Web Technologies Project @ PoliMi, 2025

Creating a Auction Site with HTML5 & Thymeleaf & JS & CSS

Sun Jinian
jinian.sun@mail.polimi.it
https://github.com/sun-jinian

Vittorio Robecchi
vittorio.robecchi@gmail.com
https://github.com/VictuarVi

Contents

Abstract

Overview The source code can be found on Github – for a web server that handles an auction management system. A user is able to register, log in, create an item, create an auction, and make an offer. Users can create an auction using items they have created.

There are two subprojects: a (pure) **HTML version**, which is structured as a series of separate webpages; and a **RIA version**, which is structured as a single-page webapp. The functionalities are quite the same, the code changes mostly at a frontend level.

Tools This project was built using the following technologies:

Java for the backend server, leveraging Jakarta' s Servlet API; **Apache Tomcat** to run the server; for the HTML version, **Thymeleaf**, a template engine; and for the RIA version, **JavaScript**.

We decided to use **MySQL** version 8.0

This document has been typeset with **Typst**. To create sequence diagrams we use `chronos` package.

Configuration & Running In order to run this project, the following packages and their respective versions are to be installed:

- Java JDK 23
- Apache Maven
- Apache Tomcat 10
- MySQL
- JDBC driver

The credentials are stored in plain text in the database, while the items' images are stored on the same disk where your Tomcat server is running.

1

Project submission breakdown

1.1 Database logic

Legend		
	Entity	**Attribute**
	Attribute specification	**Relationship**

Each **user** has a **id**, **username**, **password**, **first_name**, **last_name**, **address**. Each **auction** has **id**, **creator**, **title**, **start price**, **minimum increment**, **expiration**, **status**, **time of creation** and **file**. Furthermore:

- Suppose the id is auto-increment
- Suppose status of auction can be closed only on creator' s request, even expired
- Suppose user can make offer if and only if 2 conditions are met: auction is open & not expired
- One can not make an offer on his own auction
- There are no deletion in database after auction closed or items sold, only make it not visible

After the login, the user is able to **create items** by loading their data and then put them in an auction. A **auction** contains **a set of items**. A playlist has a **title**.

1.2 Behaviour

Legend		
	User action	**Server action**
	HTML page	**Page element**

After the login, the user **accesses** the **HOME PAGE** which has two link sublink that direct to **SELL** or **BUY**, **SELL displays** the **list of closed and open auctions** ordered by descending creation date and **list of items** created that are available, ; a **form to create item** and a **form to create a auction**. The auction form:

- **displays** the **list of open auction and closed auction** ordered by expiration date descending
- Allows to **select** one or more items

The item form:

- **displays** the **list of user items** available to put in auction

When a user **clicks** on a **auction id** in the **SELL**, the application **loads** the **DETTAGLIO**; It contains a **table of offers made by users** and a **block of information on this auction**.

- Every cell contains the offer' s id, user' name who made that offer, price they offered and when they made offer
- The offers are ordered by the time they made offer

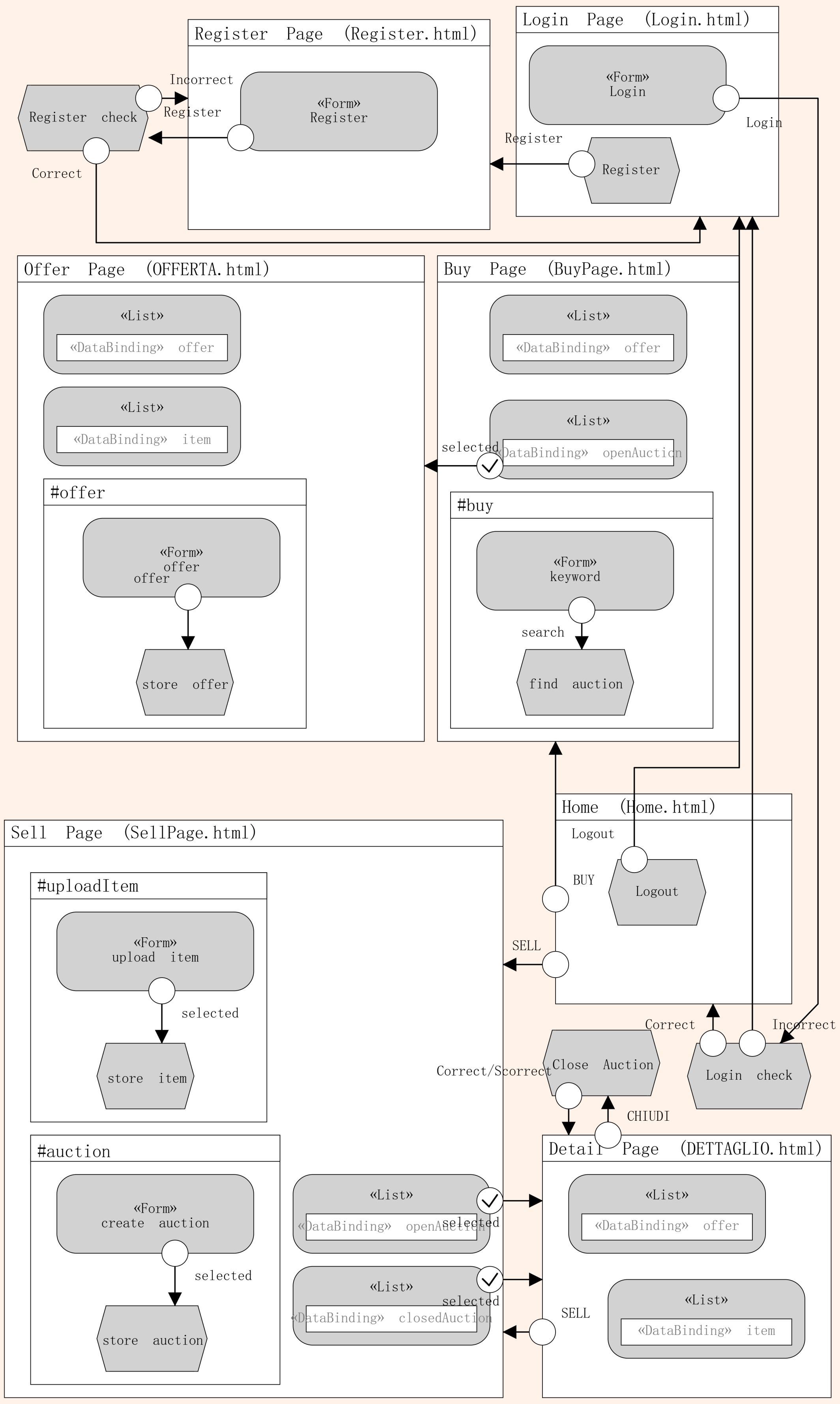


Figure 1: IFML diagram (HTML).

1.3 JavaScript version

Create a client-server web application that modifies the previous specification as follows:

- After the login, the entire application is built as a single webapp `// RIA`
- If the user accesses the application for the first time, it displays the content of the BUY page. If the user has already used the application, it displays the content of the SELL page if the user’s last action was the creation of an auction; otherwise, it displays the content of the BUY page with the list (possibly empty) of auctions that the user previously clicked on and that are still open.

  The information about the last action performed and the visited auctions is stored on the client side for a duration of one month.
- Every user interaction is handled without completely reloading the page, but instead triggers an asynchronous server call and, if necessary, updates only the content that needs to be refreshed as a result of the event.

Saving history The application must allow the user to save last action and all historical visit locally. From the **LOGIN VIEW**, the user is able to **access** a either **BUY VIEW** if last action is NOT **creation of an auction**, otherwise user **access** **SELL VIEW**, every time the user **visit** **OFFERTA VIEW**, it will save a history locally for 30 days.

Databse remains the same as before.

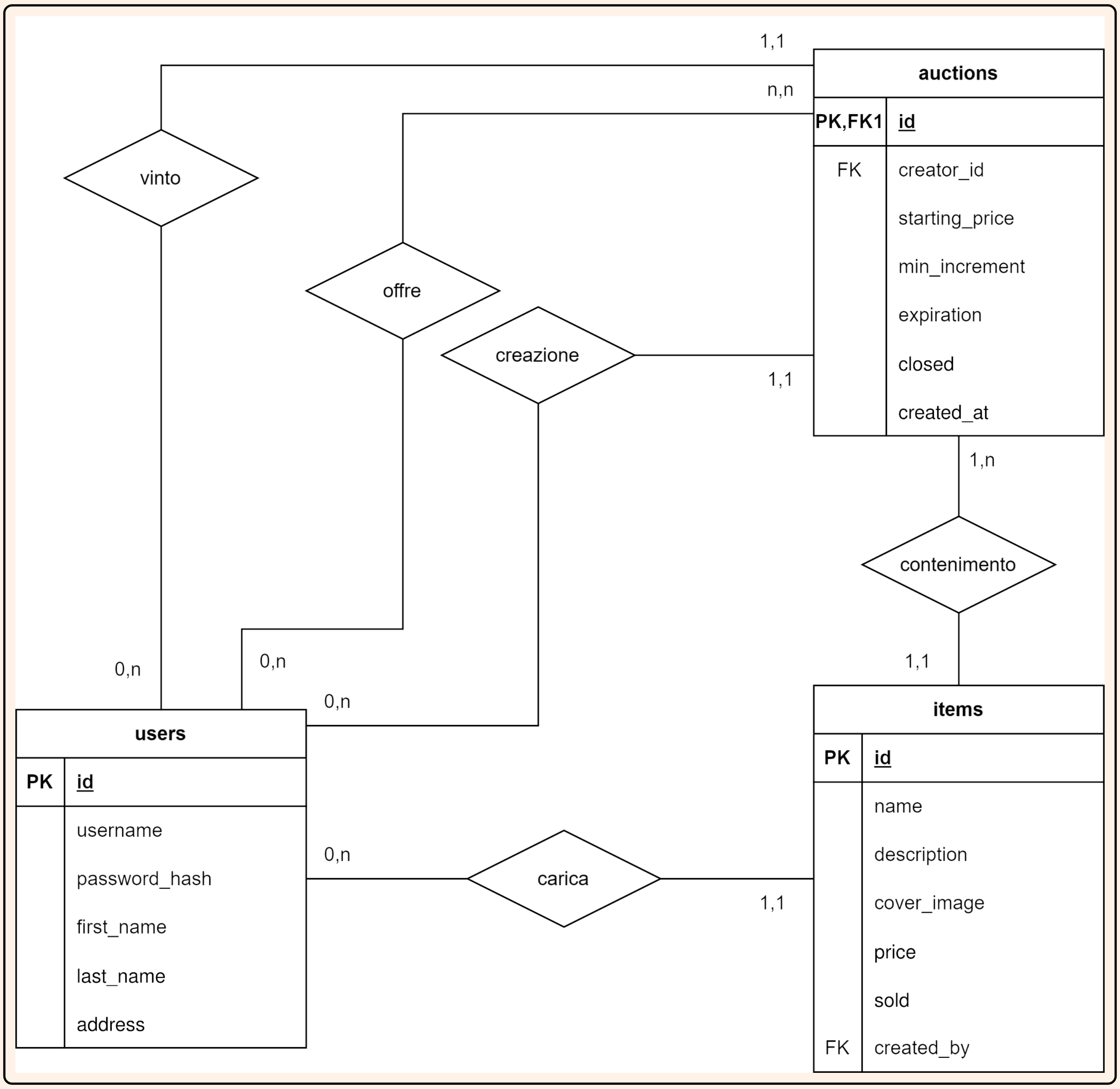


Figure 2: ER diagram (HTML).

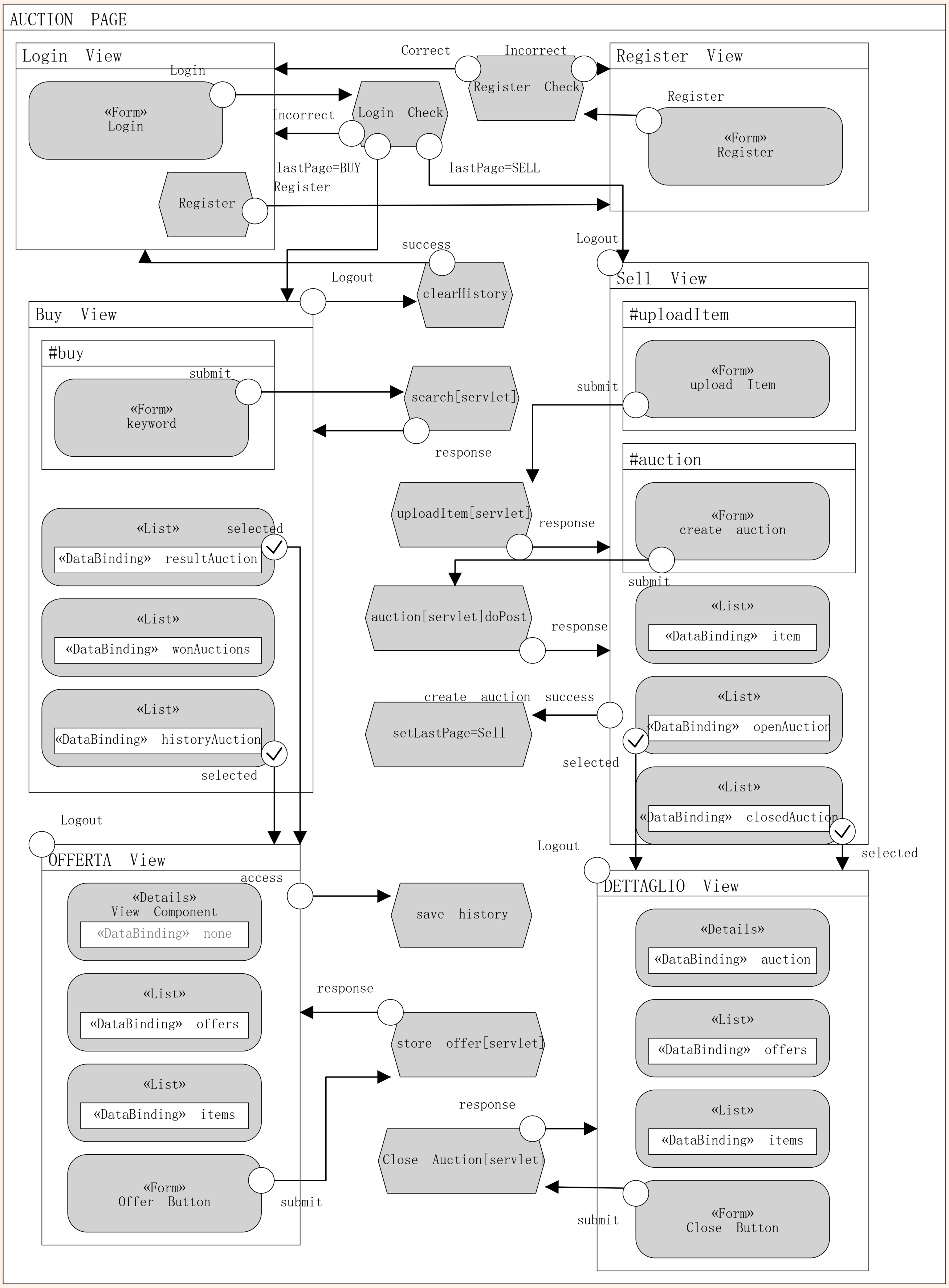


Figure 3: IFML diagram (RIA).

2

Specifications completion

2.1 HTML-specific features

In addition to the requirements, we implemented a series of new features:

- logout button in Login.html
- The top navigation bar, where the username is shown and buttons to relocate are located
- Success and error messages for different operations

In the traditional HTML-based application, every user action results in a full page reload. This means that whenever the user interacts with the system—such as navigating to another view, submitting a form, or clicking a button—the browser either redirects to a different page or reloads the current one.

To ensure that users see the most up-to-date information, especially in a competitive environment like auctions, certain pages are configured with an automatic refresh timer. In particular, the BuyPage, DetailPage, and OfferPage are refreshed every 5 seconds. This periodic reloading guarantees that users always receive the latest information about active offers and auction expiration times. However, while this approach ensures data consistency, it comes at the cost of performance overhead and user experience disruption, since each refresh reloads the entire page including static content that has not changed. Every action from user will be redirected to another page, or same page but reloading it.

2.2 RIA-specific features

In contrast, within a Rich Internet Application (RIA), we eliminate the reliance on page-level refresh timers. Instead of reloading the entire page, the application uses partial updates (e.g., via asynchronous requests) to keep the interface synchronized with the latest server state.

For example, when a user places a new offer, the system automatically reloads only the offer table rather than refreshing the entire page. This ensures that the most recent bids from all participants are displayed without unnecessary re-rendering of unrelated content. Similarly, other data tables—such as auction listings, user history, or closed auction results—are refreshed individually only when needed.

This RIA approach significantly improves responsiveness, reduces bandwidth consumption, and provides a smoother user experience, since users are no longer interrupted by constant full-page reloads. It also better reflects real-time interactions, which is particularly valuable in time-sensitive scenarios like online auctions.

3

SQL database schema

3.1 Overview

The project requirements slightly change from `pure_html` and `js`, where the latter requires the tracks to support an individual custom order within the playlist to which they are associated – this is achieved via a simple addition in the SQL tables schema.

In both scenarios, the schema is composed by four tables: `user`, `track`, `playlist` and `playlist_tracks`.

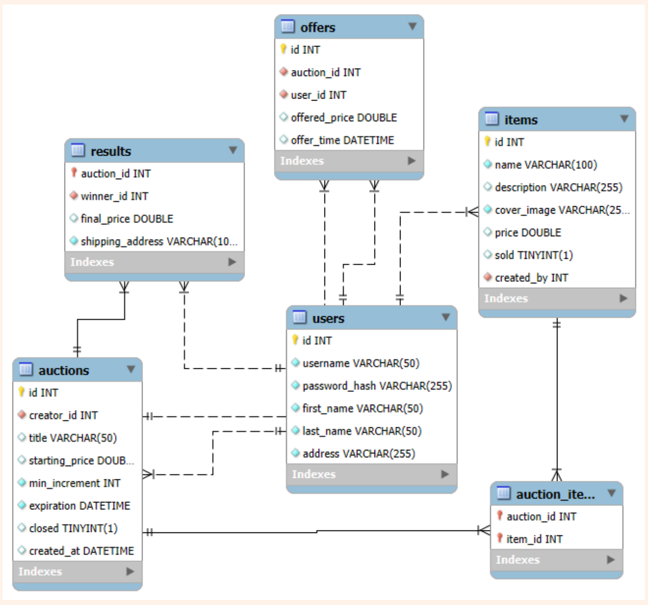


Figure 4: UML diagram.

3.2 The tables

- `users` table

```
CREATE TABLE IF NOT EXISTS users (
  id INT PRIMARY KEY AUTO_INCREMENT,
  username VARCHAR(50) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50) NOT NULL,
  address VARCHAR(255) NOT NULL
);
```

This table is straightforward and standard.

- The `id` attribute is the primary key, ensuring each user is uniquely identified.
- The `username` attribute has a unique constraint to prevent duplicate usernames.
- All other attributes (`password_hash`, `first_name`, `last_name`, `address`) are required but do not have uniqueness constraints.
- There is no composite primary key because each user must have a unique `id`, and uniqueness for the username is enforced separately.
- `items` table

```
CREATE TABLE IF NOT EXISTS items (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    description VARCHAR(255),
    cover_image VARCHAR(255) NOT NULL,
    price DOUBLE DEFAULT 0,
    sold BOOLEAN DEFAULT FALSE,
    created_by INT NOT NULL,
    FOREIGN KEY (created_by)
        REFERENCES users (id)
        ON DELETE CASCADE ON UPDATE CASCADE
);
```

This table stores all items sold and not sold.

- `id` is the primary key.
- `created_by` is a foreign key referencing `users.id`, ensuring each item is strictly associated with a user.
- Attributes `name`, `description`, `price`, and `sold` are standard.
- `cover_image` contains the relative path to the image file. The images are physically stored on disk under `\opt\auction\uploads`
- `auctions` table

```
CREATE TABLE IF NOT EXISTS auctions (
      id INT AUTO_INCREMENT PRIMARY KEY,
      creator_id INT NOT NULL,
      starting_price DOUBLE NOT NULL,
      min_increment INT NOT NULL,
      expiration DATETIME NOT NULL,
      closed BOOLEAN DEFAULT FALSE,
      created_at DATETIME DEFAULT
CURRENT_TIMESTAMP,
        FOREIGN KEY (creator_id) REFERENCES
users(id)
);
```

This table stores auctions created by users.

- `id` is the primary key.
- `creator_id` is a foreign key referencing `users.id`, ensuring each auction is associated with a user.
- Attributes `starting_price`, `min_increment`, `expiration`, `closed`, and `created_at` are standard.
- `created_at` defaults to the current timestamp when the auction is created.
- `closed` indicates whether the auction has ended, defaulting to FALSE.
- `auction_items` table

```
CREATE TABLE IF NOT EXISTS auction_items (
        auction_id INT,
        item_id INT,
        PRIMARY KEY (auction_id, item_id),
        FOREIGN KEY (auction_id) REFERENCES
auctions(id) ON DELETE CASCADE,
        FOREIGN KEY (item_id) REFERENCES
items(id) ON DELETE CASCADE
);
```

This table represents the many-to-many relationship between auctions and items.

- `auction_id` is a foreign key referencing `auctions.id` with ON DELETE CASCADE.
- `item_id` is a foreign key referencing `items.id` with ON DELETE CASCADE.
- The primary key is a composite of `(auction_id, item_id)` to ensure that each item appears only once in a given auction.
- `offers` table

```
CREATE TABLE IF NOT EXISTS offers (
        id INT AUTO_INCREMENT PRIMARY KEY,
        auction_id INT NOT NULL,
        user_id INT NOT NULL,
        offered_price DOUBLE NOT NULL,
        offer_time DATETIME DEFAULT
CURRENT_TIMESTAMP,
        FOREIGN KEY (auction_id) REFERENCES
auctions(id) ON DELETE CASCADE,
        FOREIGN KEY (user_id) REFERENCES
users(id)
);
```

This table stores bid records for auctions.

- The `id` attribute is the primary key, uniquely identifying each offer.
- The `auction_id` attribute is a foreign key referencing `auctions.id`, linking the offer to its auction and using ON DELETE CASCADE.
- The `user_id` attribute is a foreign key referencing `users.id`, linking the offer to the bidding user.
- The `offered_price` attribute stores the bid amount.
- The `offer_time` attribute records the time the offer was made, defaulting to the current timestamp.
-
- `results` table

```
CREATE TABLE results (
                                auction_id INT
PRIMARY KEY,
                                winner_id INT
NOT NULL,
                                final_price DOUBLE
NOT NULL,
                                shipping_address
VARCHAR(100) NOT NULL,
                                FOREIGN KEY
(auction_id) REFERENCES auctions(id),
                                FOREIGN KEY
(winner_id) REFERENCES users(id)
);
```

This table stores the results of auctions after closure.

- The `auction_id` attribute is the primary key, linking the result to a specific auction.
- The `winner_id` attribute is a foreign key referencing `users.id`, indicating the user who won the auction.

- The `final_price` attribute stores the final winning bid amount.
- The `shipping_address` attribute stores the shipping address for the auction item.

4

Codebase overview

4.1 Components

Introduction The projects is built upon the following components:

1. DAOs
   - AuctionDAO
   - ItemDAO
   - UserDAO

1. Entities
   - Auction
   - Item
   - User
   - Offer
   - ClosedAuction
   - OpenAuction
   - Result

These are coded following the JavaBeans model.

1. Servlets
   - AuctionServlet
   - BuyServlet
   - CloseServlet
   - HomeServlet
   - LoginServlet
   - OfferServlet
   - RegisterServlet
   - SellServlet
   - UploadItemServlet
2. Utils (short-term for Utilities)
   - DBUtils
   - Util
   - ThymeleafConfig

4.2 DAOs methods

AuctionDAO methods:

- `getResultsByAuction(List<Auction> auctions)`
- `insertOffer(int userId, int auctionId, double offeredPrice)`
- `getMaxOfferOfAuction(int auctionId)`
- `getMaxOffersByAuction(List<Auction> auctions)`
- `allItemsByAuction(List<Auction> auctions)`
- `findAllAuctionsNotClosed(int userId)`
- `findAllOpenAuction(int userId)`
- `findAllClosedAuctionsAndResult(int userId)`
- `findAllAuctionClosed(int userId)`
- `findById(int auctionId)`
- `findAllOffersByAuction(int auctionId)`
- `closeAuction(int auctionId)`
- `findUserAddressById(int userId)`
- `updateResult(int auctionId)`
- `createAuction(int userId, String title, double startingPrice, int minIncrement, LocalDateTime ending_at)`
- `insertItems(int auctionId, int[] items)`
- `findAllOpenAuctionByKeywords(String[] keywords)`
- `findAllWonAuctions(int userId)`
- `find_open_historical_auctions(String[] visitedAuctions)`

ItemDAO methods:

- `uploadItem(String name, String description, String file_path, double price, int user_id)`
- `findAllItemInAuction(int auction_id)`
- `findAllItemNotInAuction(int userId)`
- `sellItemInAuction(int auction_id)`
- `calculateTotalPrice(int[] itemIds)`

UserDAO methods:

- `createUser(String username, String password, String first_name, String last_name, String address)`
- `login(String username, String password)`
- `findByUsername(String username)`
- `mapRowToUser(ResultSet rs)`
- `findNameById(int id)`

4.3 RIA subproject

One single javascript,in which exists following fucntion, everything will be initiallized after page is loaded.

- `showRegisterLink()` — Load form for regitration
- `loginLink()` — Load form for login
- `showPage(page, auctionId)` — Show the page based on the page parameter and auctionId(optional)
- `saveHistory(auctionId)` — Save a history of a auction in the localStorage
- `showError(message)` — Load error messages on the top nav bar
- `showMessage(message)` — Load notification messages on the top nav bar
- `getBuyPageInfo()` — load search history table, won auction table
- `getSellPageInfo()` — Load open auction table, closed auction table, available item table, set minimum endDate
- `getOfferPageInfo()` — Load open auction table, closed auction table, available item table, set minimum offer
- `getDetailPageInfo(auctionId)` — Load detailed information of an auction and its offer table
- `login()` — Load last page visited after successful login
- `register()` — Load login page after successful registration
- `uploadItem()` — Load newly uploaded item to the available item table after successful upload
- `createAuction()` — Load newly created auction to the open auction table after successful creation
- `offer()` — Load renewed offer table in OFFERTA.html after successful offer
- `logout()` — Load loginView and invalidate session

Utils:

- `formatDate(rawDateObj)` — Format a LocalDateTime object to a string in the format "YYYY-MM-DD HH:mm:ss"
- `verifyInputs(username, password, firstName, lastName, address)` — verify if the register inputs are valid

And finally the interfaces, which are the Typescript translation of the Record classes.

Client side		Server side	
Event	**Action**	**Event**	**Action**
LoginView ⇒ Login form ⇒ Submit	Data validation	POST (`username`, `password`)	Credentials check
LoginView ⇒ LastPage = 'sell_page'	getSellPageInfo()	GET (`openAuctions`, `closedAuctions`, `availableItems`)	Queries openAuctions, closedAuctions, availableItems
LoginView ⇒ LastPage = 'buy_page'	getBuyPageInfo()	GET (`wonAuctions`, `historyAuctions`)	Queries wonAuctions, historyAuctions
BuyView ⇒ Search button	load search result table	GET (`resultAuctions`)	Queries resultAuctions
BuyView ⇒ Click on an auction	load OfferView	GET (`auction`, `offers`, `items`)	Queries auction, offers, items
OfferView ⇒ Offer form ⇒ offer	load offer table	POST (`offeredPrice`)	Insert offer in the offers table
SellView ⇒ Click on an auction	load DetailView	GET (`auction`, `offers`, `items`)	Queries auction, offers, items
SellView ⇒ Auction form ⇒ Submit	load openAuctions table	POST (`title`, `startingPrice`, `minIncrement`, `ending_at`)	Insert auction in the auction table
SellView ⇒ Item form ⇒ Submit	load availableItems table	POST (`name`, `description`, `image`, `price`)	Insert item in the items table
DetailView ⇒ Click on close button	change auction status to closed	POST (`auctionId`)	Check if request is valid and update the auctions table
navbar ⇒ Sell Button	load SellView	GET (`openAuctions`, `closedAuctions`, `availableItems`)	Queries openAuctions, closedAuctions, availableItems
navbar ⇒ Buy Button	load BuyView	GET (`wonAuctions`, `historyAuctions`)	Queries wonAuctions, historyAuctions
navbar ⇒ Logout Button	load LoginView and clear localStorage	POST(`logout`)	invalidate session

Table 1: Events & Actions.

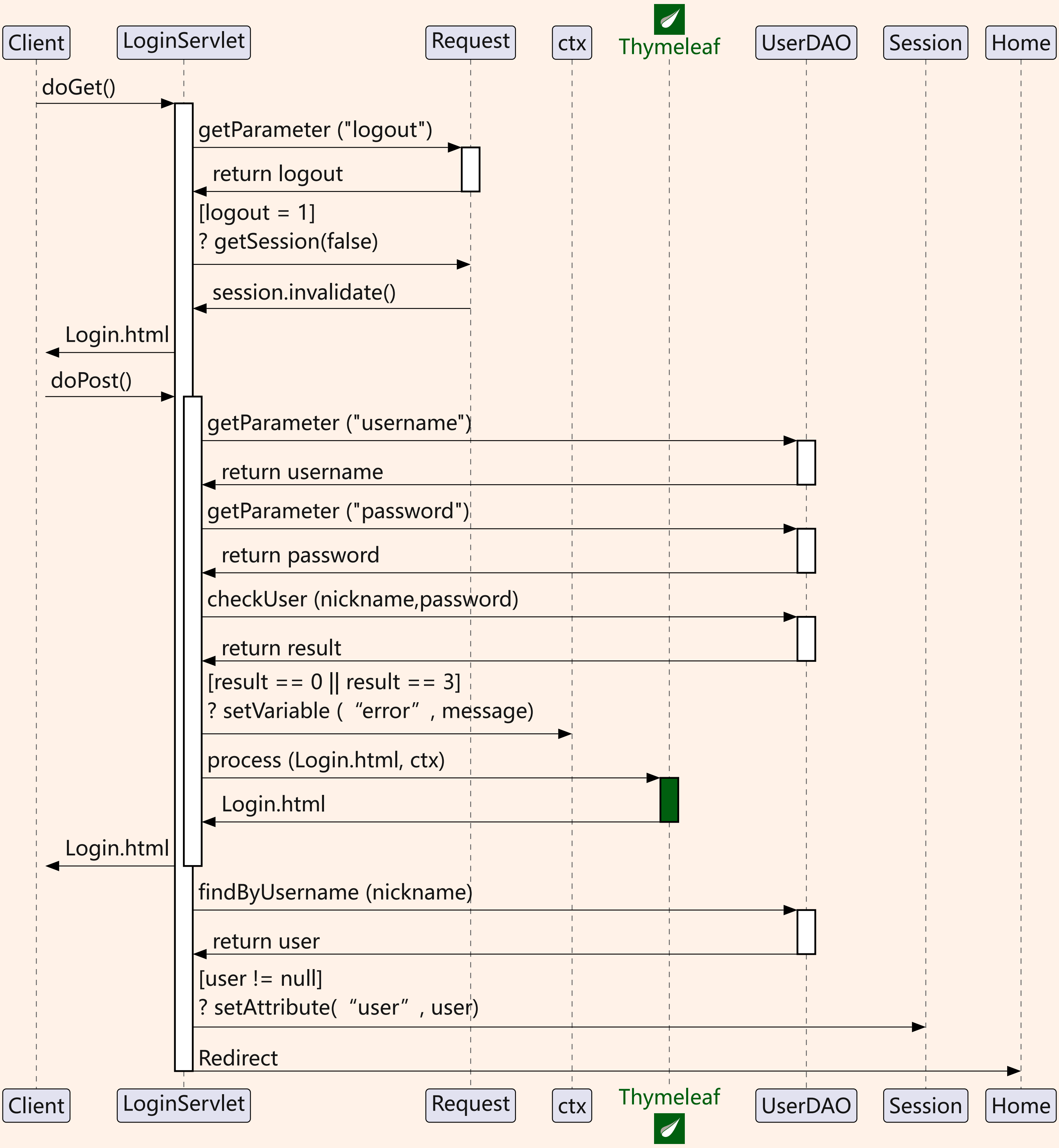
Client side		Server side	
Event	**Controller**	**Event**	**Controller**
LoginView ⇒ Login form ⇒ Submit	login()	POST (`username`, `password`)	loginServlet
LoginView ⇒ LastPage = 'sell_page'	getSellPageInfo()	GET (`openAuctions`, `closedAuctions`, `availableItems`)	SellServlet
LoginView ⇒ LastPage = 'buy_page'	getBuyPageInfo()	GET (`wonAuctions`, `historyAuctions`)	BuyServlet
BuyView ⇒ Search button	search()	GET (`resultAuctions`)	BuyServlet
BuyView ⇒ Click on an auction	getOfferPageInfo()	GET (`auction`, `offers`, `items`)	OfferServlet
OfferView ⇒ Offer form ⇒ offer	offer()	POST (`offeredPrice`)	OfferServlet
SellView ⇒ Click on an auction	getDetailPageInfo()	GET (`auction`, `offers`, `items`)	AuctionServlet
SellView ⇒ Auction form ⇒ Submit	createAuction()	POST (`title`, `startingPrice`, `minIncrement`, `ending_at`)	AuctionServlet
SellView ⇒ Item form ⇒ Submit	uploadItem()	POST (`name`, `description`, `image`, `price`)	uploadItemServlet
DetailView ⇒ Click on close button	closeAuction()	POST (`auctionId`)	CloseServlet
navbar ⇒ Sell Button	getSellPageInfo()	GET (`openAuctions`, `closedAuctions`, `availableItems`)	SellServlet
navbar ⇒ Buy Button	getBuyPageInfo()	GET (`wonAuctions`, `historyAuctions`)	BuyServlet
navbar ⇒ Logout Button	logout()	GET(`logout`)	LoginServlet

Table 2: Events & Controllers (or event handlers).

5

Sequence diagrams

5.1 LoginServlet sequence diagram



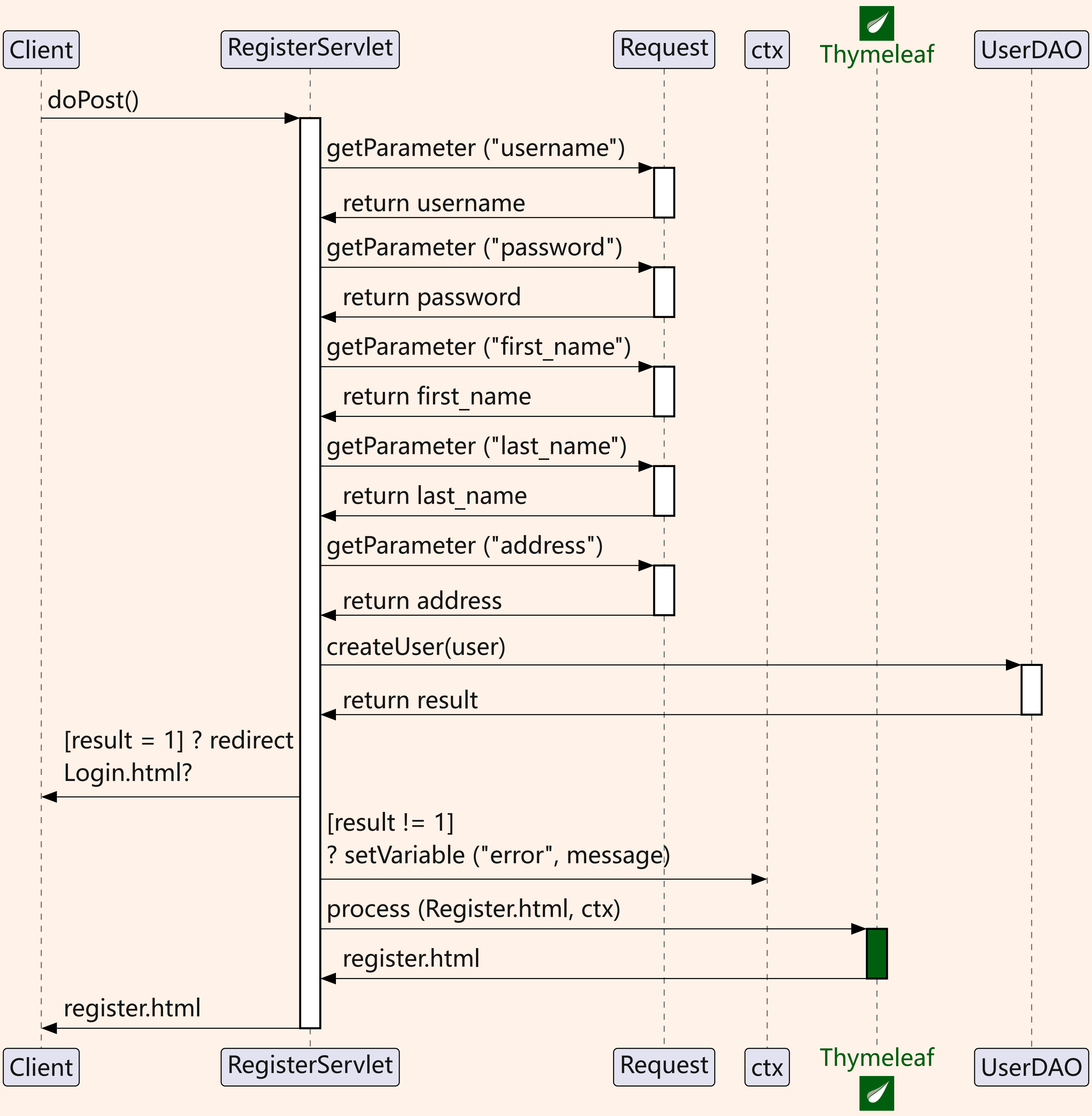
Comment

Upon first doGet() request from Client, server redirect it to static Login page.Afterwards, the User inserts their credentials. Those values are sent to LoginServlet via doPost(), and then passed to the `checkUser()` function that return an integer,0 if password is wrong, 3 if username not found, 1 if all goes well, then `findByUsername()` returns `user`, then Client is redirected to their Home and the `user` variable is set for the current session.

If there has been some error in the process – the credentials are incorrect, database can' t be accessed… – then the servlet will set context variable `error` to `message`, which then will be process by thymeleaf engine and print a Login.html with `message`.

If user click on logout button, it will send a `doGet()` with parameter `logout` = 1, then the current session will be invalidated.

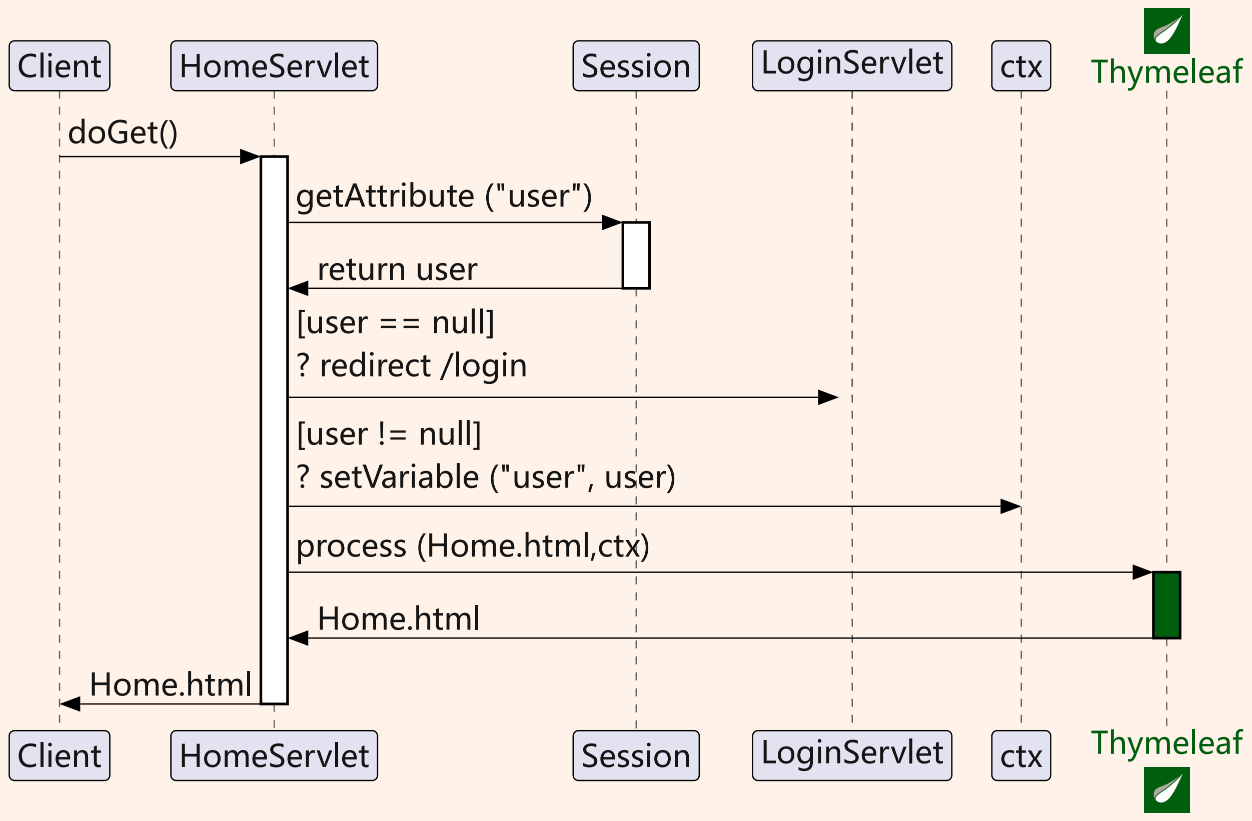
5.2 RegisterServlet sequence diagram



Comment

In Login.html, user can be directed to a initially static Register.html page, as per the Login sequence diagram, once all the parameters are gathered and verified (omitted for simplicity), if registration is successful, Client will be redirected to a static Login.html, if operation fails, there can’ t be two Users with the same useranme. If that happen, then *`createUser()`* returns *`0`* and then the servlet will set context variable *`error`* to *`message`*, which then will be process by thymeleaf engine and print a Register.html with *`message`*, same goes for other type of errors (omitted for simplicity).

5.3 HomeServlet sequence diagram

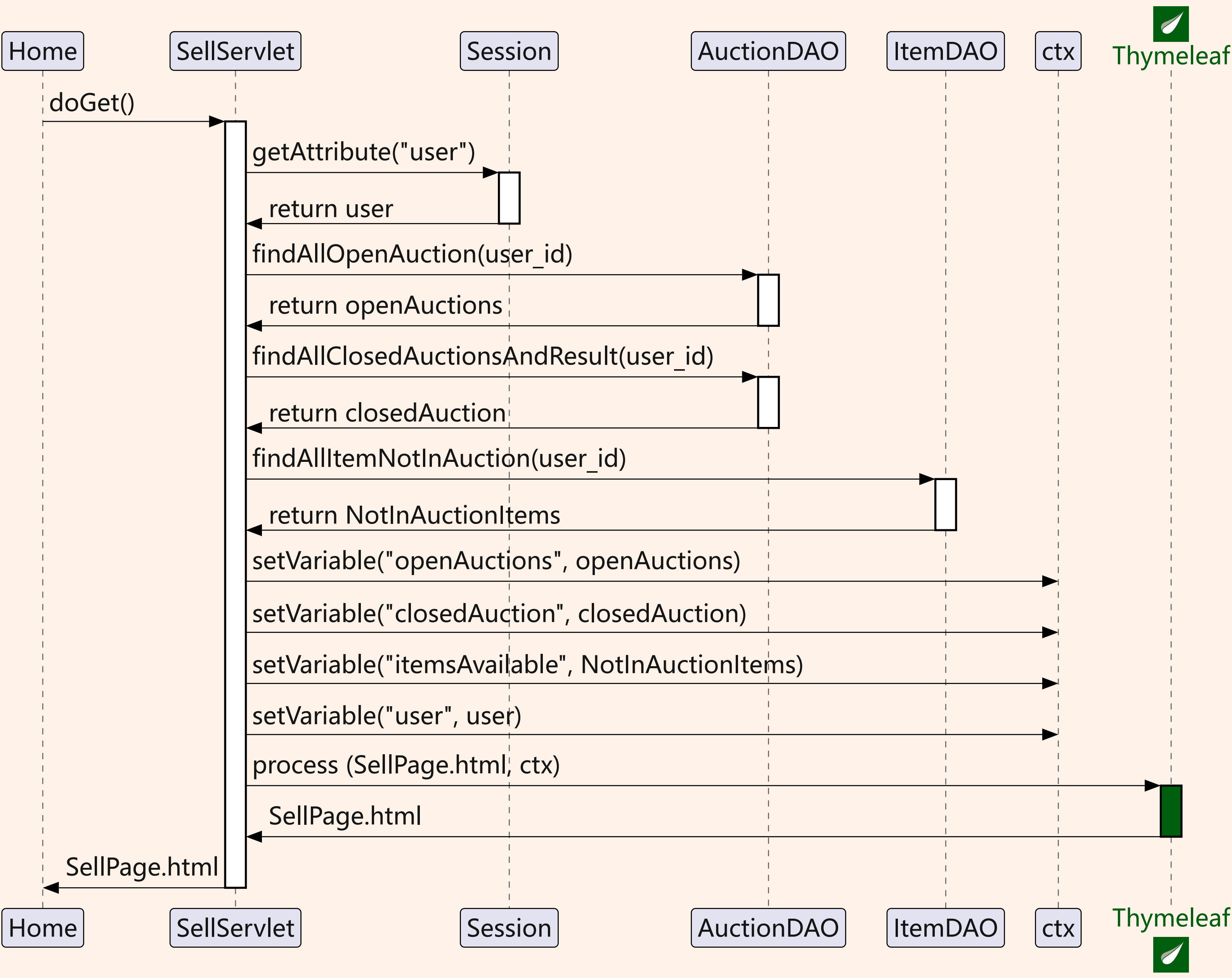


Comment

Once the Login is complete, the User is redirected to their HomePage, which only has 2 button, *`SELL`* and *`BUY`*, first is handled by *`SellServlet`*, second handled by *`BuyServlet`*.

From now on, verify user’ s session will be omitted since it is the same for every Servlet, unless it has another use.

5.4 SellServlet sequence diagram


Home
SellServlet
Session
AuctionDAO
ItemDAO
ctx
Thymeleaf
doGet()
getAttribute("user")
return user
findAllOpenAuction(user_id)
return openAuctions
findAllClosedAuctionsAndResult(user_id)
return closedAuction
findAllItemNotInAuction(user_id)
return NotInAuctionItems
setVariable("openAuctions", openAuctions)
setVariable("closedAuction", closedAuction)
setVariable("itemsAvailable", NotInAuctionItems)
setVariable("user", user)
process (SellPage.html, ctx)
SellPage.html
SellPage.html

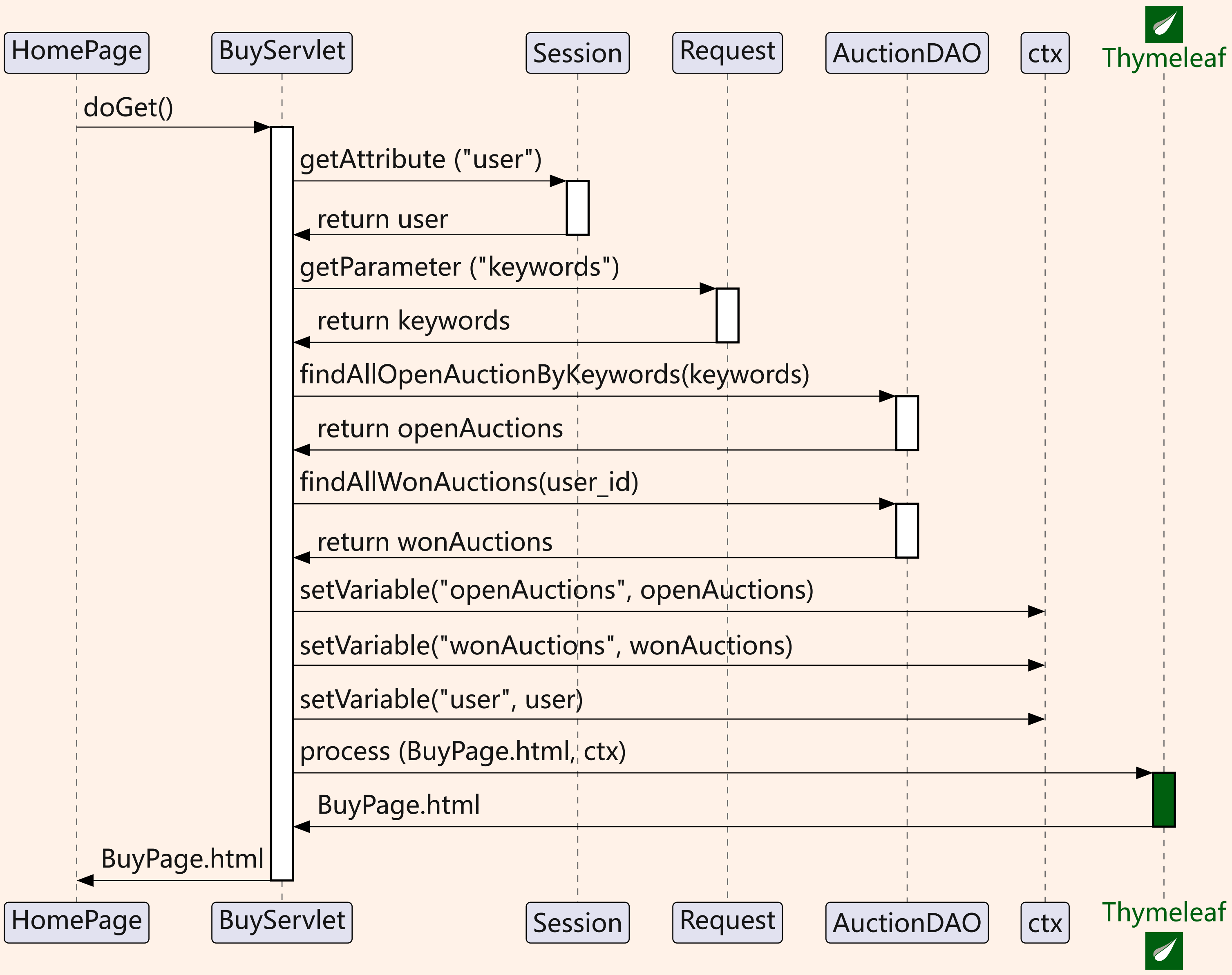
Comment

From the HomePage, by clicking on the link, the site redirects to the `SellPage`, which lists all the auctions associated to that user, also all items that can be added to an auction.

In order to do so, the program needs the user attribute – which is again retrieved via the session, user attribute contains all information.

Then user id are passed to the `findAllOpenAuction()`, `findAllClosedAuctionsAndResult()`, `findAllItemNotInAuction()` method, that returns all the auctions and sellable items. Finally, thymeleaf processes the context and display the `SellPage`.

5.5 BuyServlet sequence diagram



Comment

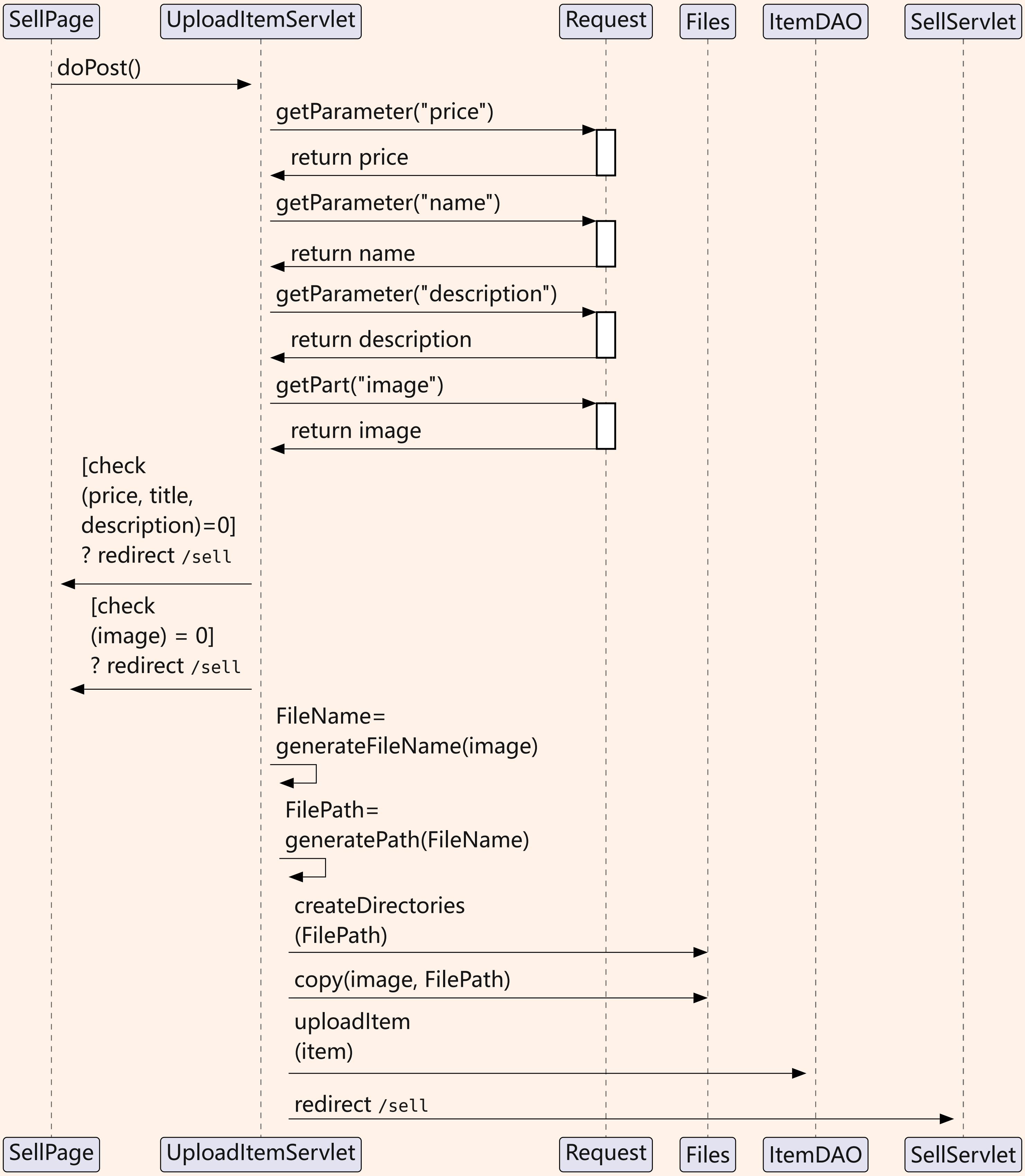
From the HomePage, by clicking on the link, the site redirects to the `BuyPage`, which lists all the auctions won by the user.

On the BuyPage, there is also a form where the user can enter keywords and submit them to the server to search for auctions containing items with at least one word matching the keywords.

In order to do so, the program needs the user attribute, parameter keywords is passed with request.

Then keywords are passed to the `findAllOpenAuctionByKeywords()` method, the user id is passed to `findAllWonAuctions()` method, then the result set of auctions and won auction are save in context, thymeleaf processes the context and display the `BuyPage`.

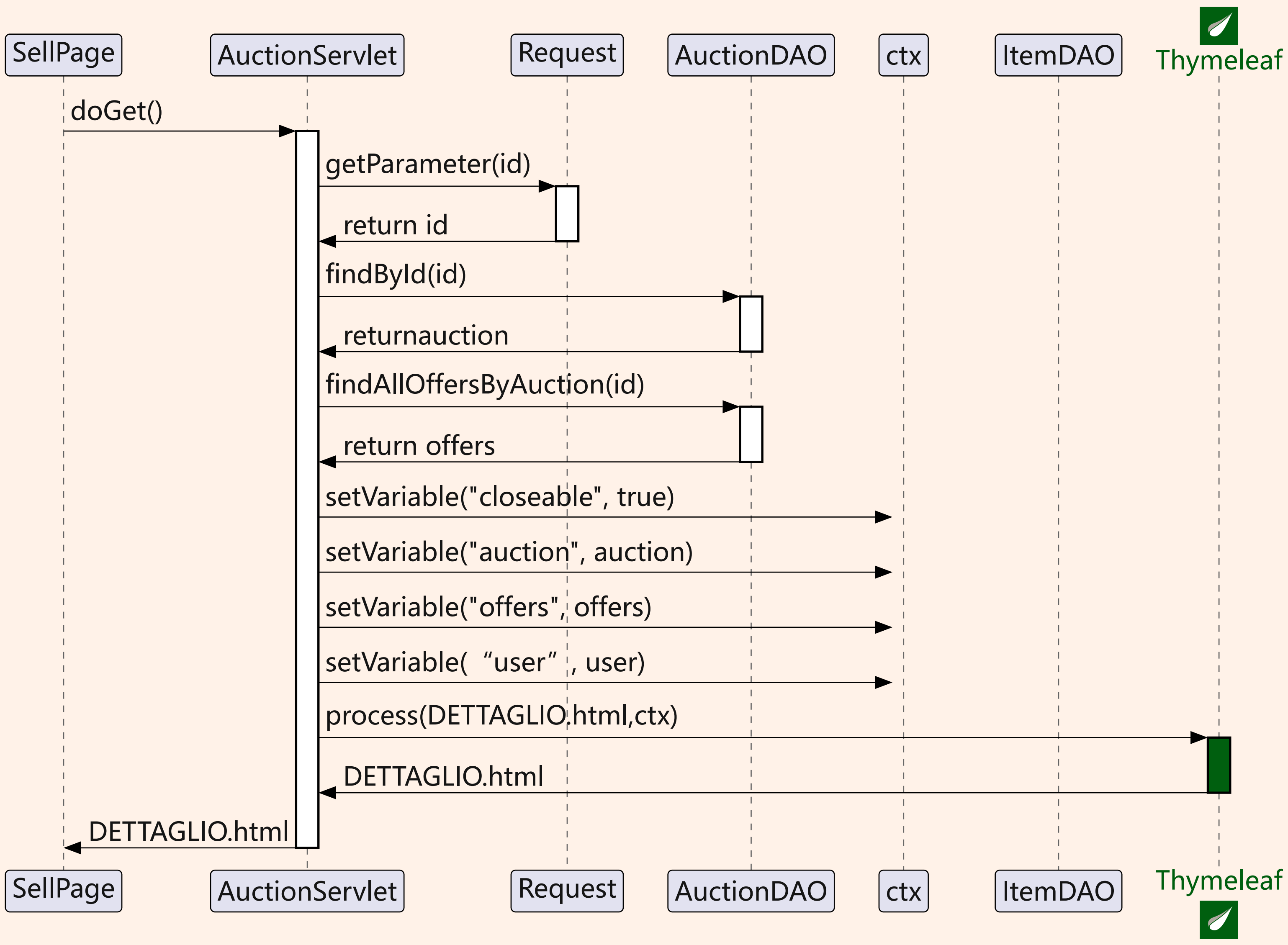
5.6 UploadItemServlet sequence diagram



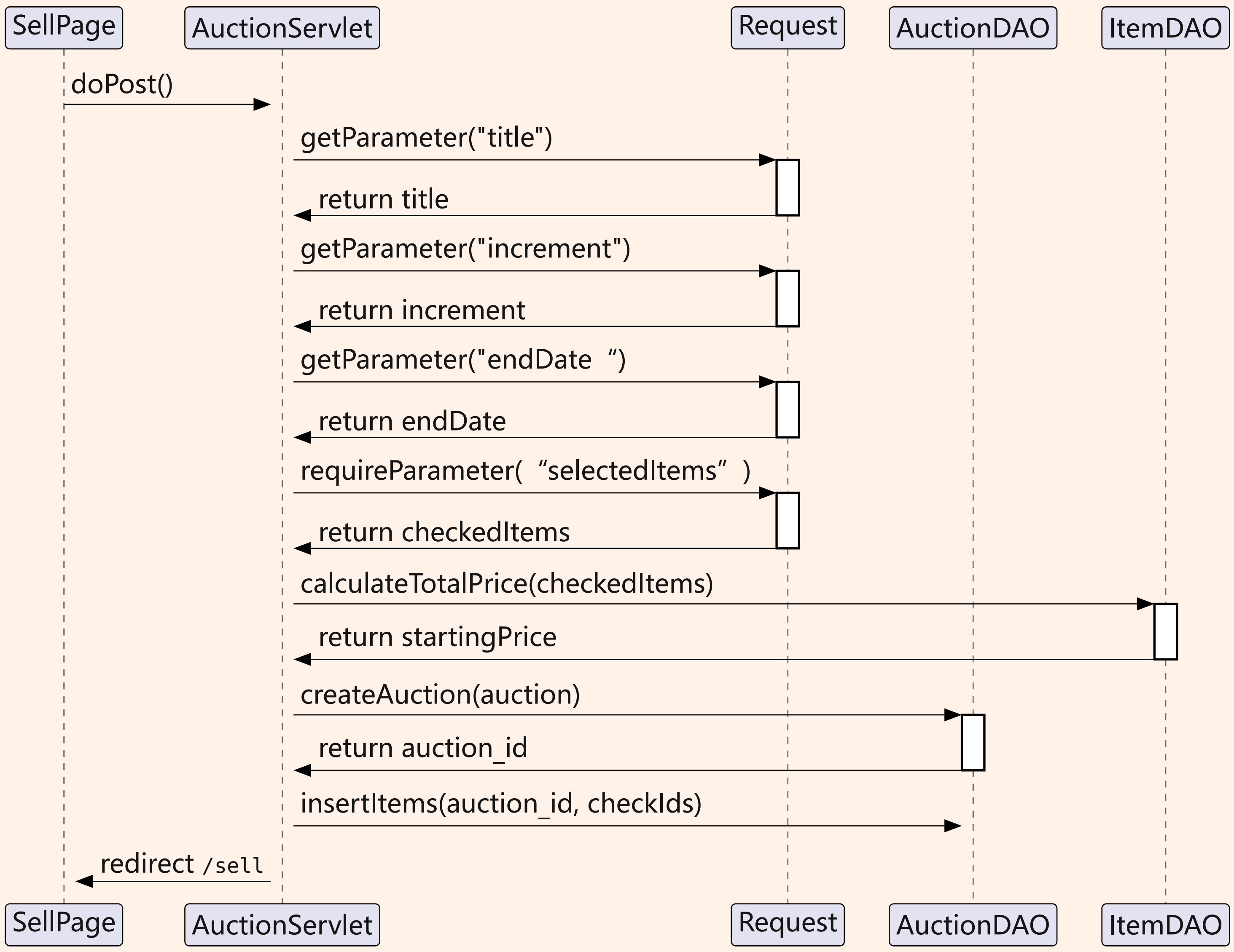
Comment

The User can upload item from the appropriate form in the SellPage (Section 5.4). When the POST request is received, the parameters are checked for null values and emptiness. Image will be renamed with generateSafeFileName(image) to a random UUID-based file name, which is 36 characters long (including dashes), plus the file extension. Before writing the file to disk, the method `createDirectories(relativeFilePath)` create a subdirectory for the user, hence each user has one directory contain all the images. Image are written to disk by the `copy(image.getInputStream(), relativeFilePath)()` method, which has two parameters: a Part object, that was received within a multipart/form-data POST request, and a Path object. Once this is completed, item' s information is passed to `uploadItem(name, description, relativeFilePath, price, user_id)` method of ItemDAO. If anything goes wrong, image will be deleted with `Files.deleteIfExists(relativeFilePath)` (omitted).

5.7 AuctionServlet sequence diagram


SellPage
AuctionServlet
Request
AuctionDAO
ctx
ItemDAO
Thymeleaf
doGet()
getParameter(id)
return id
findById(id)
returnauction
findAllOffersByAuction(id)
return offers
setVariable("closeable", true)
setVariable("auction", auction)
setVariable("offers", offers)
setVariable(“user” , user)
process(DETTAGLIO.html,ctx)
DETTAGLIO.html
DETTAGLIO.html

5.8 AuctionServlet sequence diagram



Comment

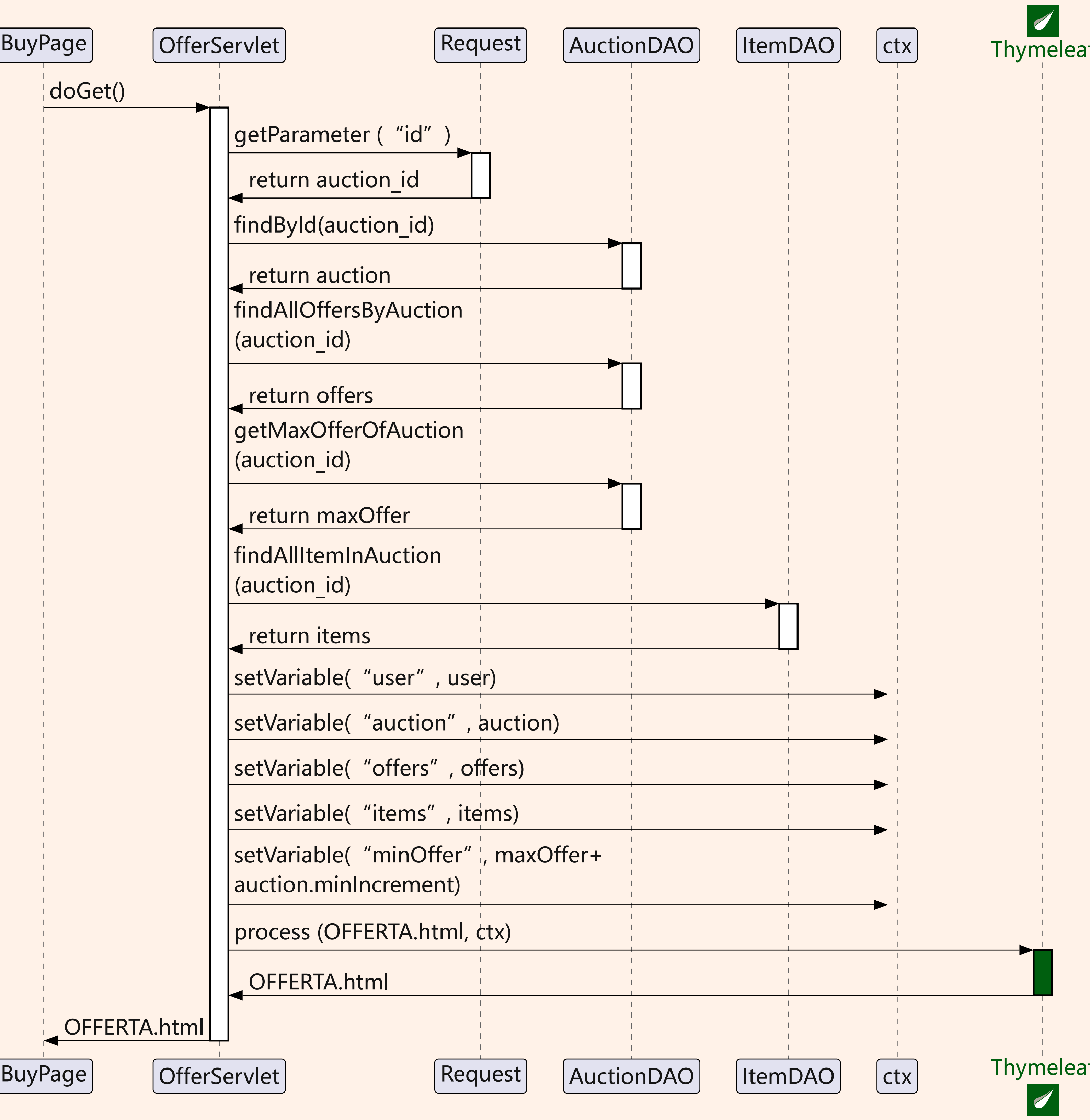
The User can create auction filling out form with all required information, in particular at least one auction.When the servlet gets the POST request, it interacts with the AuctionDAO to create the auction with the `createAuction()` method and to add the items with the `insertItems()` method.

Note that checkIds is a list of integers obtained by converting the strings inside the array returned by the `requireParameter("selectedItems")` method and parsing them with `Integer.parseInt()`.

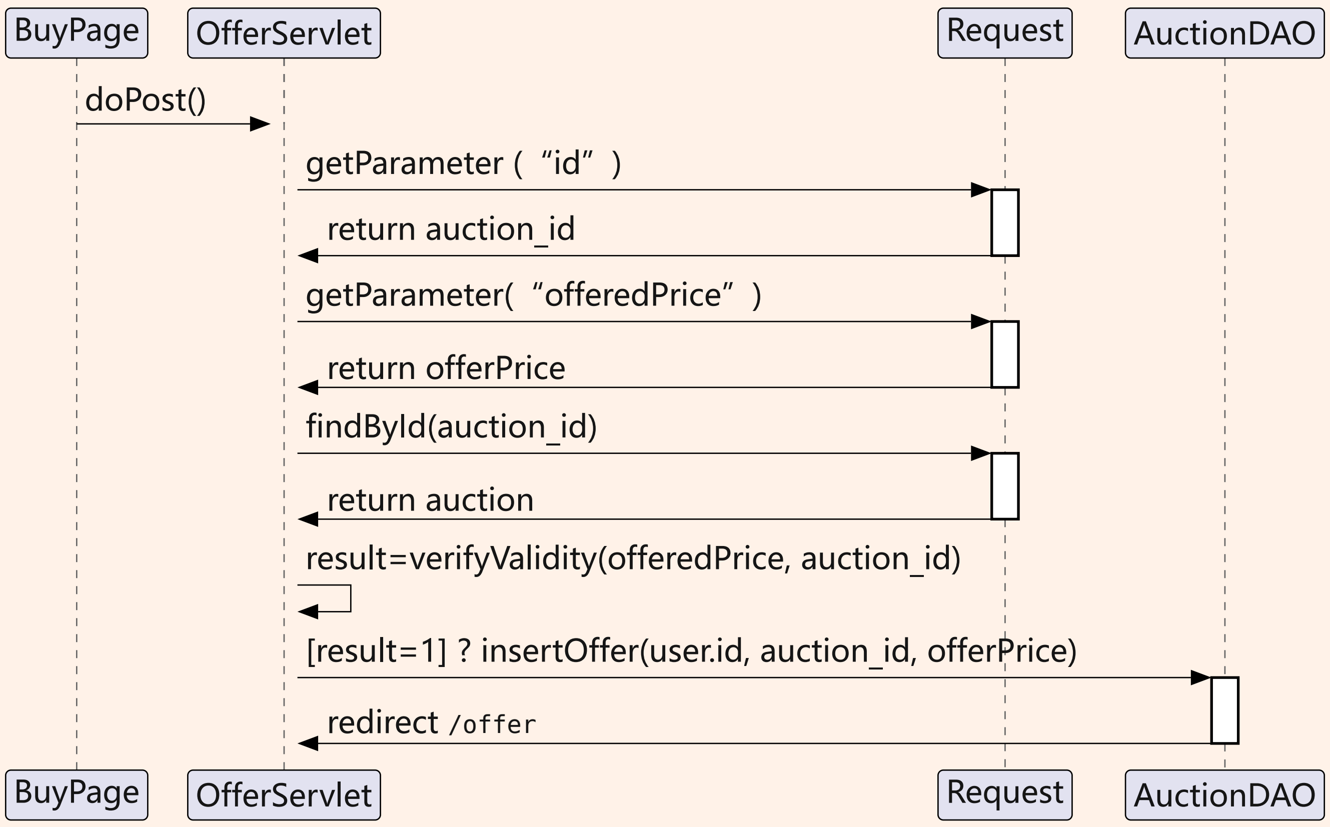
`requireParameter(param)` is a custom-method that make sure parameter with name `param` is obtained via request.

startingPrice of auction is calculated as sum of items inserted.

5.9 OfferServlet sequence diagram


BuyPage
OfferServlet
Request
AuctionDAO
ItemDAO
ctx
Thymeleaf
doGet()
getParameter (“id”)
return auction_id
findById(auction_id)
return auction
findAllOffersByAuction (auction_id)
return offers
getMaxOfferOfAuction (auction_id)
return maxOffer
findAllItemInAuction (auction_id)
return items
setVariable(“user” , user)
setVariable(“auction” , auction)
setVariable(“offers” , offers)
setVariable(“items” , items)
setVariable(“minOffer” , maxOffer+ auction.minIncrement)
process (OFFERTA.html, ctx)
OFFERTA.html
OFFERTA.html

5.10 OfferServlet sequence diagram

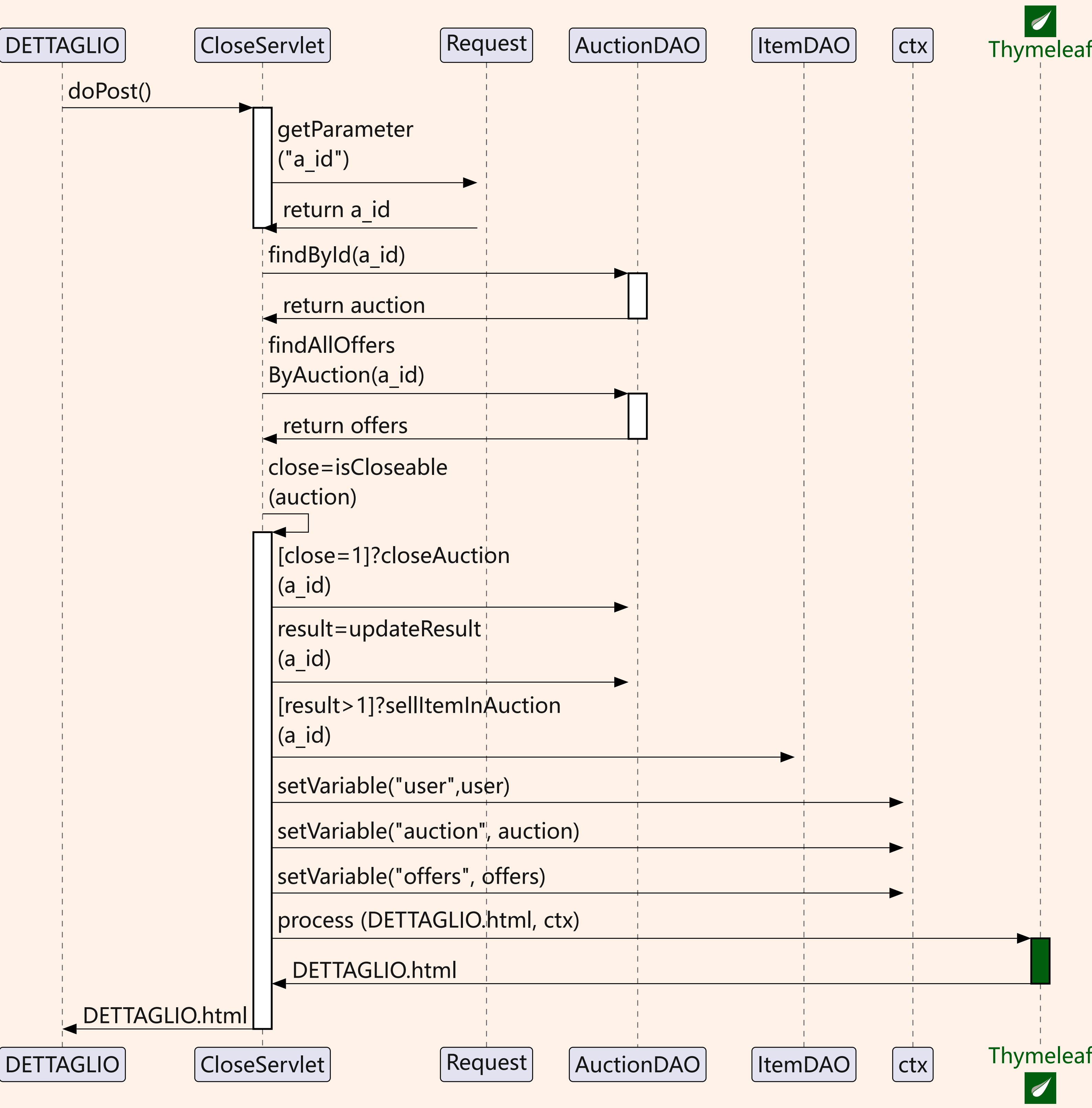


Comment

From BuyPage.html, after getting the renewed BuyPage.html from BuyServlet as result of *`doPost()`* send with keywords, here you click on any auction sends a *`doGet()`* to OfferServlet, getting list of offers and items related to that specific auction.

There is a form in OFFERTA.html, where you can make an offer that is greater than minimum offer imposed by Server, you will be redirected to same page afterwards using PRP.

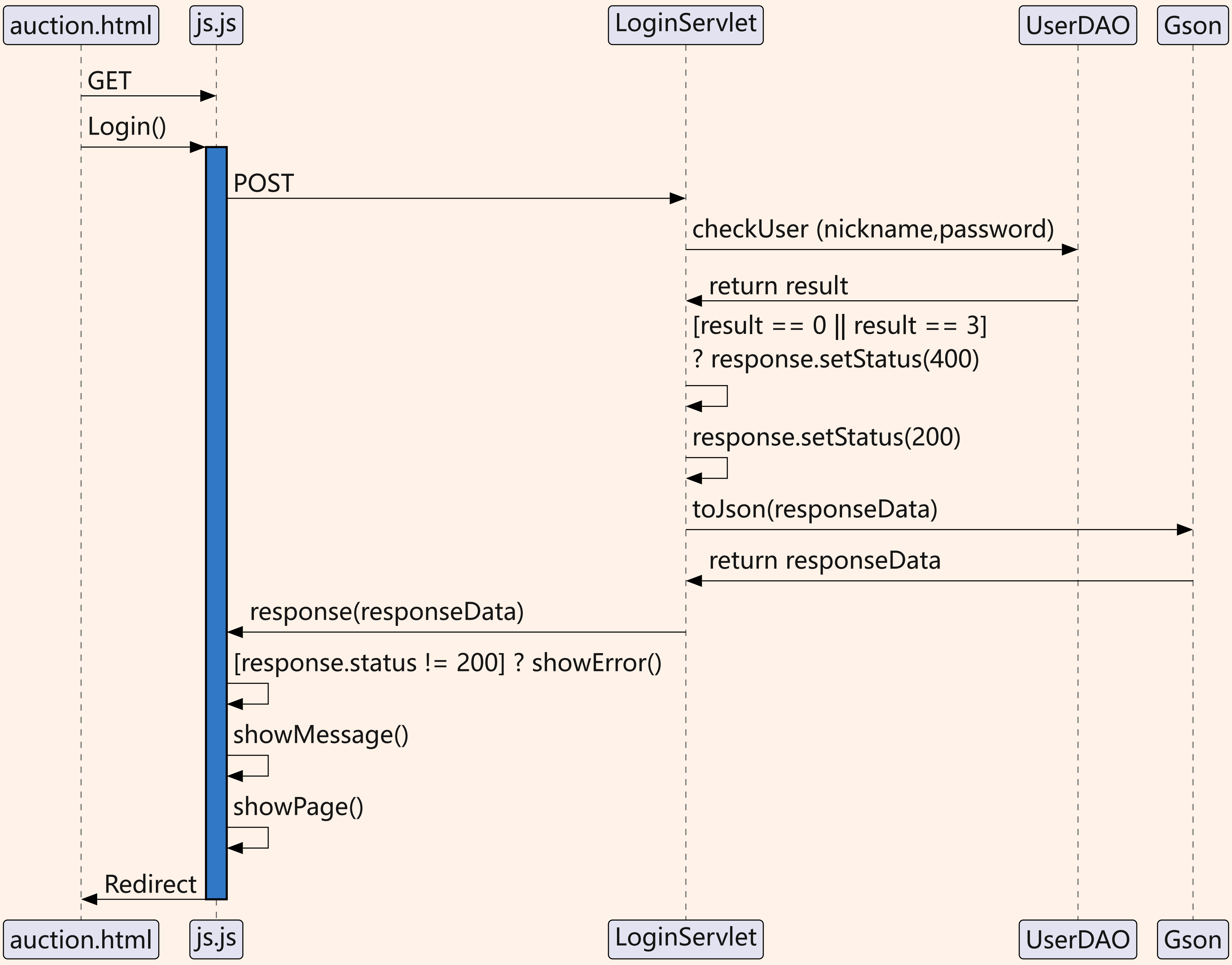
5.11 CloseServlet sequence diagram


DETTAGLIO
CloseServlet
Request
AuctionDAO
ItemDAO
ctx
Thymeleaf
doPost()
getParameter ("a_id")
return a_id
findById(a_id)
return auction
findAllOffers ByAuction(a_id)
return offers
close=isCloseable (auction)
[close=1]?closeAuction (a_id)
result=updateResult (a_id)
[result>1]?sellItemInAuction (a_id)
setVariable("user",user)
setVariable("auction", auction)
setVariable("offers", offers)
process (DETTAGLIO.html, ctx)
DETTAGLIO.html
DETTAGLIO.html

Comment

In DETTAGLIO.html, owner of the current auction can try to close the auction, which is will be handled by CloseServlet, firstly will verify if auction is expired and is currently open with *`isCloseable()`* method, afterwards it will update Result table in database with *`updateResult()`* method, it will insert a new record if and only if the auction has a winner, with return value = 1. If result > 1, all items in the auction will be marked as sold with *`sellItemInAuction()`* method.

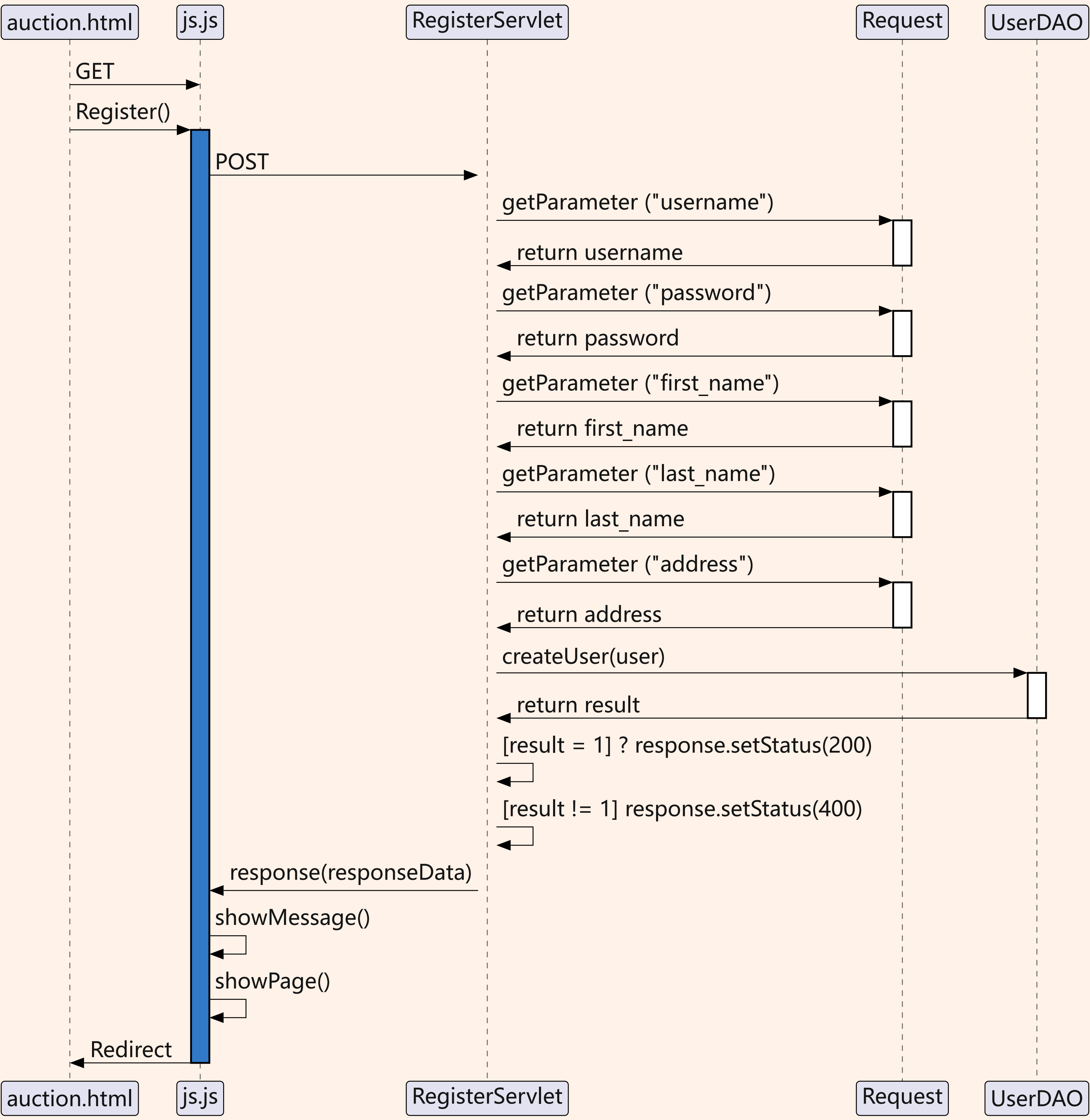
5.12 TS Event: Login



Comment

As the server is deployed the *`auction.html`* requests the associated Javascript files. As they have been loaded thanks to the IIFE, the User is able to Login. Once the button has been clicked, Javascript performs a POST request – via *`login()`* method – to the Login servlet, which, as seen in the Login sequence diagram (Section 5.1), checks if the User exists: if that' s the case it returns a 200 OK status code, otherwise it returns a 400 Bad Request status code. If not, then a error div will appear on top of the navbar. Body of response always contains a *`Map<key,Object> responseData`* containing needed information, it will be omitted for simplicity.

5.13 TS Event: Register



Comment

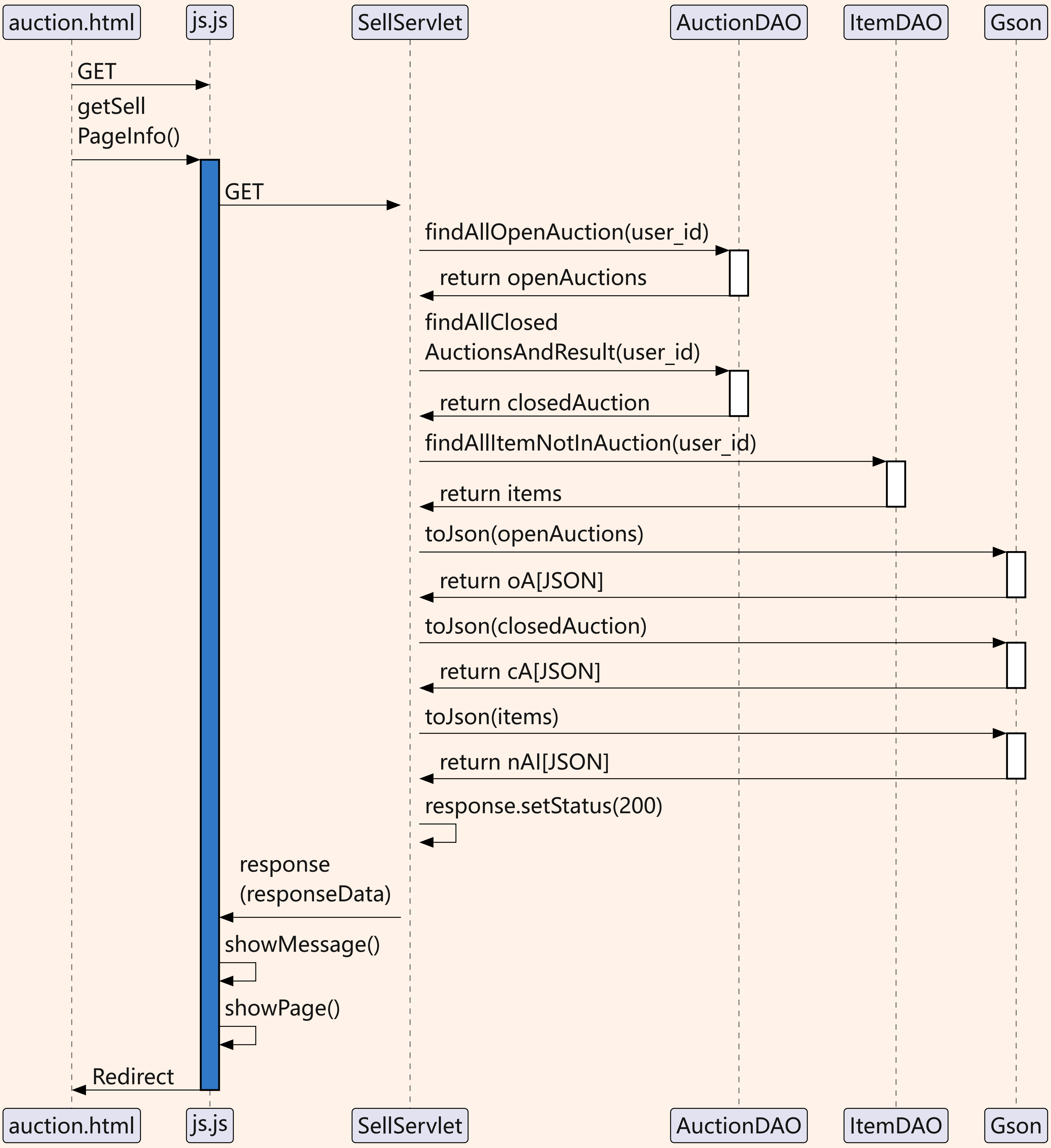
In *`auction.html`*, the User can register by clicking on the *`Register`* button, which will send a POST request to the *`RegisterServlet`*, which will create a new User in the database.

The *`createUser()`* method of the *`UserDAO`* class is called, which returns a boolean value, 1 if the User has been created, 0 otherwise.

Response status code is set to 200 OK if the User has been created, 400 Bad Request otherwise.

responseData is omitted, it contains normal message or error message.

5.14 getSellPageInfo sequence diagram



Comment

When user clicks on `Sell` button, javascript sends a GET request to the `SellServlet`, which returns a list of open auctions and a list of closed auctions and a list of items not in any auction.

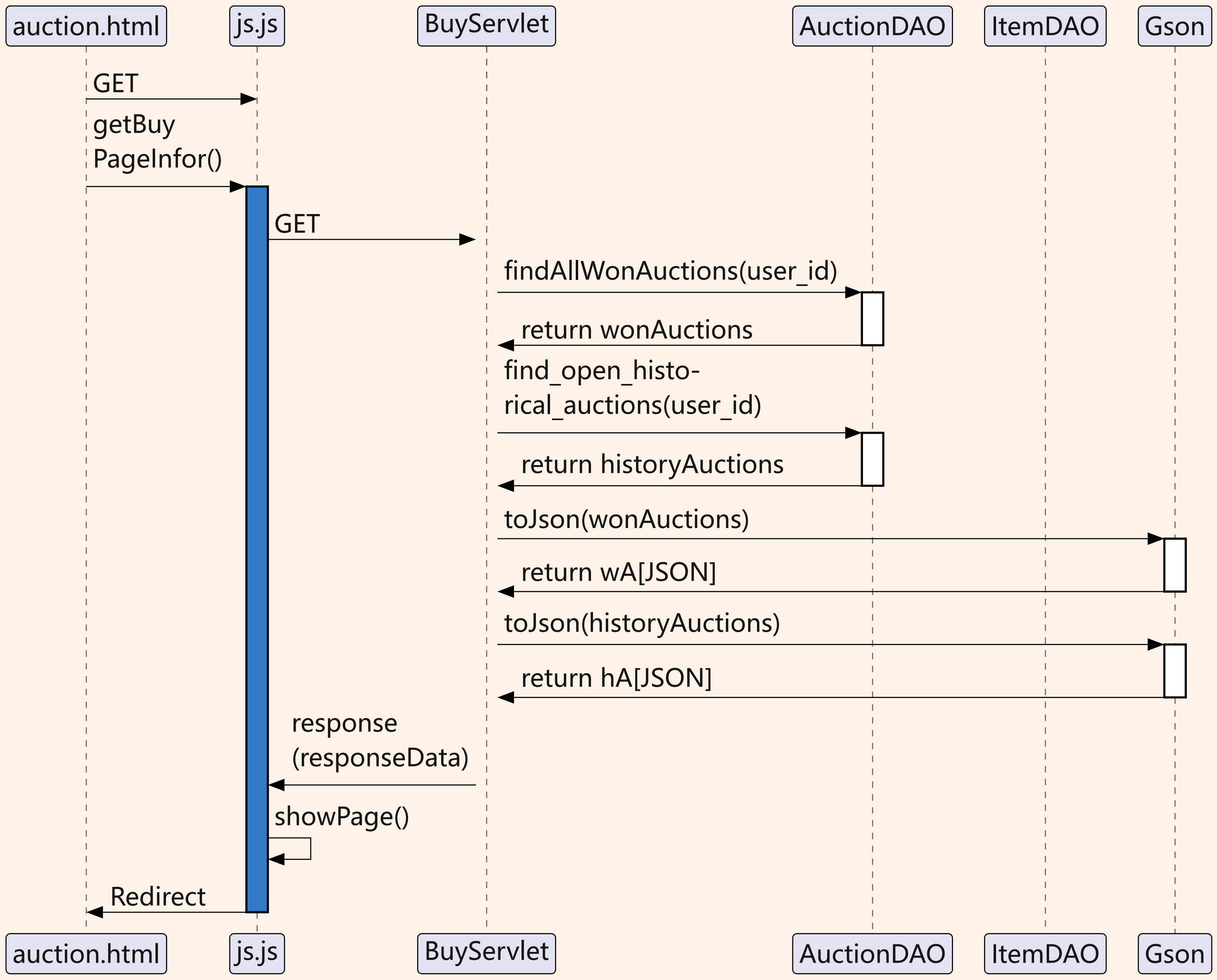
Session object is used to get the current user, which is used to filter the auctions, it will be omitted, user_id will be used directly instead.

The responseData is a JSON object containing the list of open auctions, the list of closed auctions and the list of available items.

If response status code is 200 OK, then page will display the `SellView`, also part of setting status code will be omitted for simplicity.

Same goes for showMessage() and showError(), since they are always invoked after the response is sent.

5.15 TS getBuyPageInfo sequence diagram



Comment

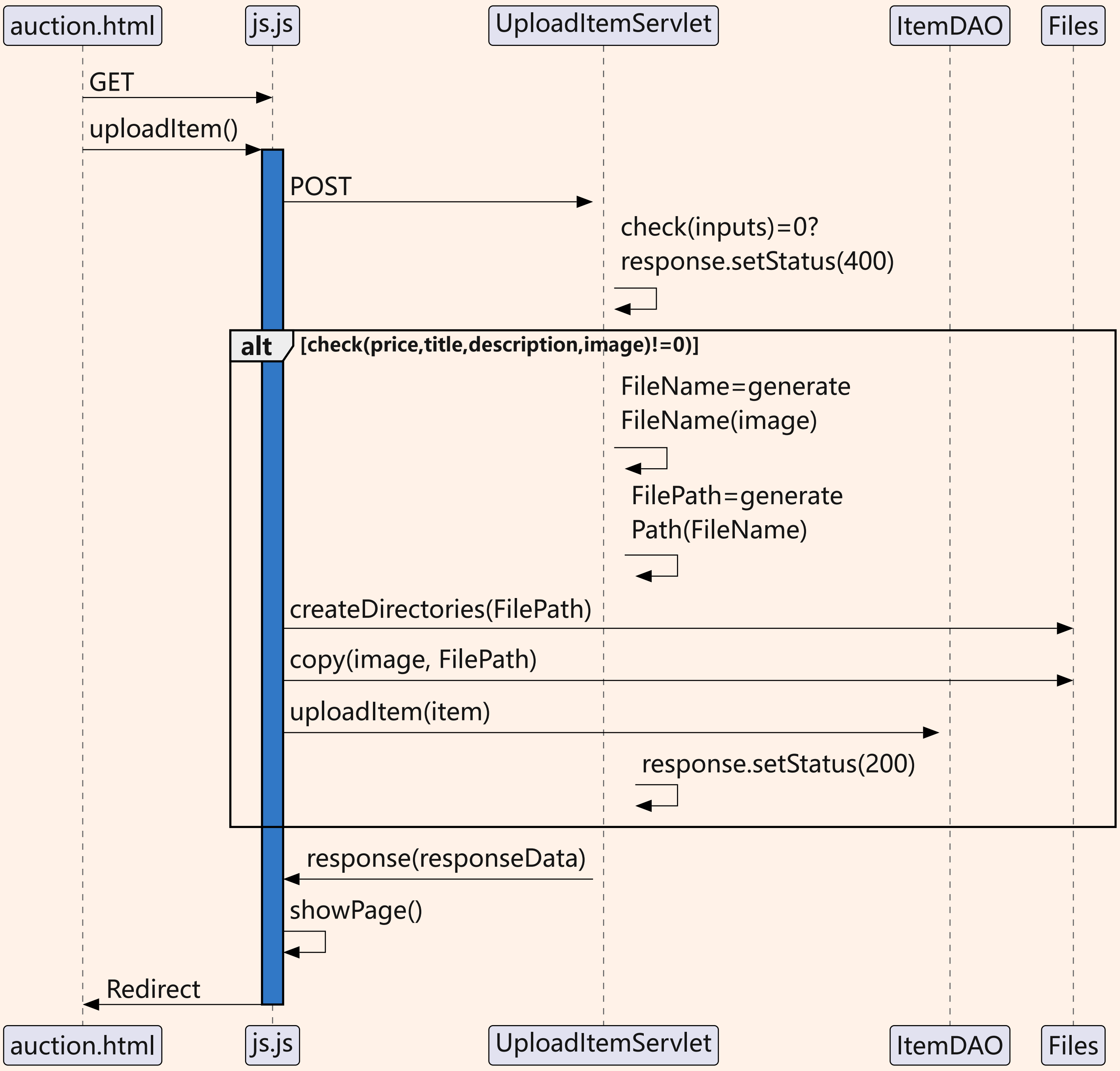
When user clicks on `Buy` button, javascript sends a GET request to the `BuyServlet`, which returns a list of won auctions and a list of historical auctions.

Session object is used to get the current user, which is used to filter the auctions.

The responseData is a JSON object containing the list of won auctions and the list of historical auctions.

If response status code is 200 OK, then page will display the `BuyView`.

5.16 uploadItem sequence diagram



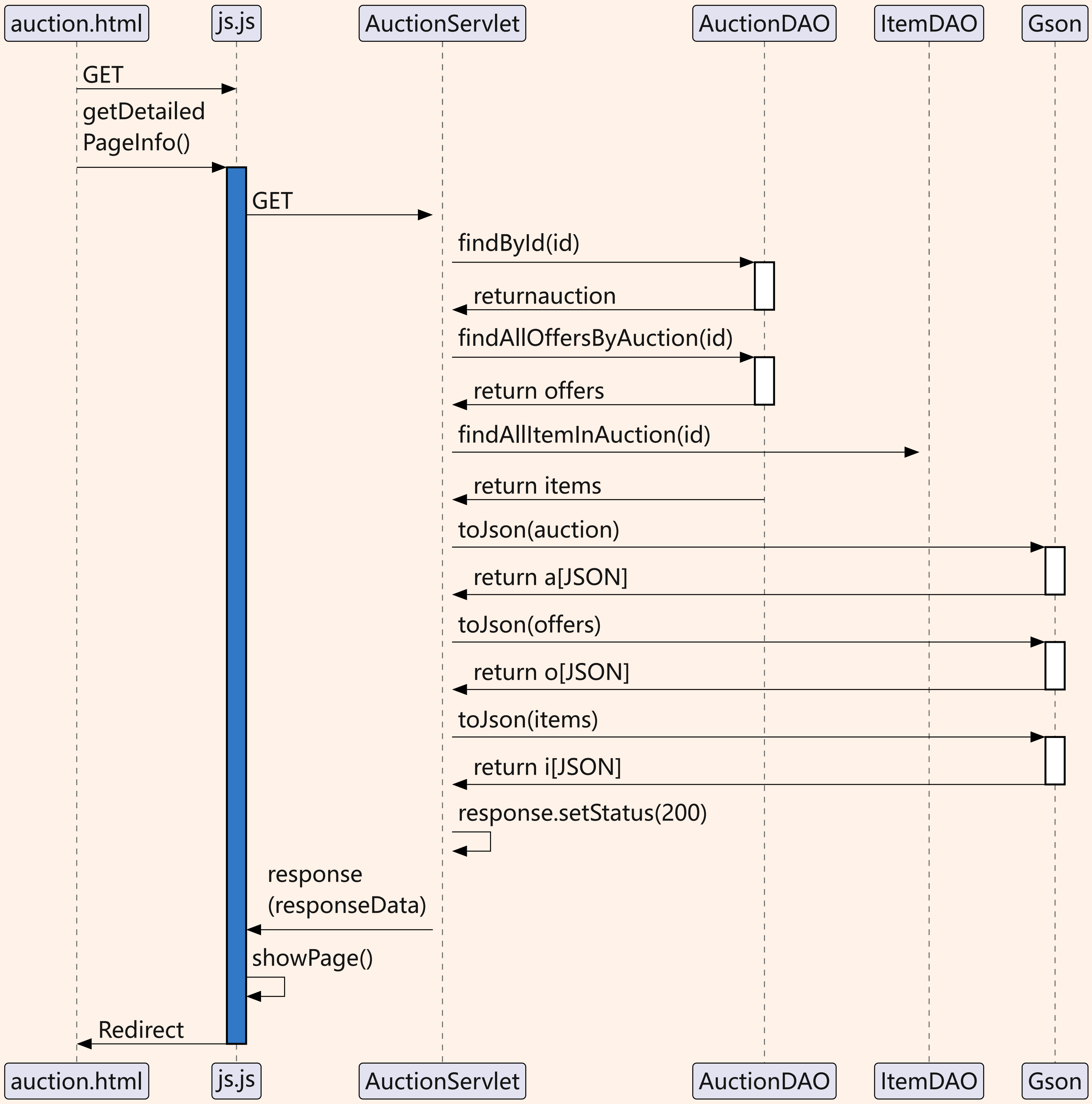
Comment

Inputs contains (title, description, price, image) This is quite similar to the *`uploadItem`* sequence diagram Section 5.6, except this time responseData is sent instead of an entire page.

When user clicks on *`uploadItem`* button, javascript sends a POST request to the *`UploadItemServlet`*, which receives the following parameters: *`price`*, *`title`*, *`description`*, *`image`*.

During processing, the *`check()`* method is called to validate the input parameters. If any of them is invalid, the response status code is set to 400 Bad Request and return immediately.

5.17 TS getDetailedPageInfo sequence diagram



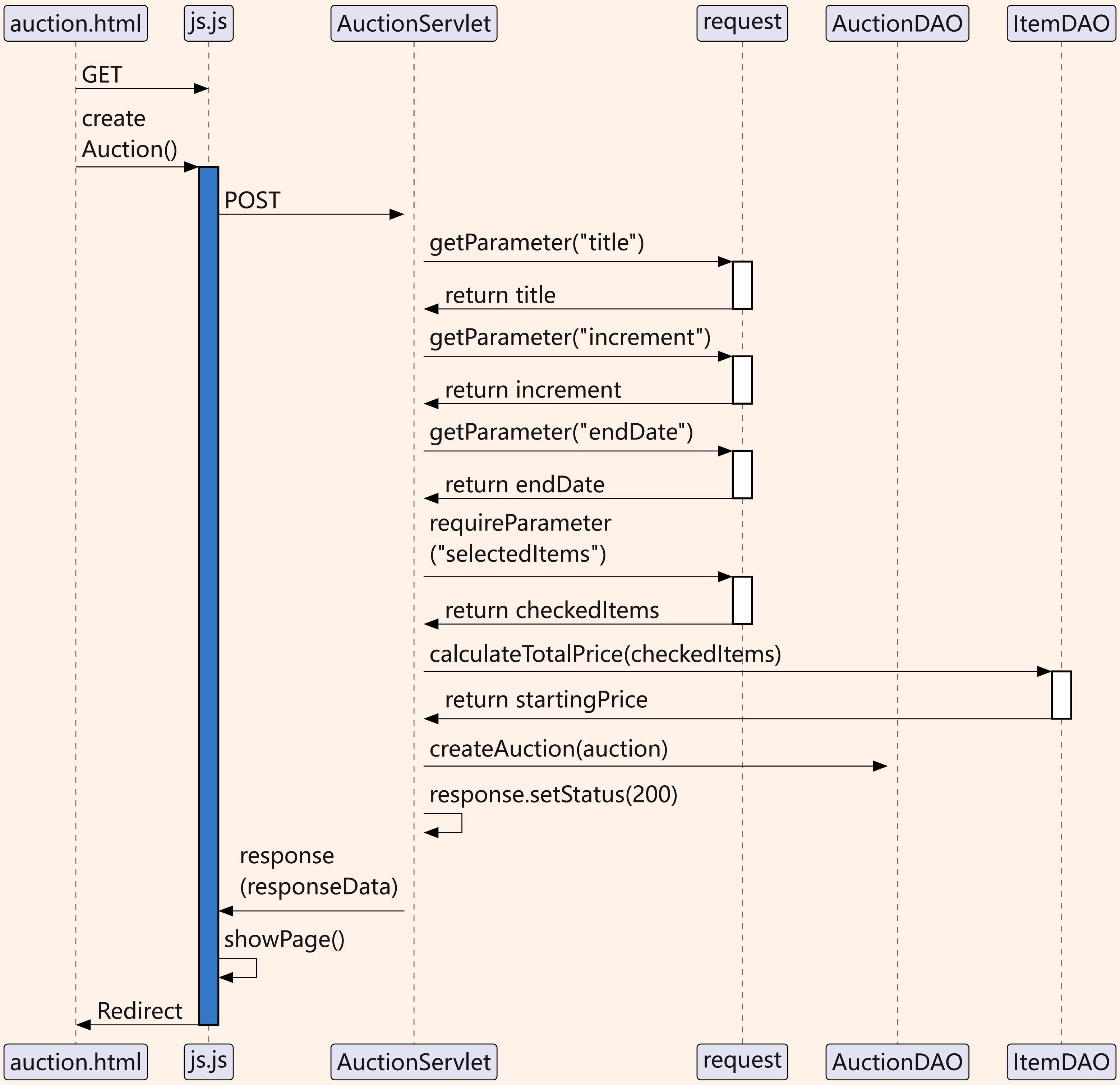
Comment

When user clicks on any auction in `SellView`, javascript sends a GET request to the `AuctionServlet`, which returns the auction details along with the list of offers and items related to that specific auction.

The responseData is a JSON object containing the auction details, the list of offers and the list of items.

If response status code is 200 OK, then page will display the `DetailedView`.

5.18 TS createAuction sequence diagram



Comment

When user clicks on `createAuction` button, javascript sends a POST request to the `AuctionServlet`, which receives the following parameters: `title`, `increment`, `endDate`, `selectedItems`.

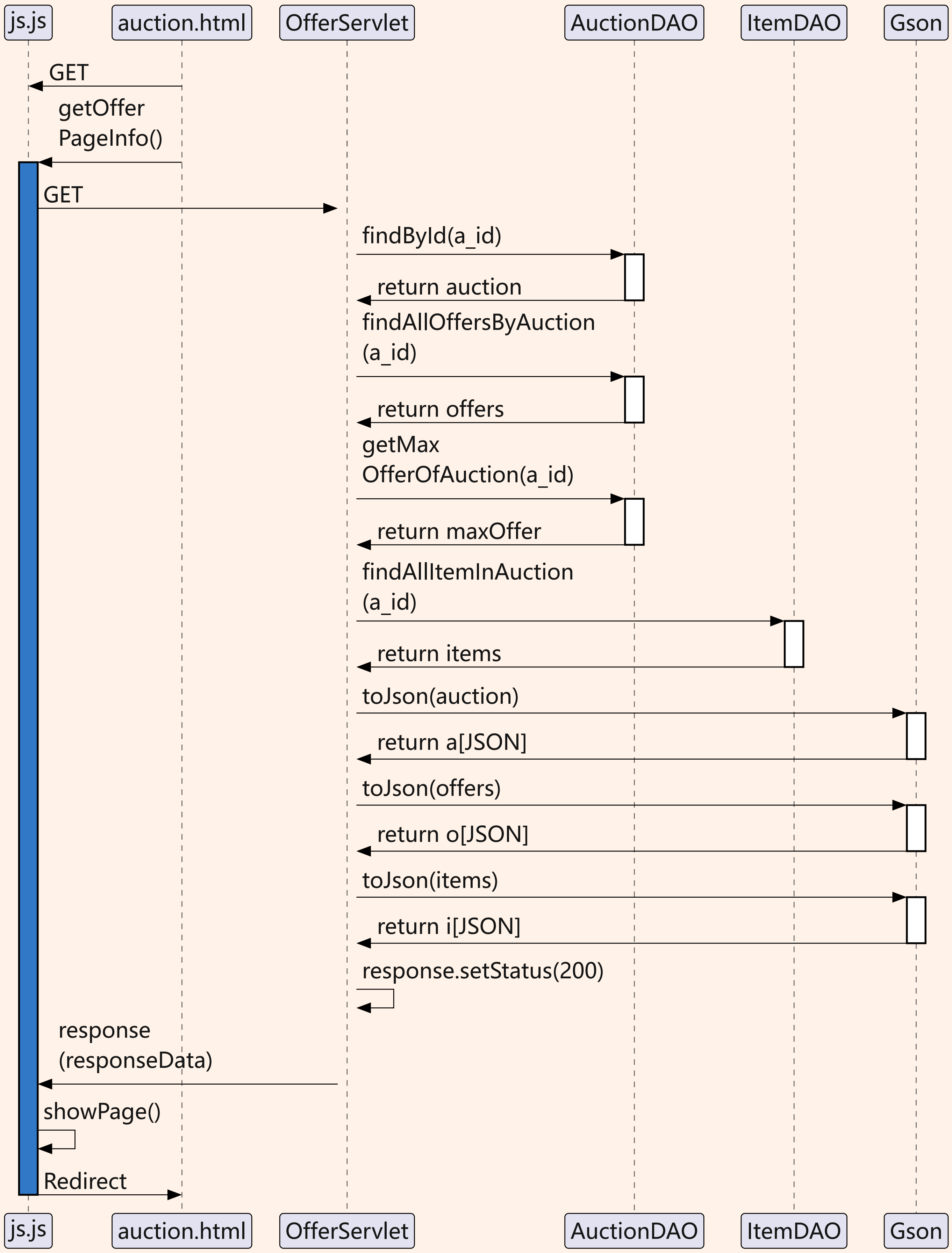
The `createAuction()` method is called to create a new auction with the given parameters.

If the auction is successfully created, the response status code is set to 200 OK and the responseData is a JSON object containing the newly created auction.

If the auction creation fails, the response status code is set to 400 Bad Request, here omitted for simplicity, and the responseData contains an error message.

The page will display the SellView with newly created auction.

5.19 TS getOfferPageInfo sequence diagram



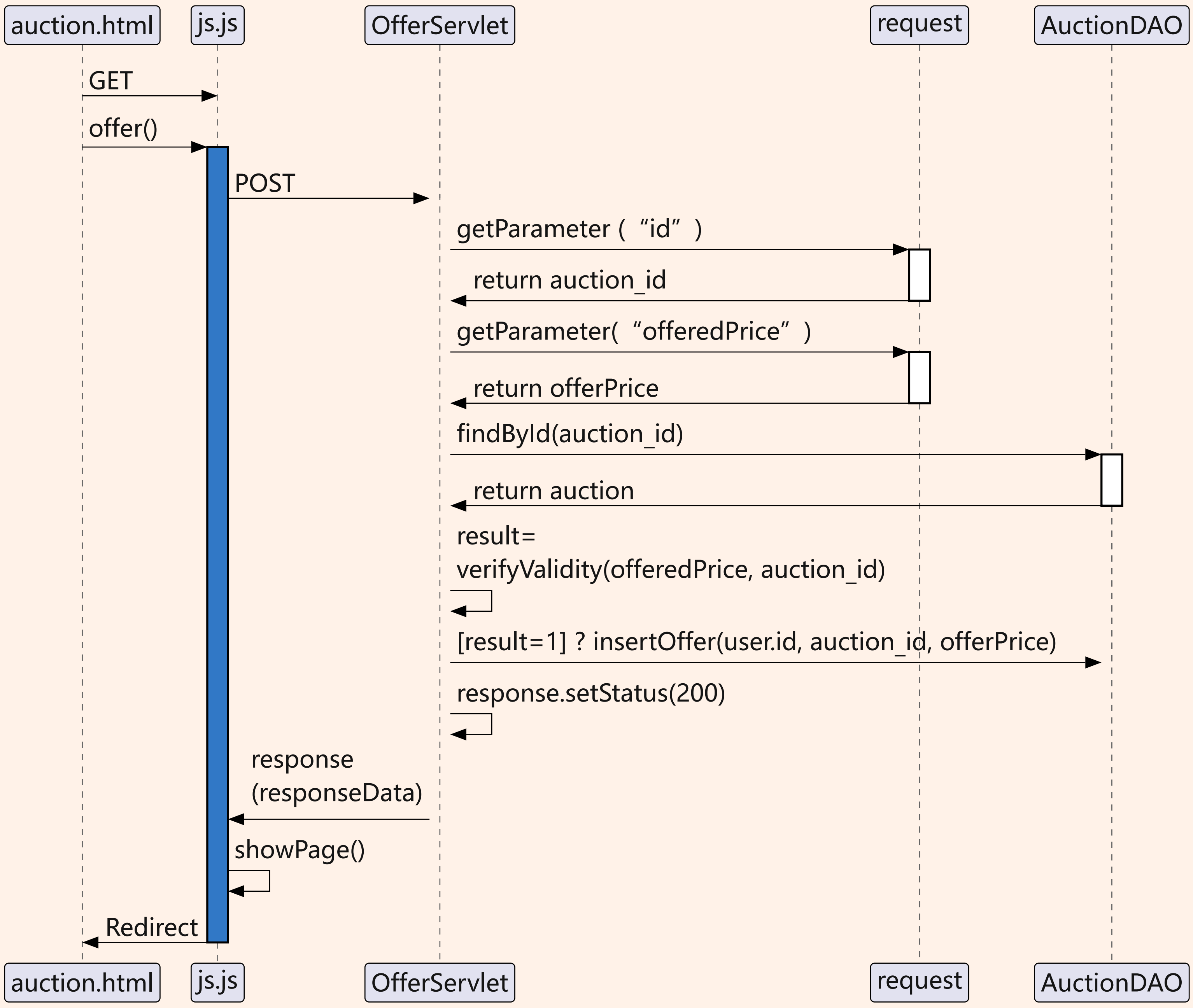
Comment

When user clicks on any auction in `BuyView`, javascript sends a GET request to the `OfferServlet`, which returns the auction details along with the list of offers and items related to that specific auction.

The responseData is a JSON object containing the auction details, the list of offers and the list of items.

If response status code is 200 OK, then page will display the `OfferView`.

5.20 TS offer sequence diagram



Comment

When user clicks on *`offer`* button in *`OfferView`*, javascript sends a POST request to the *`OfferServlet`*, which receives the following parameters: *`id`*, *`offeredPrice`*.

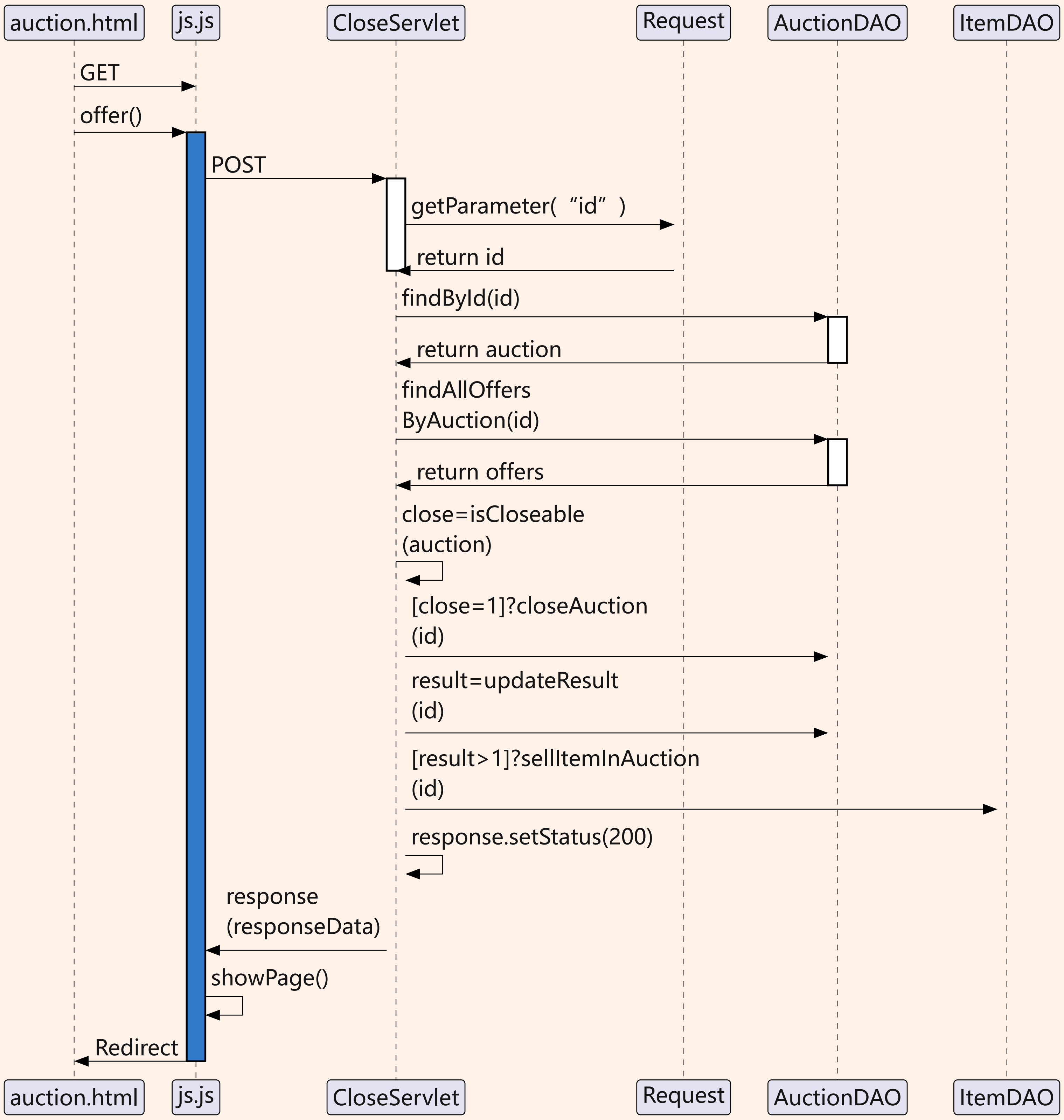
The *`offer()`* method is called to create a new offer with the given parameters.

If the offer is successfully created, the response status code is set to 200 OK and the responseData is a JSON object containing the newly created offer.

If the offer creation fails, the response status code is set to 400 Bad Request, here omitted for simplicity, and the responseData contains an error message.

The page will display the *`OfferView`* with newly created offer.

5.21 TS close sequence diagram



Comment

When user clicks on *`close`* button in *`DetailView`*, javascript sends a POST request to the *`CloseServlet`*, which receives the following parameters: *`id`*.

The *`close()`* method is called to close the auction with the given ID.

If the auction is successfully closed, the response status code is set to 200 OK and the responseData is a JSON object containing the result of the closing process.

If the auction closing fails, the response status code is set to 400 Bad Request, here omitted for simplicity, and the responseData contains an error message.

The page will display the *`CloseView`* depending on the result of the closing process, status will be either closed or open.