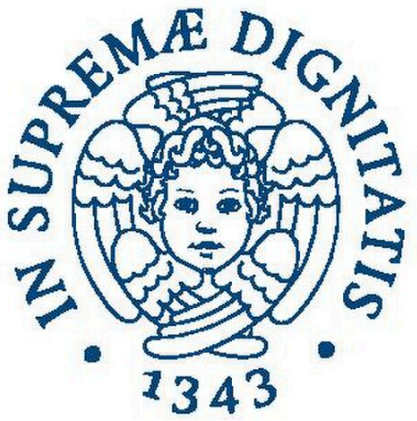


UNIVERSITÀ DI PISA

MSc in Computer Engineering

Distributed Systems and Middleware Technologies

Project Documentation

Mariella Melechì, Riccardo Fiorini, Erica Raffa

2023/2024

1. **Project Specifications**

1.1. Introduction

Our idea is a Pokemon trade app. Every user can create or search for **listings** to exchange their **Pokemon**. Users can select a listing and offer their pokemon to be exchanged with the one proposed by the listing's owner. Moreover they can decide the pokemon's winner from their own listings.

1.2 Functional Requirements

In our application we implement 2 types of actors:

- Unregistered user
- Registered user

The possible action that each actor can perform are the following:

1. An **unregistered** user can:

    - Create an account

2. A **registered** user can:
    - Login/logout
    - Browse Listings
    - Select Listings
    - Make/Delete offers
    - Add/Delete Pokemon listing
    - Browse his pokemon box
    - Browse his past trades

1.3 Non-functional Requirements

Concurrent service accesses management:

- Multiple users can create/delete listings or offer Pokemon at the same time
- Strong **consistency** during trades and listings
    - A pokemon is available exclusively for one listing or one offer
- Allow high horizontal scalability

Synchronization and timing issues:

Internal coordination for Real Time synchronization of offer and listings

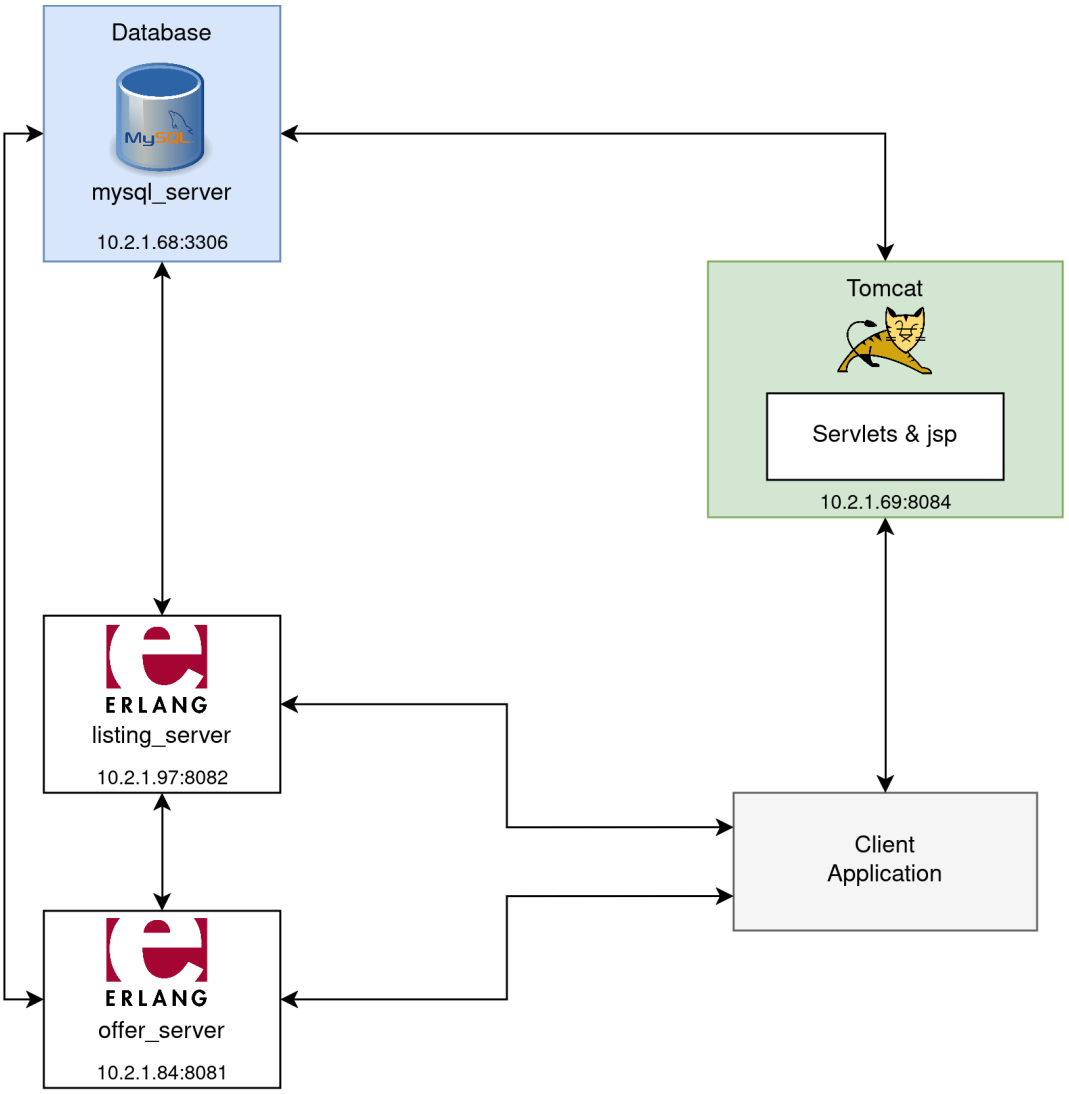
- Each time a listing is created:
    - The Home is updated in real time
- Each time a listing is deleted:
    - The Home is updated in real time
    - Inside the Listing page, the users are notified that the listing is closed and redirected to the Home
- Each time Pokemon are traded:
    - The Home is updated in real time
    - Inside the Listing page, the users are notified that the listing is closed and redirected to the Home
    - The owners of the traded Pokemon are switched

These issues are managed with **Erlang**, spawning different Erlang processes for each functionality and notifying the changes thanks to the websockets between server and client.

There are two server nodes written in Erlang: one for the listings and one for the offers. They will both communicate with each other, with the client and with the MYSQL database. We plan to introduce one node for the **Tomcat** server.

2 System Architecture

The application system architecture schema shows how we decided to exploit the 4 available nodes:



The web application is composed by:

- One **Tomcat server**
- Two **Erlang servers** with different roles:
    - listing_server
    - offer_server

The functionalities are handled by 3 main modules: *socket_listener.erl*, *mysql_handler.erl* and *listing_registry.erl*. The **socket_listener** is a process spawned for each user, and intercepts a new JavaScript request, unpacking it and forwards it to the registry. The registry sorts the arriving requests and forwards them to the **mysql_handler**, unless it is a register/unregister operation. At last, the **mysql_handler** module runs every database-related operation.

2.1 Listing Server

The listing server application leverages the **supervisor** behavior.
Upon startup, the application process initiates the creation of a supervisor process, known as "**listing_server_sup**" callback module. This supervisor is responsible for handling the lifecycle of the process that operates the Cowboy listener. The process responsible for running Cowboy is designed as a **gen_server** module, known as "**cowboy_listener**" callback module. The Cowboy server listens on port 8082, awaiting incoming WebSocket connection requests. Once a request is received, it spawns a new process to manage that specific connection. As a result, each user is assigned to a distinct Erlang process.
When the process starts, a "**socket_listener**" module, a "**listing_registry**" module and a "**mysql_handler**" module are initiated. Then the listing_registry is charged to coordinate the socket connections and forward the messages to the users of the registry.

The "**mysql_handler**" module is an Erlang process that receives the requests from the registry process, runs the MySQL queries and responses with the related information to the registry.

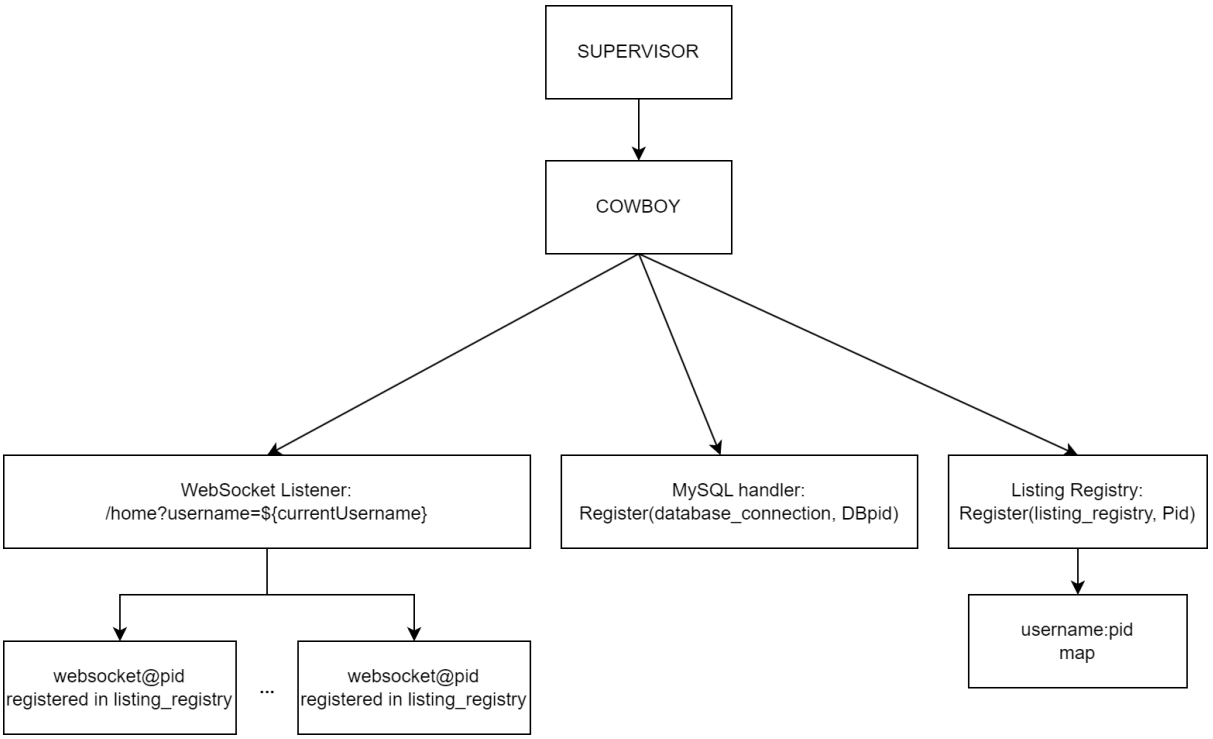


Figure 2.1 - Listing Server Architecture

Socket listener module operations:

- **init/1**: this function is invoked every time a request is received, spawning a new process for the Cowboy WebSocket

- **websocket_init/1**: it passes its related PID and the username to the registry for the registration
- **websocket_handle/2**: it is responsible for handling the reception and deserialization of JSON objects sent by the client, depending on the operation requested. This method is called whenever a frame is received from the client
- **websocket_info/2**: whenever an Erlang message is received by the registry, this method encodes the required information into a JSON object and forwards it to the client
- **terminate/3**: this function is called when the WebSocket connection gets closed. It removes the associated user from the list of active ones, sending an “unregister” message to the registry process.

Listing registry module operations:

- **start_listing_registry/0**: this function spawn two processes: one handles the registry, the other handles MySQL operations
- **registry_loop/1**: it’s a receiving loop that can:
    - register/unregister the users
    - spawn processes to forward insert/delete operations to MySQL
    - forward the messages received from the other Erlang server to MySQL, to update a listing
- **handle_mysql/6**: it is responsible for handling the communication between the listing registry and MySQL. It forwards a request to the process that handles MySQL operations and forwards its response to the users of the registry.

2.2 Offer Server

The **offer server** application utilizes supervisor behavior to manage its processes. Upon initialization, the application process starts by creating a **supervisor** process referred to as the "**offer_server_sup**" callback module. This supervisor is tasked with managing the lifecycle of the process that runs the **Cowboy** listener. The process that handles Cowboy is implemented as a gen_server and uses the "**cowboy_listener**" module as its callback. The Cowboy server listens for incoming **WebSocket** connection requests on port 8081. When a request is received, a new ad hoc process is spawned to handle that specific connection, ensuring each user is assigned to a separate Erlang process. Upon startup, the process also initiates the "**socket_listener**," "**offer_registry**", and "**mysql_handler**" modules.
The offer registry is then responsible for coordinating the socket connections using Erlang messages.

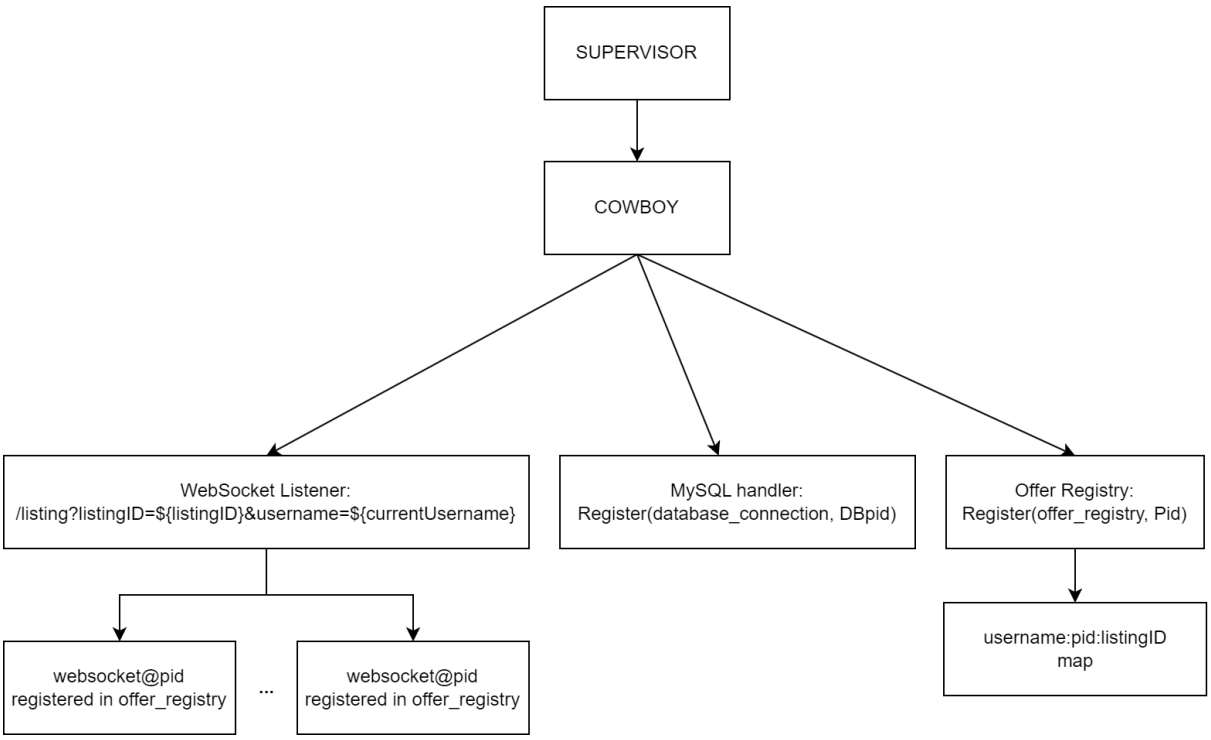


Figure 2.2 - Offer Server Architecture

Socket listener module operations:

- **init/1**: this function is triggered upon each request reception, initiating the creation of a new process for the **Cowboy WebSocket**.
- **websocket_init/1**: upon invocation, it transmits the associated PID and the username for registration.
- **websocket_handle/2**: activated upon receipt of a frame from the client, this function manages the reception and JSON object deserialization sent by the client.
- **websocket_info/2**: when the registry process receives an Erlang message, it forwards messages to the client's browser connected to this specific process.
- **terminate/3**: executed upon WebSocket connection closure, this function's primary task is to remove the corresponding user from the active user list by dispatching an unregistered message to the registry process.

Offer registry module operation:

- **Register**: insert a key value pair for the username, listing ID and its associated WebSocket PID in a map.

- **Forward**: this operation is useful to add a new offer in a listing. Specifically, it looks for the destination PID in the map, then **spawn** a new process in order to save the offer into the database and eventually chooses if forward the offer to the other processes mapped with the same listing ID to make this new offer visible in real time.

- **Unregister**: remove the specific key value pair from the map.

- **Delete**: this operation is used to delete the specific user's offer from the database and sends the delete notification to all other processes mapped with the same listing ID.

- **Trade**: the trade operation affects the winnings of the listing featuring the selected Pokémon. It ensures the update of all pertinent fields in the database to uphold consistency with the Pokémon boxes. Additionally, the trade notification is dispatched to other processes associated with the same listing ID to inform other users about the listing's closure. Simultaneously, the notification is relayed to the listing server node to update and change visibility within the application's home.

- **Listing_deleteUpdate**: this operation allows the deletion of a listing to users who are viewing it in real time. It is received by the Erlang listing server node and is sent to all processes mapped with the same received listing ID.

Mysql Handler module operation:

- **save_offer:** insert offer in the database.

- **delete_offer**: delete offer from the database and modify the **listed** value of pokemon box.

- **trade**: insert the listing’s winner and modify all the listed value of the pokemon’s box offer, delete offers and change the pokemon owner.

- **update_listing**: delete offers related to the deleted **listing** and change all the listed value of the pokemon’s box

2.3.Tomcat

2.3.1. Data folders

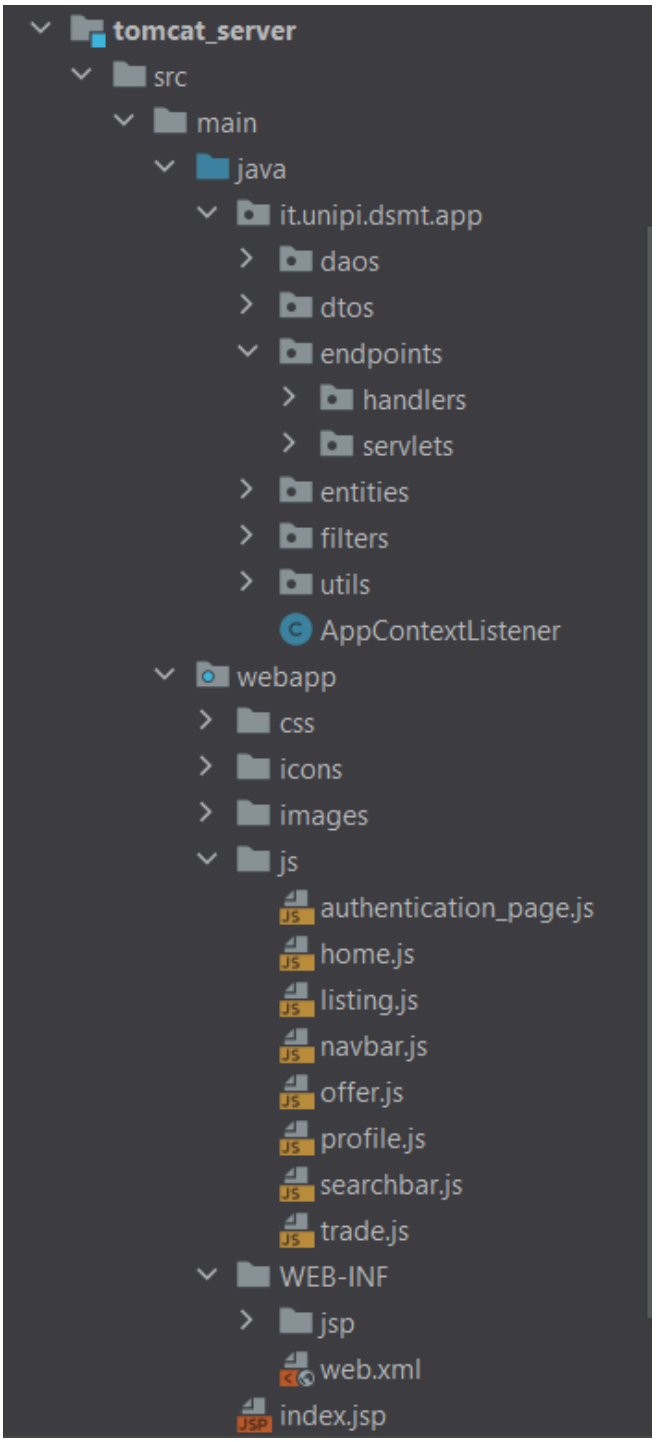


Figure 2.4 - Data Folder Structure

The project is organized as follows:

- **daos**: Contains Data Access Objects (DAOs) for initializing and interacting with the database or other data sources.
- **dtos**: Houses Data Transfer Objects (DTOs) that represent data objects used for transferring data between different parts of the application.
- **endpoints**: Contains classes related to application endpoints or routes.
- **handlers**: Includes request handlers for specific endpoints (login and signup authentication).
- **servlets**: Contains servlet classes for handling web requests.
- **entities**: Holds entity classes that represent objects in the application domain, facilitating easier handling of database-related objects.
- **filters**: Contains filter classes for intercepting requests, particularly the access filter class that validates authorization tokens.
- **utils**: Includes utility classes that provide various helper functions and methods for the application, such as error handling and access control functions.
- **webapp**: Contain web-related resources.
- **WEB-INF**: A standard folder for configuration files and resources, including JSP components and pages.

2.3.2 Java

Java exploits an app context listener to initialize the database connection and store it in the servlet context, so that it is shared among all the servlets.
The Java application interacts with the database through **DAOs** (Data Access Objects) exploiting the entity classes to model MySQL tables. Each DAO exposes the main operations towards the database, and interacts with the client application through the **DTOs** (Data Transfer Objects), modeling the data as they are required.

Servlet Endpoints

Name	Endpoint	Method	Description
HomeServlet	“/home”	GET	Retrieve the listings to show in the home page
ListingServlet	“/listing”	GET	Retrieve the info and the offers of the selected listing. If the current user is not the owner of the listing, it also retrieves his box
LoginServlet	“/login”	GET	Check if the authorization token is still valid: if it is, the user is redirected to the home page; otherwise it is redirected to the login page
		POST	Validate user credentials and eventually set an authorization token for the access. The user is then redirected to the home page
LogoutServlet	“/logout”	GET	Invalidate the session and redirect the user to the login page
ProfileServlet	“/profile”	GET	Retrieve the info, the box and the listings of the current user to show them in the profile page, in the “box” section
		POST	A request is sent to this endpoint when a new listing is created. It retrieves the info and the listings of the current user to them in the profile page, in the “listings” section.
SignUp	“/signup”	GET	Redirect to the signup page
		POST	Get the info of the form, register the new user and set the authorization token. Then redirect to the home page

2.3.3 Webapp

The project's **webapp** section contains resources for formatting and building the pages, including **CSS** and **JavaScript** files, as well as secondary assets like icons. The most important content is the **WEB-INF** folder, which includes a jsp folder with all necessary **JSP** files.

2.3.4 Listing JavaScript functions

In the **home.js** file, a **WebSocket** connection relative to the home of the current user is opened. This file contains the functions that implement the real-time updates of the home.
The "**onmessage**" function receives the message from the Erlang server and processes it depending on the message type:

- **insert:** the listing is created with the informations of the message and displayed in the home through the "**appendListingComponent**"

- **delete:** the listing with the specified ID is deleted from the home and it is hidden to the users through the "**HideListingComponent**"

- **update:** the listing with the specified ID is hidden from the home through the "**HideListingComponent**"

In the **profile.js** file, a **WebSocket** connection relative to the profile of the current user is opened. This file contains the functions that implement the real-time updates of the profile.
Specifically, when a user creates a new listing using the "**handleNewListing**" function, the **Erlang Listing_Server** receives the insert request, communicates with MySQL and delivers it to the other processes, as said before. If the Erlang server encounters an issue, it returns an error.
Similarly the user can delete a listing via the "**handleDeleteListing**" function, and this follows the same logic as before.

The "**onmessage**" function receives the message from the Erlang server and processes it depending on the message type:

- **insert:** this notifies that the listing has been inserted in the database, as requested; the user is then redirected to the "listings" section of its profile

- **delete:** this notifies that the listing has been removed from the database, as requested; the selected listing is then hidden from the profile

2.3.5 Offer JavaScript functions

In the **offer.js** file, a **WebSocket** connection relative to the specific listing is opened. This file contains important functions to handle the **WebSocket** operations such as:

1. **handleSend**: allows to send a message in the **JSON** format through the connection established.
2. **onmessage**: handles the incoming message.

Specifically when a user adds a new offer using the "**handleSend**" function the **Erlang Offer_Server** receives the insert request, saves and delivers it to the other processes, as said before. If the Erlang server encounters an issue, it returns an error.
Similarly the user can delete the offer via the "**delete**" function or carry out the trade via the "**trade**" function.

The "**onmessage**" function receives the message from the Erlang server and processes it depending on the message type:

- **offer:** the offer is added to the specific listing and displayed to the users through the "**appendOfferComponent**" that appends the new component and discriminates whether the logged in user is the owner of the listing or the offer's creator to select the right button to display.

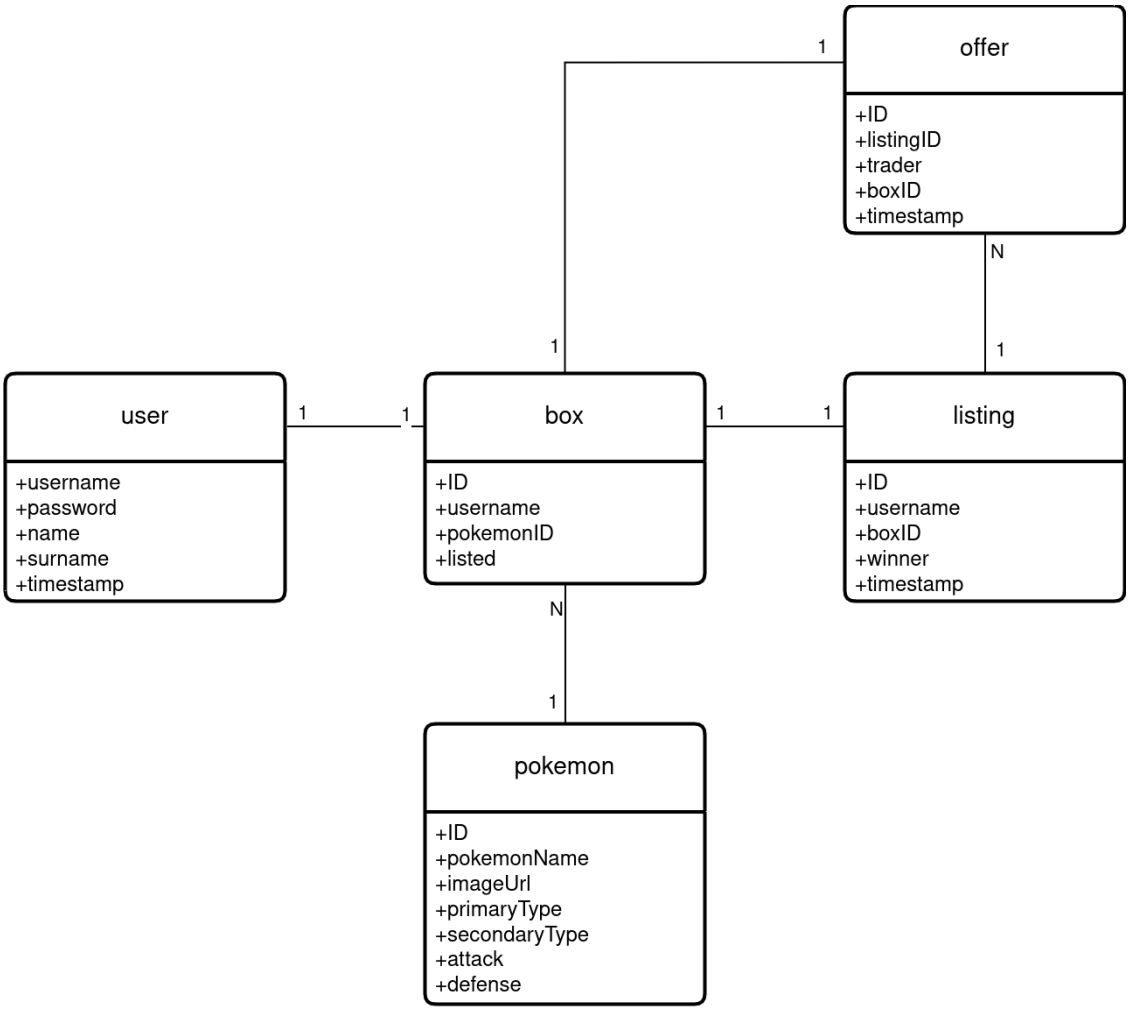
- **delete:** the offer is deleted from the specific listing and it is hidden to the users through the "**HideOfferComponent**". It discriminates whether the logged in user is the offer's owner to select the right button to display.

- **listing:** the listing is deleted and a popup is shown to the users that are in that listing page.

- **trade:** the winner of the list is changed, and a popup is shown informing of the winner.

2.4 MySQL Server

One of the nodes has been dedicated to host the Database. The schema below shows the final architecture of the Database:



- **User:** It is identified by the unique username, chosen during the signup. The passwords are encrypted and the table is completed with the name and surname of the user.
- **Pokemon:** This table has the purpose to represent every kind of existing Pokemon, each represented by its own ID and other information such as type, attack and defense.
- **Box:** This table represents the ownership of one pokemon to the respective user. A user can have more than one pokemon of the same type and pokemon id (previously described table) but each of them has a unique box ID.
- **Listing:** A user can create one listing for each boxed Pokemon.
- **Offer:** A user can offer multiple boxed Pokemon for one or more Listings.

3 **User manual**

3.1. Login Page

The login page of the web app allows users to log in by entering their username and password. After clicking the login button, the user will be redirected to the home page if the login is successful. If the login attempt fails, an error message will appear. There is also a link to the signup page for unregistered users.

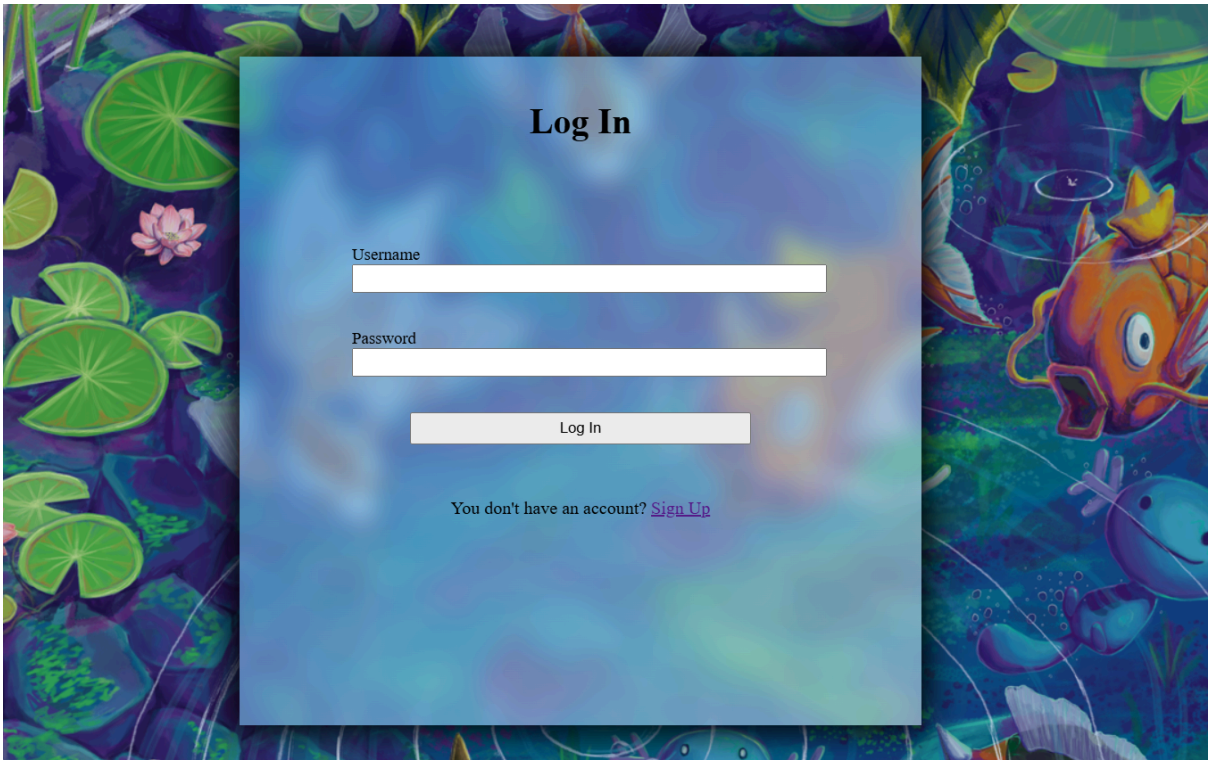


Figure 3.1 - Login page

3.2 Sign-up Page

The Sign-Up page enables unregistered users to create an account in the app. It can be accessed from the Login page and requires users to fill in their personal information. Upon successful registration, the user is redirected to the Home page. If the sign-up fails, an error message is displayed. Additionally, there is a link for users to return to the Login page.

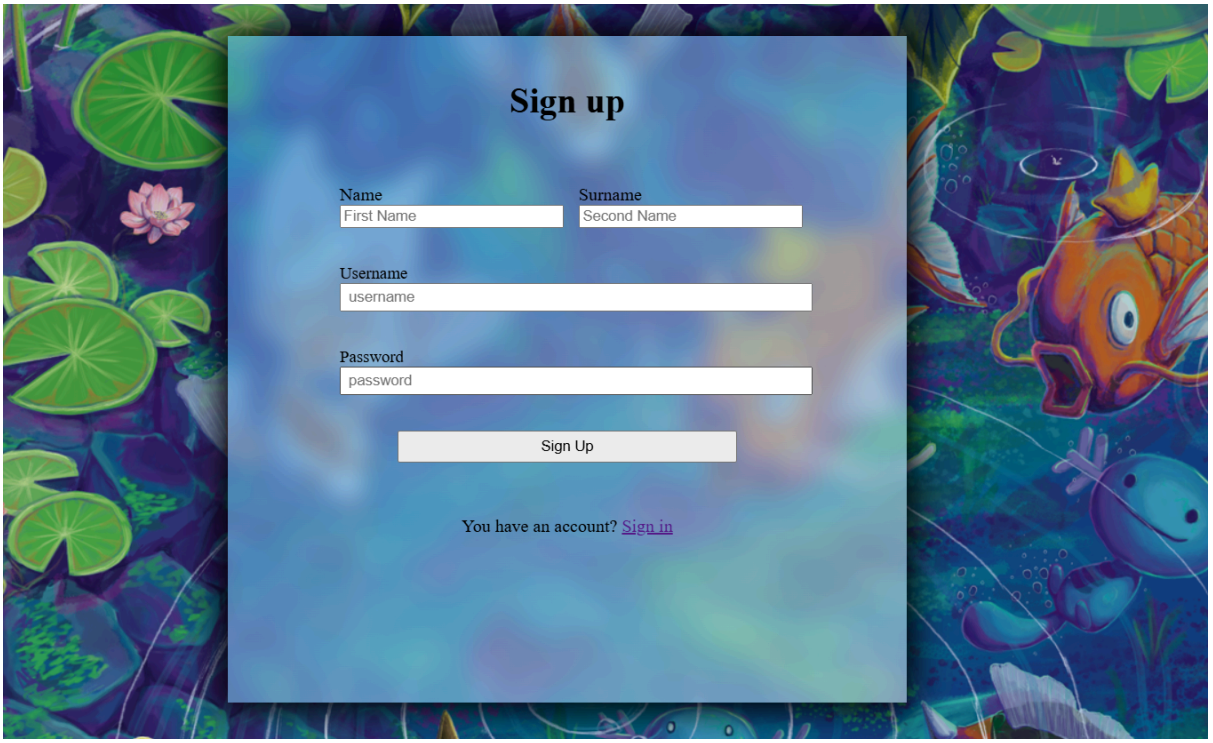


Figure 3.2 - Sign-up page

3.3 Home Page

It shows all ongoing listings that do not belong to the logged in user. He can search for a particular listing via the search bar or show past listings via the "**past**" listings filter.
Furthermore, each listing shows its characteristics, i.e. the owner and the Pokemon offered for exchange. From here, the user can select the listing, or the "**Profile**" button to be redirected in his/her personal profile or the "**Logout**" button to logout from the application.

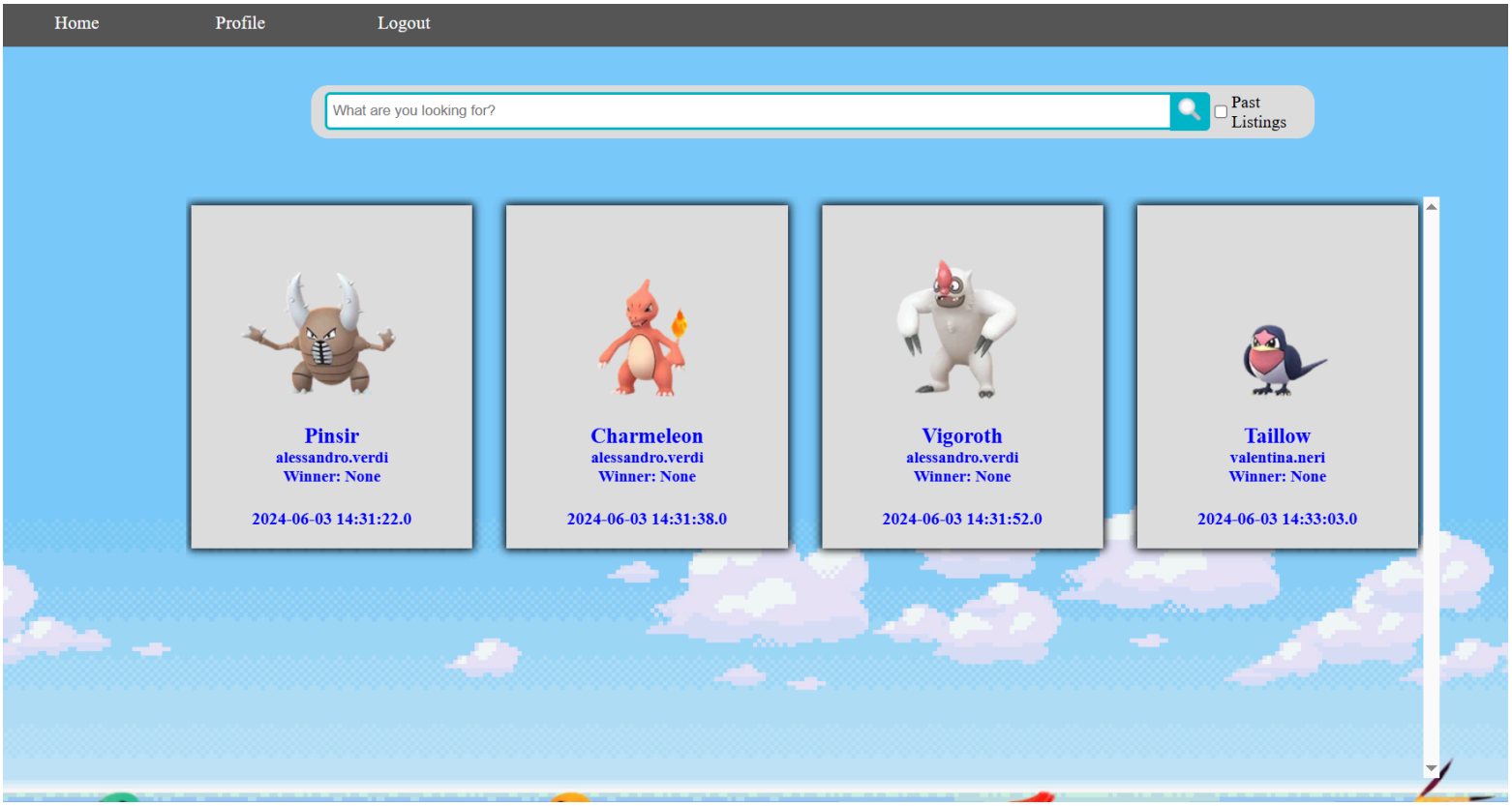


Figure 3.3 - Home page

3.4 Listing Page

It shows the listing information and any offers already present for the specific listing. If the user wants to make an offer, he can click on "**MAKE A OFFER**" and can select from all the pokemon he owns the ones he wants to offer. Once this is done, the "**Select**" button relating to the chosen pokemon will be eliminated and the "**Delete**" button will be displayed on the new offer to be able to delete it.

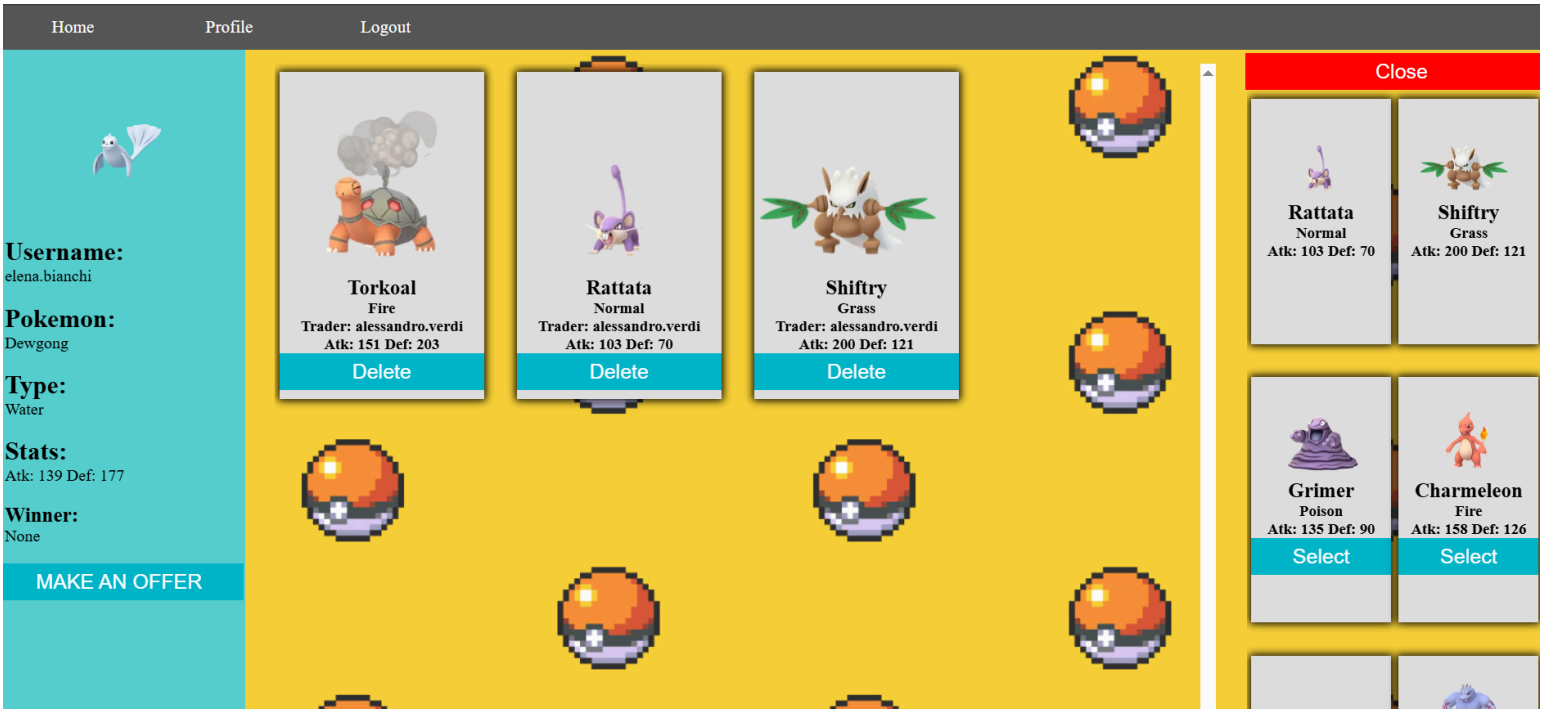


Figure 3.4 - Listing page

3.5 Profile Page

It shows all the user's personal information. Furthermore, depending on the section selected:

- **Listings**: it shows the active listings owned by the user, which he can select to carry out the trade, or even just to view current offers or delete them.
- **Box**: shows the owned pokemon. From here he can add a new listing.

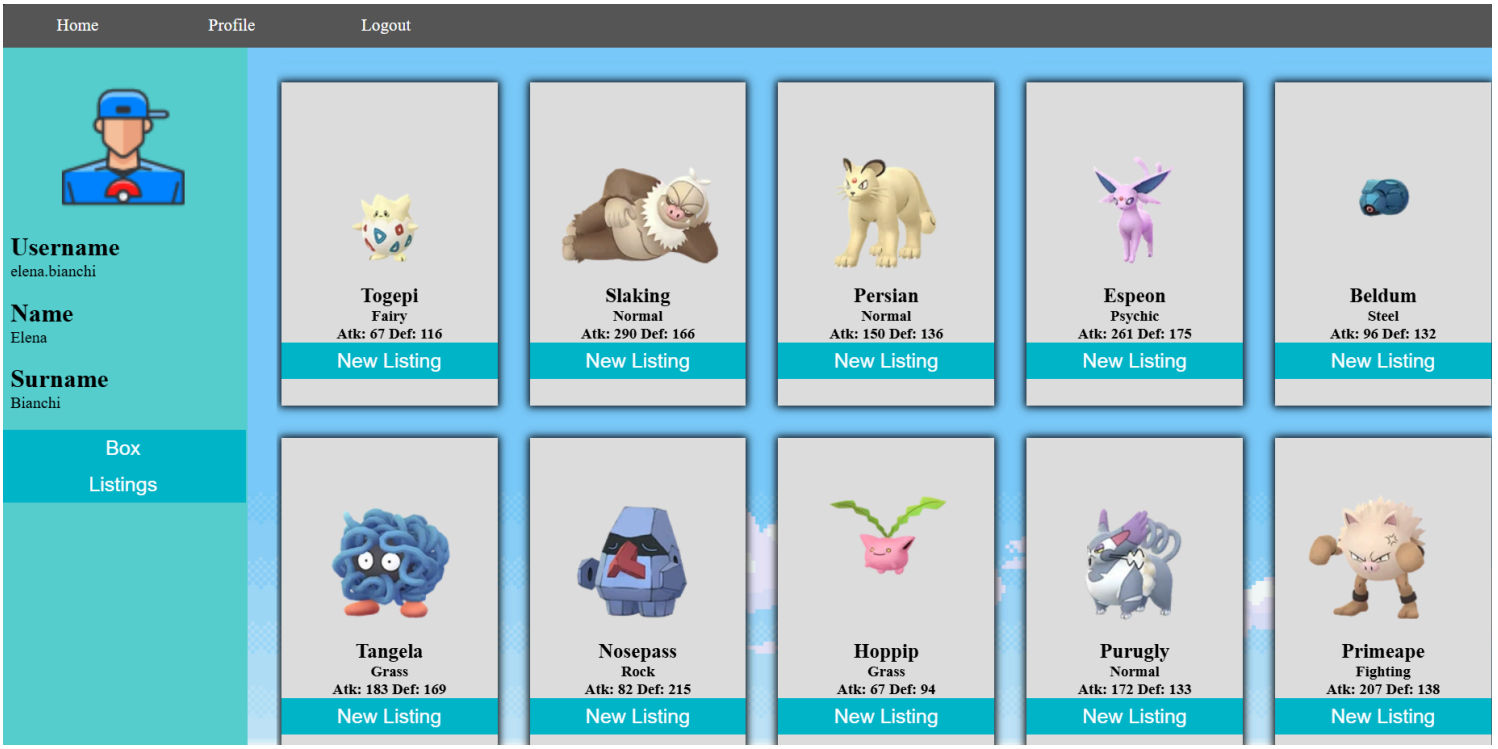


Figure 3.4 - Profile page Box section

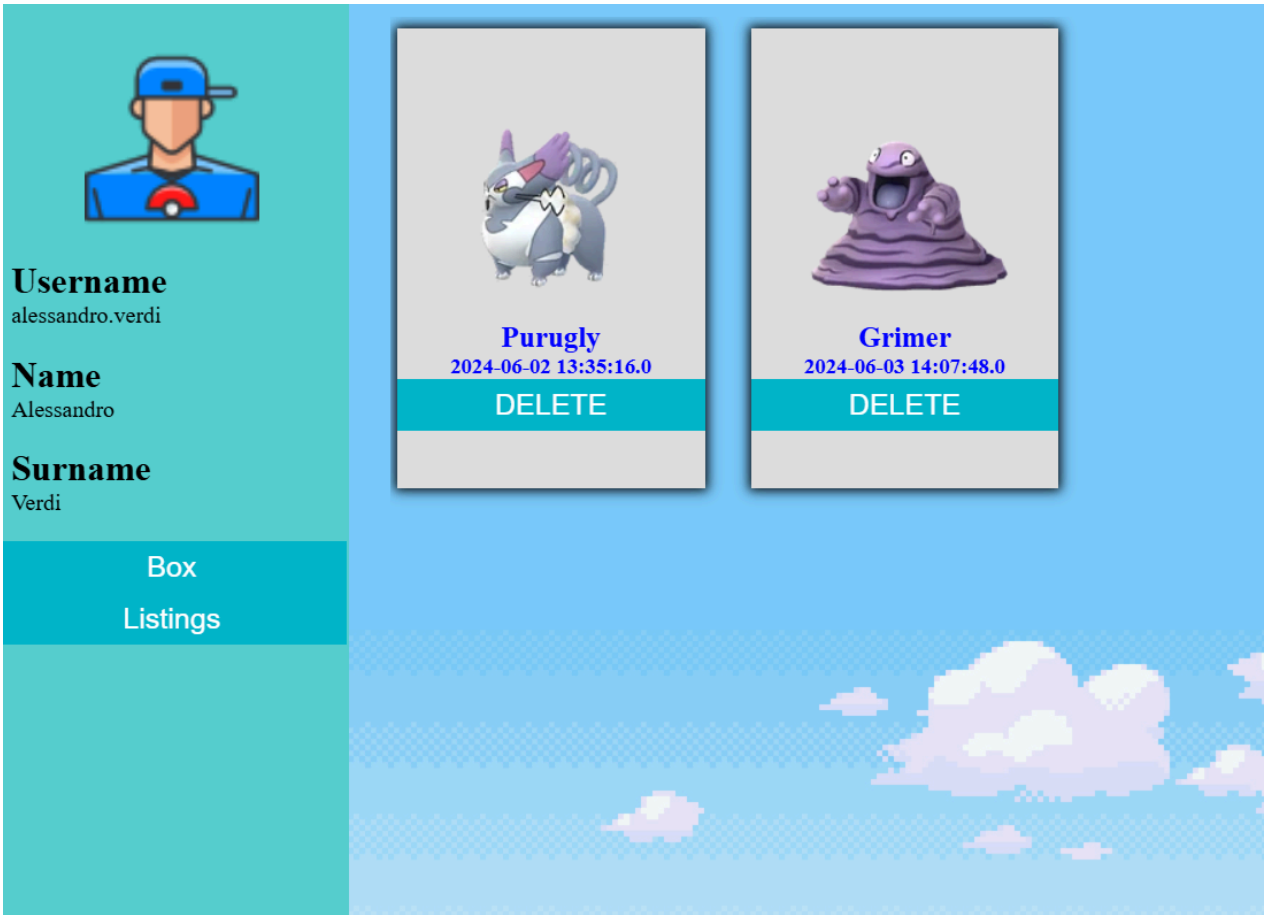


Figure 3.4 - Profile page Listings Section

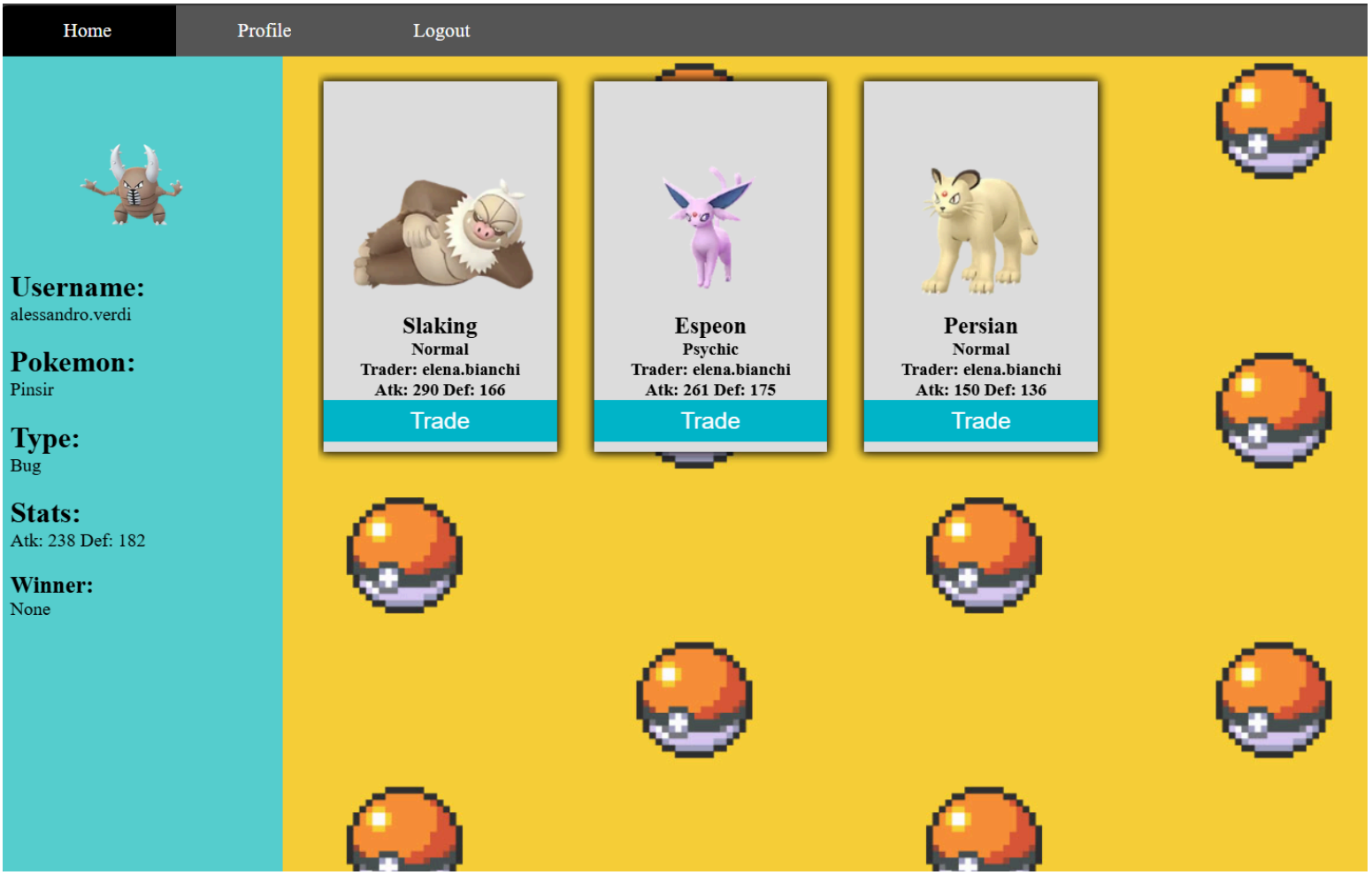


Figure 3.4 - Profile page Listings Section trade