17: Debug Level Logging Statement

Database access. The Commentary project is an innovative solution in its own field, therefore to allow better adoption and simpler usage it is a good decision to allow flexibility for end users to choose the supporting database technologies. As explained before, incorporates a Java Persistence API [61] specification-based implementation named Hibernate [62]. The Hibernate JPA model allows creating SQL table structures during application runtime based on the object – relational model definitions, JDBC driver and dialect used. The object – relational model definitions are annotations inside of every Java class that is to be persisted in the database. The annotations allow JPA implementations to dynamically create SQL queries, rather than having them statically added to the code. Thus, the resulting code is clean, highly testable and easy to maintain. It is then only the end users' decision on which type of relational database to use and how to connect to it.

5.4. Version Control

Storage. The entire Commentary code is stored in a private Git repository on a GitLab server. All three components of the system are stored as a single project. Development work is done in branches, while 'master' and 'stable' branches are locked for selected releases.

Versioning. The software is being versioned along with features implemented (not commits made). This means that each iteration of the software introduces new features which should be fully integrated and tested. The releases are marked using GitLab release tags (a release points to a specific commit id).

Planning. The agile sprint tasks are also noted in the GitLab repository. All the tasks created have a milestone (sprint) assigned, and several tags, which can be selected from: 'pending', 'in development', 'testing' or 'done'. Using such planning methods, it can be easily evaluated how well the sprint is progressing and how much work is still left.

Deployment. The deployment instructions can be found in Appendix 5.

6. Evaluation

6.1. Overview

It is critical for an application of such scale to be thoroughly tested not only because of the number of end users that it faces, but also because of the sensitive data that it handles. Both factors are very important, therefore testing is a critical part of Commentary development process. To achieve both high test coverage and predict problems before occurring, a development methodology – Test Driven Development (or TDD) was chosen. The workflow using this methodology is further explained in the Testing Workflow section below.

An overall goal of 80% unit test line coverage for all Commentary components was set in advance, while the project was still in design phase. To achieve such coverage, most of the component code had to be unit tested even before writing the code itself (following TDD patterns) and adjusted accordingly as development progressed.

Unfortunately, as development progressed, the initial coverage plan was changed, as the Plugin Tool was designed not to use any framework. This led to difficulty while unit testing, as no convenient test tools were found and the Plugin Tool has relatively coupled code due to space limitations. As a result, current iteration of the Plugin Tool is only acceptance and integration tested on many different browsers and operating systems. Other components' test plan was not changed.

6.2. Testing workflow

As explained in the subsection above, the Administration Portal and Backend Server were developed using test driven development as a core development methodology. Ideally, following best TDD practices, each commit to the Git repository should be accompanied by tests, covering the new or enhanced feature. The tests must be written before actual production code is written to ensure robustness [30]. However, this may be difficult to achieve because of challenges during development. In such cases where it becomes impossible to create tests together with code, it is aimed to have the required test coverage by the end of each sprint.

To make testing successful and worth doing, it is important to set clear goals that the tests should help achieve. For the Commentary project, it is very important to ensure that not only features work correctly, but also to assert that the non-functional requirements are met. This ultimately leads to two required types of tests –unit tests that will be developed simultaneously as new code is being written and manual integration tests that are manually executed as the entire system is deployed. The testing is done individually for each of the project's components.

6.3. Plugin Tool testing

As explained before, unit test support for the Plugin Tool was dropped during the design phase.

The current version of Commentary Plugin Tool is manually tested (acceptance tests) on a variety of different operating system and browser combinations to ensure compatibility. Such a type of testing is required as the complex drag-and-drop and screenshot features work very differently in every environment. A table with current testing environments is shown below.

	Browser Version	**Browser Type**	**Operating System**
1.	57	Google Chrome	macOS 10.12.2
2.	58	Google Chrome	macOS 10.12.2
3.	10.0.2	Apple Safari	macOS 10.12.2
4.	11 TP	Apple Safari	macOS 10.12.2
5.	52	Mozilla Firefox	macOS 12.12.2
6.	11 (10240)	Internet Explorer	Windows 10 (10240)
7.	38 (14393)	Microsoft Edge	Windows 10 (10240)

8.	57	Google Chrome	Windows 10 (10240)
9.	52	Mozilla Firefox	Windows 10 (10240)
10.	58	Google Chrome	Windows 10 (10240)

Table 1: Testing Environments

6.4. Administration Portal testing

6.4.1. Unit Testing

The Administration Portal is tested with Jasmine [63], a universal testing framework for JavaScript. Jasmine, combined with the Angular testing tools allows simple and convenient testing of virtually any component of the system.

However, due to time limitations, a decision was made that only parts of the system that contain most logic should be tested. Fortunately, good separation of concerns and the state management model that Administration Portal uses means that if all the non-visual components are tested, the application has most of the logic tested. Therefore, the highest focus was to test all the reducers, actions, effects, services and utilities. The resulting code coverage can be seen in the table below.

Component Type	Line Cov. (%)	Function Cov. (%)	Branch Cov. (%)	Statement Cov.(%)
Actions	100%	100%	100%	100%
Reducers	98.91%	96.43%	100%	99.16%
Services	100%	100%	50%	100%
Utilities	100%	100%	100%	100%
Directives	100%	100%	100%	100%
Models	100%	100%	100%	100%

Table 2

However, the remaining untested components, such as all visual components and effects should be tested in future releases following the same testing patterns.

The tests are written in the so-called ‘when, given, then’ pattern, where each test is divided into three sections. It is formally known as Behaviour Driven Development [64], a type of Test Driven Development. However, BDD has more unique patterns than this, therefore it cannot be considered that Commentary is developed following BDD practices. In the tests, the ‘given’ section describes the context of the test (inputs), ‘when’ describes the action that is happening (usually the tested method execution) and ‘then’ section is a set of evaluations performed to assert the results.

```
it('should create a load last issues action', () => {
    //given
    const expectedType = actions.ActionTypes.LOAD_LAST_ISSUES;
    const expectedPayload = null;

    //when
    const action = new actions.LoadLastIssuesAction(expectedPayload);

    //then
    expect(action.type).toEqual(expectedType);
    expect(action.payload).toEqual(expectedPayload);
});
```

Code 18: A sample of a 'given/when/then' test

6.4.2. Acceptance Testing

As well as unit tests, a series of acceptance tests (end user tests) are carried out with every new iteration of the software.

These tests are performed to evaluate that the entire combined Commentary system can execute new features that were planned for the release. The current state of the application tests does not have this process automated, therefore all the tests are executed by the project shareholders.

6.5. Backend Server testing

6.5.1. Unit testing

The Backend Server is thoroughly tested with JUnit [65] and Mockito [66] testing tools for Java. As this Commentary component is the most vulnerable to any logical errors and attempts to misuse the system, it is very important to ensure that it is covered by isolated unit tests.

All the Commentary tests are run manually before pushing to the Git repository. A continuous testing environment would normally be set up in production environments, where suites of tests would automatically be executed before building new releases of Commentary.

The Commentary Backend Server follows the same 'given, when, then' testing pattern from BDD. As an extension to this, it is considered a good practice to write all of the test case definitions in advance of writing the component itself.

The Backend Server test coverage is as follows:

Package	**Class, %**	**Method, %**	**Line, %**
core	79%	65%	64%
dao	100%	100%	100%
exceptions	65%	55%	54%
model	100%	100%	100%
resources	100%	100%	99%
security	65%	66%	65%
services	94%	94%	95%
utility	60%	40%	38%

Table 3: Backend Server test coverage (as of 01/03/2017, before some latest refactoring)

6.5.2. Integration testing

Integration testing for the Backend Server is performed with every new release version as part of the Administration Portal acceptance testing, as it is dependent on and involves testing all capabilities of the Service itself.

All the responses are evaluated according to the given input, and in case a mismatch is found, the release is not accepted.

7. Summary and Reflections

The progress breakdown. The entire requirements, design, implementation and documentation work was planned to be split across 10 sprints, each of them two weeks long. Listed below are the sprints, their goals and what was the result:

1. Sprint 1. 31/10/16 – 11/11/16.

   Goal: The first sprint will be focused on setting up the core of the project (Java backend, SQL database), the administration user interface (an Angular 2 project) and a very basic implementation of the plug-in script, as described before. Furthermore, the first sprint will include basic test setup to make sure all of the tools work as intended.
   Result: Two out of three major components of the system were set up, including gathering of primary requirements and doing research about existing tools. The third component was not started during the sprint, as a result of prolonged research.

2. Sprint 2. 14/11/16 – 25/11/16.
   Goal: The second sprint will be focused on implementing main features of the plug-in – network, event and console tracing, as well as implementing a user interface, which adapts to any type of screen and application type (manually tested on most frameworks and plain JavaScript).
   Result: The plug-in tool implementation was started and all goals were done. However, more research had to be done, which led to extra requirements and changing of the plan slightly. Any further development of the backend and administration panel was not started.

3. Sprint 3. 28/11/16 – 09/12/16.

   Goal: This sprint will have a focus on ensuring high quality of the tests and software, will increase test coverage to reach the planned test coverage. As well as ensuring quality, a lot of the sprint's time will be spent writing documentation and the interim report.
   Result: The third sprint was one of the most intensive sprints, as a lot of development was done on all three components, including documenting the code and writing tests.

As a result, the aim of all three passed sprints was reached and no required tasks were left behind.

4. Sprint 4. 12/12/16 – 20/12/16.

   Goal: Focus will be oriented towards developing the administration portal. In the case that development is done to cover all plugin features available, time will be spent on tests.
   Result: The sprint went as planned.

5. Sprint 5. 23/01/17 – 03/02/17.

   Goal: Adding new features in Java backend for administration portal, writing integration tests for various input types from the users to the backend service.
   Result: The sprint went as planned.

6. Sprint 6. 06/02/17 – 17/02/17.

   Goal: Extending administration portal features and possibly making final touches to plugin tool. In case there is more time, adding free-form drawing to plugin tool.
   Result: The sprint went as planned, free-form drawing implementation was started.

7. Sprint 7. 20/02/17 – 03/03/17.

   Goal: Partially working on final dissertation document and documenting the software. Further extensions and testing for administration portal.
   Result: The seventh sprint was intensive, since documentation took part in all the components. This led to major refactoring of all components and code clean-up.

8. Sprint 8. 06/03/17 – 17/03/17.

   Goal: Partially working on final dissertation document. Running manual and automated end to end tests on most modern browser types and operating systems.
   Result: The dissertation document preparation was started. End to end and other manual system tests were ran.

9. Sprint 9. 20/03/17 – 31/03/17.

   Goal: Final administration portal and backend service integration and deployment of last feature set. This sprint will have a strong focus on eliminating most available user interface imperfections.
   Result: Since most of the development work was already done in advance, minor user interface tweaks were done. Most of the sprint focus was done to write and finish the dissertation document, deploy the software and test it further.

10. Sprint 10. 03/04/17 – 06/04/17.

    Goal: Short sprint focused on final touches to the dissertation document and submission.
    Result: This sprint was omitted, since all of the work was done in advance due to early deadline set by the student.

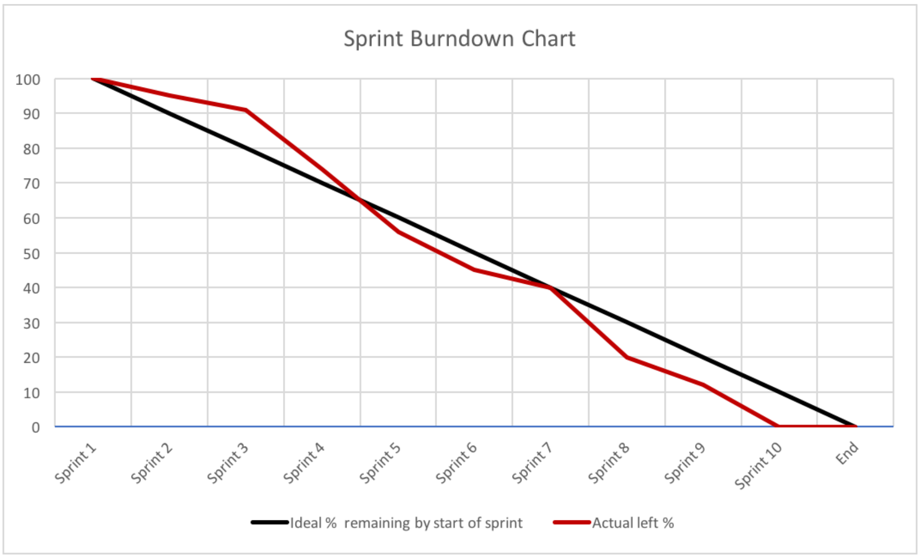

Figure 9: Project Burndown Chart

Major mistakes in project planning. As seen in the project burndown chart, the first sprints had a slower pace than was expected.

One of the major mistakes made in planning was the underestimation of requirements gathering and design steps. As a result, both tasks had to be incorporated into first sprints along with simultaneous development. It is a common agile methodology practice to develop software while still gathering the requirements, however the development part should have much lower focus during first sprints.

Another mistake was not planning for any time spent for code clean-up and refactoring. As a result, problems were only found in final steps of the project and a lot of important documentation and testing time had to be spent to refactor the codebase.

Risks. Even though the goal of the project is mostly reached, a few features from the initial requirements were left out due to lack of time. This includes user event replay option and additional validation mechanisms on user input to prevent accidental sensitive data input. Before the project can be made publicly (commercially) available, these requirements must be met and undergo proper testing.

Another major risk is getting user consent for private data collection. According to the Data Protection Act, data can only be collected after "the data subject has given his consent to the processing.” [22] It is left as a commercial client responsibility to inform and get consent from every user of the application.

Future development. There are multiple opportunities to expand the application further. The biggest improvement would be to create application-specific Plugin Tool versions, which would be context aware (such as a plugin tool for IBM Cognos, for which a use case is presented at the start of this document) and further customizable.

It would also be a huge convenience if the software could be offered with a SaaS model (Software as a Service), deployed in the cloud. This would need further improvements to the backend logic to support additional permission levels and relationships between entities.

Personal development. The Commentary project has helped me learn a vast number of programming techniques, new technologies and gain experience in planning projects. I consider these three factors to be of highest value for my future work and life:

1. Experience in time self-management, project planning and estimation. It has been a challenging project from these perspectives, as all the work was done by myself without any prior knowledge of project planning.

2. New technologies learned both in the front-end and the back-end of the application. It will be very helpful in future career, even if I decide not to pursue a web development career.

3. The programming techniques and ways to approach and find solutions to problems. It was a difficult project from the technical point of view, as many of the intended features of the application are not documented anywhere, therefore much research and attempts to develop solutions were needed. I have successfully managed to tackle the problems and find solutions to them even in situations where it seemed almost technically impossible at first.

Summary. All in all, the project went very well. Except for a few planning issues at the beginning of the project, the time spent on individual components is almost as expected. Unfortunately, several minor features are missing from the result software, but the system is convenient to use, high quality, tested and performs how it is expected to perform.

8. Bibliography

Note: Due to the nature of the project (browser-based applications), many references refer to websites of the tools and frameworks used during the project.

[1] M. M. Group, “Internet World Stats,” 2016. [Online]. Available: http://www.internetworldstats.com/stats.htm.

[2] Clock, “New JavaScript Frameworks In 2016,” [Online]. Available: http://www.clock.co.uk/blog/javascript-frameworks-in-2016.

[3] Wikipedia, “Business Intelligence,” [Online]. Available: https://en.wikipedia.org/wiki/Business_intelligence.

[4] IBM, “Cognos Analytics,” [Online]. Available: http://www-03.ibm.com/software/products/en/cognos-analytics.

[5] SAP, “Business Objects,” [Online]. Available: https://www.sap.com/solution/platform-technology/analytics/business-intelligence-bi.html.

[6] Qlik. [Online]. Available: http://www.qlik.com/en-gb.

[7] TrackDuck, “TrackDuck,” 2014. [Online]. Available: https://trackduck.com/. [Accessed 7 12 2016].

[8] UserSnap, “Usersnap - The visual bug tracker,” Usersnap, 2016. [Online]. Available: https://usersnap.com/.

[9] BugMuncher, “BugMuncher - Free User Feedback,” BugMuncher, 2011. [Online]. Available: https://www.bugmuncher.com/. [Accessed 2016].

[10] StackOverflow, “Stack Overflow,” 2016. [Online]. Available: http://stackoverflow.com/.

[11] Reddit, “Reddit Programming,” 2016. [Online]. Available: https://www.reddit.com/r/programming/.

[12] Google, “Google Trends,” 2016. [Online]. Available: https://goo.gl/iNAN7c.

[13] C. S. Miltiadis Allamanis, “Why, When, and What: Analyzing Stack Overflow Questions by Topic, Type, and Code,” 2013. [Online]. Available: http://homepages.inf.ed.ac.uk/csutton/publications/msrCh2013.pdf.

[14] Atlassian, “Jira - Issue & Project Tracking,” Atlassian, 2016. [Online]. Available: https://www.atlassian.com/software/jira.

[15] GitHub, “Mastering GitHub Issues,” 2016. [Online]. Available: https://guides.github.com/features/issues/.

[16] StackOverflow, 2010. [Online]. Available: http://stackoverflow.com/questions/1694184/what-do-you-look-for-in-a-bug-tracker.

[17] I. Sommerville, Software Engineering, Tenth ed., Glasgow: Pearson, 2015.

[18] Wikipedia, “Java Database Connectivity,” 2016. [Online]. Available: https://en.wikipedia.org/wiki/Java_Database_Connectivity.

[19] S. C. M. A. a. P. T. Italo Dacosta, “One-Time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens,” Laboratory Georgia Institute of Technology, 2012. [Online]. Available: http://b.gatech.edu/2gVfbdZ.

[20] G. A.Vakali, “Content delivery networks: Status and trends,” 2003.

[21] G. P. M. v. S. G. A. Swaminathan Sivasubramanian, “Analysis of Caching and Replication Strategies for Web Applications,” 2007. [Online]. Available: http://bit.ly/2gDaoL2.

[22] U. K. Parliament, *Data Protection Act 1998,* 1998.

[23] L. R. a. S. Ruby, RESTful Web Services, O'REILLY, 2007, p. 311.

[24] R. Martin, Agile software development: principles, patterns, and practices, 2003.

[25] A. Rodriguez, Restful web services: The basics, IBM developerWorks, 2008.

[26] P. o. E. A. Architecture, Martin Fowler, Addison-Wesley Longman Publishing Co., 2002.

[27] S. R. L Richardson, RESTful web services, O'Reilly, 2008.

[28] J. L. A. M. C. M. G Kiczales, “Aspect-oriented programming,” 1997.

[29] R. C. Martin, Clean Code - A Handbook of Agile Software Craftsmanship, Fourteenth Printing ed., Prentice Hall, 2015.

[30] R. C. Martin, Clean Code, Prentice Hall, 2015, p. 121.

[31] Google, “Material Design,” [Online]. Available: https://material.io/.

[32] Google, “Material Design - Introduction,” [Online]. Available: https://material.io/guidelines/.

[33] Google, “Material Design - Color Style Guidelines,” [Online]. Available: https://material.io/guidelines/style/color.html.

[34] Google, “Material Design - Motion Guidelines,” [Online]. Available: https://material.io/guidelines/motion/material-motion.html.

[35] Google, “Material Design - Elevation Guidelines,” [Online]. Available: https://material.io/guidelines/material-design/elevation-shadows.html.

[36] Google, “Angular2,” [Online]. Available: https://angular.io/.

[37] Facebook, “A Javascript Library for building user interfaces - React,” [Online]. Available: https://facebook.github.io/react/.

[38] Microsoft, “TypeScript,” [Online]. Available: https://www.typescriptlang.org/.

[39] D. Abramov, “Hashnode - How do you explain the term predictable state container?,” [Online]. Available: https://hashnode.com/post/how-do-you-explain-the-term-predictable-state-container-in-simple-words-ciizdac5300wege53dogz8aqk.

[40] GitHub, “ngrx/store,” [Online]. Available: https://github.com/ngrx/store.

[41] Medium, “Redux Best Practises,” [Online]. Available: https://medium.com/lexical-labs-engineering/redux-best-practices-64d59775802e#.lgbq42epo.

[42] Wikipedia, “Pure Function,” [Online]. Available: https://en.wikipedia.org/wiki/Pure_function.

[43] D. Abramov, “Redux - Core Concepts,” [Online]. Available: http://redux.js.org/docs/introduction/CoreConcepts.html.

[44] “Side effect model for @ngrx/store,” [Online]. Available: https://github.com/ngrx/effects.

[45] Google, “Google Guice,” [Online]. Available: https://github.com/google/guice.

[46] S. Framework, “Spring,” [Online]. Available: https://spring.io.

[47] Oracle, “Java Servlet Technology,” [Online]. Available: http://bit.ly/2hktmpY.

[48] Jackson, “Jackson,” [Online]. Available: https://github.com/FasterXML/jackson.

[49] Hibernate, “Hibernate. Everything data.,” [Online]. Available: http://hibernate.org.

[50] “Javaslang,” [Online]. Available: http://www.javaslang.io.

[51] Apache, “Log4j2,” [Online]. Available: https://logging.apache.org/log4j/2.x/.

[52] Auth0, “JWT Toolkit,” [Online]. Available: https://auth0.com/docs/jwt.

[53] IETF, “JSON Web Token Specification,” [Online]. Available: https://tools.ietf.org/html/rfc7519.

[54] “JUnit,” [Online]. Available: http://junit.org/junit4/.

[55] “Mockito,” [Online]. Available: http://site.mockito.org.

[56] Gradle. [Online]. Available: https://gradle.org.

[57] Apache, “Maven,” [Online]. Available: https://maven.apache.org.

[58] Wikipedia, “Plain Old Java Object,” [Online]. Available: https://en.wikipedia.org/wiki/Plain_old_Java_object.

[59] Oracle, “Java Generic Types,” [Online]. Available: https://docs.oracle.com/javase/tutorial/java/generics/types.html.

[60] Oracle, “Java Properties,” [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/environment/properties.html.

[61] Oracle, “Java Persistence API,” [Online]. Available: http://bit.ly/1zRydVT.

[62] Hibernate, “Hibernate,” [Online]. Available: http://hibernate.org/.

[63] “Jasmine,” [Online]. Available: https://jasmine.github.io.

[64] Wikipedia, “Behavior Driven Development,” [Online]. Available: https://en.wikipedia.org/wiki/Behavior-driven_development.

[65] JUnit, “JUnit,” [Online]. Available: http://junit.org/junit4/.

[66] Mockito, “Mockito,” [Online]. Available: http://site.mockito.org/.

[67] Mozilla Developer Network, “Introduction to the DOM,” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.

[68] M. S. M. a. J. C. Powell, “Single Page Web Applications,” Manning Publications.

[69] Wikipedia, “Multitier architecture,” 2016. [Online]. Available: http://bit.ly/28YXaXL.

[70] SmartBear, “How Testing changes in Component Based Architectures,” 2015. [Online]. Available: http://bit.ly/1Jk9OIG.

[71] L. Richardson, RESTful Web APIs, 2013.

[72] Wikipedia, “Google Guice,” [Online]. Available: https://en.wikipedia.org/wiki/Google_Guice.

[73] Oracle, “Jersey,” [Online]. Available: https://jersey.java.net/.