Exchange Rates application architecture

Abstract

The Architecture document is aimed at providing information about the system to different stakeholders e.g. developers, end users, management, administrators etc. Since different stakeholders are interested in different aspects of the system, the Architecture is actually a set of documents (or views), each document describing the system from a specific aspect. This document describes the system from a developer’s perspective.

Assumptions

Since for this particular task there is no way to clarify requirements with product owners, a few assumptions had to be made.

The chosen external server for getting exchange rates is CurrencyLayer. A free account was created which has several important limitations: only a limited number of requests can be made to the server; there is no support for getting exchange rate for anything but USD. These limitations mandate some design decisions

First, a decision was made to implement server side cache to limit backend server hits. Normally, the cache would reside in the browser and then if fresh data is needed, a request to the backend would be made which in turn would make a request to the external server without caching anything in the server. However, due to the assumption that only Free Access can be used and only a limited number of hits to the external server are allowed, the data is cached on the server. Therefore, once fresh data is obtained from the external server, it is available for all users thus avoiding each user hitting the external server.

Second, a decision was made to compute exchange rates to currencies other than USD using a somehow naïve algorithm – just divide the destination currency exchange rate to USD by the exchange rate of the source currency to USD. This approach gives good approximate results. An assumption was made that this is acceptable since there is anyways no other way to achieve that using the Free Access account.

The Free Access account password is stored in clear text in the configuration file because it is not really considered sensitive information (it is free after all). Note that this can be easily avoided by removing the password from the config file and setting it as an environmental variable instead(Spring supports property override out of the box).

Limitations

Unit test coverage is deemed adequate for a simple application like ExchangeRates. However, more tests can be added in the future to cover the frontend functionality (AngularJS has great support for unit testing) and the full end to end black box integration tests (ideally through a BDD framework like JBehave). This is by far outside the estimated 10 hours for this task.

Technology stack decisions

Spring vs JEE

The application framework for this application will be Spring instead of JEE(e.g. CDI, EJB etc). Spring is by far the most popular framework for building enterprise java applications. Spring implementation of IoC is easier to use and understand than the JEE alternative - CDI. In addition, Spring WebMVC makes the implementation of REST Services a breeze. Then, Spring does not require an (allegedly) heavyweight JEE application container(Weblogic, WebSphere etc) but only needs a lightweight Servlet Container like tomcat. This also means that migrating the application to cloud infrastructure like AWS would be also easier because AWS has great support for tomcat through ElasticBeanstalk.

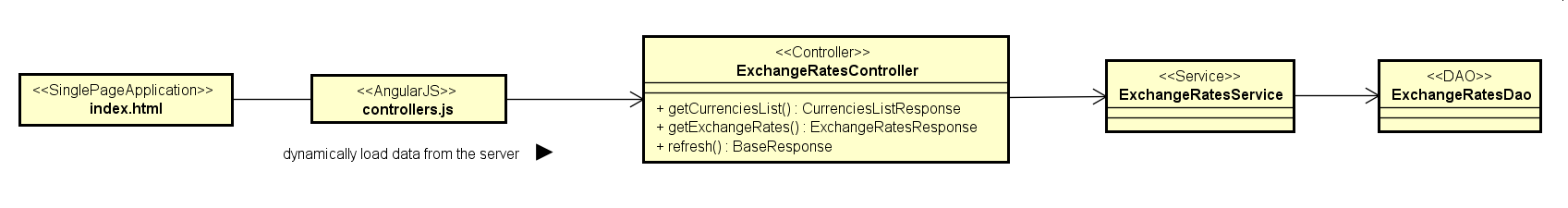
AngularJS

Although there are quite a few alternatives for the frontend at present, the choice here again seems very easy. With the raise of JavaScipt in the last couple of years(the most popular language in the world currently), one should have a very good reason to use other fronted technology. The JEE standard JSF never managed to gain widespread acceptance within the Java community. JSF has quite steep learning curve, it is quite heavy weight and quite restrictive. Struts is a good and powerful framework but it is legacy by now and the open source community around it is not as vivid as it once was. Spring Webflow is not a bad alternative but Angular is just better. It is light weight and very easy to master, it has big ever increasing community around it and therefore many widgets that can be obtained for free. It is also defacto the default framework to use for building Single Page Applications.

Bootstrap

Angluar is conveniently complemented by Bootstrap with a wide variety of widgets so one can build rich SPAs very quickly. Most of the components in Bootstrap have been adapted to Angular already so using them is fairly straight forward.

Class Diagram



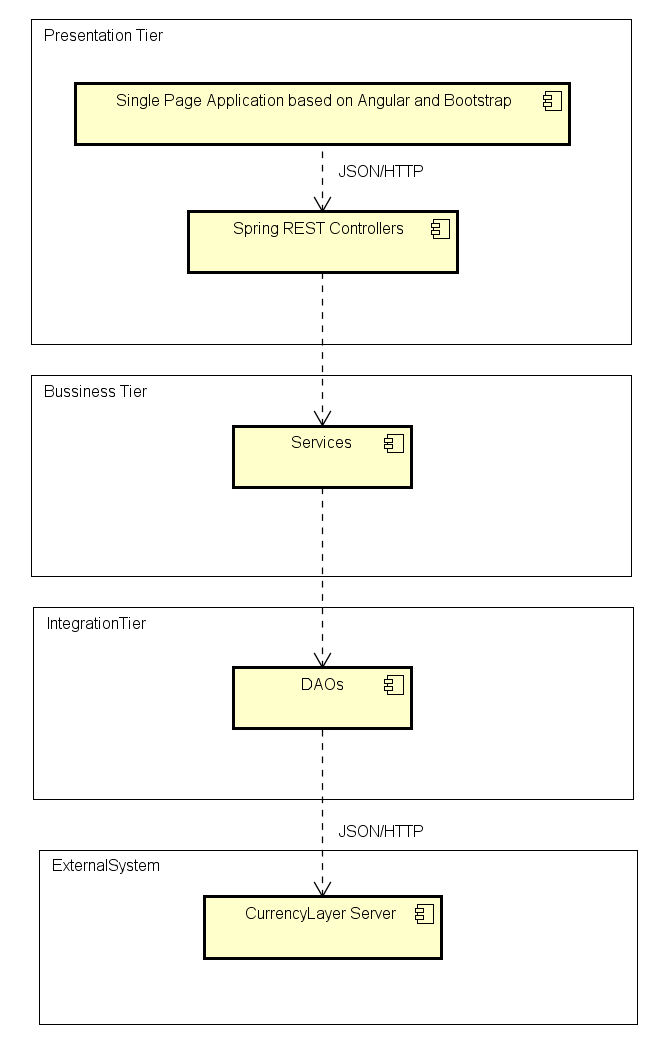
The class diagram shows the main classes in the application. The frontend is fairly simple with hardly any business logic, just asynchronous calls to the backend and updating the view with the values. Again, exchanges rates are not cached here to save external server hits.

A few third party widgets are used to enhance the user experience. First, since the drop down to select base currency contains 162 values, a component that allows easy search in the values is used. In much the same way, the table with the destination currencies can be filtered out to contain only certain currency.

Since there is so little business logic in the angular controller, there is no need for a frontend angular service which is the right place for more complex, reusable business logic in the fronted.

The Spring REST controller is intentionally kept light weight but one important thing for every web application is to catch all exceptions and convert them to appropriate JSON response. All the heavy lifting is in the Service e.g. computing rates, caching etc. The DAO has low level functionality for communicating with the external server by executing the live and list requests.

Component Diagram



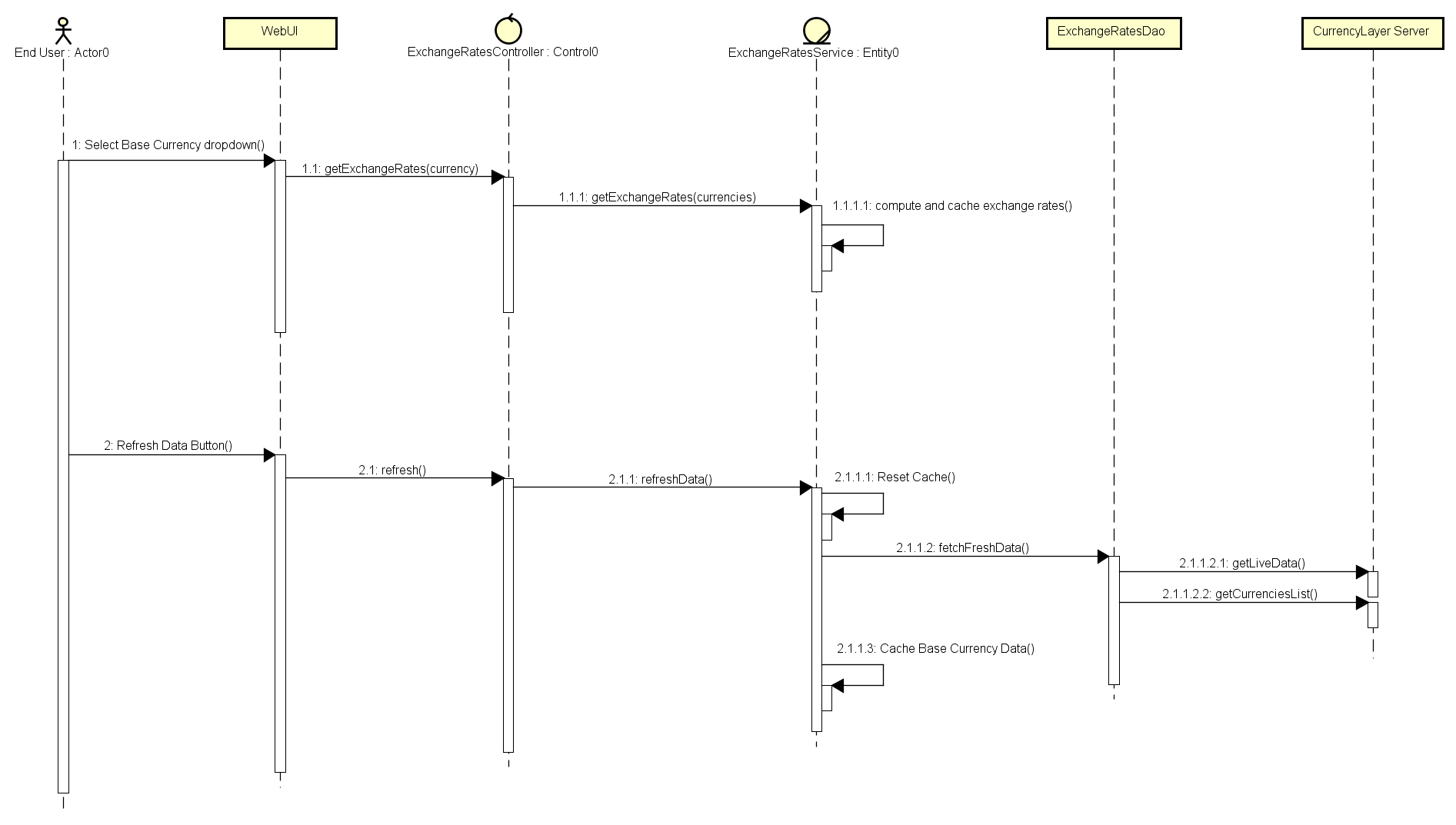
The component diagram is intended to illustrate the clear separation of concerns by isolating functionality into different tiers.

The presentation tier consists of the angular frontend and the rest controllers on the server. It is perfectly acceptable to put more heavy business logic on the client to make the application really feel like a SPA. However, due to the simplicity of the application functionality, there is not much logic in the frontend. The spring REST controllers should be kept lightweight.

All business logic should go into the services in the Business Tier. The services contain coarse grained functionality the is aware of the business domain of the application unlike the Integration Tier.

The integration tier abstracts away the external services consumed by the application making it easy to replace those in the future. For example, if we need to switch from CurrencyLayer to another provider, the changes needed are isolated only in the respective dao.

Sequence Diagram



The sequence diagram details how get exchange rates and refresh data methods work. The important part is the server side caching and rate computation.

Non Functional Requirements

The architecture is usually mainly about non functional requirements and risks. The most important non functional requirement is called Architecture Driver and it should be considered when making any decision. The risk analysis is important because it is crucial to address the big risks as early as possible and mitigate them as much as possible.

After analysing the requirements, the most important non functional requirements we identified: performance, user experience and flexibility. The architecture driver is performance.

To address performance requirements, server side cache was implemented thus decreasing the application response time drastically.

Superior user experience comes naturally with Angular Single Page applications. In addition, some great third party widgets were used for searching in drop down and displaying rates in a great looking filterable table.

The flexibility is achieved through implementing the Separation of Concerns principle – the application consists of distinct tiers. Therefore it is easy to maintain and change.

The main risk associated with the application implementation is to not deliver on time. The application functionality is very simple, however building an entire application from the ground up in 10 hours along with documentation and architecture is a challenge. To mitigate this risk, the important functionality of the system was implemented first, the so-called MVP (minimal viable product). All testing was left at the end but still the system has adequate amount of automated tests.