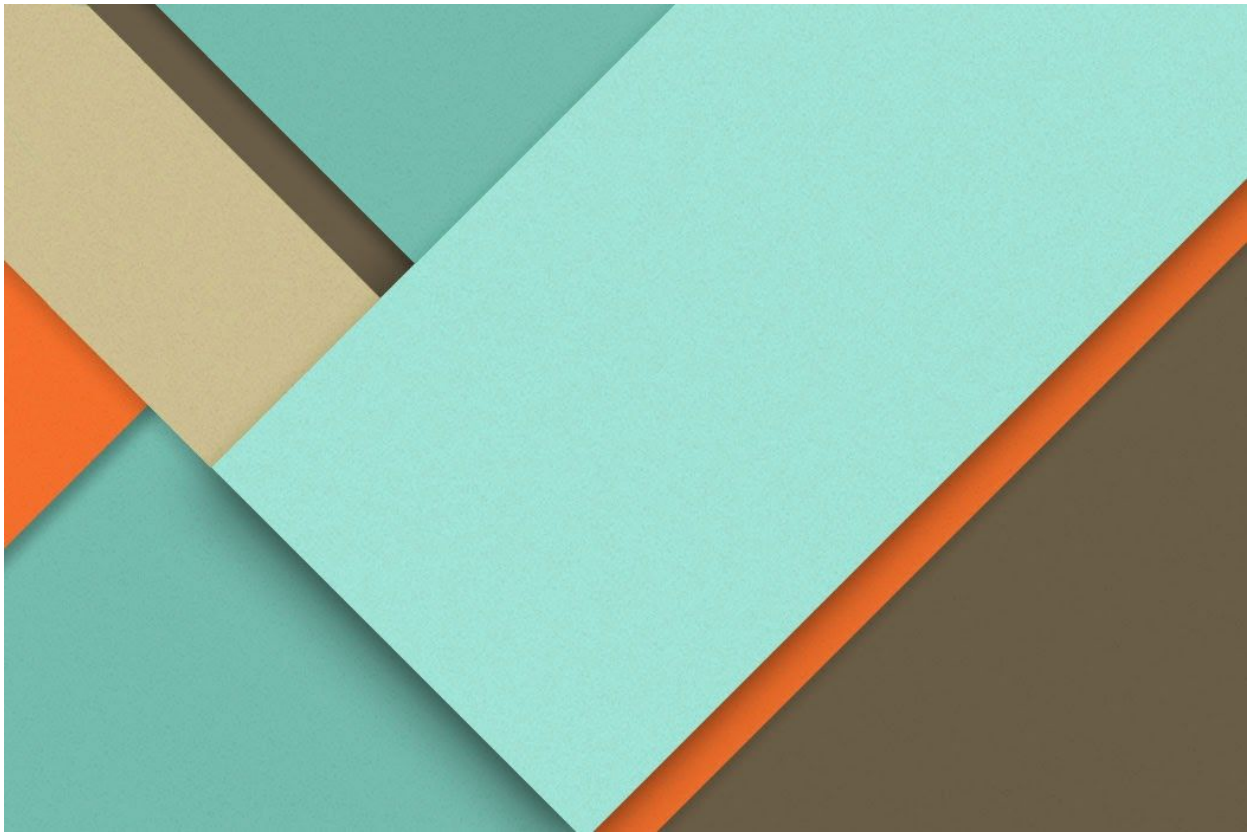

SPRINT 3

28.09.2020

Laura Mazzuca, Nicholas Antonio Carroll, Giuseppe Giorgio

Requirements analysis

Assumptions

- we **will** consider the presence of **multiple clients**;
- The client, the barman and the smartbell will be simulated and grouped together with the waiter context-wise;
- we **will not** consider the existence of the manager and his interface, but we will build the Architecture on which it will run;
- we **will not** take into account optimization of tasks;
- The waiter **will not** be able to interrupt a task;
- We **won't** take into account the MaxStayTime and MaxWaitTime timeouts;
- The Waiterwalker **will** be divided into Waiterwalker and Walker;
- The Client **will** be able to interact with the Waiter through an Interface.

Problem Analysis

System Architecture

The system will be enriched with a Server to handle communication between the tearoom Context and the User Interface.

From the very beginning, we have been applying an architecture based on Microservices. This has led us to think of the **Server as a Model-View-Controller** to implement the Web Application, **where the Model** won't be directly "built-in" in the Server, but **will be represented by the Tearoom Context**. To communicate with it, the Server will use the messages interface we have defined in the previous SPRINT and will then communicate to the Web UI what updates to make.

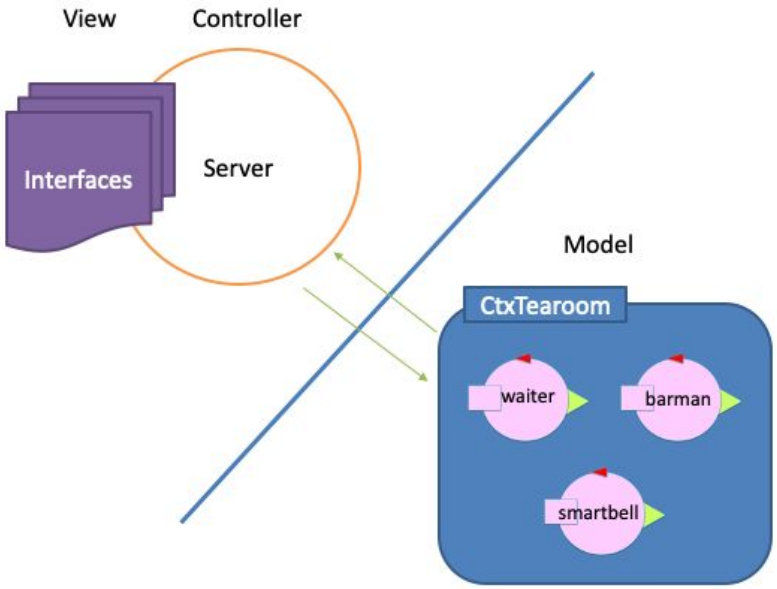

Fig. 1: our MVC where the model is the CtxTearoom

Our web application is composed by the following elements:

- **Client UI**: The client view shows data to the user and lets him interact with the system through the controller;
- **Manager UI**: Similar to the client UI but it's behavior is passive as it only shows the state of the tearoom to the manager;

- **Controller**: Deals with updating the views as the internal state of the tearoom changes. it also receives commands from the client, processes them and sends them to the server;
- **TeaRoom**: the model, it represents the internal state of the server. This level defines where the application's data objects are stored.

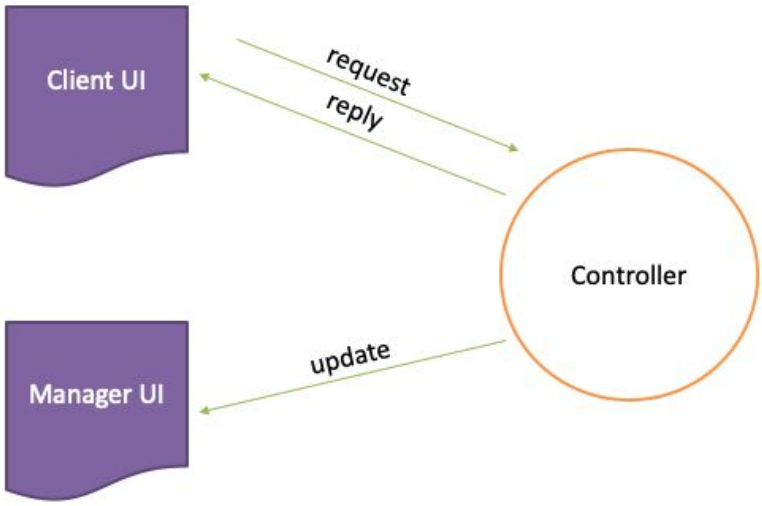

Fig. 2: the exchange that will need to happen between the controller and the Web Interfaces

Test plans

The test plan of the previous sprints remains valid thanks to the MVC framework, which lets us test the model independently.

In addition to that, we want to test **how the Waiter currently handles multiple clients** showing up at the tearoom.

Project

CoAP as the connection support for our Microservices

Since our system is composed of many different Entities working on different nodes scattered around the tearoom and we need to be able to check on and communicate with them, we need a way to both be able to broadcast an update message from the Actors and connect for an exchange from the Server side (or a generic *alien*), while also keeping the impact on the system itself as low as possible.

The Constrained Application Protocol comes to our aid, since it's built exactly for this kind of communication. We will use the **Californium library**, which implements the CoAProtocol for Java, and the adaptation library **connQak**, already developed by our Software House.

Changes to `connQakCoap.kt`

To properly work in a request-reply message exchange, though, we had to make a small change in the request function:

```
override fun request(msg: ApplMessage): ApplMessage {
        val respPut = client.put(msg.toString(), MediaTypeRegistry.TEXT_PLAIN)
        println("connQakCoap | answer= ${respPut.getResponseText()}")
        return ApplMessage(respPut.getResponseText())
}
```

Now, **the request function returns the replied message** to the class that called it, this way it's possible to read and elaborate the data contained in it.

`connQak.utils.ApplMessageUtils.java`

To let the class that called the request extract the payload data, we implemented this static class that returns an Array containing, at the corresponding index in the message the payload found.

Spring Framework to Develop the Server

To develop our web application we use Spring Boot as it offers native methods to work on an MVC application, greatly simplifying the job. It also provides automatic configuration for many common functionalities.

Model-Controller Exchange

The Controller is implemented as a standard `@Controller` and it exchanges messages with the CtxTearoom Model through a **CoAP** connection. The connection uses the `connQakCoap` library, this way we can specify the information about the Actor we want to connect to and it will be the library to handle the QAktor-specific url path construction. As a

plus, we also get that the interface of the messages is the same and we can receive and analyze messages directly as `ApplMessage` types with the `utils.ApplMessageUtils` static class.

Client-Server Interaction

The Controller handles the communication with the Client through **Websocket [RFC 6455]**, which is a lightweight layer above TCP. Then we apply the **STOMPProtocol**, a simple text-based sub-protocol layered on top of Websocket, to enable the exchange of Json messages, which Server side are mapped as POJOs. In particular, we have the classes:

- **`ClientRequest`**, which contains a maximum of 3 payload values that represent the information sent from the Client interface;
- **`ServerReply`**, which contains the answer from the Server, with a redir value, that contains the eventual redir path, and a maximum number of 5 payload values.

As described in the Spring documentation, their approach to working with STOMP messages is to route them to `@Controller` classes. So, we introduced the **mappings**:

- `/smartbell`: for messages directed to the Smartbell Actor;
- `/waiter`: for messages directed to the Waiter Actor.

These mappings are used to receive messages, which will be bound to a `ClientRequest` class.

To answer the request, we send a message back by using the mapping `/topic/display`, which for now we'll keep as a broadcast, and we send back a `ServerReply` with whatever payload can answer the Client's request.

To register these mapping and the stomp endpoint `/it-unibo-iss` we use the class `WebSocketConfig.java`, which extends `WebSocketMessageBrokerConfigurer.java`, and acts as a Configurator (`@Configuration` annotation). The endpoint we register is also registered to fallback on **SockJS** options, this way we ensure the possibility for the connection to work also in browsers that do not support Websocket.

Client-Controller Interaction in-depth

More in particular, the interaction between Client and Controller are:

- **`/smartbell`**: when the Controller receives this message he sends a request to the smartbell actor when he receives the reply he redirects the user to either badTemp or teaRoom depending on the temperature measured by it.
    - `/badtemp`: if the client's temperature is above 37.5˚ Celsius so he is denied entrance and sent an html error page in the response.

    - `/tearoom`: if the client's temperature is below 37.5° Celsius and if there are free tables he is let in and the controller sends the teaRoom html page back to the client.

- **`/waiter`**: the user calls the Waiter to be deployed to the table or the exitdoor, to order tea or to pay the bill. Depending on the Request ID there will be different behaviors. In each of these the Controller communicates with the waiter through the CoAP connection and then communicates the answer to the user through the usual ServerReply. If the request is accomplished correctly, the user will receive a success message and his webpage view will be updated accordingly. The request ids are:
    - *`waitTime`*: waitTime is **automatically asked** and communicated to the client during the smartbell interaction;
    - `deployEntrance`, `deployExit`: the waiter is asked to escort the client to and from the table;
    - `serviceOrder`, `servicePayment`: The waiter receives a request to deal with an order or a payment.
    - `order`: the order is forwarded to the waiter.
    - `pay`: the payment is forwarded to the waiter.

WebApp User Interface

The Client Interface has been developed in html-css-javascript using `Bootstrap` and `Ajax`. The connection is handled with:

- a **SocketJS**, that connects to the Server's URL `'/it-unibo-iss'`;
- A **STOMP Client** that connects through the SocketJS and, once connected, subscribes to the `'topic/display'` mapping to receive the messages.

When the Client wants to send a message, she uses the connection to send a Json formatted message, which has to be built with attributes with the same name as those of ClientRequest, and send them to the mappings mentioned above, depending on what Actor they need to contact.

Test

The test class for Multi-clients can be found at `tearoom.SPRINT3/test/TestMultiClient.kt`. To test its behavior we check that the value `resourceRep` in the object Waiter and the object Smartbell is correct with the input sent, again directly to the object.

Backlog Update

- We need to implement tests that work by exchanging messages instead of working by directly accessing the object we want to check.
- Waiter hits obstacles and crashes.