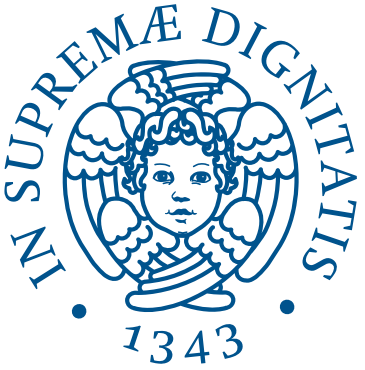

UNIVERSITÀ DI PISA

Computer Engineering

Distributed Systems and Middleware Technologies

Project Documentation

TEAM MEMBERS:
Tommaso Burlon
Francesco Iemma
Olgerti Xhanej

Academic Year: 2021/2022

Contents

1 — Project Specifications

AuctionHandler is a distributed web-app in which users can sell their goods by creating Online Auctions. Registered users have the possibility to join an ongoing auction in order to buy a good in case they beat the concurrence by setting an higher offer on a given limited time.

1.1 Use Cases

An *Unregistered User* can:

- Register to the service

A *Unlogged User* can:

- Login to the service

A *Logged User* can:

- View the list of ongoing Auctions
- Create a new Auction
- Join an ongoing Auction
- Logout
- After Joining an Auction:
  - Make an offer
  - View list of participants
  - View past offer history
  - View the remaining time
  - Wait until the end of the Auction and then exit
  - View Auction result

The *System* must:

- Remember registered users
- Remember ongoing auctions
- Remember auction participants
- Choose in a unique way the auction winner
- Remember offers history
- Synchronize the remaining time, the auction participants, the offer history for an auction for each user
- Synchronize the list of ongoing auctions for each user

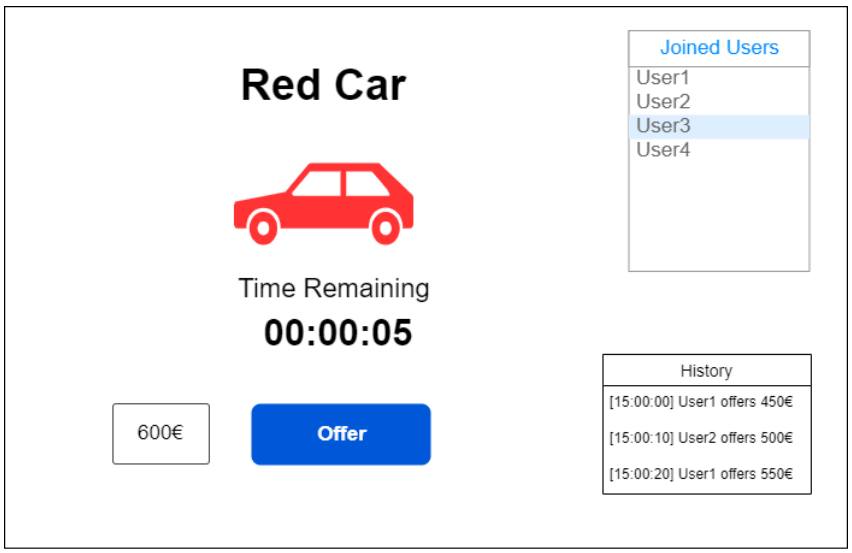

Figure 1: Mock-up of the main interface of the auction

1.2 Synchronization and Communication Issues

On the application we have the following synchronization and communication issues:

- Client nodes need to be synchronized with the same remaining time of the auction, the same offer history for a given auction, the same list of joined users on a given auction and the same list of available ongoing auctions.
- In case a client makes a valid offer, the server will be in charge of communicating to other clients nodes the information regarding the made offer.
- In case a client creates a new auction, the server will be in charge of communicating to other clients the information regarding the newly created auction.

1.3 Design Ideas

We were thinking of implementing the system in the following way:

- **Client Nodes**: User Interface via HTML/CSS, generated via Java Servlets and JSP. Each client node will have a dedicated Erlang node for communicating with the server
- **Server**: Made entirely in Erlang. The responsibility of the server will be performing persistent data storage and handling the communication and synchronization between different client nodes

2 — System Architecture

A graphical representation of the system architecture can be seen in figure 2.

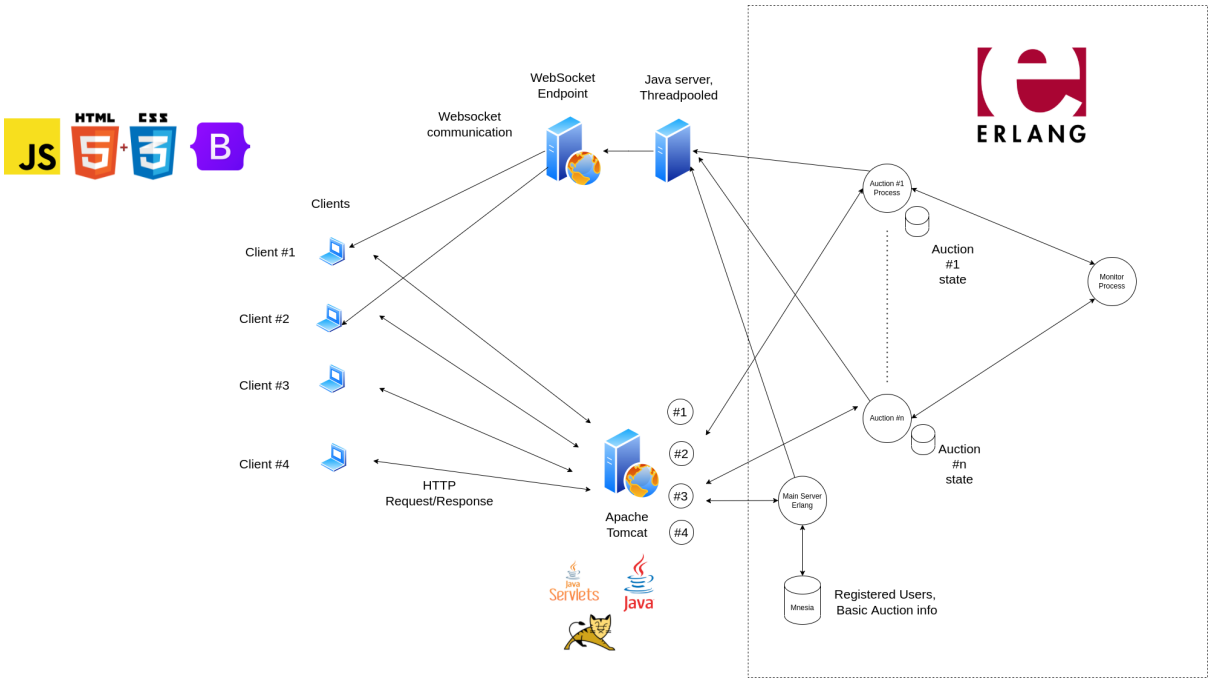

Figure 2: System Architecture graphical representation

We can divide the overall system in two part:

- The server side part: developed in Erlang, it is in charge of handling the request coming from the users and it has to handle the auctions and to maintain the global view of an auctions consistent among the users.
- The client side part: depeloved in Java, using EJB e JSP, and in Javascript for what regards the websocket. It is in charge of retrieving information from the server, create the GUI, update the GUI in order to let the user have a consistent view of the state of the auction and of the overall system.

2.1 Server Side

2.1.1 Main Server

2.1.2 Auction Handler

2.1.3 Monitor And Supervisor

2.2 MnesiaDB

2.3 Client Side

The web application shown to the user is generated via Apache Tomcat web-server, the page update is managed asynchronously via web-sockets, the front-end part is done with Bootstrap (a pretty famous CSS & Javascript framework) and Javascript "vanilla".

2.4 Web-server with Apache Tomcat

In Apache Tomcat web-server HTTP requests arriving from the user are processed by Java Servlets:

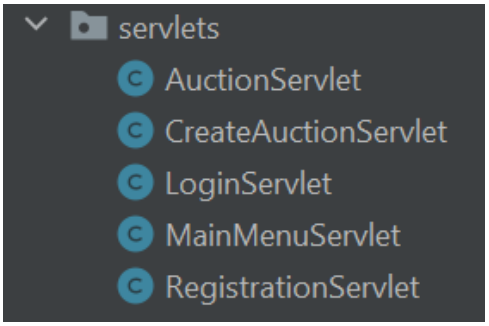

Figure 3

As shown in the above figure there is one Java Servlet for each page of the web-application. From the generated page is possible to follow some hyperlinks to find other pages of the web-app.
The Webapp folder is organized in the following way:

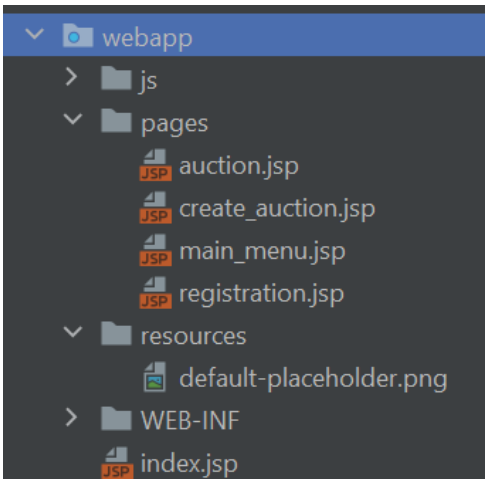

Figure 4

The actual HTML is generated via JSP technology, with some script-lets, for each page, in order to have a logic separation between the "View" part and the rest of the application. In example the *main_menu.jsp* scriptlet contains some logic to generate different cards by iterating the list of auction returned by the Erlang server.

In order to add some layers of security to the application a couple of Filters were added:

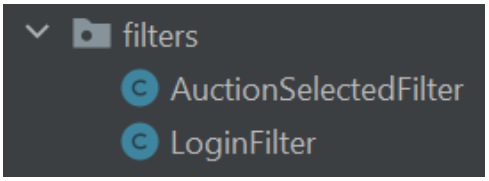

Figure 5

- **LoginFilter** is responsible for checking that the request to some "protected" pages arrives with users already logged in (this can be assured by analyzing the session of the particular user, indexed by a cookie).
- **AuctionSelectedFilter** prevents a logged user to access an auction without having pressed the related button. This is done because at the same time multiple auctions can be running at the same time.

In order to update the "Model" part of our application some requests to the Server-Side part in Erlang needs to be done. This is dealt in the communication package:

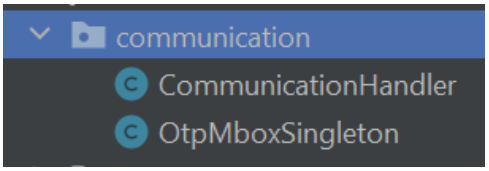

Figure 6

With the usage of Jinterface library: for each client is adopted a different mailbox of one single "Erlang" node, obtained from the cookie of the client. Then a request/reply communication happens with the Erlang Main Server and if needed with the related Auction Handler, in case an auction page is requested. The communication will return a result to communicate if the request of the client has succeeded or not (i.e. a client registration to the service may fail in case of a duplicate username).

2.5 Web-sockets

In order to update the application state without requesting periodically a web-page, Web-socket technology is used.

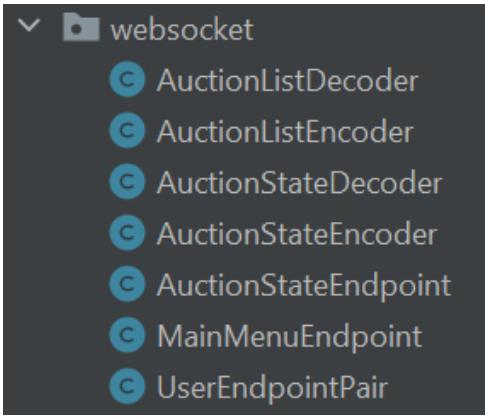

Figure 7

As we can see in the image above two different web-socket endpoint are used:

- **MainMenuEndpoint**: responsible for updating the state of the main menu w.r.t. the list of active and inactive auctions. Clients connected to this endpoint are registered through the usage of a Java static *CopyOnWriteArraySet* for dealing with concurrent changes of the connected client list.
- **AuctionEndpoint**: responsible for updating the state of the auction w.r.t. the list of joined users, the list of valid bids, the auction time, and the winner election. Clients connected to this endpoint are registered through the usage of a Java static ConcurrentMap¡String, Set¡UserEndpointPair¿¿ data structure. With the latter, for each auction name is associated a list of client endpoints, in order to correctly send messages only to clients of a particular auction, with a publish/subscribe fashion.

2.6 Java Listener

There's one actor missing in the whole picture, the Java Listener. The web-socket state update happens withe the following sequence of steps:

1. A client requests a change to the state of the model (i.e. creates an auction)
2. Apache Tomcat intercepts the requests and start the communication with the related Erlang server-side which actually stores the application state
3. The Erlang server-side sends in general **two messages for each requests**: one is fore the servlet itself for giving an ACK to the made request, one is sent to the **java listener** for triggering the state change to other clients connected (without need of a page refresh)
4. Java listener broadcast the state change to all interested users via web-socket

The advantages of using this approach are the following:

- Since messages are sent by the erlang server-side (which actually is the source of truth of the web-application state), the state returned is consistent.

- Thanks to web-sockets, there's no need to periodically sending request to Apache Tomcat. **Page updates happens only when there is a state change**, avoiding a considerable amount computation.

- Some events are triggered by erlang side itself and not by the client (i.e. auction ending and winner election), so this approach update the web-application state correctly by simply avoiding to send one message to Apache Tomcat

The **cost** of this approach is the implementation of the Java Listener.

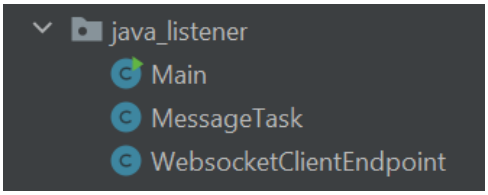

Figure 8

The Java Listener is implemented with a listener Jinterface node for getting requests from Erlang server-side. For handling multiple requests in parallel, threads are used: in particular, for avoiding the overhead due to the creation and termination of threads, **thread-pooling** is used via ExecutorService interface that implements a fixed thread pool. The task (called MessageTask) computed by each thread is related with sending a message (with the state information) to the correct web-socket endpoint.

2.7 User Interface

In order to ease the UI developing a framework like **Bootstrap** has been used, in order to create a responsive web-application in a simple way, with a set of ready-to-be-used css classes, without the need of reinventing the wheel by defining our set of css classes for getting the same result.

A couple of javascript scripts are then defined:

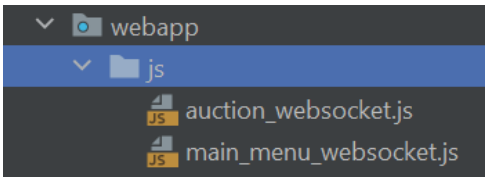

Figure 9

- **main_menu_websocket**: Responsible for connecting with the main menu web-socket endpoint and to update the DOM of the HTML page when a message arrives.

- **auction_websocket**: Responsible for connecting the client to the auction web-socket endpoint, to update the DOM of the HTML page when a message arrives and to update locally the timer.

For what concern form validation, **browser-default validation** was adopted (i.e. some fields are required to be added, some fields, like the auction duration, must contain a number).

2.8 Synchronizations Issues