Architecture Design The First Order

The First Order

June 17, 2016

1 Introduction

1.1 Purpose

This document describes the architectural software design of the system by providing different architectural views. It's purpose is to explain the architectural decisions made on the system.

1.2 Design goals

While designing the system, we've maintained the following design goals:

1.2.1 Component independence

To maintain modularity and extensiblity, it's important for components for be independent of each other. This should allow us to easily replace, change or extend components in the system. Several design principles will be applied for this, like inversion of control by using springboot, and by applying the model-view-controller pattern to decouple the presentation, logic and the models.

1.2.2 Code quality

To keep our code maintainable, code quality is an important design aspect. A new member should be able to quickly understand what the code does. This means all methods should be commented with Java-doc. Testing is also an important aspect. Not only does it verify the workings of a class, it also provides a form of documentation.
To enforce this, we use several static code analysis tools (Checkstyle, Findbugs, PMD). These have to pass before any code may be merged with the main branches. We have also added a constraint to the dev branch that checks for testing coverage. If coverage is down in a branch that wants to be pulled it cannot be merged.

1.2.3 Cross-platform compatibility

The application should be able to run not only on desktops, but on tablets as well. The UI should adjust itself to provide a smooth experience for both desktop and tablet users.

1.2.4 Scalability

The system should not only work for small projects, but also for larger projects. The performance should be consistent, which means importing a large project shouldn't slow the application too much.

1.2.5 Reliability

The system should not crash or produce unknown errors. If an exception is thrown, it should either be handled transparently or, if it is fixable by the user, shown to the user.

1.2.6 Security

An user must log on before he or she can make changes. Every user should have a personal account. To keep the system secure, the application and it's data should not be accessible to unauthenticated users.
Users should also have different levels of authorization. I.e. a director user should have both viewing and editing permissions, while an operator user may only have viewing permissions. Depending on their authorization, access to certain pages may also be restricted.
To implement these security measures, we use SpringSecurity that provides an OAuth 2 implementation for us to use, a widely-used web services authentication service.

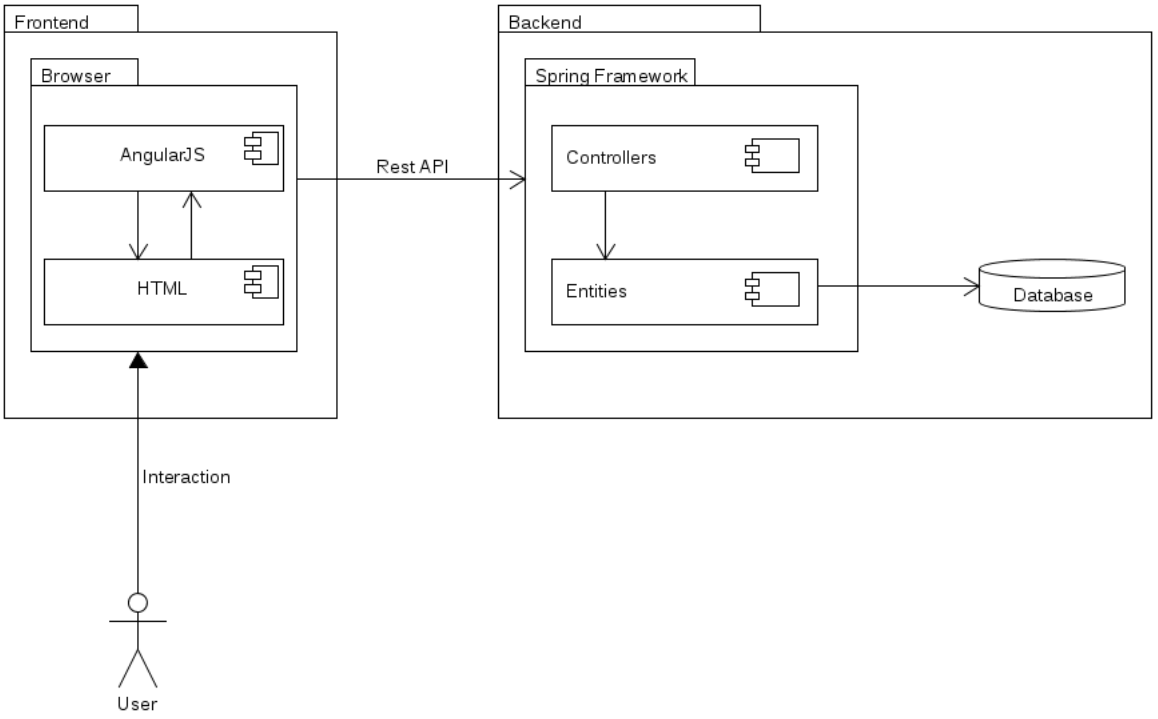
2 Software architecture

2.1 Programming languages and frameworks

The application is written in Java and JavaScript. To allow cross-platform compatibility, we chose to make this a web application.
Out of all the Java web frameworks, Spring Boot seemed the best choice. Spring is a dependency injection framework that has been extended with support for numerous features. It has it's own model-view-controller framework, support for object-relational mapping frameworks and more. Spring Boot is also well documented and used by many developers, which means we can troubleshoot online easily.
Spring Boot is a pre-configured suite, built on the Spring framework. It also provides the convention-over-configuration paradigm for Spring. This makes it fairly simple to setup a Spring application without configuring XML files.
For the frontend, we use AngularJS. This is a JavaScript web application framework and provides a client-side model-view-controller architecture. To keep our AngularJS code organized, we aim to follow the John Papa AngularJS 1 Style Guide [4].

We used JHipster to setup the configuration of the frontend and backend [1]. This is a commonly used method in web development for building applications. What this means, practically, is that a Bower script is set up that handles all packages and dependencies that are used in building the frontend. These dependencies have been removed from the git repository, as they are automatically generated when a build is done. Bower recommends checking in all these dependencies, but on the ta's request, we have removed them from the repository. Library imports (for javascript, angular, visualization) are injected into the index.html file and the libraries themselves are also downloaded into the repository.
We have used JHipster to setup the 'framing' (skeleton) of the system, the same way PHPMyAdmin would generate a MySQL database. Some service classes, like the PDF export utility class were created from the ground up and we will be working and have been working in classes like the ProjectResource class and other services classes. As the project progressed, we've edited and maintained these database classes ourselves in order to get a better understanding of the workings of the code and to make sure our edits would not be overwritten. In the front-end, we've created the map editor, scripting page, live preview and different user profiles.

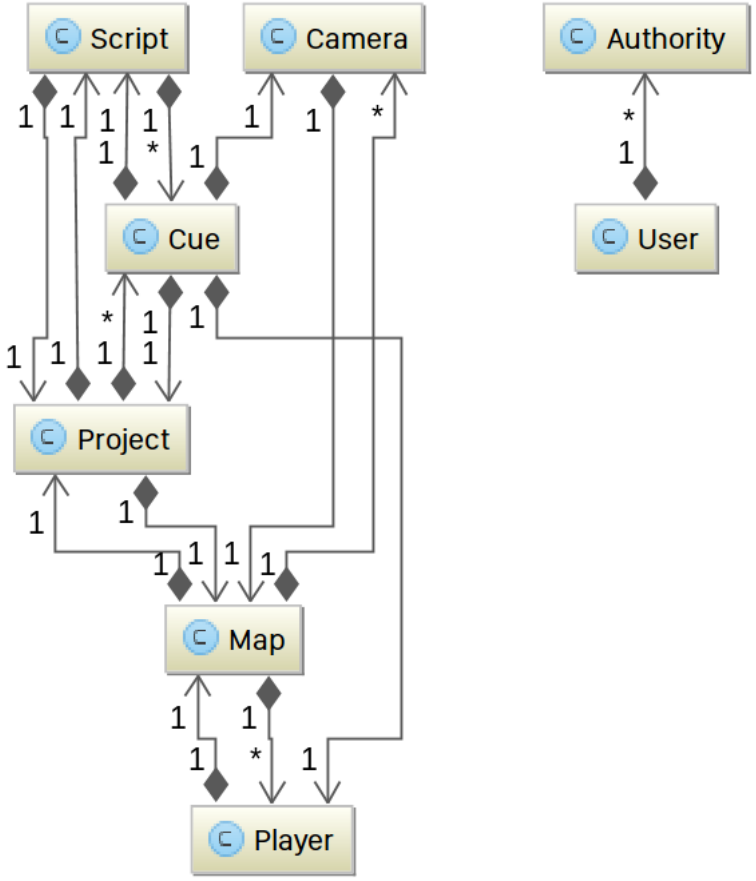
2.2 Subsystem decomposition



Frontend The frontend is the graphical user interface. It's a interactive web application, created with AngularJS, and runs in the browser. Using RESTful API calls, the web application can interact with the backend.

Backend The backend is the Spring Boot server which handles the server logic and manages the data. It serves the web pages to the clients. It has a database for persistent storage of entities. To interact with these entities, it provides a RESTful API.

2.2.1 Entities

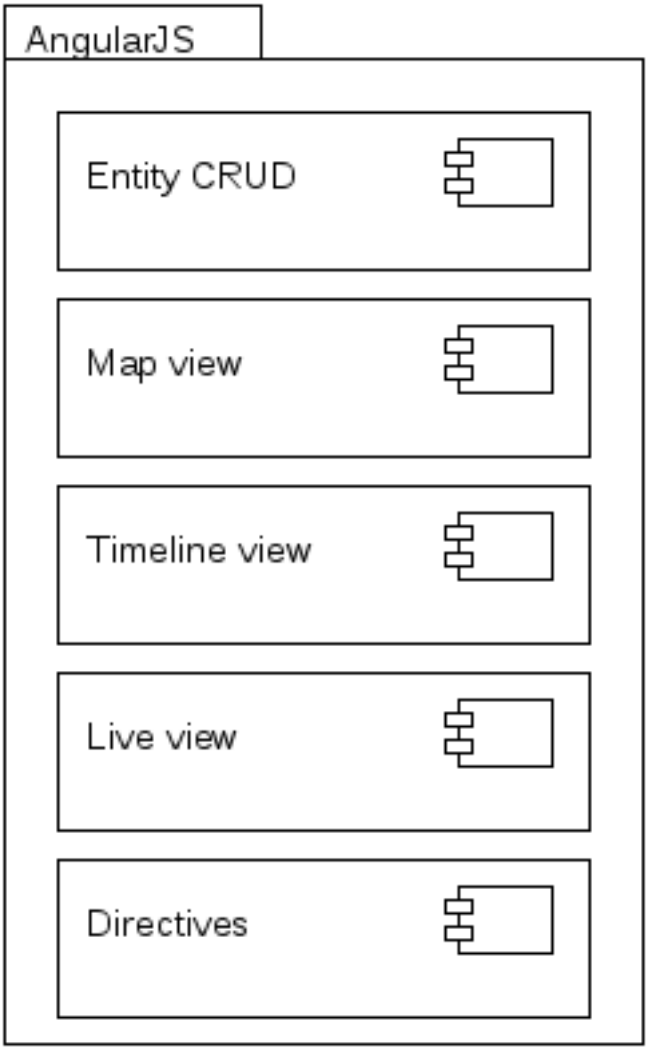


In sprint 5 and 6, we changed the relationships between the entities.
All entities now have a Project entity as owner, except for Players and Cameras, they are owned by a Map. This makes sense, because it allows us to isolate changes to a single project. Due to how Hibernate works, a Project entity needs a reference to those entities as well. An user has a reference to a Project_Id. This allows us to persist the same project across all pages.
Most of the *OneToMany* relations are bi-directional. Hibernate needs this to map these entities to a relational database. In the database schema, the Project table has no references to other entities. Instead, all entities have an additional column with the Project_Id. So, even though the relations in Java are a bit confusing, they make a lot of sense from a database perspective.

In Sprint 6 we have also removed the TimePoint and CameraAction entities. These were part of the Cue, but turned out to be impractical. To add a Cue, one had to first create a TimePoint entity and when a Cue was deleted, the corresponding

TimePoint had to be deleted too. It was easier to just add a bar number and duration to the cue itself. The CameraAction would be more practical as a simple string, because we wanted the director to be able to enter anything as the action for a camera and actions would not need to be reused.

2.2.2 AngularJS



Entity CRUD The web application provides an interface where the user can easily create, read, update and delete entities.

Map view The map view provides a graphical user interface to create, view and edit maps.

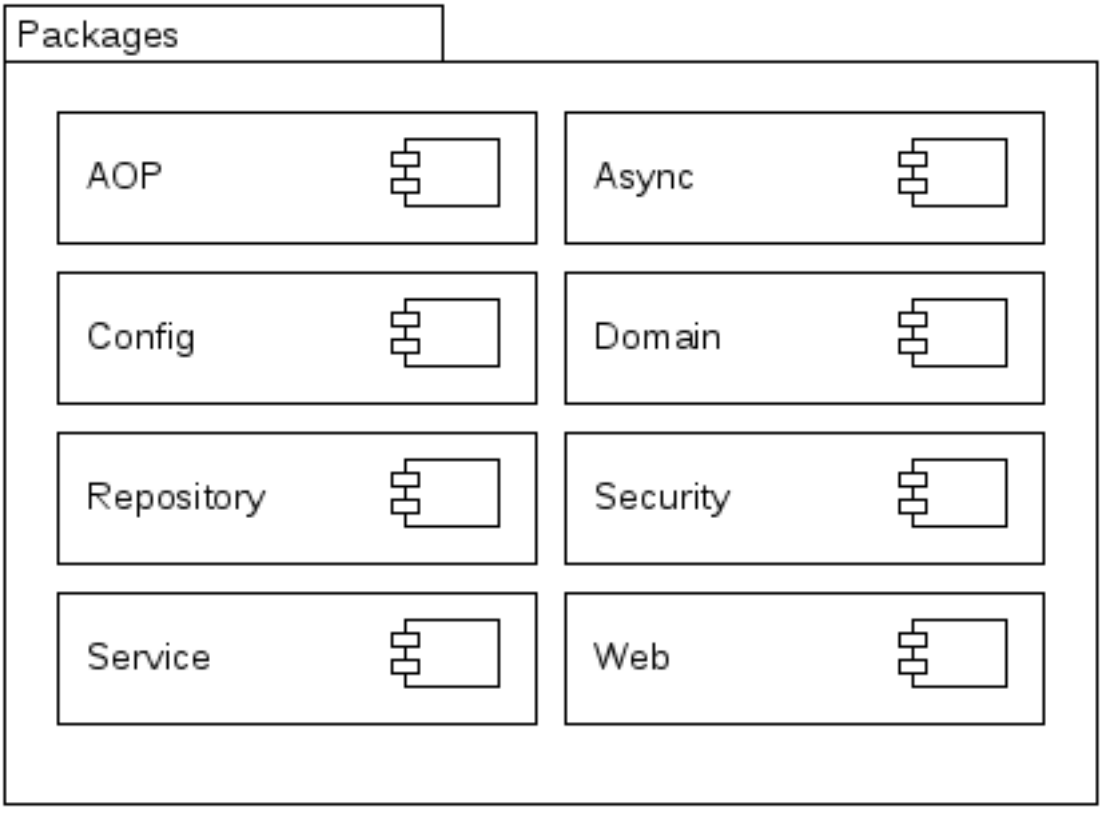
Scripting view The scripting view provides a graphical user interface to create, view and edit a script. By having a map directly above it, the user can more easily visualize the script creation. Cues can be set to specific intervals with actions.

Live view The live view provides a graphical user interface to walk through the script in a manner that is constructive for a live workflow. This means that the current and next cues for each camera are shown for that camera. These cues can be progressed through a score reader view that will be added in the next sprint.

Map directives Angular also provides directives. These allow you to create custom HTML elements or attributes. The map directive draws the map and connects a map controller to it that loads, shows, and updates the map canvas on user input.

Timeline directive The timeline directive draws the timeline and handles user interaction with the timeline.

2.2.3 Spring Boot



AOP is Aspect-orient-programming logger package. This is used for aspect oriented logging execution of service and repository Spring components.

Async This is a helper for asynchronous handling of exceptions.

Config Spring Boot allows and recommends you to use a Java-based configuration instead of a XML-based configuration. Classes annotated with $@Configuration$ are configuration classes.

Domain The domain package stores all the entity classes.

Repository Spring Data Repositories are interfaces that can be used to access and store entities in a database. Spring allows you to only define an interface. It's implementation is automatically created during runtime.

Security Spring Security is a framework to provide both authentication and authorization to Java applications.

Service This contains all service related classes. Service related classes are classes that hold logic to interface between database and APIs. An example of this class is the PDF utility class that reads data from the database and uses logic to export this to a PDF. IntelliJ (or other IDE's) might not that these classes are not used, but SpringBoot uses them nonetheless. If these classes are deleted, the project will not run.

Web The web component is responsible for the REST API. It provides the API to the client and connects it to the corresponding services classes. One important type of API calls is the DTO. This is a Data Transfer Object, which packages different data in one object, so Angular can easily work with our data.

2.3 Hardware/software mapping

The Spring Boot backend runs as a server on a single machine. This server can serve multiple clients. To keep our application scalable, we use a RESTful architecture[2] in the backend server. The UI can not only be viewed on the same machine as the backend, but also on other machines. It can be accessed through the browser, which means other devices could also access it, such as tablets.

2.4 Persistent data management

The Spring Boot framework has support for relational databases. For deployment, we will most likely use a MySQL database, but for development we currently use an in-memory database.
To store and retrieve Java objects in a relational database, we use Hibernate [3]. This is a object-relational mapping framework, which maps a Java object to a relational model in a SQL database.

2.5 Concurrency

Multiple users might be using the application simultaneously. To accommodate for this, we designed the system to be RESTful. RESTful means that the server does not hold the state for each user, but the client does. A client requests a new state from the server through links (GET requests) and the server responds with this new state. Because the server does not have to hold the state for each user, it can server

many users at once.

We are now also working on using WebSockets to provide real-time collaboration in the application. WebSockets allow for full-duplex communication over a tcp-connection. Here, one user can send data to another user and this information gets updated real time. We implemented this for the live-view.

References

[1] URL http://jhipster.github.io/.

[2] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 407–416, New York, NY, USA, 2000. ACM. ISBN 1-58113-206-9. doi: 10.1145/337180.337228. URL http://doi.acm.org/10.1145/337180.337228.

[3] Red Hat. *Hibernate documentation*. URL http://hibernate.org/orm/documentation/5.1/.

[4] John Papa. *Angular 1 Style Guide*. URL https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md.