1.REST API, Error Responses & JSON

Our API consists of 3 REST services; AddressResource, CompanyResource and PersonResource. Each of these REST services have their own CompanyResource and PersonResource both have the HTTP CRUD methods POST, GET(1 entity and all), PUT and DELETE, along with a couple of others that were defined in the assignment, such as “getAllPersonsFromZip”.

We managed to implement 2 “specific” exceptions, and 1 “Generic” exception.
EntityNotFoundException is used to return a 404 response the specified entity is not found in our database.
NotNumericException is used to return a 404 response if a certain call does not contain integers, though it should (writing a name in a field for phone numbers for example).
We use a GenericExceptionMapper to take of exceptions that are not supported by EntityNotFoundExceptionMapper and NotNumericExceptionMapper.

PersonResource uses a createJsonObjectFromPerson(Person p) method to create Json objects from Person objects, and CompanyResource uses one much like it. We chose this method to simplify the process of creating Json objects in the REST Services, instead of having to create a Json object and format from scratch in every single method.

How to use our API with AJAX and JQuery:

1:
Create a function, and make a request with AJAX like this (either in the html page or a separate javascript file):

```
function createPerson()
        { var request = $.ajax({
```

2:
In the request, include information you want to send in the request, in this example we will send data used to create a person, alongside information used to define our request.
url: The url of the REST Service method you want to call. (“api/person/create”).
dataType: Which kind of datatype you want to POST (since we’re working with json objects, we will post it as “json”).
type: The type defines which kind of HTTP method it is (in our case it’s “POST”).
contentType: the format of which our REST service produces or consumes data (“application/json”)
data: This field is used to determine the actual data, that we want to POST. In our case we want to create a Person Json object, so we want to parse and post values from fields on our website.

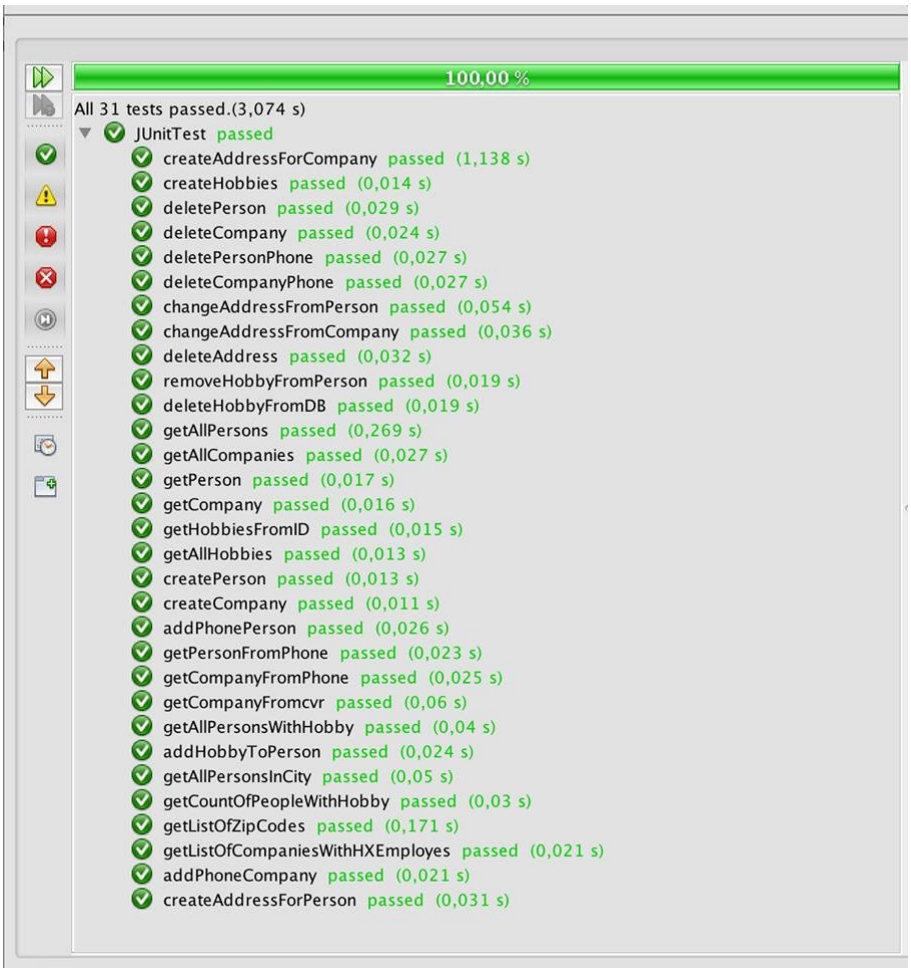
```
data: JSON.stringify({"firstname": $("#fname").val(), "lastname": $("#lname").val(), etc.
```

In this example “firstname” is the attribute we’re POSTing, and $(“#fname”).val() is defining which div etc, we’re getting our data from, and then .val to extract the value of the field.

At the end of our function, we might want something to happen once our request is done.
request.done(function (jqXHR, textStatus)
{ alert("Person created!");
This will create an alert box saying ”Person created!”.

2.Test and jUnit Test

First and foremost, we did not feel like we had enough time to implement jUnit tests using Restassured, because we spent too much time implementing REST Service methods, and getting them to work on our webpage with jQuery (and a probably too covering façade jUnit test).
Our Test Strategy for our Façade has been to test all of its methods. To insert, remove, edit and find data in our MySQL database. We also implemented a test package with a few test classes that we used to test our database before we implemented jUnit tests for our Façade.
Instead of using Restassured to test our REST Services, we used Postman to test their “actual” functionality. If we had more time, we would’ve very much liked to implement Restassured, but we chose to focus primarily on the functionality of our program.



3.Inheritance

When given the assignment, we were given a business domain, describing how our entities should look, and how they should inherit from each other.
Both Company and Person extend the InfoEntity class, which means they both have an id, and an email. Phone and Address are both related to Person and Company, and Hobby is related to only Person. The entity CityInfo is related to Address.
We chose the inheritance strategy to be TABLE_PER_CLASS to avoid null values in our database. We considered using the strategy JOINED, but it created additional tables and null values in our database, so we decided to use TABLE_PER_CLASS.
This results in us having a table for each of our entities, which at first look might seem unheard of, since there's a lot of navigating through tables and many different IDs, but we find it more manageable this way.
And since the code doesn't care about how our database tables look in this scenario, we decided to do it this way.

4.Who did what?

Openshift: Christian
Web pages: Jonas & Andreas
jQuery/Javascript: Jonas & Andreas
Entities: Jonas & Christian & Andreas
Façade: Andreas & Jonas
REST: Andreas & Jonas
Exceptions: Jonas & Christian
Tests & jUnit: Jonas & Christian

5.Demo of the program

Postman demo: A list of all Postman inputs are documented on our website.
Website demo: Navigate through the links at the top of our webpage. Press buttons or fill out forms to create, delete or view persons/companies.
(Many of our HTTP methods will work in Postman, but are not yet implemented on our website, due to lack of time)