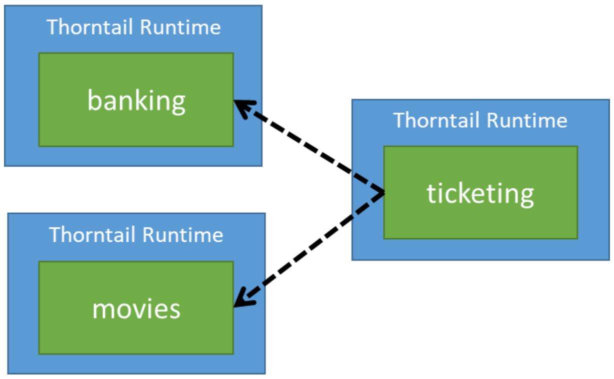
4. Microservices homework

Dr. Balázs Simon (sbalazs@iit.bme.hu), BME IIT, 2019

In the following text use your own Neptun code with capital letters instead of the word "NEPTUN".

1 Description of the task

The task is to create a ticket selling system for movies. The architecture of the system is the following:



The server is built of three microservices:

- **movies**: a REST resource service for handling movies
- **banking**: a REST RPC service for simulating banking operations
- **ticketing**: a REST RPC service for buying tickets

Each microservice runs on its own microserver. The ticketing service is built on the movie and banking services.

2 The movie service

The service should be implemented in a Thorntail Maven application similar to the one in the tutorial. The pom.xml must be the attached **movies/pom.xml**, but in this file the word NEPTUN should be replaced with your own Neptun code.

The name of the application should be: **Microservices_NEPTUN_Movies**

The service has to store data between calls. In a real world application the storage should be a database. In the homework, the storage should be static variables.

The base URL of the service must be:

```
http://localhost:8081
```

The movie service has the exact same functionality as the REST homework (2nd homework). The messages exchanged by the service is described by the attached **movies/movies.proto** file, which should be copied into the **src/main/resources** folder.

The service has to support the following calls added to the base URL above:

- `GET /movies`
    - input HTTP body: empty
    - output HTTP body: `MovieList`
- `GET /movies/{id}`
    - input HTTP body: empty
    - output HTTP body: `Movie`
- `POST /movies`
    - input HTTP body: `Movie`
    - output HTTP body: `MovieId`
- `PUT /movies/{id}`
    - input HTTP body: `Movie`
    - output HTTP body: empty
- `DELETE /movies/{id}`
    - input HTTP body: empty
    - output HTTP body: empty
- `GET /movies/find?year={year}&orderby={field}`
    - input HTTP body: empty
    - output HTTP body: `MovieIdList`

3 The banking service

The service should be implemented in a Thorntail Maven application similar to the one in the tutorial. The pom.xml must be the attached **banking/pom.xml**, but in this file the word NEPTUN should be replaced with your own Neptun code.

The name of the application should be: **Microservices_NEPTUN_Banking**

The base URL of the service must be:

```
http://localhost:8082
```

The following pseudo-code describes the interface of the service:

```
interface Banking
{
    bool ChargeCard(string cardNumber, int amount);
}
```

The **ChargeCard** function simulates the charging of a credit card. The **cardNumber** is and arbitrary string value, the **amount** is an arbitrary integer number. The function returns true if the **amount** is positive and the length of the **cardNumber** is an even number. Otherwise, the function returns false.

The service has to be implemented as a REST RPC service. The format of the input and output messages are described by the attached **banking/banking.proto** file, which should be copied into the **src/main/resources** folder. The functions of the service can be called by the following REST calls added to the base URL above:

- `POST /banking/ChargeCard`
    - input HTTP body: `ChargeCardRequest`
    - output HTTP body: `ChargeCardResponse`

4 The ticketing service

The service should be implemented in a Thorntail Maven application similar to the one in the tutorial. The pom.xml must be the attached **ticketing/pom.xml**, but in this file the word NEPTUN should be replaced with your own Neptun code.

The name of the application should be: **Microservices_NEPTUN_Ticketing**

The service has to store data between calls. In a real world application the storage should be a database. In the homework, the storage should be static variables.

The base URL of the service must be:

```
http://localhost:8080
```

The following pseudo-code describes the interface of the service:

```
struct Movie
{
    int id;
    string title;
}

struct Ticket
{
    int movieId;
    int count;
}

interface Ticketing
{
    Movie[] GetMovies(int year);
    bool BuyTickets(int movieId, int count, string cardNumber);
    Ticket[] GetTickets();
}
```

The **GetMovies** function returns the list of movies in the given **year** ordered by their titles. To get the list of identifiers it has to call the movie service using `GET /movies/find?year={year}&orderby={field}`. Then it has to call the movie service using `GET /movies/{id}` to get the titles of the movies. The results combined from these calls must be returned by the **GetMovies** function.

The **BuyTickets** function buys **count** number of tickets for the movie identified by **movieId**. The price of the tickets is **10*count**, i.e. each ticket costs 10 units of money. The **cardNumber** has to be forwarded to the **ChargeCard** function unchanged. The return value of the **BuyTickets** function is the same as the return value of the **ChargeCard** function.

The **Ticketing** service must store the number of sold tickets for each movie. Whenever a **BuyTickets** call is successful (the **ChargeCard** function returns true), the **Ticketing** service must add the number of sold tickets to the ticket counter of the movie.

The **GetTickets** function returns the movie ticket sales data. Exactly those movies must be in the returned list for which there has been at least one ticket sold. Each item (**Ticket** object) of the list contains the identifier of the movie (**movieId**), and the number of sold tickets for the movie (**count**).

The service has to be implemented as a REST RPC service. The format of the input and output messages are described by the attached **ticketing/ticketing.proto** file, which should be copied into the **src/main/resources** folder. Since the **Ticketing** service uses the **Movies** and **Banking** services, the proto files of those services should also be copied into the **src/main/resources** folder.

The functions of the **Ticketing** service can be called by the following REST calls added to the base URL above:

- **`POST /ticketing/GetMovies`**
  - input HTTP body: **`GetMoviesRequest`**
  - output HTTP body: **`GetMoviesResponse`**
- **`POST /ticketing/BuyTickets`**
  - input HTTP body: **`BuyTicketsRequest`**
  - output HTTP body: **`BuyTicketsResponse`**
- **`POST /ticketing/GetTickets`**
  - input HTTP body: **`GetTicketsRequest`**
  - output HTTP body: **`GetTicketsResponse`**

Important: it is forbidden to hardwire the URLs of the movie and banking services into the ticketing project! Instead, the URLs of the required services must be configured as in the attached **ticketing/project-defaults.yml** configuration file, which should be placed into the **src/main/resources** as in the tutorial. The configuration part for the two required services looks like this:

```
microservices:
    movies:
        url: http://localhost:8081
    banking:
        url: http://localhost:8082
```

5 Message formats

Every service must support JSON and Protocol Buffers at the same time! The MIME types for these formats are the following:

- JSON: **application/json**
- Protocol Buffers: **application/x-protobuf**

The **protobuf-java-util** library can be used to serialize proto messages to JSON. The attached **pom.xml** files already contain this library as a dependency.

(The JSON messages should not be the same as in the REST homework (2nd homework). Instead, they should be the JSON serialized versions of the proto messages.)

6 The client

No client is required for this homework. However, creating a client for testing is strongly recommended, but do not upload this client.

7 To be uploaded

A single ZIP file has to be uploaded. Other archive formats must not be used!

The root of the ZIP file must contain three folders:

- **Microservices_NEPTUN_Movies:** the Maven application of the movie project
- **Microservices_NEPTUN_Banking:** the Maven application of the banking project
- **Microservices_NEPTUN_Ticketing:** the Maven application of the ticketing project

The compiled class files (target folder) may be omitted from the ZIP file.

Each application must compile by issuing the following command from the project folder:

```
mvn clean
mvn -P=generate-sources generate-sources
mvn package
```

Each application must run by issuing the following command from the project folder:

```
mvn thorntail:run
```

The services must support the exact proto message formats attached to this assignment. Changing the proto files is forbidden.

The solution must compile and run in the given environment using the given pom.xml files. These pom.xml files must not be modified apart from replacing the word NEPTUN with your own Neptun code.

Important: the instructions and naming constraints described in this document must be precisely followed!

Again: use your own Neptun code with capital letters instead of the word "NEPTUN".