3. WebSocket homework

Dr. Balázs Simon (sbalazs@iit.bme.hu), BME IIT, 2017

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

1 The task

The task is to create and publish a WebSocket service for cinema seat reservations.

2 The service

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

The name of the web application should be: **WebSocket_NEPTUN**

The service should listen on the following URL:

`ws://localhost:8080/WebSocket_NEPTUN/cinema`

The service should receive and return messages in JSON format. The rows and columns are positive integer numbers.

The service can receive the following messages from a client:

Operation	Sample message	Description
Initializing the room	`{ "type": "initRoom", "rows": 10, "columns": 20 }`	Initializes the room with the given **rows** and **columns**. The valid rows are numbered from 1 to **rows**, the valid columns are numbered from 1 **columns**. If the value of **rows** or **columns** is not a positive integer, then an error should be sent back (see the **error** message on the client side).
Getting the size of the room	`{"type":"getRoomSize"}`	Returns the size of the room (see the **roomSize** on the client side).
Getting the status of the seats	`{"type":"updateSeats"}`	Returns the status of all seats one by one (see the **seatStatus** message on the client side).

Lock a seat	`{`
	`    "type": "lockSeat",`
	`    "row": 3,`
	`    "column": 11`
	`}`
		Locks the seat at the given **row** and **column**. Returns the identifier of the lock (see the **lockResult** message on the client side), and also the new status of the seat (see the **seatStatus** message on the client side). The status of the new seat must be sent to all open Sessions so that all the clients can see the new status.
		
		If the given seat is not free (locked or reserved), or the row or column is invalid, an error must be returned (see the **error** message on the client side).

Unlock a seat	`{`
	`    "type": "unlockSeat",`
	`    "lockId": "lock5"`
	`}`
		Releases the lock with the given **lockId**. Returns the status of the seat made free (see the **seatStatus** message on the client side). The status must be sent on all open Sessions so that all the clients can see the new status.
		
		If the **lockId** is invalid, an error is returned (see the **error** message on the client side).

Reserve a locked seat	`{`
	`    "type": "reserveSeat",`
	`    "lockId": "lock7"`
	`}`
		Reserves the locked seat with the given **lockId**. Returns the status of the reserved seat (see the **seatStatus** message on the client side). The status must be sent on all open Sessions so that all the clients can see the new status.
		
		If the **lockId** is invalid, an error is returned (see the **error** message on the client side).

The service must store data between calls. In a real application, data should be stored in a database. In the homework, the storage should be static variables. For example, the open Sessions should also be stored in a static list.

3 The client

The client should be a JavaScript application running in a browser. The HTML should be under the **webapp** folder inside the same web application as the service. The initial HTML code called **cinema.html** is attached to this document.

After the application is deployed on the server, the client should be accessible from the following URL:

`http://localhost:8080/WebSocket_NEPTUN/cinema.html`

The client can receive the following messages from the service:

Operation	Sample message	Description

Size of the room	`{`
	`    "type": "roomSize",`
	`    "rows": 10,`
	`    "columns": 20`
	`}`
		Arrives as a response for the **getRoomSize** operation. It gives the current size (**rows**, **columns**) of the room stored on the server side.
		
		After the arrival of this message the client must retrieve the status of all seats by sending the **updateSeats** message to the service. The seats must be drawn to the canvas (see the **seatStatus** message).

Status of a seat	`{`
	`    "type": "seatStatus",`
	`    "row": 4,`
	`    "column": 5,`
	`    "status": "free"`
	`}`
		An update message coming from the service. It gives the **row** and **column** of the seat along with its status.
		The status can have the following values:
		• **free**
		• **locked**
		• **reserved**
		
		Whenever the client receives such a message, it must redraw this seat on the canvas with the updated status.

Result of a lock	`{`
	`    "type": "lockResult",`
	`    "lockId": "lock45"`
	`}`
		Arrives as a result of a lock. Gives the identifier of the lock (**lockId**). This identifier may be used later for unlocking or reserving the seat.

Error	`{`
	`    "type": "error",`
	`    "message": "Seat is not`
	`free."`
	`}`
		An error message returned from the server. This **message** should be displayed as an alert in JavaScript.

4 Hints

A custom **Encoder/Decoder** should be written for JSON serialization on the server side. The serialization can be solved by JAXB or by the classes in **javax.json** package (the latter is recommended).

The messages can be told from each other by the **type** field of the message. This field appears in all messages.

Some messages generate more than one messages to be sent back. This cannot be solved by just simply returning a result. In this case a parameter of type **Session** should be added to the method with the **@OnMessage** annotation. The object returned by the **getBasicRemote()** method of the **Session** can be used to send multiple messages on this session. But it may be possible that the custom **Encoder** will not be used in this case automatically, so it may have to be called manually.

The HTML client provides only a skeleton for the solution. The event handler functions in JavaScript must be registered for the buttons. Look for the **TODO** word, which indicates the places in the code, where it should be changed. The HTML already contains the functions required for drawing the seats. These can be reused.

Some APIs were not introduced in the lecture (e.g. **javax.json** package, JavaScript code). Collecting the missing knowledge is also a challenge in this task.

5 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 a single folder:

- **WebSocket_NEPTUN:** the Maven application of the service, which also contains the client code in the cinema.html

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

The service application must compile from the command line using the following command:

```
...\WebService_NEPTUN>mvn package
```

The service application must deploy to the server from the command line using the following command:

```
...\WebService_NEPTUN>mvn wildfly:deploy
```

The service must support the exact JSON message formats described in the examples above.

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”.