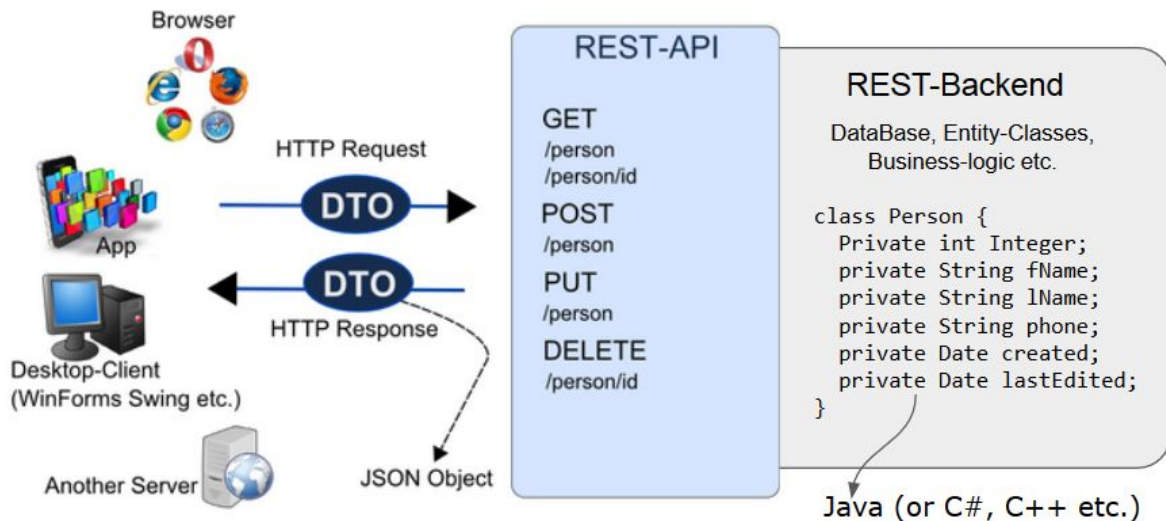


REST with JAX RS

In this exercise we will go through all the steps necessary to create a REST driven application as sketched below:



This exercise is meant for two days, so when you get to the part related to error-handling, this will be covered Thursday.

In the backend, you will implement an Entity class `Person` with the following fields: **firstName**, **lastName**, **phone** (String), **created**, **lastEdited** (java.util.Date) and **id** (Integer).

You will also create a `PersonFacade`, that implements this interface:

```
public interface IPersonFacade {
    public Person addPerson(String fName, String lName, String phone);
    public Person deletePerson(int id);
    public Person getPerson(int id);
    public List<Person> getAllPersons();
    public Person editPerson(Person p);
}
```

By now, you should know that an important REST constraint is to have a layered system, with *Resources Decoupled from their Representation*. For this exercise, we will expose data as JSON.

For the REST-endpoints that returns a `Person`, the JSON object should be built like this (observe the absence of the two date-fields):

```
{"fName":"Kurt","lName":"Wonnegut", "phone":"12345678","id":0}
```

For the REST-endpoints that *creates* a `Person`, send JSON as above, without the `id` property.

For the **GET** method that returns all `Persons`, the JSON must have this format:

```
{
  "all" :[
    {"fName":"Kurt", "lName":"Wonnegut", "phone":"12345678", "id":0},
    {"fName":"Peter", "lName":"Hansen", "phone":"12345678", "id":1}
  ]
}
```

The facade and the matching endpoints

- 1) Use our start code as the starting point for this exercise. Initially, just skip the steps related to Travis and Deployment. It will provide you with a starting point for the Rest Assured tests you have to write.
- 2) Create a person database and, either use your existing, or create a new test database for this exercise.
- 3) Implement a facade class that implements the `IPersonFacade` given above. Initially just add dummy methods, since we like you to implement the methods, step-by-step, including their matching REST-endpoints. When you implement the methods, make sure to create corresponding JUnit tests.
- 4) Complete the **GET** methods in the facade, and implementing matching REST endpoints. Test via a browser or Postman (Rest Assured will be requested later).
- 5) Complete the **addPerson(..)** method in the facade, and implement a matching REST endpoint. Test using Postman
- 6) Complete the **editPerson(..)** method in the facade, and implement a matching REST endpoint. Test using Postman
- 7) Complete the **deletePerson(..)** method in the facade, and implement a matching REST endpoint. Test using Postman

Hints: If you use these DTO classes as mappers for your gson-conversions, they will provide you with JSON exactly as specified initially for this exercise.

*An instance of this DTO class, used as input to Gson will provide you with JSON as requested for a single Person.
It can also be used as a mapper to convert an incoming JSON string for a POST request to a Java Object.
The second constructor can be used by your RestAssured Tests when you need to provide a Post request with JSON data.*

```
public class PersonDTO {
    private long id;
    private String fName;
    private String lName;
    private String phone;
    public PersonDTO(Person p) {
        this.fName = p.getFirstName();
        this.lName = p.getLastName();
        this.phone = p.getPhone();
        this.id = p.getId();
    }
    public PersonDTO(String fn,String ln, String phone) {
        this.fName = fn;
        this.lName = ln;
        this.phone = phone;
    }
    public PersonDTO() {}
    // getters setters hashCode and equals
}
```




An instance of this DTO class, used as input to Gson will provide you with JSON as requested for a list of Persons

```
public class PersonsDTO {
    List<PersonDTO> all = new ArrayList();

    public PersonsDTO(List<Person> personEntities) {
        personEntities.forEach((p) -> {
            all.add(new PersonDTO(p));
        });
    }
}
```

Rest Assured Tests

For each of the REST-endpoints created above, implement one (or more) Rest Assured test which should mirror the way you initially tested the endpoint using Postman

   You don't necessarily have to complete this for all your endpoints. For green students, make sure to implement a few, and red should implement all.

Hints

Error Handling with JAX RS and ExceptionMappers

This exercise is meant for Thursday

1) Add the following Exceptions to the system (in a package exceptions)

```
public class PersonNotFoundException extends Exception {
    public PersonNotFoundException(String message) {
        super(message);
    }
}
```

2) Change the interface (and implementation) as sketched below:

```
public interface IPersonFacade {
    public Person addPerson(String fName, String lName, String phone);
    public Person deletePerson(int id) throws PersonNotFoundException;
    public Person getPerson(int id) throws PersonNotFoundException;
    public List<Person> getAllPersons();
    public Person editPerson(Person p) throws PersonNotFoundException ;
}
```

The API-description for this exercise will now be supplemented with a description for the responses for Error Scenarios.

Error responses for GET: /person/{id}

```
{"code": 404, "message": "No person with provided id found"}
```

Error responses for DELETE:

```
{"code": 404, "message": "Could not delete, provided id does not exist"}
```



For all RuntimeExceptions

```
{"code": 500, "message": "Internal Server Problem. We are sorry for the inconvenience"}
```

For those errors where the server by default throws (an HTTP encapsulated) error (calling an endpoint that does not exist, calling a method (DELETE for example) that does not exist)

{ "code": nnn, "message": "xxx"} where code and message is taken from the original error.

Hints: Everything requested above can be handled by our suggested ExceptionMappers and ExceptionDTO. See references given for your class (slides, guide or whatever was used)

3) Test your error responses, first using Postman, and   then with JUnit and Rest Assured.

Hint: See how the original startcode verified that the server was running, use this strategy (with the expected status code), and the fact that errors are now returned as JSON.

Additional Error Responses

Add a few more exceptions with corresponding ExceptionMappers, for example the one given below (if you have time only)

```
public class MissingInputException extends Exception {  
    public MissingInputException (String message) {  
        super(message);  
    }  
}
```

Change the interface to throw this Exception for addPerson(..) and editPerson(..) also, if you like

Error responses for POST:

A person must have both a firstName and a lastName

```
{"code": 400, "message": "First Name and/or Last Name is missing"}
```

Error responses for PUT:

```
{"code": 400, "message": "First Name and/or Last Name is missing"}
```

Test with JUnit and RestAssured

Entity Classes with relations


Create a separate branch for this part, since it will require changes in many places in your project. To speed up the process, and to focus on relations only, you could also (initially) disable your tests.

1) Create a new java class, entity class `Address` and create a few fields to represent an address (street, zip and city).

Implement a *bi-directional* relationship, so that a `Person` has a single `Address`, and to simplify matters, only one person can live at an address (one-to-one relationship).

Change the `PersonDTO` to include the address fields. Observe, we don't need another DTO since a person can have only one address.



2) Change all relevant places in the code to use this structure. If you delete a `Person`, also delete the `Address` since no one else "can live here" given the one-to-one relationship between the two.

3)  Change the relationship between `Person` and `Address`, so that a `Person` still can have only one `Address`, but an `Address` can belong to several `Persons` (many-to-one, between `Address` and `Person`).

You still don't need an additional DTO since we won't require an endpoint that can return `Addresses` (which can relate to several persons).

It will however probably cause a problem when you try to create a new `Person` from a DTO with both name, phone, and address info. For this exercise, just check whether an address with street, zip and city already exists (spelled exactly as it comes in). If it is found, use that existing address, if not just create it as a new.

This exercise is meant for week 2 (JavaScript)

1. Implement a read-only page to show all Persons in a table. The table must be built in the browser using plain JavaScript, and data fetched via a REST call.
2. Add a refresh button that should refresh the page designed in the previous step. Use Postman to add a new Person to verify that we actually get an updated list (without having to create a new page on the server).
3.  Add an option to create new Persons (inspired by the figure below) on the same page as the one with the table. Use the REST API to create the new person on the server (hint-3).
4.  Add an option to delete/edit persons as sketched below

ID	First Name	Last name	Phone	
1	Peter	Olsen	1234	delete / edit
2	Hanne	Olsen	1234	delete / edit
3	Kurt	Wonnegut	7985476	delete / edit
4	Heidi	Petersen	1234	delete / edit

Reload DataAdd New Person

Hints for the JavaScript part:

Posting data with fetch. Use this [stackoverflow](#) article for how-to information

When you press the Add New Person button: Bring up a modal form as explained in this [article](#).

Handling the delete/edit links: One way to do this, would be to add the person's id as the id-value for the anchor-tag, and a class declaration used to distinguish this link from others (edit-links) as sketched for a single row below

```
<a href="#" class="btndelete" id="1">delete</a>
```

Now attach a click handler to your tbody-tag and use the fact that events, by default, bubbles up to handle all "delete-events".