|  |
| --- |
| CROSSOVER |
| Sample Application |
| Software Design Document |
|  |
| **Meltem Yesiltas** |
| **10/14/2016** |

|  |
| --- |
| High-level Software Design Document for the project that is developed for Crosssover Tech Trial |

Contents

[Introduction 2](#_Toc464030983)

[Requirement Analysis 2](#_Toc464030984)

[Functional Requirements 2](#_Toc464030985)

[Non-functional Requirements 2](#_Toc464030986)

[Data Model 3](#_Toc464030987)

[System Architecture 3](#_Toc464030988)

[Deployment Architecture 3](#_Toc464030989)

[Component Breakdown 4](#_Toc464030990)

[Front Web 4](#_Toc464030991)

[Component1 6](#_Toc464030992)

[Component1 8](#_Toc464030993)

[Common Component 8](#_Toc464030994)

[Other concerns 8](#_Toc464030995)

[Class Diagrams 8](#_Toc464030996)

[User Interfaces 8](#_Toc464030997)

[Future Enhancements 9](#_Toc464030998)

## Introduction

(Purpose, scope, overview, assumptions)

This document describes the architecture and design of the “sample-app” system. The potential customer/users of system and the team members that will contribute to the development of the system can read this document.

“Sample-app” system is used by a …. company to serve its customers through Internet.

Customers can do….

Also, there is another type of user, which is defined as admin. Admin users can use the system to make necessary definitions, ….

## Requirement Analysis

(Functional, nonfunctional requirements)

### Functional Requirements

### Non-functional Requirements

**Scalability**: The system must scale when necessary. Each part of the system (component) may scale independently, without affecting the other components of the system.

**Transaction Integrity**: Each component of the system must provide transactional integrity. It is assumed that there is no need for a transactional integrity between different components of the system, i.e. there will be no transaction that its boundaries span more than one system component.

**Logging:** A proper and configurable logging solution must be provided.

**Exception Handling**: All the exceptions must be logged properly by the framework.

**Mobile-readiness**: The system must be implemented in a way that its core components can be reused and integrated easily with mobile client applications.

**Security**: The user must be authenticated with a username and password in a secure way. Since, the system is broken down to different components; these components should communicate to each other in a secure way. The system must save the user and itself from a CSRF attack. It is assumed that all the components of the system will be deployed in application servers that only communicate through HTTP over TLS (HTTPS).

**User Interface**: User Interface must be a Single Page Application with a responsive design to support different size of client devices.

**Cloud Ready:** The system must be easily deployable to a cloud platform (e.g. AWS ElasticBeanstalk)

## Data Model

(ER diagram)

## System Architecture

Diagrams…

(Describing the composition and working of the system, explaining the component interaction and process, control and data flows)

Crosscutting concerns: Exception handling, trx. Management, logging

### Deployment Architecture

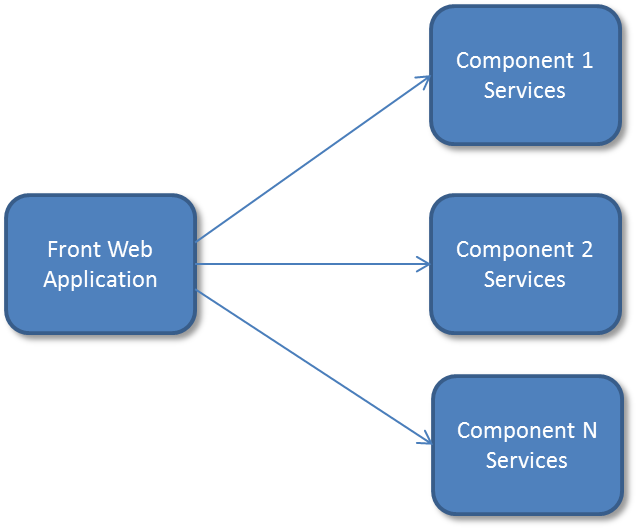
System is broken down in components in a way that each component will be responsible from a narrow business domain. Each component (or module) will serve RESTFUL services related to its domain and will be deployable independently. This approach is called “Micro-services Architecture”. This design will provide us to:

1. Easily extend the system in the future, without affecting the existing components.
2. Have a loosely coupled system that each component is deployable and testable independently.
3. Narrow the effects of a future change in an existing component. As long as the component keeps its RESTFUL API stable, there is no risk of affecting the other components.
4. Scale each component independently.
5. Deploy components to containers (e.g. Docker) for better scalability and portability
6. Easily develop additional UI applications for new channels or devices (tablet, mobile, etc.)

The disadvantages of this design:

1. Runtime performance overhead of HTTP communication between components.
2. May introduce some transaction integrity issues if component borders are not defined well.
3. Harder to implement at the beginning, but for future changes its more flexible.
4. Increases the complexity of the system.

The following diagram depicts the deployment architecture of the system:



Front Web Application

Component 1 Services

Component 2 Services

Component N

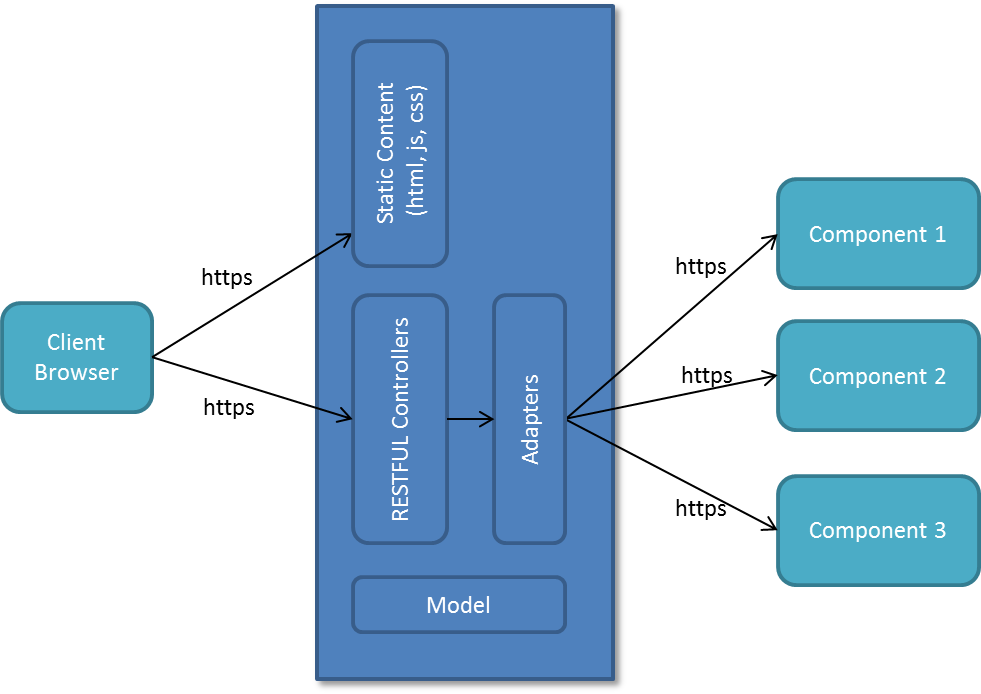
Services

## Component Breakdown

(Technical implementation details of each component, along with the design patterns involved with reasons that justify your choices)

### Front Web

This component is merely responsible from serving the client application that will run in the browser. It will contain all the user interface assets like html, js, css, etc. This component should not contain any business logic. It should not connect to database for any transaction. It should gather/send all the information from/to other components by sending them http requests. The following is the internal architecture of this component:



Client communicates to web application via HTTPS protocol. It can either get some static resources or make some ajax calls to RESTFUL services. RESTFUL services are implemented as Spring controllers, which actually have some filters in front of them for authentication and authorization. Authorization is not implemented in the framework, it may be when needed. But authentication is implemented using Spring Security (see WebSecurityConfig). A form login is required to authenticate the user. Also CSRF protection is enabled in a way that it will support AngularJS CSRF token header. These filters are not included in the above diagram to keep it simple.

Once a request arrives to a controller, the controller is responsible to reply it using one or more backend services from the related components. Here in front web application controllers are not allowed to contain business logic. They can only implement some logic specific to that channel or user interface. They will directly call backend service(s) from the other component(s) using the adapter defined for the corresponding component(s).

**A design discussion:** It is possible to call backend services directly from client running on browser, thus not having a controller layer in “front web” application. But implementing all the coordination and UI logic in the javascript in a secure way seemed difficult to me. Having an extra controller layer as a mediator seems more secure and flexible and makes me feel more comfortable.

**Adapter** structure is implemented to provide an extension point for making it possible to talk different backends with different requirements. It is implemented using singleton and factory design patterns. A concrete adapter called BasicAuthRestAdapter is already implemented to talk to RESTFUL backends that are using http basic authentication. In future there may be a need to integrate a backend using some custom headers, etc. The necessary connection information about a component (also referred as module) is declaratively defined as Spring context property (in application.properties file). This information includes which adapter to use, and other information specific to corresponding adapter. For example, for a module named “mymodule” that requires basic authentication, the following definition can be used:

modules.mymodule.adapter.class = net.yesiltas.sample.client.adapter.BasicAuthRestAdapter

modules.mymodule.api.url = http://localhost:8080/sample-app-mymodule/

modules.mymodule.auth.client.id = "sample-app-client"

modules.mymodule.auth.client.secret = "some secret key"

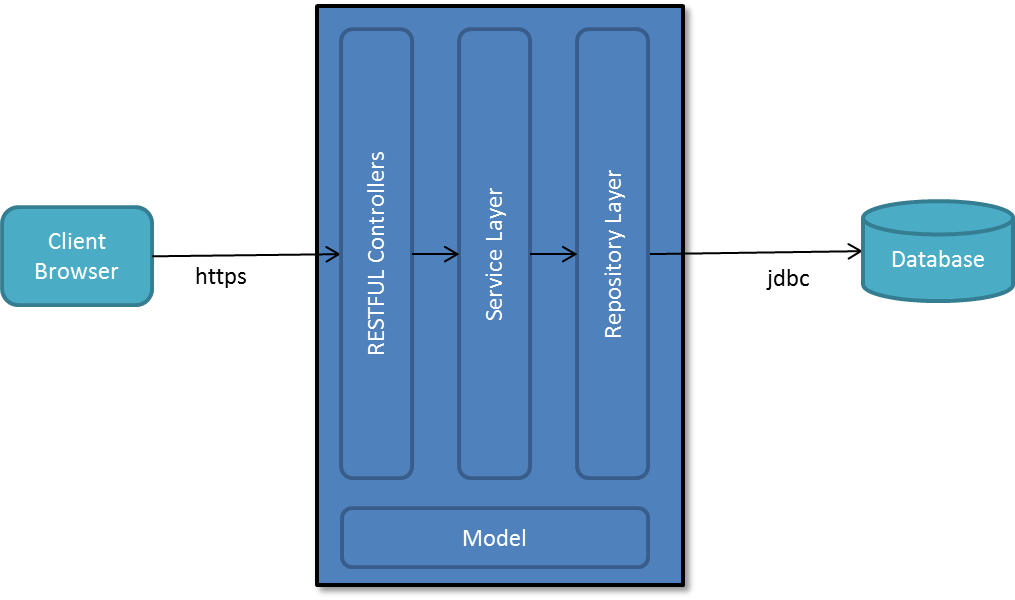
Browser will actually execute a JavaScript MVC application using AngularJS. It is designed as a single page application. Responsiveness is supplied using Bootstrap. All the routing logic of UI is implemented in client; server never should send redirects or forwards to client to manage the routing logic. Routing is managed by client using corresponding AngularJS modules.

Here I will not talk about the AngularJS concepts in detail. The following are the main design concerns for client application:

1. There is also modularity and structure in the client application itself, similar to server application. Each module will have a different subfolder under webapp. Each module will have its own angular module definition to contain related controllers, constants, factories, routing definitions, etc.
2. View is separated from the remaining logic and made as reusable as possible. It is plain HTML file containing no javascript. Thus it can be designed independently using any design tool. It does not even specify which controller to use, so it is possible to reuse view with different controllers, or vice versa, reuse controller with different views. View and controller are attached to each other using routing configuration.

Component1

This component is responsible from implementing all the business logic for xxx domain and serving this logic as RESTFUL services to Front Web application and any other third-party application. The following is the internal architecture of this component:



This module is secured with basic authentication. It is implemented using Spring security (see WebSecurityConfig). Credentials for the client applications are defined as Spring context property (application.properties). This application and other module applications are designed as stateless, they will have no session or any other state, thus they can scale well behind a load balancer in a production environment.

RESTFUL Controllers call service classes for executing business logic. Controller classes are not expected to use repository layer directly. Controllers are responsible to serve a clear and standard API to clients. For example any method that will change the state must not be served as http GET method.

Service layer will implement business logic . It will talk to database through repository layer. Spring’s JPA facilities are used to do that. If a service changes the state it must be annotated with @Transactional annotation. That is all we have to do for transactional integrity. Spring boot will handle rest. If necessary it is possible to set transaction isolation and propagation. But the defaults (READ\_COMMITED, REQUIRED) will be ideal for most cases.

The following properties is defined in Spring context to connect to database. You can use different configuration for different environments (stages) using Spring’s profile facility.

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/sampleDB?profileSQL=false&useSSL=false

spring.datasource.username=root

spring.datasource.password=123456

The information about the active Spring profiles is defined in main application.properties file.

spring.profiles.active=dev, mysql-local

All three layers (controller, service, repository) use model classes as input and output object types. They are mostly JPA’s Entity classes. To avoid duplication and boilerplate code, there are no DTO classes. Entity classes are used as DTO classes as well. Since the clients of RESTFUL Controllers will need them, they are put on separate Java project, so that packaged as a separate JAR file and handed over the client applications. In our case “Front Web” application is a client for this component, it will depend this component’s model jar. This dependency is preferred over re-implementing the same classes in the client application. Also sharing model between RESTFUL provider and its client is a good choice for managing consistency. There is a side-effect and applied solution: Since our model classes are Entity classes, client project needs to JPA and other related libraries in its build path to compile. When you have these libraries in your build path Spring will automatically try to initialize JPA and look for a database connection, dirver class, etc. To prevent this situation related classes are excluded from auto-configuration in client application’s initializer class (ClientApplicationInitializer)

@EnableAutoConfiguration(exclude = { DataSourceAutoConfiguration.**class**, HibernateJpaAutoConfiguration.**class**, DataSourceTransactionManagerAutoConfiguration.**class** })

Component1

Common Component

Common component is not an independent/deployable component itself, it is designed as a JAR library to include in every other component except client application. If a lot of common code to share between both server components and client emerge, then another common project may be added to the system instead of using this one. Because this common project may include code related to service and persistency layer, it can reach database if necessary, if client project includes this common component, this will destroy the design principle about client project. It must be thin, have no business logic and no db connection. Client project will depend only to model part of this component to populate input and output of RESTFUL services it will call.

Other concerns

A sample logging is implemented using logback . More logging should be added to application to trace the problems. Logging level is configured through application.properties. It is set debug level for development profile, and warning for production profile. More fine-grained configuration can be done when necessary.

A generic exception handler called DefaultExceptionHandler is implemented in the common project. It makes sure that every exception is logged properly.

## Class Diagrams

(UML Diagram)

## User Interfaces

(Manuel or Balsamiq Mockup)

## Future Enhancements

1. Authorization should be implemented using Spring Security for all the components of system.
2. Proper unit test implementation for javascript part is missing
3. Unit testing of Controllers in client project will require all component application up and running to execute. A proper mocking for the remote services may be implemented to eliminate this.
4. Multilingual support may be added if required. It should be implemented both on javascript side and Java side.
5. Exception handling may be enhanced. An “error report” button for user would be very useful during production phase of the system. When the user clicks all the details about the current problem (including stack trace, root server, screenshot, etc.) is sent to an email address. Although it is good to have all the details reported, we must be careful about exploiting our application’s technical details to the public. Storing the incidents in database with all details and giving them id, and sending only the id to the client would be a solution.
6. TLS must be enabled for both client application and other components. For communication between front web and backend components basic authentication is used, but is must be used in conjunction with TLS in a production environment.
7. Client names and secrets are stored in plain text in application.properties. It must be stored as encoded in the database.
8. A caching solution may be implemented in front web project for data that is frequently read but not changed a lot. This will improve the performance by decreasing roundtrips to the backend components.