Table of Contents

This chapter describes the goals that motivated the development of this software. It also specifies the format of this document.

Motivation

Back in 2012, I was researching about the death era of monolithic applications, and how the industry was moving towards a separation of front-end and backend. I talked to the community about this research in the conference session *"Single-page applications and Grails"* [spi-greach], given at Greach Madrid in January. 2013.

Traditionally, software developers have been used to create applications where the views and the business logic, although probably separated in software layers according to best practices and design patterns, are packaged together in the resulting application deployable artifact. This approach has several disadvantages:

- **Front-end** and **backend** developers have to work in the same project, using probably a server-side based framework such as Spring MVC for Java EE or Rails for Ruby. This will require so called full-stack developers, as the views are not simply HTML, but a framework-specific view templating technology.
- **Deploying** and **scaling** such monolithic applications is complicated, as it all have to consider the whole package.
- **UI changes** cannot be made and deployed independently of the business logic, due to the way of packaging and deployment. This reduces or even prevents doing frequent interface changes to improve conversation, A/B testing, etc.

Those problems were not significant to the development industry because the majority of the developers were considering themselves **full-stack developers**: proficient in doing backend and front-end. However, a technology was created and it changed the game rules: **Node JS**.

Node JS not only enabled developers for the first time to use the same language for front-end and backend (Javascript), but also set the foundation for a large number of tools, such as Grunt or Gulp as bulding tools, Bower as dependency manager and Yeoman as project creation and scaffolding tool. Those projects attracted a large number of developers that were highly skilled in HTML5, CSS3 and Javascript, and for the first time in forever had a specific toolset where no backend was involved.

On the other hand, backend development was experiencing the rise of the new buzzword of the moment: **microservices**. After several years since REST architecture was first introduced, and more and more teams trying to adopt such architecture with relative success, the industry finally agreed on a preferred way of building the business logic: they should be REST API's, producing and consuming JSON over HTTP.

The last necessary contributors in this shift were the frontend development frameworks, like Ember but specially Angular JS (built by Google), that effectively enabled the possibility of creating applications where the UI was entirely executed in the browser, and the communication with the server was using the mentioned JSON over HTTP.

The API's that the backends were converted into had a special requirement per the definition of REST (*REpresentational State Transfer*): they had to be **stateless**.

In my job at that moment, as Web Architect, I introduced the necessary changes in the company to break down a existing set of monolithic Grails-based applications, into Grails REST backends plus Angular JS front-ends.

In another conference in December 2013, the Groovy & Grails eXchange (GGX) in London, I was talking to the audience about this idea to separate front-ends and backends ([spi-ggx]), and eventually one of the attendees asked a relevant question:

> How do you handle authentication/authorization with this setup?.
>
> — @javazquez, 12th December 2013

More or less, developers managed to create stateless business logic, using the front-end frameworks to keep client state and ensuring that such state was sent to the server on each and every round-trip. However, authentication and authorisation was different. **Security** in a web application has traditionally been a stateful service: it heavily relies on the HTTP Session, which itself represents a state in the server side. Therefore, there was a friction when applying stateless restrictions to operations like login or logout.

Therefore, to not only answer that attendee but also provide a potential solution, I continued my research on this subject that ended up in the development of **Spring Security REST**: a plugin for Grails framework, on top of Spring Security, that offered developers a way to implement authentication and authorization in a stateless way, as well as it made the necessary transformations over core Spring Security to make it more REST-friendly.

Goals

The goals of this project are closely related to the motivations described above, and are the following:

1. Provide an implementation for the Grails framework.
2. Make core Spring Security, which only offered form-based authentication using the HTTP Session, more REST-friendly.
3. Offer developers a way to implement authentication and authorization in a stateless way.
4. Analise, design, implement and test a robust solution, stable and highly covered by tests.
5. Contributing back to the community by creating an open-source solution.

This document describes in detail all the stages of the process followed to achieve the mentioned goals.

Rest of the document

This document is structured in the following way:

1. **Introduction**. This chapter explains the motivations that originated the need to create this project, the goals that were pretended to achieve with it, as well as this document structure.
2. **State of the art**. In this chapter an evaluation of the starting point is performed, as well as a comparision of the existing solutions, to conclude that none is able to achieve all the goals.
3. **Analysis, design, development and deployment**. Here are described in detail the requirements that the system must complete, including documentation about the use cases.
4. **Evaluation**. It contains an analysis of the project success, based on different metrics.
5. **Planning and budget**. In this chapter it is described the planning of the different tasks performed to complete the project. It also contains a budget estimation, considering the cost of the resources used to develop the project.
6. **Conclusions and future work**. It is explained how the goals have been met, what is the result of having applied the software development process, which are the personal improvements obtained, as well as possible future improvements.
7. **Appendices**. It contains the official software documentation.