

# Miniproject Report

---

**Course: Development of Large Systems**

Bachelor in Software Development

EASV, 1st Semester 2015

**Name:** Christoffer Bjerregaard

**URL:** <http://client.c-bjerregaard.dk>

## Indhold

1. Introduction.....	2
2. Fulfillment of Requirements .....	2
3. Architectural design .....	3
4. Client Design and Implementation .....	4
5. Server(s) Design and Implementation.....	7
6. Data Base(s) Design .....	8
7. Conclusions and Further Work.....	8
8. Bibliography .....	10
9. Appendix.....	10

## 1. Introduction

My miniproject is a country information website, where you can view a lot of information about countries. You can personally choose what countries you wish to know more about. I wanted it to be fast and responsive for the user to use, and I believe I have accomplished that.

You can visit the website via this link and see the solution in action:

<http://client.c-bjerregaard.dk>

## 2. Fulfillment of Requirements

### 1. Mashup pages.

I have chosen to make my mashup pages client side, in the Angular JS[1] framework. This makes for a snappy client, which loads the content dynamically, and all resources (html, css etc.) on the first page load. This means that mostly all assets are served with the first GET request.

### 2. Information about country.

In my client I show the user the following information about a country:

Name, Alpha2, Alpha3, Region, Subregion, Population, Short description, Google Maps location and Currency.

### 3. Webservices.

I use a few webservices.

- Rescountries.eu[2]
  - o For getting simple country information (Population, Region, Subregion, Name of currency, Lat and Long coordinates). It looks after a country matching the Alpha2 code.
- Freebase API[3]
  - o For getting the short country description. It looks on Wikipedia for articles matching the country name.
- Yahoo finance[4]
  - o In my RMI Server I use Yahoo Finance to get the currency exchange rates for the countries, and for calculating the exchangerate between two chosen countries.

### 4. Currency Exchange rate conversion utility

It is implemented on the RMI server, which is linked to the client, via my Spring server.

### 5. GET, POST, PUT, DELETE verbs.

Most of the verbs are used in the client. The RMI server GETs the currency exchange rate from Yahoo Finance. In my client I use GET request up against my Spring server, which returns a list of countries, and GET requests against several API's which returns a lot of data in JSON format to my client.

POST has been implemented on the “Add new country” view, which POSTS some data to my Spring server, that in turn creates a new country in the database.

DELETE has been implemented. It’s simply for deleting a country. The Spring server accepts an ID as parameter, and simply deletes the country with that corresponding ID.

PUT is implemented as a simple Edit country function, where you can alter a country’s name or both alpha codes.

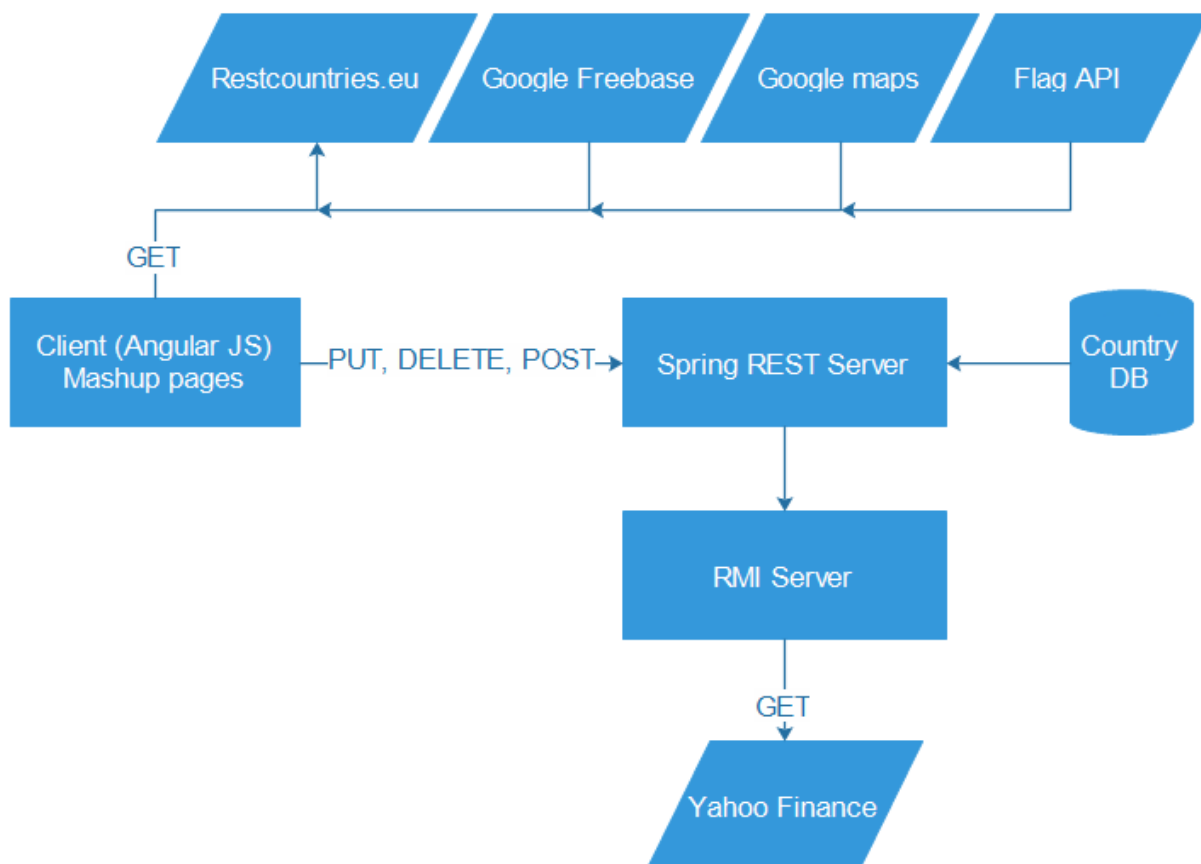
## 6. 3 servers.

I have implemented two servers. The mashup server I felt was unnecessary, because I am using AngularJS to handle the templating, and putting in the JSON responses dynamically if the request is successful.

## 7. Client – Server communication.

The client communicates via http verbs to the server, which in turn communicates with the RMI server via an interface defined in both projects.

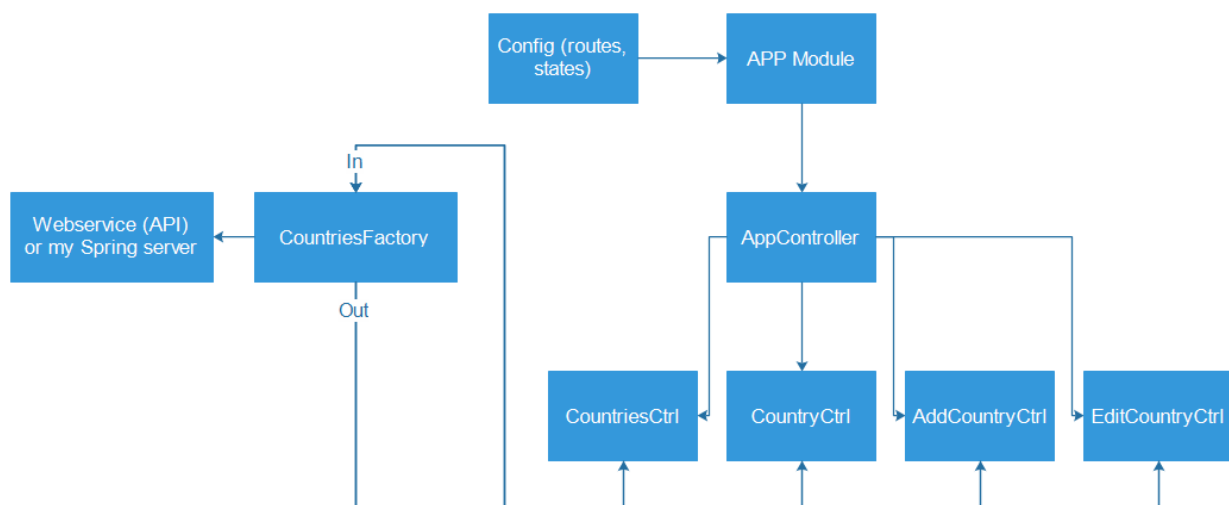
## 3. Architectural design



This is my architectural design. I have scrapped the mashup server in favor of an Angular JS client, and most of my API requests are handled in the client. My server however is still the heart of the website, as it is the one controlling what the user gets to see.

#### 4. Client Design and Implementation

My Spring[5] server accepts GET, PUT, POST and DELETE requests, and returns JSON. This is perfect for an Angular JS client. Angular is a Javascript Model View framework, which makes it possible to make snappy and responsive web applications. I felt that with the relatively simple data, that Angular would perform well. All assets are loaded when you visit the page, this means that you wont ever have to load again, until of course you request data from some webservice. However this is seamless, and you can view the template while the data is getting loaded.



This is a model of my Angular application layout. I have some config .js file, which loads the module and dependencies. And also creates the url states, and links these states with the right template HTML file, and controller. I chose to have a controller at the top level, that I named “AppController”. I did this, because every controller beneath it then inherits the functionality of the parent controller. This means that I can easily send data to the template of the parent controller, without having to use a service.

Every controller, except for the AppController is injected with a CountriesFactory factory, which I’ve made. It contains a set of functions to communicate with different web services and my Spring server. Additionally I created an RMIFactory, which is injected into the AppController, to get the status of my RMI server.

I like the Controller -> Factory -> Controller pattern for when I’m communicating with API’s because it is relatively simple. Take for example this function from the “CountriesCtrl”.

```
CountriesFactory.getCountries()
    .then(function (response) {
        $scope.countries = response.data;
    });
```

controllers.js

```
getCountries: function () {
    return $http({
        url: BASE_URL + '/get/countries',
        method: 'GET'
    })
    .success(function () {
        console.log('getCountries success.');
```

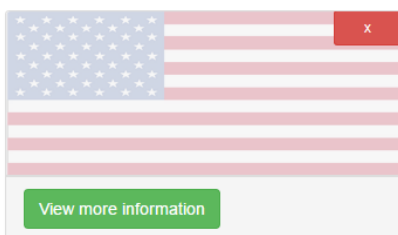
factories.js

This function uses the CountriesFactory that I injected into the controller, to access the functions in the factory. In this case the “getCountries” function. The getCountries function, uses a native Angular JS service, called \$http, which takes a few parameters, like url and method. The getCountries function returns this \$http service, and when it completes, it returns a promise, containing a response. This response is accessed by calling a .then on the function, which then holds the data returned by the \$http service. Then I can take the data from that response, and

make it available on the scope of the controller. Then I can access it in my HTML template like this:

```
<div class="col-lg-4" ng-repeat="country in countries">
    <div class="panel panel-default">
        <div class="panel-heading w_flag">
            
            <input type="button" value="x" ng-click="deleteCountry(country.id)" class="btn btn-xs btn-dange
        </div>
        <div class="panel-footer">
            <input type="button" class="btn btn-success" ui-sref="countries/country({country: country.name,
        </div>
    </div>
</div>
```

The “ng-repeat” directive is placed in an HTML block, which will be repeated for each element in the “countries” object that we made available. Our object contains simple JSON, so it is easy to put into our template. The data from the object is accessed within the {{ brackets }}. The final result is shown here:



It has the image that we fetched from the geonames flag api, and the button which redirects the user to the country information page.

View Countries
Add new country
RMI Status:

Info

Name: Denmark

Alpha2: dk ISO 3166-1 alpha-2

Alpha3: DNK ISO 3166-1 alpha-3

Capital: Copenhagen


Region: Europe

Subregion: Northern Europe

Population: 5655750

Information about Denmark

Denmark (/ˈdɛnmɑːrk/; Danish: Danmark, pronounced [ˈd̥ænmɑɡ̊] (listen)), officially the Kingdom of Denmark (Danish: Kongenget Danmark, pronounced [ˈkɔŋəɪ̯əðˌd̥ɑnmɑɡ̊] (listen)), is a sovereign state in Northern Europe, with two additional overseas constituent countries also forming integral parts of the kingdom; the Faroe Islands in the North Atlantic and Greenland in North America. Denmark proper is the southernmost of the Nordic countries, located southwest of Sweden and south of Norway, and bordered to the south by Germany. The country consists of a large peninsula, Jutland and many islands, most notably Zealand, Funen, Lolland, Falster and Bornholm, as well as hundreds of minor islands often referred to as the Danish Archipelago. The Kingdom of Denmark is a constitutional monarchy organised in the form of a parliamentary democracy, with its seat of government in the capital city of Copenhagen. The kingdom is unitary, with some power being devolved from the central government to Greenland and the Faroe Islands, this polity is referred to as the Danish Realm. Denmark proper is the hegemonial area, where judicial, executive, and legislative power resides. The Faroe Islands



Currency converter with amount.

Source: DKK

Target: EUR

100

Result: 13.4 EUR

Convert

Edit Denmark

Currency converter

Source: DKK

Target: EUR

Result: 0.134 EUR

Convert

This is what you'll see if you click on View more Information. It will list some information about a country, that is fetched from several API's, as well as from the RMI server.

There is also a currency converter, and an exchange rate viewer that you can use. This view of course has its own Angular JS controller as well, called "CountriesCtrl". This view is also cached in the client, which means that if you visit the same country once more, the data is stored in the cache, and will be served instantly.

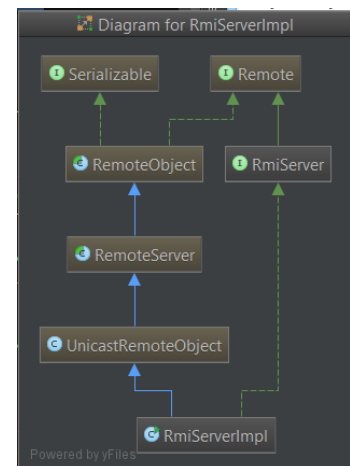
## 5. Server(s) Design and Implementation

### RMI Server:

The RMI server is implemented with three main classes.

1. RmiServer – Interface
2. RmiServerImpl – Implemented methods from the interface.
3. CurrencyLoader – Enum, for loading currencies.

The CurrencyLoader is implemented as a Singleton, and is accessed by calling `.INSTANCE`. This means that the CurrencyLoader object is threadsafe, and won't be able to be accessed from any other threads. When the RMI server is started, it will register the RMI server to a port and a hostname, so that it is easier to access via the REST server, just by listening on the port and checking if a service is running there with that specific hostname. When the server is running, it will start scanning the currencies from the text file and querying Yahoo finance for the exchange rates. The UML diagram shows that the RmiServerImpl class implements RmiServer, which implements Remote. This makes it possible to access the methods from this class on a remote server.



### Connecting the two servers:

Both the RMI server and the Spring REST server has the class "RmiServer". They have to, for security reasons, be in the same namespace and have the same name and the same methods. This is to combat someone else trying to hijack your remote server, and execute methods on it to possibly harm your service. The two classes, RegistryConfig and ServerConfig are also implemented on both the RMI server and the REST server, to give them the same configurations, so that the port and hostnames are the same, so that the REST server will be able to connect to the RMI server.

On the REST server I have an "RmiConnector" class, and of course the same interface implemented on the RMIServer. The RmiConnector class is, like the "RmiServerImpl" also an enum, to make it a singleton. The RmiConnector has a constructor method, that when the object is created, it will fire the `connectToRmi()` method. This method uses the aforementioned config classes from the Shared package, to know what port and hostname to listen on. If connection is successful it will return an instance of the RmiServer and you will be ready to use the methods remotely.

### Spring REST server:

My Spring REST server is using SQLite[6], and therefore the DatabaseConnection class is pretty simple, but also a singleton. It simply scans the folder for a SQLite database file, and returns a connection so that it can be used in the CountryDBManager enum. In the CountryDBManager I have implemented several CRUD methods, to access and alter the database and return the countries to the user. The CountryController, which is a Spring Controller, has an instance of the CountryDBManager so that it can access the methods that accesses the database, and also has an instance of the RmiConnector, so that it can access the remote server.

```
private static CountryDBManager _cdbManager = CountryDBManager.INSTANCE;
private static RmiConnector _rmiConnector = RmiConnector.INSTANCE;
```

This is how simple it is to create singletons with the enum design pattern. The CountryController wires the url paths so that it can be accessed from the client. All of the methods in the CountryController returns JSON, that the client supports.

It is very simple to implement GET, PUT, POST and DELETE verbs in the Spring controllers, and also to implement request parameters.

```
@RequestMapping(value = "/post/country", method = RequestMethod.POST)
public String addCountry( @RequestParam(value = "name") String name,
                          @RequestParam(value = "alpha2") String alpha2,
                          @RequestParam(value = "alpha3") String alpha3) throws SQLException {
    _cdbManager.addCountry(name, alpha2, alpha3);
    return "{\"message\":\"Successfully added country '" + name + "' with id\"}";
}
```

This is the POST method for adding a country. It accepts 3 parameters (the database autoincrements the ID, so we don't need to think about that). It then accesses the CountryDBManager, to add the country to the database.

```
@RequestMapping(value = "/get/RmiServer", method = RequestMethod.GET)
public String getRmiServer() throws RemoteException, MalformedURLException, NotBoundException, ServerNotActiveException {
    try (Socket ignored = new Socket(ServerConfig.SERVER_IP, RegistryConfig.REGISTRY_PORT)) {
        try {
            server = (RmiServer) Naming.lookup(ServerConfig.SERVER_ADDRESS);
            // Log client on RMI server
            _rmiConnector.getRmiServer().getClientInfo();
            return "{\"status\":\"1\"}";
        } catch (NotBoundException | MalformedURLException | RemoteException | ServerNotActiveException e) {
            System.out.println(e.getMessage());
        }
    } catch (IOException ex) {
        return "{\"status\":\"0\"}";
    }
    return "{\"status\":\"0\"}";
}
```

This is the method to GET the status of the RMI server. In this you can see how simple it is to access remote methods. I simply create a new Socket, which scans the server ip and the registered port. And then I try to access the RMI server by looking up via the server address. Then I use the instance of the RmiServer to try and get the client info. If this is successful, I return status 1 to my client, who then knows if the RMI is online, and notifies the user. If the server is unreachable, it will return 0, and the Server will appear offline, but you can still access the methods of the Spring server, that does not utilize the RMI server.

## 6. Data Base(s) Design

I have a very simple SQLite database with one table for the countries. It has 3 attributes, which are Name, Alpha2 and Alpha3. I have chosen this, because nearly every API can be accessed using a combination of these attributes as parameters, or simply one of them.

## 7. Conclusions and Further Work

All in all I am quite content with my solution. I liked working with Angular JS, and I found Spring to be a quite powerful framework. I like how snappy the website feels, and how fast the client fetches data, both from my server which is located in France, and from the API's. I think the performance is quite good, and even with 3000 countries in the list, it performed well. The limitations lie in the client when displaying that much JSON data, because it has to render all the HTML blocks. JSON is extremely fast, because all you have is text data, and therefore it does not take long to get the data.



I also like how all the HTML, styling and images are downloaded and implemented at first load, so that you won't get the "click url -> load data w. white screen -> display data". It instantly switches pages, and the data is loaded in the background.

I feel that I have learned a lot by implementing these two servers and to get them to communicate with each other. I know I have learned a lot about http requests. I had some trouble due to CORS being an issue when you're using Angular JS. I had to implement a filter class, which is fired when the server is launched, that then adds the response header: Access-Allow-Control-Origin = \* header, which then will allow API requests from all sources. Instead of a star, I could have written an URL if I wanted, for added security, but I decided that it could just be an open API. This also meant that a classmate found my API url, and made 3000 add country requests via a rest client. But that's how I tested how it handled under a bit of stress. So that was fun. Adding the filter also allowed me to specify that only GET, PUT, POST and DELETE verbs were allowed in the API.

I could improve the solution showing a list of the chosen country's exchange rates against some popular currencies. But overall I am happy with the speed of the application and also how it is implemented.

All server and client code is written by myself. Some of the code in the RMI server is from Programming Assignment 2, where I cooperated with Klaus Andresen.

## 8. Bibliography

In addition to google and Stackoverflow, which has been my partners in this project, I have used the following documentation and API's.

1. Angular <http://docs.angularjs.org/api>
2. Restcountries <http://restcountries.eu>
3. Freebase <https://www.freebase.com/>
4. Yahoo <http://finance.yahoo.com/>
5. Spring <https://spring.io/>
6. SQLite <https://www.sqlite.org/>

## 9. Appendix

### User guide

If you wish to run the miniproject locally, there are a few steps:

1. First, run the main method in the RMI Server. It's located in the RmiServerImpl class.
2. Secondly run the Spring application. The main method is located in the Server class.
3. Then put the client files in a www environment, such as wamp or xampp, and then redirect your browser to the index.html file.
4. If the port for the spring server is not the standard :8080 you will have to alter the URL in the app.js file, where it configures Angular.

```
9 // Local hosted
10 var BASE_URL = "http://localhost:8080";
```

*App.js*