UNIVERSITÀ DI PISA

Master Degree in Computer Engineering

Distributed Systems and Middleware Technologies
Project Report

Roice Web App

Team Members:

Francesco Martoccia
Luca Tartaglia
Salvatore Lombardi

Academic Year 2022/2023

Contents

1 - Introduction

1.1 Use Cases

The developed web application is designed for cell phone auctions. Users can create accounts, browse a wide selection of available phones, and participate in auctions to bid on and secure their desired phones at the best possible prices.

An **Unregistered user** can:

- Register to the service

A **Registered user** can:

- Login to the service

A **Logged user** can:

- View phone list
- Browse phones
- View phone details
- View live auctions
- View auction details (remaining time, current bid, current winner)
- Bid on a phone
- Logout

The **Administrator** can:

- Add a new cell phone
- Create a new auction

1.2 Synchronization and Communication

The application guarantees the following aspects regarding synchronization and communication:

- **Bid synchronization:** All users must see the same latest offer, current winner and remaining time.
- **Live auctions:** Each user must see an up-to-date view of all current live auctions, which is synchronized across all users. This view updates in real-time whenever a new auction is added or an existing auction ends.

1.3 Non-functional requirements

The application is designed with a distributed architecture, which offers several advantages:

- **Availability:** The distributed design ensures that the application remains available and responsive even under high load conditions.
- **Load Balancing:** By distributing the load across multiple nodes, the application can efficiently manage user requests and maintain optimal performance.
- **Fault Tolerance:** The system is capable of tolerating failures of auction processes on each node, ensuring continuous operation without service disruption.

2 - Application Structure

2.1 Code Organization

The application's codebase is organized into two main components, developed using Java and Erlang languages respectively.

2.1.1 Java Application

The Java portion of the application is a **Spring Boot Web Application**, which follows the **Model-View-Controller** (**MVC**) pattern. It defines the web pages using **JavaServer Pages** (**JSP**) and handles operations on the **Apache Tomcat** server. A detailed explanation of the Java application can be found in Chapter 3.

2.1.2 Erlang Applications

The Erlang portion of the application is designed to satisfy requirements in terms of **synchronization** and **communication**, discussed in Chapter 1.
This part consists of two Erlang applications, both described in Chapter 4:

- **ERWS (ERlang WebSocket) Application:** This application starts a Cowboy WebSocket server, which listens for requests from clients and sends updates to them.
- **Master Application:** Responsible for initializing the communication between the Erlang nodes and setting up the Mnesia DB on all of them. It has to be executed only after the ERWS application has been launched on the nodes we want to use as servers.

2.2 System Architecture and Components

As described previously, the application has been developed and deployed according to a distributed paradigm. The architecture is shown in Figure 2.1.

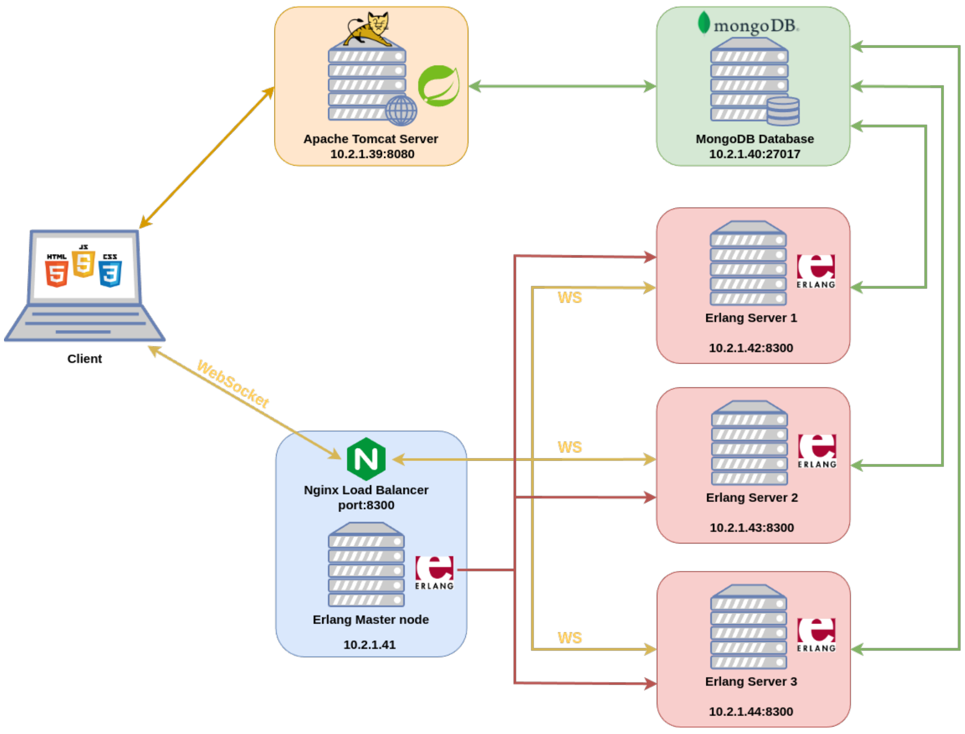


Figure 2.1: System Architecture

2.2.1 Interactions between Components

The client requests a web page from the **Tomcat server** using HTTP. The server processes these requests and returns the HTML content to the client, utilizing **JavaServer Pages** (**JSP**) to dynamically generate the HTML based on the server-side logic.
Whenever necessary, the Tomcat server contacts the **MongoDB database** to access or save the required information.
To handle communication with the **Erlang** part of the application, which is responsible for synchronization and communication as described in Section 1.2, the **JavaScript** code executed by the webpage opens a **WebSocket** connection on **port 8300**.
This request is received by the **Nginx load balancer**, which acts as a proxy and redirects the request to one of the available **Erlang servers**.
When the **Master Node Application** is started, it communicates with the other Erlang server nodes, in order to start Mnesia on them.
When an auction ends, the **Erlang server node** that managed the auction process sends an HTTP request to **MongoDB**, in order to ensure the **deletion** of the information that are not needed anymore.

2.2.2 Load Balancer Configuration

The **Nginx configuration** used is as follows (located at `/etc/nginx/nginx.conf`):

```
worker_processes 1;
worker_rlimit_nofile 10000;

events {
    worker_connections 1024;
    # multi_accept on;
}

http {
    map $http_upgrade $connection_upgrade {
        default upgrade;
        '' close;
    }

    upstream websocket {
        ip_hash;
        server 10.2.1.42:8300;
        server 10.2.1.43:8300;
        server 10.2.1.44:8300;
    }

    server {
        listen 8300;
        location / {
            proxy_pass http://websocket;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection $connection_upgrade;
            proxy_set_header Host $host;
        }
    }
}
```

The ***events*** block configures Nginx to handle up to 1024 simultaneous connections.
In the ***http*** block, the ***map*** directive ensures that WebSocket upgrade requests are properly handled.
The ***upstream*** block named ***websocket*** defines a pool of Erlang servers using the ***ip_hash*** method to distribute connections based on the client's IP address, which helps in maintaining session persistence.
Finally, the ***server*** block allows to specify that Nginx listens on port 8300 for WebSocket connections and proxies these requests to the upstream WebSocket servers, ensuring that the necessary headers for WebSocket communication are set correctly.
This setup ensures efficient load balancing and fault tolerance, distributing client connections across multiple Erlang servers while maintaining seamless communication and synchronization.

3 - Java Application

It was decided to develop the Java application using **Spring Boot** technology, which facilitates the creation of much more flexible and dynamic stand-alone web applications and microservices. In addition, Spring Boot offers the ability to automatically integrate the **Tomcat Web Server** without the need for additional installations and a more efficient deployment.

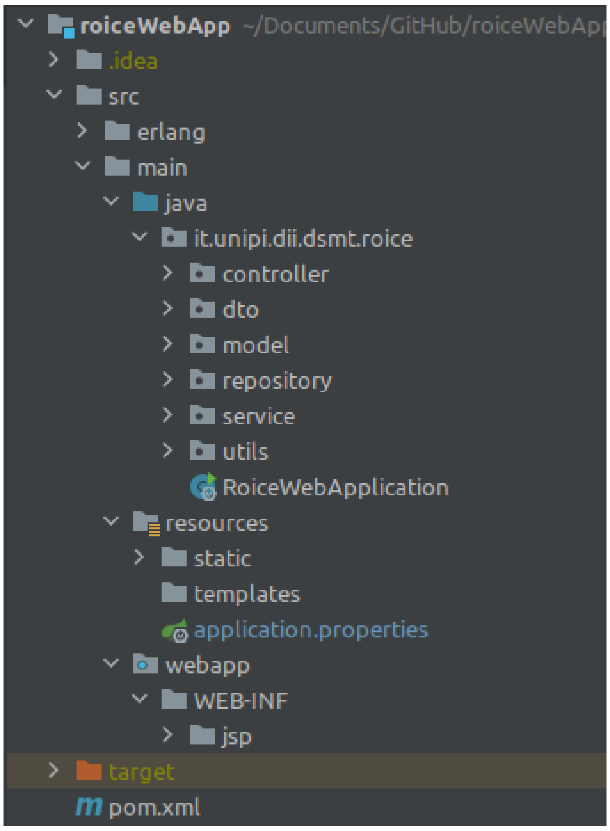
The application follows the **Model-View-Controller (MVC)** paradigm for component and service management, utilizing **Spring Data** for database interaction and management. Regarding data storage, we opted for the use of a non-relational database to store information about users (categorized into simple users and administrators) and cell phones. Each cell phone document includes details about auctions for that specific model, such as start and end dates, and the starting price.

Additionally, The Java Application employs **WebSocket APIs** for communication with the Erlang application. These Web Sockets are specifically used for managing connections between the Java client and the Erlang server, as well as handling all functionalities related to auctions, such as participating in auctions, placing bids, and receiving real-time updates.

The **structure of the Java project** is outlined below, detailing the components of the MVC design pattern, available WebSocket functionalities, and the Database used for storing information about users and cell phones.

3.1 Java Application Structure (MVC)

As mentioned above, the web application follows the MVC design pattern, so the division of classes and files is organized according to this paradigm.



```
roiceWebApp ~/Documents/GitHub/roiceWebAp
  .idea
  src
    erlang
    main
      java
        it.unipi.dii.dsmt.roice
          controller
          dto
          model
          repository
          service
          utils
          RoiceWebApplication
      resources
        static
        templates
        application.properties
      webapp
        WEB-INF
          jsp
  target
  pom.xml
```

Figure 3.1: Java Application Structure

the project contains the following components within the package ***"it.unipi.dii.dsmt.roice"***:

- **controller**: Contains all classes responsible for the web application business logic operations, managing all the client requests and mapping URLs to its methods
- **dto**: In the Data Transfer Object (DTO) package, are defined the object structures used to encapsulate the *User*, *Admin*, and *Phone* models for data transfer. Indeed, are also defined mapper classes for conversion from the normal model into the DTO object and vice versa (e.g User to UserDTO)
- **model**: Are defined the structure of the models and the main functions of interaction with them, used in service and controller classes
- **repository**: Contains the interfaces used for the communication with MongoDB (extends *MongoRepository*). Are defined the queries needed to retrieve and store information about users and cell phones.
- **service**: It contains the ***"PhoneService"*** and ***"UserService"*** classes. They take care of all the operations needed by the controllers on both *Users* and *Phones* repositories. The functions are as follows:
  - **PhoneService functions**
    - *getPhones*
    - *searchPhonesByname*
    - *getPhonesWithLiveAuctions*

    * *addPhone*
    * *removeAuctionByName*
  - **UserService functions**
    * *registerUser*
    * *addUser*
    * *findByEmail*
    * *addPhonePreview*
    * *deleteFavoritePhone*
    * *addWonAuction*

- **utils**: Contains a utility class, in particular the class used for the security of the web app, which contains the methods to hash the passwords and to generate the salt added to the latter (*getHashedPassword* and *getSalt)* in order to securely save user and administrator passwords to the DB.

In addition to the java code, inside the ***"resources"*** and ***"webapp"*** packages,there are all the **jsp files** (which include the view part of the model) and the **static files** needed by the web app (css, js, images etc.)

3.1.1 Model Classes

The classes defined in this component are responsible for representing the objects used in the web application and manipulating their data.

Specifically, classes are defined to represent the **User** objects (both simple and administrators), **Phones**, and **Auction** and **AuctionWon** (present in the User and Phone documents, respectively).

The classes are as follows:

- **Generic User**: This is the superclass used for the two types of users in the web app (User and Admin). It defines the common attributes shared by both user types.
- **User**: As a subclass of the GenericUser class, it defines all the personal information of a user. Additionally, it contains an array list of all auctions won by that specific user.
- **Admin**: This is the other subclass of the GenericUser class. It defines the information specific to an admin, which includes fewer attributes compared to a user since admins cannot actively participate in auctions.
- **Phone**: This model represents objects for cell phones. In addition to information on the phone's technical features, they also contain an Auction object, which specifies the start and end dates of the auction for that mobile phone model and its starting price.
- **PhonePreview**: This model is only used to show the main information about a phone.

- **Auction**: As mentioned in Phone, this model is used to be inserted inside phone type objects, specifying the auction start and end date and the minimum price.
- **AuctionWon**: Similar to the Auction model, this model is used to enter information about auctions won within a User object. This model specifically includes the name of the phone won, the date when the auction concluded, and the winning bid amount.

3.1.2 View Templates

View classes are responsible for displaying model data to the user and relaying user input to the controller classes.
Specifically, the view component includes JSP pages (Jakarta Server Pages), where dynamic logic is implemented using JSTL tags (Jakarta Standard Tag Library) and JavaScript files.

Each JSP is associated with its respective controller and represents a specific page of the web application, each with its own unique features. Below are the JSP pages developed, along with the features available to both users and administrators.

login.jsp

On this page, a user can log in as an admin or a simple user. Depending on their privileges, they can perform different operations. This page also allows the user to navigate to the "Signup" page, where an unregistered can perform the sign up operation.

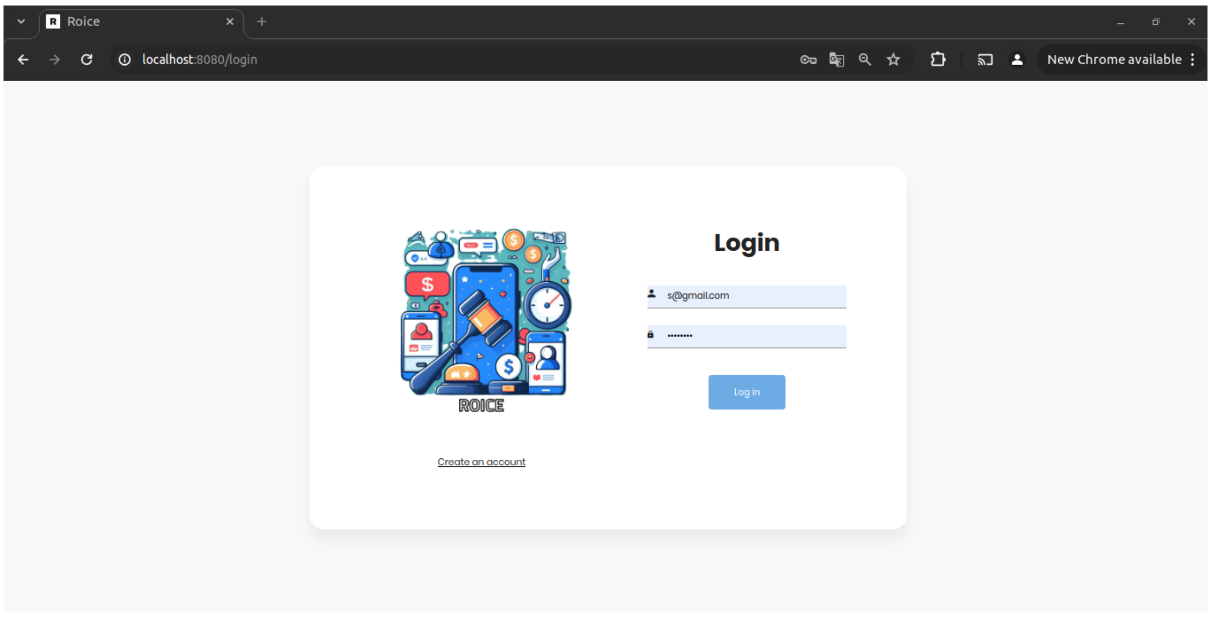


Figure 3.2: login.jsp Page

signUp.jsp

On this page, a user can subscribe to the site by creating a user profile. This allows them to perform all available operations (such as viewing available phones and participating in auctions etc.).

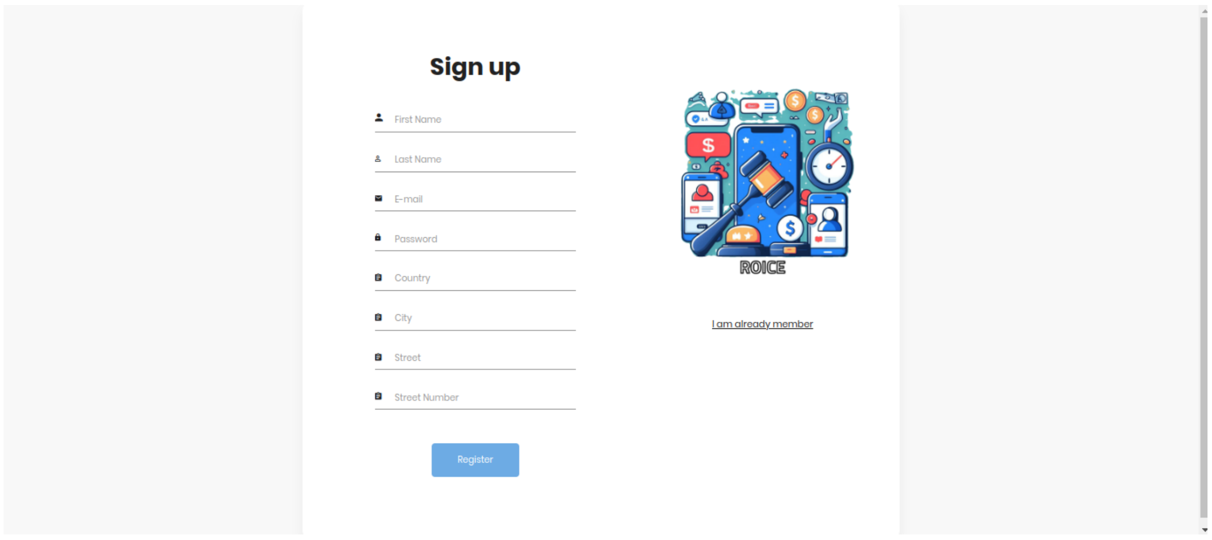


Figure 3.3: signUp.jsp Page

homePage.jsp

This is the main page of the web application, which changes dynamically based on the type of performed access (user or admin). Specifically, an admin, unlike a regular user, has additional features such as the ability to insert new phones by accessing the dedicated page (see adminAddPhone.jsp) and create or remove auctions (see createAuction.jsp). However, an admin cannot participate in auctions like a user but can only observe their progress (see phoneDetails.jsp).

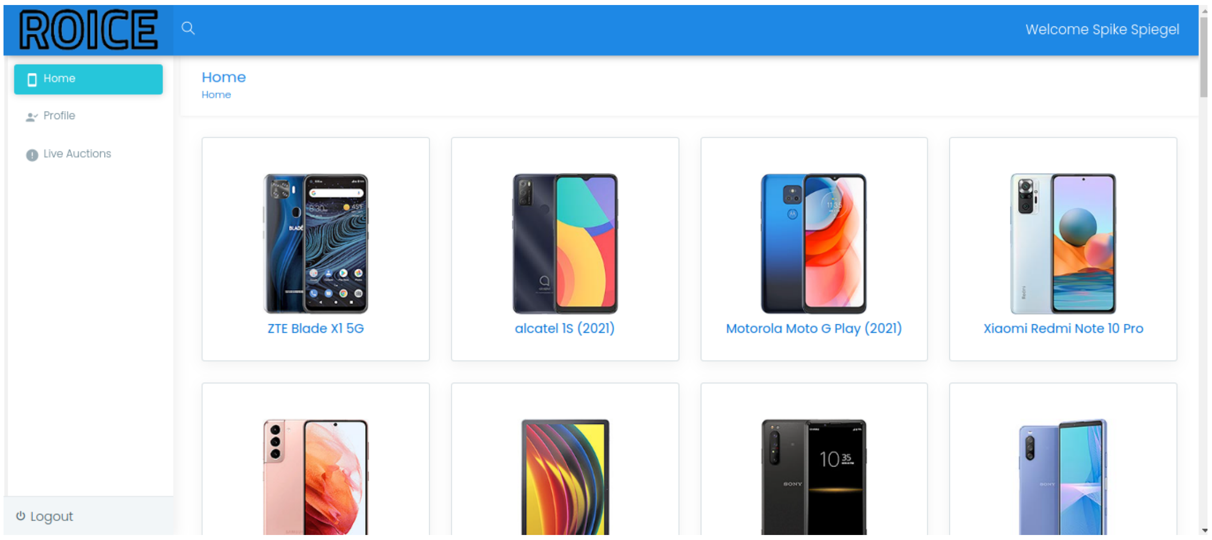


Figure 3.4: homePage.jsp Page

searchLiveAuctions.jsp

This page is displayed when a user clicks on the **Live Auctions button** in the home page menu. It searches for all the auctions currently in progress and displays the phones related to them in a similar manner to how it is presented on the home page.

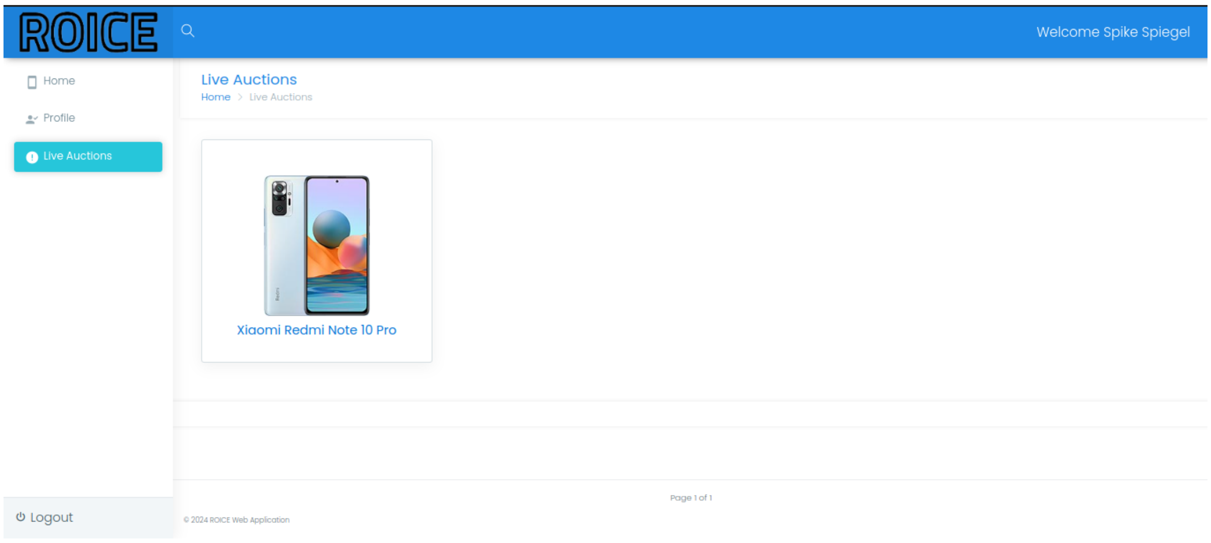


Figure 3.5: searchLiveAuctions.jsp Page

searchPhones.jsp

Similar to the previous scenario, this page is displayed when a user searches by name through the top bar of the home page. It showcases all the phones retrieved from the results based on the research by name.

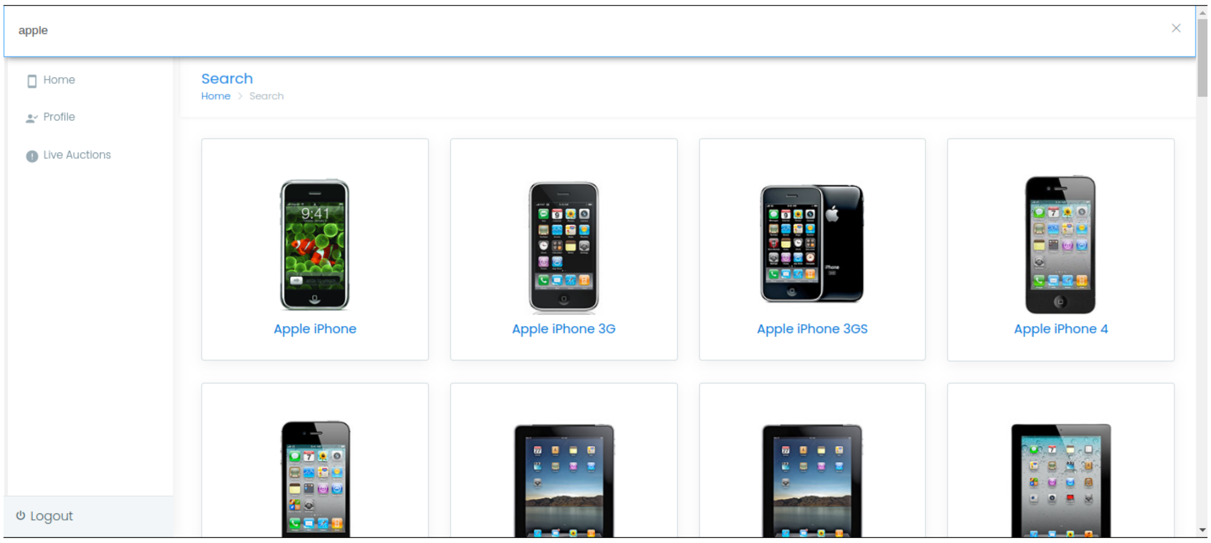
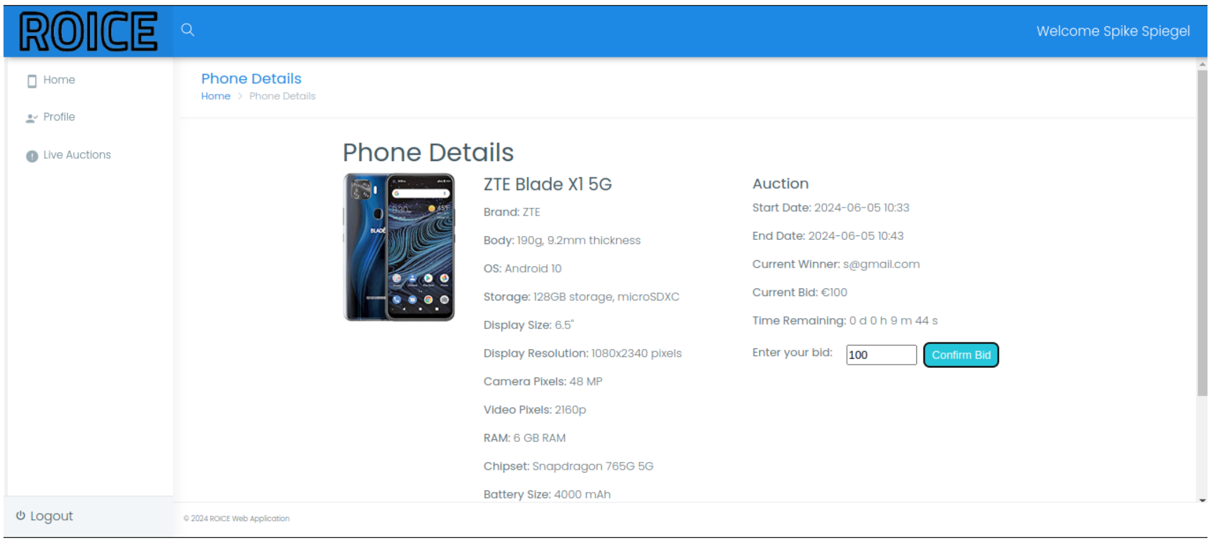


Figure 3.6: searchPhones.jsp Page

phoneDetails.jsp

This page is accessed when a user or administrator selects a specific phone. It provides detailed information about the phone and, if an auction is ongoing, displays the auction details. This includes the auction's start and end dates, the current highest bid, and the name of the leading bidder. In the auction window, users can submit new bids using a text box and a submit button. Additionally, the page features a local timer that periodically synchronizes with the Erlang Server to ensure accurate timing information.

Additionally, administrators have access to an auction creation feature, which allows them to create new auctions by navigating to the *createAuction.jsp* page. Regular users, on the other hand, see buttons to add or remove the specific phone from their favorites list.



Phone Details

ZTE Blade X1 5G

Brand: ZTE

Body: 190g, 9.2mm thickness

OS: Android 10

Storage: 128GB storage, microSDXC

Display Size: 6.5"

Display Resolution: 1080x2340 pixels

Camera Pixels: 48 MP

Video Pixels: 2160p

RAM: 6 GB RAM

Chipset: Snapdragon 765G 5G

Battery Size: 4000 mAh

Auction

Start Date: 2024-06-05 10:33

End Date: 2024-06-05 10:43

Current Winner: s@gmail.com

Current Bid: €100

Time Remaining: 0 d 0 h 9 m 44 s

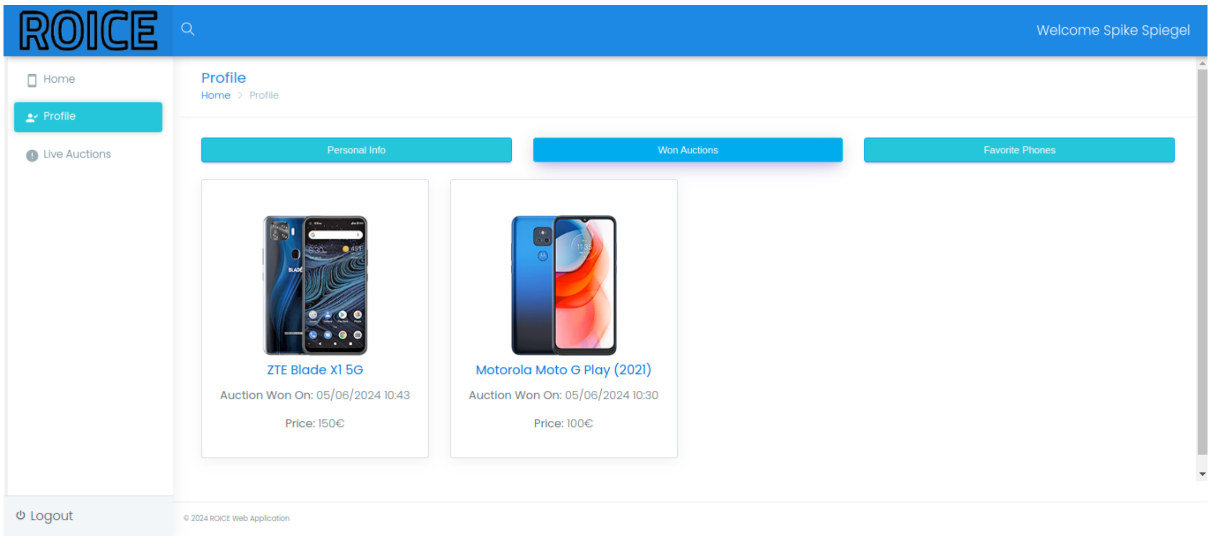
Enter your bid: 100

© 2024 ROICE Web Application

Figure 3.7: phoneDetails.jsp Page

userPage.jsp

The user profile page displays personal information, a list of all previously won auctions with details (such as auction date, winning bid amount etc.), and a list of all the phones marked as favorites.



ZTE Blade X1 5G

Auction Won On: 05/06/2024 10:43

Price: 150€

Motorola Moto G Play (2021)

Auction Won On: 05/06/2024 10:30

Price: 100€

© 2024 ROICE Web Application

Figure 3.8: userPage.jsp Page

adminAddPhone.jsp

This page, accessible directly from the home page, enables an administrator to add a new phone to the database, including all its technical specifications.

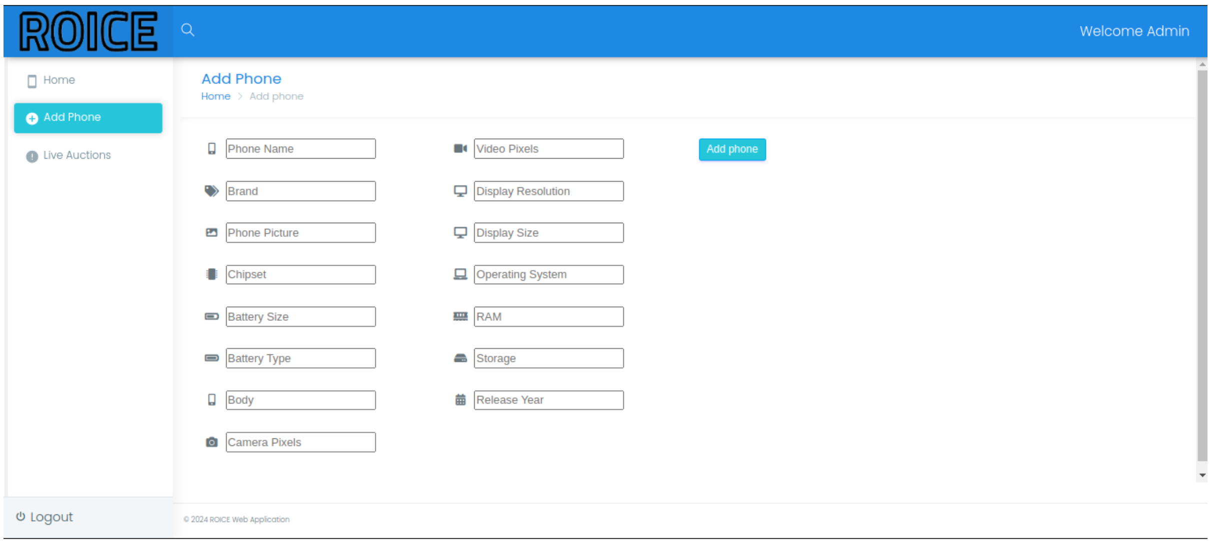


Figure 3.9: adminAddPhone.jsp Page

createAuction.jsp

As detailed in *phoneDetails.jsp*, this page allows an administrator to create a new auction for a phone (if one does not already exist), specifying the auction's start and end dates as well as the starting price.

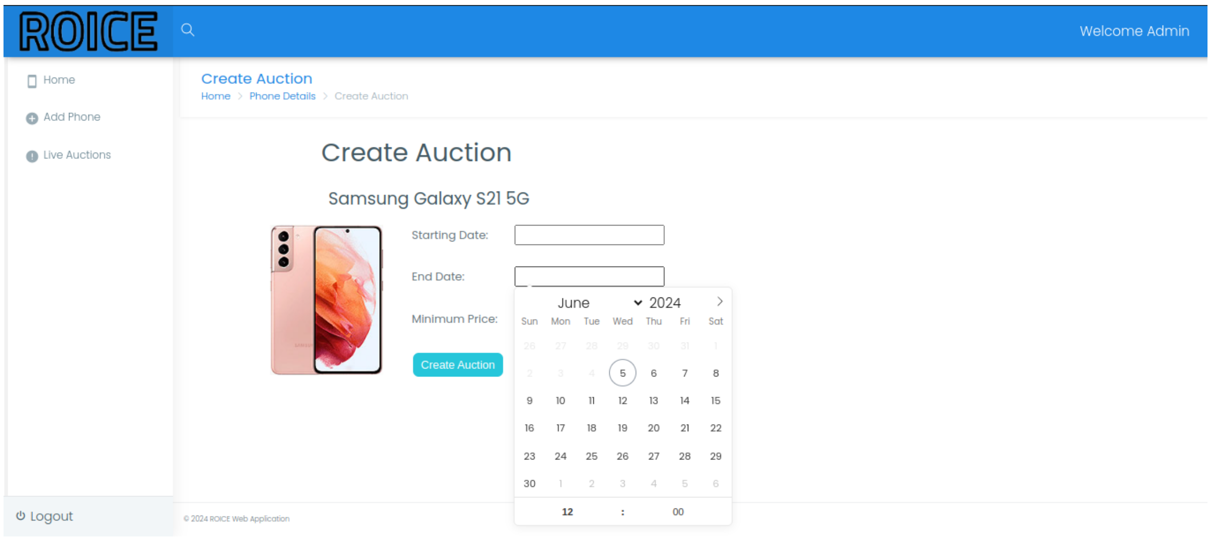


Figure 3.10: createAuction.jsp Page

error.jsp

This is the standard error page. It includes a button to return to the home page.

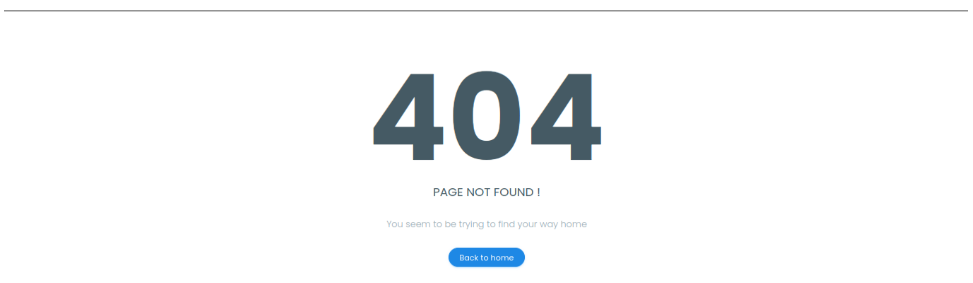


Figure 3.11: error.jsp Page

3.1.3 Controller Classes

The controller classes handle the business logic of the application, taking the input provided by the user in the view querying the model,and finally updating the models and the views accordingly.
As mentioned above, controller classes are linked directly to JSP pages and exploits the service classes to perform required operations (such as retrieve information and update the database).

Following are described the controller classes and their main functionalities:

LoginController
This controller takes care of managing user authentication. It performs the login operation (differentiating from simple user or administrator) by retrieving and validating credentials ensuring secure and efficient handling of login requests.

The handler method is called ***login()*** and it is a POST request mapped to the *login.jsp* page.

SignUpController
This controller handles user registration operations. The core functionality is encapsulated in the ***signUp()*** method, which processes POST requests submitted by users via the *signUp.jsp* page.

HomeController
This controller manages all user requests related to the home page. Its main features include displaying phones using pagination, searching for phones by name, and searching for live auctions. The methods provided are as follows:

- ***homePage()***: It gets a page of phones from the PhoneService service and adds the information to the template for the view. It maps a GET requests and returns to the user the **homePage.jsp** page.

- ***searchPhones()***: Performs a phone search by name using the PhoneService service and adds the results to the model for the view. It maps a GET request and returns to the user the **searchPhone.jsp** page.
- ***searchLiveAuctions()***: Performs a search for phones with live auctions using the PhoneService service and adds the results to the model for the view. It maps a GET request and returns to the user the **searchLiveAuctions.jsp** page.

PhoneDetailsController
This class handles requests concerning the details of individual phones. It manages displaying phone details and, if present, associated auction information. Additionally, it provides methods for users to add or remove phones from their favorites list. The functions provided are the following:

- ***showPhoneDetails()***: This class maps a GET request initiated by the user upon accessing a specific phone page. It is responsible for retrieving all information associated with the phone, including details related to its scheduled auction, if one exists.
- ***addFavoritePhone()***: This class maps a POST request initiated by the user upon clicking the "add to favorites" button. It handles adding the main information of the selected phone to the user's document in the database.
- ***removeFromFavorites()***: This function performs the reverse operation of the previous one. It handles a POST request triggered by the user when clicking the "remove from favorites" button. Its task is to delete the selected phone from the list of favorites in the user's document stored in the database.
- ***handleAuctionEnd()***: This function maps a POST request received directly from the Erlang server. It manages the deletion of the auction related to a specific phone from the database and adds it to the list of auctions won in the winning user's document.

UserPageController
This controller is responsible for mapping the GET request when a user navigates to their personal profile page. The function ***userPage()*** retrieves all the information related to the user and updates the *userPage.jsp* to display their personal information, previously won auctions, and favorite phones.

AdminAddPhoneController
This controller maps both the GET request when the administrator enters the page **adminAddPhone.jsp** (from the homePage.jsp when the user is logged as an admin) and the POST request when they want to insert a new phone into the database. The main function, ***addPhone()***, facilitates the addition of the phone to the collection once all the required fields have been entered correctly.

CreateAuctionController
This controller handles both the GET request when the administrator accesses the page for creating a new auction related to a specific phone (**createAuction.jsp**) and the POST request to save the auction in the database, specifically as an object within a

phone document. The functions responsible for these operations are named ***showCreateAuction()*** and ***createAuction()***, respectively.

3.2 Web Sockets

It was decided to use the **WebSocket APIs** as a method of data exchange between Java Application and Erlang Server. These provide a method of data exchange in both directions through a persistent connection, taking advantage of being particularly suitable for applications that require continuous data exchange.

The creation and management of websocket connections, the reception, management and transmission of websocket messages is defined in the ***connection.js***, ***handleAuction.js*** and ***liveAuctions.js*** files, each of which performs specific operations based on the type of message it receives or the function called by the user through the JSP files. Below are described the main functions belonging to these JavaScript files.

connection.js

This class defines the ***connect()*** function, which handles the following operations when and admin wants to create a new auction for a phone or a user wants to participate in an ongoing auction:

- **WebSocket object creation and connection establishment**: Creates a WebSocket object and establishes a connection the the specified URL.
- **Incoming WebSocket messages:** This function differentiates the operations to be performed according to the message received by the Erlang Server, the various cases are as follows:
  - **Current Winner and New Bid Messages**: The current winner is changed due to higher bid. Updates the current winner and the current bid labels inside the auction section (phoneDetails.jsp).
  - **Update Timer Message**: Receives a message to containing the auction remaining timer, due to a client synchronization requests. Updates the time remaining label.
  - **Winning Bid and User Messages**: Receives a message containing the winning user and the winning bid. Updates the labels.
- **Connection close messages**: Displays a connection closing message.
- **Websocket Error messages**: Displays a websocket error message.

handleAuction.js

This file contains the functions to handle the WebSocket messages related to an ongoing auction. The main functions are the following:

- ***send()***: This is the general function to send a WebSocket message (in JSON format)

- ***confirmBid(email, phoneName)***: This function is called by a user who wants to make a new bid, send a WebSocket message containing the bid value along with his email and name of the phone to which is making the offer.
- ***createErlangAuction(phoneName)***: This function sends a message to the Erlang Server to create a new auction. It sends all the data needed to the creation, such as the name of the phone for which to create the auction, the starting price and the auction end and start date.
- ***sendJoinAuctionRequest***: This function sends a WebSocket message to the Erlang Server when a user connects to a phone page (PhoneDetails.jsp) that has an auction in progress. By sending this message the user starts to listen for receiving all the update messages related to that specific auction, such as the current winner, current bid, or updates on the remaining time for synchronization.
- ***sendGetTimerRequest***: This function send a constant "get auction timer request" to the Erlang Server (set to 10 seconds), in order to perform a synchronization operation about the remaining time of the ongoing auction to which the user is connected.

liveAuctions.js
This file takes care of opening a new WebSocket when a user is connecting to the live auctions page (*searchLiveAuctions.jsp*) in order to send and receive a real-time update message of the live auctions to/from the Erlang Server as soon as the auction starts. The functions defined to fulfil this task are ***createWebSocketConnection()*** and ***sendLiveAuctionsMessage()***.

3.3 DataBase

Regarding data persistence, we have opted to use a document database to store all information related to users and phones and we have specifically chosen **MongoDB** for this purpose. MongoDB's schema-less design provides greater flexibility in handling various types of data without requiring a predefined schema. It also eliminates the need for complex joins, as it can store related data together in a single document, enabling more efficient retrieval of complete documents rather than individual fields. These features collectively result in superior performance and scalability compared to a traditional relational database, making MongoDB an ideal choice for our application.
We have decided to create two collections in our MongoDB database to organize our data efficiently:

- **Users Collection**: used to store information about all users of the system, which can be administrators or regular ones.
- **Phones Collection**: it contains detailed information about the phones available on the platform. Users can search for these phones and participate in auctions to purchase them. Each document in this collection includes specific phone features and, eventually, some auction-related details.

4 - Erlang Applications

Both Erlang applications, the **Master Node Application** and the **Erws Application**, were developed using **rebar3** as the build tool. Rebar3 is an advanced build tool for Erlang that helps in managing dependencies, running tests, and building releases. Each application has its own configuration file, ***rebar.config***, where specific dependencies, project settings, and build configurations are specified. This setup simplifies the development process and ensures consistent builds across different environments.

4.1 Master Node Application

The Master Node Application is responsible for managing the **cluster** of Erlang nodes, initializing **Mnesia**, and maintaining the overall state of the system.
The following subsections describe the modules and their functionalities in detail.

4.1.1 Application Configuration

The application configuration file (***master.app.src***) specifies various settings for the Master Node Application, such as dependencies, environment variables, and registered modules. It defines the description, version, and dependencies on other applications (e.g., kernel and stdlib). It also sets the nodes that are part of the Erlang cluster.

4.1.2 master_app

The ***master_app*** module is the core of the Erlang application, responsible for managing the master node and coordinating the setup and operation of the cluster. Let's break down its behavior:

Start

The ***start/2*** function initializes the application. It retrieves the list of cluster **nodes** from the **application environment**. Then, it **connects** to each node in the cluster using the ***connect_nodes/1*** function. The latter iterates over the list of cluster nodes and connects to each one of them using the standard function ***net_kernel:connect node/1***.
After connecting to all nodes, it initializes Mnesia on the cluster using the ***mnesia_setup:init/1*** function.

Stop

The ***stop/1*** function stops the application. First, it stops Mnesia using the ***mnesia:stop/0*** function. Then, it stops each remote node in the cluster using the ***stop_nodes/1*** function. The latter sends a shutdown signal to each node using ***spawn/3***.

4.1.3 mnesia_setup

The ***mnesia_setup*** module is responsible for setting up Mnesia on the **cluster nodes**.

Initialization

The initialization function in the ***mnesia_setup*** module (***init/1***) creates the Mnesia schema and starts Mnesia on both the local node and the remote nodes specified. It then calls functions to create the auction and bid tables, ensuring that these tables are set up correctly across the cluster.

Table Creation

The module provides functions to create the auction and bid tables (***create_auction_table/1*** and ***create_bid_table/1***). These functions check if the tables already exist and, if not, create them with the specified attributes and replication settings. The tables are configured to be stored as disk copies on the specified nodes, ensuring data durability and availability.

4.2 Erws Application

The Erws application is responsible for managing WebSocket connections, handling auctions, and maintaining the overall state of the Application.

4.2.1 Application Configuration

The application configuration file specifies various settings for the Erws application, including dependencies, environment variables, and registered modules. It defines the description, version, dependencies on other applications (e.g., kernel, stdlib, cowboy), and environment-specific settings such as WebSocket endpoints and ports.

4.2.2 erws_app Module

The ***erws_app*** module is the main application module responsible for starting and stopping the application. It initiates the top-level supervisor, **erws_sup**, to manage WebSocket connections and auction processes.

Start

The ***start/2*** function initializes the application by starting the top-level supervisor, **erws_sup**, which in turn starts the WebSocket server.

Stop

The ***stop/1*** function stops the application by terminating the WebSocket server and associated processes.

4.2.3 erws_server Module

The ***erws_server*** module is responsible for starting and configuring the WebSocket server using Cowboy. It exploits the WebSocket **endpoint** and **port** specified in the application environment to compile Cowboy dispatch rules and start the server.

4.2.4 OTP Supervision Tree

The Erws application uses a **supervision tree** to ensure fault tolerance and reliability. The application module starts the top-level supervisor, which manages the WebSocket server and a dynamic supervisor for auction processes. The supervision tree structure is illustrated in Figure 4.1.

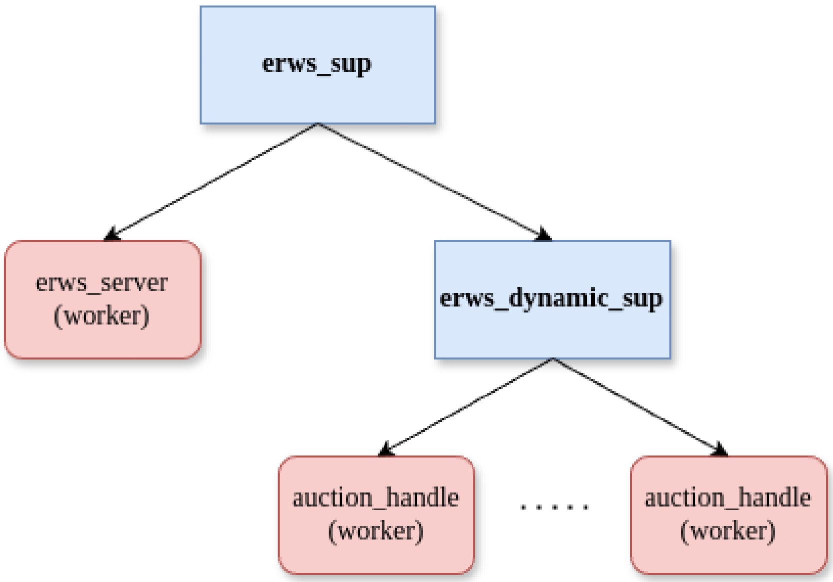


Figure 4.1: Erws Application Supervision Tree

Top-level Supervisor: *erws_sup*

The ***erws_sup*** module is the top-level supervisor for the Erws application. It is responsible for starting and monitoring the WebSocket server (***erws_server***) and the dynamic supervisor (***erws_dynamic_sup***), which manages auction processes.

Initialization

The ***init/1*** function defines the supervisor's restart strategy and child specifications:

- **Strategy**: *one_for_one* - This strategy means that if a child process terminates, only that process is restarted.
- **Maximum Restarts**: 3 - This sets the maximum number of restarts allowed within a specified time period before the supervisor itself terminates.
- **Maximum Time Between Restarts**: 30 seconds - This defines the time window within which the maximum restarts are counted.

Child Specifications

The ***erws_sup*** module has two different kinds of child specifications: one for the WebSocket server and another for the dynamic supervisor.

- ***erws_server***:
  - **Start Function**: ***erws_server:start_link/0*** - Function to start the WebSocket server.
  - **Restart Type**: *permanent* - The child process is always restarted.
  - **Shutdown Time**: 5000 ms - Time allowed for the child process to shut down gracefully.
  - **Child Type**: *worker* - Indicates that this child process performs work and is not a supervisor.
- ***erws_dynamic_sup***:
  - **Start Function**: ***erws_dynamic_sup:start_link/0*** - Function to start the dynamic supervisor.
  - **Restart Type**: *permanent* - The dynamic supervisor is always restarted.
  - **Shutdown Time**: infinity - Indicates that the child can take an unlimited amount of time to shut down.
  - **Child Type**: *supervisor* - Indicates that this child process is a supervisor.

Dynamic Supervisor: ***erws_dynamic_sup***

The ***erws_dynamic_sup*** module manages the supervision of auction processes. It allows for dynamic starting of auction processes.

Initialization

The ***init/1*** function defines the dynamic supervisor's restart strategy and child specifications:

- **Strategy**: *simple_one_for_one* - This strategy is used for dynamically attached children. It means the supervisor can dynamically attach new children which are all of the same type.
- **Maximum Restarts**: 5 - This sets the maximum number of restarts allowed within a specified time period before the supervisor itself terminates.
- **Maximum Time Between Restarts**: 30 seconds - This defines the time window within which the maximum restarts are counted.

Child Specifications

The child specifications for the ***erws_dynamic_sup*** module include:

- ***erws_auction_handler***:
  - **Start Function**: ***erws_auction_handler:start_link/0*** - Function to start an auction handler process.
  - **Restart Type**: *transient* - The child process is restarted only if it terminates abnormally.

- **Shutdown Time**: 5000 ms - Time allowed for the child process to shut down gracefully.
- **Child Type**: *worker* - Indicates that this child process performs work and is not a supervisor.

Auction Process Start

The ***start_auction_process/4*** function initiates an auction process dynamically, specifying parameters such as the phone name, minimum price, auction time, and end date. This is done using the ***supervisor:start_child/2*** function, which allows for the dynamic spawning of a child process.

4.2.5 Handlers

To correctly manage the WebSocket communications and auction processes within our system, 2 modules were implemented:

- the ***erws_auction_agent***, which is responsible for initializing, managing, and terminating WebSocket connections (via Cowboy), processing incoming WebSocket text frames, and handling various auction-related actions;
- the ***erws_auction_handler***, which focuses on starting and managing the auction processes themselves, receiving messages from bidders, sending updates, and terminating auctions.

More specifically, the erws_auction_agent module, after processing the incoming Websocket text frames and decoding the JSON object contained in them, performs operations based on the value associated with the object's **"action"** key. The following will list the possible values and an explanation of what the module does based on the cases:

- **"new_auction"**: a new auction for a given phone is scheduled to be started on the date provided and with the minimum price received. At the time the auction is to start, the supervisor module ***erws_dynamic_sup*** spawns a process associated with it dedicated to its management, invoking a method defined in the ***erws_auction_handler*** module.
- **"join_auction"**: a message is sent to the erws_auction_handler module to record a user's connection event to a live auction and allow him or her to receive updates on its progress.
- **"live_auctions"**: a message is sent with a JSON containing this action when a user enters the page relating to live auctions and allows that event to be recorded so that he can see both what auctions are in progress and what auctions are starting live.
- **"send"**: the moment, during a live auction, a user submits a bid, a message is sent to the erws_auction_handle module to manage it properly so that the current highest bid and the e-mail of the one who placed it are always displayed to those who are participating.
- **"timer"**: a message is sent to the user with information about the current bid, the current winner, and the time remaining before the auction to which he is connected is about to end.

Whenever it is necessary to give a response to a client request, a specific message is created and sent to him via the methods provided by the Cowboy server.
The erws_auction_handler module, on the other hand, interacts with the erws_auction_agent module by exchanging Erlang messages with it and, based on those messages, manages the life cycle of an auction.
The **start_link** function (invoked by the supervisor module erws_dynamic_sup) spawns a process associated with the **auction_handle** function. The latter, in case of a call after a failure while an auction is still running, is responsible for associating a new ID to the process related to the specific one. In addition to this, it has the main task of saving the new auction into the database and executing the **auction_receive** function, which handles all messages related to a specific auction that are sent from the erws_auction_agent module and performs operations based on them.
The atoms related to the different requests and what is done when they are received will be listed below:

- **bidder_join**: all information about the auction a user has just joined is retrieved and sent as an Erlang message to the erws_auction_agent module so that it can send the latest updates correctly.
- **timer**: all information about a specific auction is sent as an Erlang message to the erws_auction_agent module to allow it to update live bidders. This type of atom is received periodically and is useful for providing updates even when no bidders are bidding.
- **send**: when a new bid is received from a bidder, the necessary checks are made to see whether the current one is higher or lower than it and, in the latter case, update the data on the current winner and current bid. This information is, finally, sent as an Erlang message to the erws_auction_agent module.

Finally, the function takes care of deleting the auction from the database when it ends and creating an Erlang message to be sent to the erws_auction_agent module, which allows users who were following the auction to see the winner and its offer, in the event that at least one has been done.
To be able to properly and in real-time provide auction-related updates to those who participate in them, an advanced process registry for Erlang called **gproc**, was used. It is designed to provide more functionality than the standard Erlang process registry. Specifically, it was used to implement a **publish/subscribe** mechanism, in which processes can subscribe to events and be notified when they occur.

4.3 Mnesia DB

A distributed key-value DBMS, called **Mnesia**, was used to store the data and replicate them in a distributed manner above the various Erlang nodes in the application.
In Mnesia, the data is organized as a set of tables, each of them has an atom as its name and consists of Erlang records.
This database provides various options for the storage of the data. Among them, the one we decided to adopt is **disc_copies**, which, according to Mnesia documentation, specifies a list of Erlang nodes where a table is kept in RAM and on a disc. If a table is of type disc_copies into one node, **the entire table is written in the RAM and logged on**

the disc of that node.
The tables defined within our database and their attributes are as follows:

- **auction**: the moment an auction begins, it is saved within this table as a record whose attributes are **phone_id**, which is the name of the phone for which there may be bids, and **auction_pid**, which is the ID of the process related to the handling of the auction itself;

- **bid**: whenever, during an auction for a specific phone, a bid greater than the current bid is placed, a record is saved within this table that has as attributes the name of the phone (**phone_name**), the e-mail of the user who placed the highest bid at that time (**current_winner_user_email**), and the value of the bid (**bid_value**).

Within the ***erws_mnesia*** module, the following methods related to the tables described above have been defined:

- **save_auction(PhoneName, AuctionPid)**: to save an auction record into the auction table.

- **get_auction_pid(PhoneName)**: to retrieve an auction PID by giving as parameters a phone name and by searching between the records within the auction table.

- **delete_auction(PhoneName)**: to delete an auction record within the auction table based on a phone name as a parameter.

- **save_bid(PhoneName, WinnerEmail, BidValue)**: to save a bid record into the bid table.

- **get_winner_bidder(PhoneName)**: to get, by giving a phone name as a parameter, a tuple from the bid table that has the auction winner bidder's email and its bid.

- **delete_bid(PhoneName)**: to delete a bid record from the bid table, by giving a phone name as a parameter.

All methods listed perform operations using **atomic transactions**, which guarantees data integrity and consistency, even in the presence of concurrent operations or system failures.