Documentation
Information Technologies for the Web project

Edoardo Carlotto

edoardo.carlotto@mail.polimi.it
Person code 10730627

prof. Fraternali Piero
a.y. 2022-2023

Indice

1 Requirements

1.1 Specifications

A web application enables the management of online auctions. Users log in and can sell and buy items at auction.
The HOME page contains two links: one to access the SELL page and one to access the BUY page.
The SELL page displays a list of auctions created by the user that are still open, a list of the auctions the user has created and closed, and two forms: one to create a new item and one to create a new auction to sell the user's items. The first form inserts new items into the database, and the second shows a list of available items in the database and allows the selection of multiple items. An item has a code, name, description, image, and price. An auction includes one or more items for sale, the starting price of the set of items, the minimum bid increment (expressed as an integer in euros), and an expiration time (date and time, e.g., 19-04-2021 at 24:00). The starting price of the auction is obtained by summing the price of the items included in the offer. The same item cannot be included in different auctions. Once sold, an item must no longer be available for inclusion in other auctions. The list of auctions is ordered by ascending date and time. The list shows: code and name of the items included in the auction, the highest bid, and the remaining time (number of days and hours) between the login time (date and time) and the auction's closing date and time. Clicking on an auction opens a DETAIL AUCTION page that shows all the data of an open auction and the list of bids (username, bid amount, date, and time of the bid) ordered by descending date and time. A CLOSE button allows the user to close the auction if the expiration time has passed (the case of expired but unclosed auctions is ignored, and automatic closure of auctions after expiration is not considered). If the auction is closed, the page shows all the auction data, the name of the winning bidder, the final price, and the shipping address (fixed) of the user.
The BUY page contains a search form for keywords. When the buyer submits a keyword, the BUY page is updated and shows a list of open auctions (with an expiration time later than the submission date and time) where the keyword appears in the name or description of at least one of the items in the auction. The list is ordered in descending order based on the remaining time (number of days and hours) until the auction closes. Clicking on an open auction opens the BID page, which shows the item data, the list of received bids in descending date and time order, and an input field to submit a bid, which must be higher than the current highest bid by at least the minimum increment. After submitting the bid, the BID page updates and shows the list of updated bids. The BUY page also contains a list of the bids won by the user, with item data and the final price.

1.2 Completion of Specifications

- The initial page (index.html) must manage the user's login and store the login date and time.
- When an auction's expiration time is in the past, the auction can either be closed by the user or not yet closed. Therefore, there are closed auctions and expired auctions that have not been closed.
- The minimum bid amount is information that can be derived from the list of items or the list of bids.
- Bid amounts, the starting price of an item, and the minimum bid increment for an auction cannot be negative.
- The expiration time of an auction cannot be in the past.
- An auction must contain at least one item.
- Since a user cannot cancel an auction after creating it, the author may also place a bid on their own auction.
- A logout feature is introduced.

It is guaranteed that the constraints on values provided by the user will be validated in three ways:

- Validation of HTML5 forms to provide immediate feedback.
- Appropriate server-side checks in Java, necessary even if the user manipulates the requests.
- SQL checks and triggers in the database.

1.3 Extension of Specifications for the JavaScript Version

Implement a client-server web application that extends and/or modifies the previous specifications as follows:

- After login, the entire application is implemented as a single page.
- If the user is accessing the application for the first time, the application shows the content of the BUY page. If the user has already used the application, it shows the content of the SELL page if the user's last action was creating an auction; otherwise, it shows the content of the BUY page with the list (which may be empty) of auctions the user has previously clicked on that are still open. The information about the user's last action and visited auctions is stored on the client side for a month.
- Every user interaction is handled without reloading the entire page, but instead triggers an asynchronous server request and possibly updates only the content that needs to be refreshed due to the event.

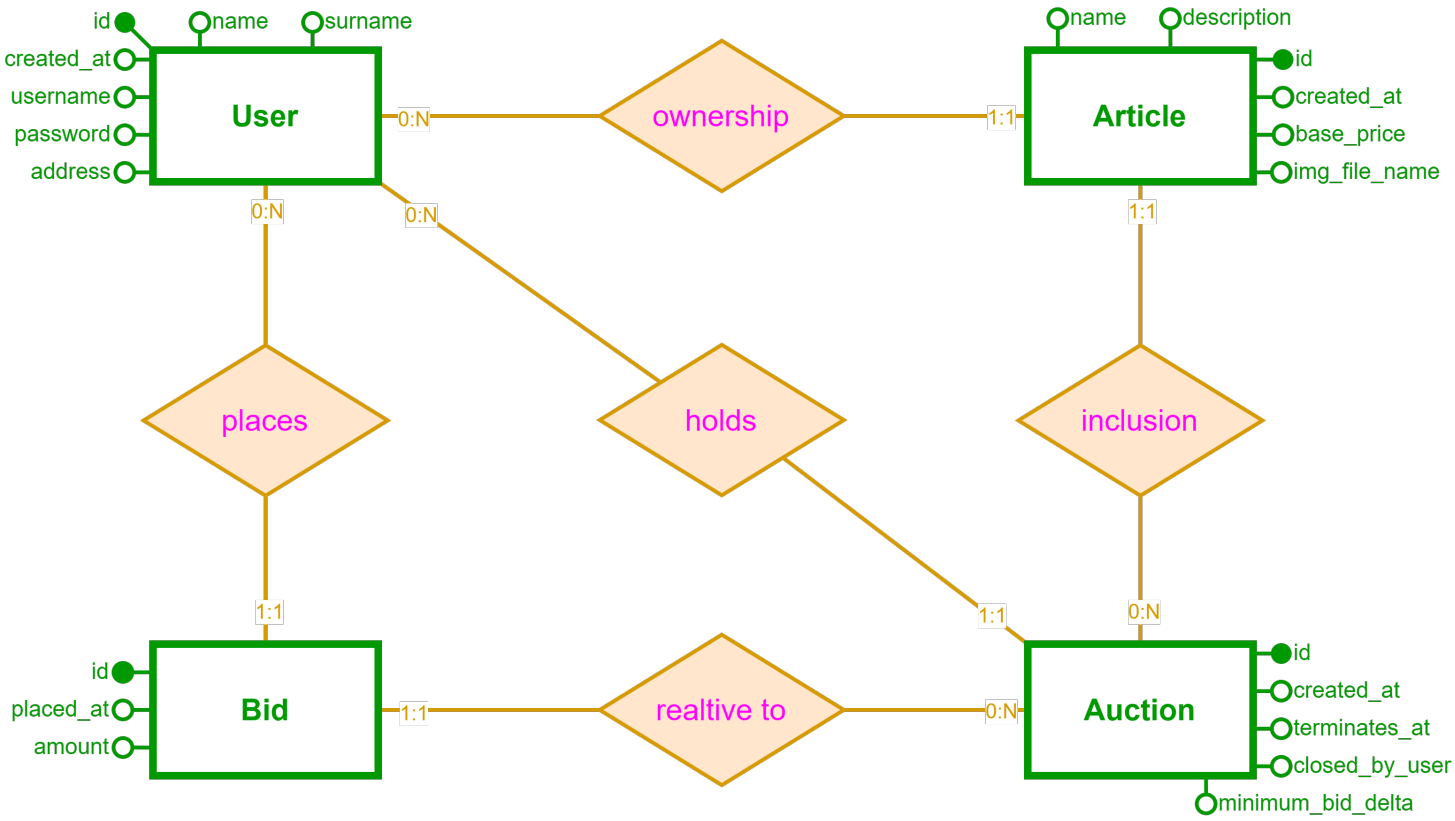
1.4 Completion of Specifications for the JavaScript Version

- The home page is replaced by a navigation menu.
- The following actions are considered for recording: creating an auction, placing a bid, creating an auction, closing an auction.
- The information saved on the client side is associated with the username, allowing multiple users to use the same browser.

2 Database Design

2.1 ER Scheme

The requirements analysis leads to the design of the following structure for the database:



Attributes that store the timestamp of when a tuple is inserted, such as *'created_at'*, are not required for all entities. However, since their value can be automatically generated by the DB during the *INSERT* phase at a negligible cost, it is preferable to include this information in the database as it might always be useful.

2.2 SQL Code for Tables

The following code, used to create the tables, shows the logical structure of the database.

2.2.1 User

```
CREATE TABLE IF NOT EXISTS `users` (
  `id` int(32) unsigned NOT NULL AUTO_INCREMENT,
  `created_at` timestamp NOT NULL DEFAULT current_timestamp(),
  `username` varchar(20) NOT NULL,
  `password` varchar(40) NOT NULL,
  `name` varchar(32) NOT NULL,
  `surname` varchar(32) NOT NULL,
  `address` varchar(50) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `username` (`username`)
);
```

2.2.2 Article

```
CREATE TABLE IF NOT EXISTS `articles` (
  `id` int(32) unsigned NOT NULL AUTO_INCREMENT,
  `owner_user_id` int(32) unsigned NOT NULL,
  `created_at` timestamp NOT NULL DEFAULT current_timestamp(),
  `base_price` decimal(20,2) NOT NULL,
  `auction_id` int(32) unsigned DEFAULT NULL,
  `name` varchar(32) NOT NULL,
  `description` varchar(200) DEFAULT NULL,
  `img_file_name` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_articles_auctions` (`auction_id`),
  KEY `FK_articles_users` (`owner_user_id`),
  CONSTRAINT `FK_articles_auctions` FOREIGN KEY (`auction_id`) REFERENCES
      `auctions` (`id`) ON DELETE NO ACTION ON UPDATE CASCADE,
  CONSTRAINT `FK_articles_users` FOREIGN KEY (`owner_user_id`) REFERENCES
      `users` (`id`) ON DELETE NO ACTION ON UPDATE CASCADE,
  CONSTRAINT `positive_price` CHECK (`base_price` > 0)
);
```

2.2.3 Bid

```
CREATE TABLE IF NOT EXISTS `bids` (
  `id` int(32) unsigned NOT NULL AUTO_INCREMENT,
  `placed_at` timestamp NOT NULL DEFAULT current_timestamp(),
  `bidder_user_id` int(32) unsigned NOT NULL,
  `auction_id` int(32) unsigned NOT NULL,
  `amount` decimal(20,2) unsigned NOT NULL,
  PRIMARY KEY (`id`),
  KEY `FK_bids_auctions` (`auction_id`),
  KEY `FK_bids_users` (`bidder_user_id`),
  CONSTRAINT `FK_bids_auctions` FOREIGN KEY (`auction_id`) REFERENCES
      `auctions` (`id`) ON DELETE NO ACTION ON UPDATE CASCADE,
  CONSTRAINT `FK_bids_users` FOREIGN KEY (`bidder_user_id`) REFERENCES
      `users` (`id`) ON DELETE NO ACTION ON UPDATE CASCADE,
  CONSTRAINT `positive_amount` CHECK (`amount` > 0)
);
```

2.2.4 Auction

```
CREATE TABLE IF NOT EXISTS `auctions` (
  `id` int(32) unsigned NOT NULL AUTO_INCREMENT,
  `creator_user_id` int(32) unsigned NOT NULL DEFAULT 0,
  `created_at` timestamp NOT NULL DEFAULT current_timestamp(),
  `terminates_at` timestamp NOT NULL,
  `closed_by_user` bit(1) NOT NULL DEFAULT b'0',
```

```
    `minimum_bid_wedge` int(32) unsigned NOT NULL,
    PRIMARY KEY (`id`),
    KEY `FK_auctions_users` (`creator_user_id`),
    CONSTRAINT `FK_auctions_users` FOREIGN KEY (`creator_user_id`) REFERENCES
        `users` (`id`) ON DELETE NO ACTION ON UPDATE CASCADE,
    CONSTRAINT `terminates_after_Created` CHECK (`terminates_at` >
        `created_at`),
    CONSTRAINT `positive_bid_delta` CHECK (`minimum_bid_wedge` > 0)
  );
```

2.3 Trigger

In addition to the constraints on individual attributes and those imposed by the presence of *foreign keys*, triggers are added (both *AFTER INSERT* and *AFTER UPDATE* on the appropriate tables) that execute a query and raise an SQL exception when non-conforming elements are found, causing the operation to fail. The following conditions are verified.

2.3.1 Article owner matches auction creator

For each item in an auction, the items owner must match the auctions creator.

```
DELIMITER //
CREATE TRIGGER `new_article_belonging_to_auction_owner` AFTER INSERT ON
    `articles` FOR EACH ROW BEGIN
 IF EXISTS (
  SELECT * FROM auctions AS au
  JOIN articles AS ar
  ON ar.auction_id = au.id
  WHERE ar.owner_user_id != au.creator_user_id
  ) THEN
   SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'New articles must belong to the auction owner';
 END IF;
END//
DELIMITER ;
```

2.3.2 Bid respects min bid delta

For each bid, the amount must be at least *minimum_bid_delta* greater than the highest of the previous bids (where a bid is considered previous by using the *placed_at* field).

```
DELIMITER //
CREATE TRIGGER `new_bid_lower_than_delta` AFTER INSERT ON `bids` FOR EACH
    ROW BEGIN
 IF EXISTS (
  SELECT * FROM bids AS b1
  JOIN auctions AS a
  ON a.id = b1.auction_id
  WHERE b1.amount - a.minimum_bid_wedge < (
   SELECT max(amount) FROM bids AS b2
   WHERE b2.placed_at < b1.placed_at AND
    b2.auction_id = b1.auction_id )
```

```
  ) THEN
   SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'New bids must respect minimum wedge';
 END IF;
END//
DELIMITER ;
```

2.3.3 **No bids after termination** Bids cannot be placed if *CURRENT_TIMESTAMP()* is greater than the auction termination timestamp.

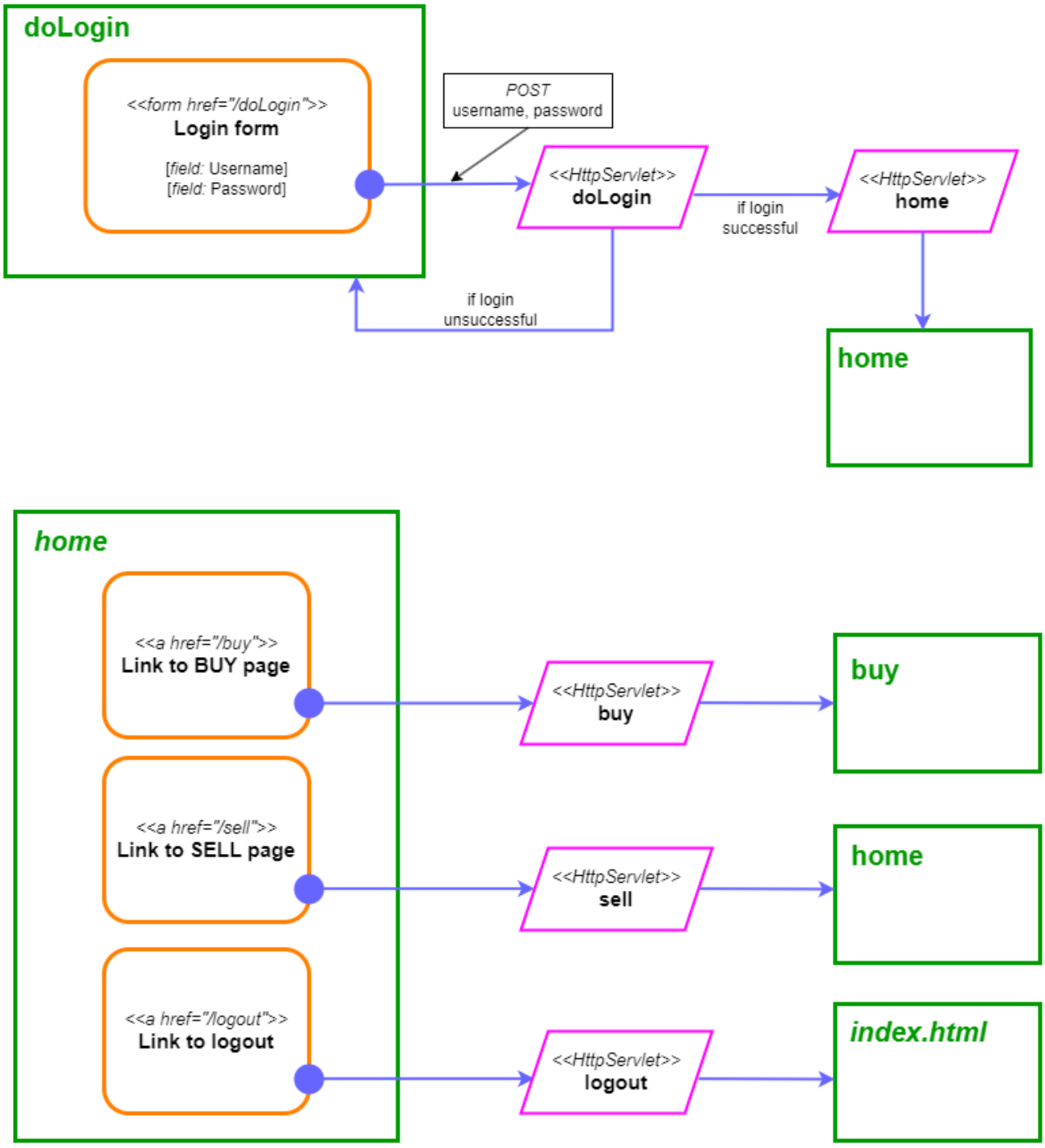
```
DELIMITER //
CREATE TRIGGER `new_bid_after_termination` AFTER INSERT ON `bids` FOR EACH
    ROW BEGIN
 IF EXISTS(
  SELECT * FROM auctions AS a
  WHERE a.id = NEW.auction_id AND
   a.terminates_at < CURRENT_TIMESTAMP()
 ) THEN
  SIGNAL SQLSTATE '45000'
   SET MESSAGE_TEXT = 'New bids cannot be placed after auction termination';
 END IF;
END//
DELIMITER ;
```

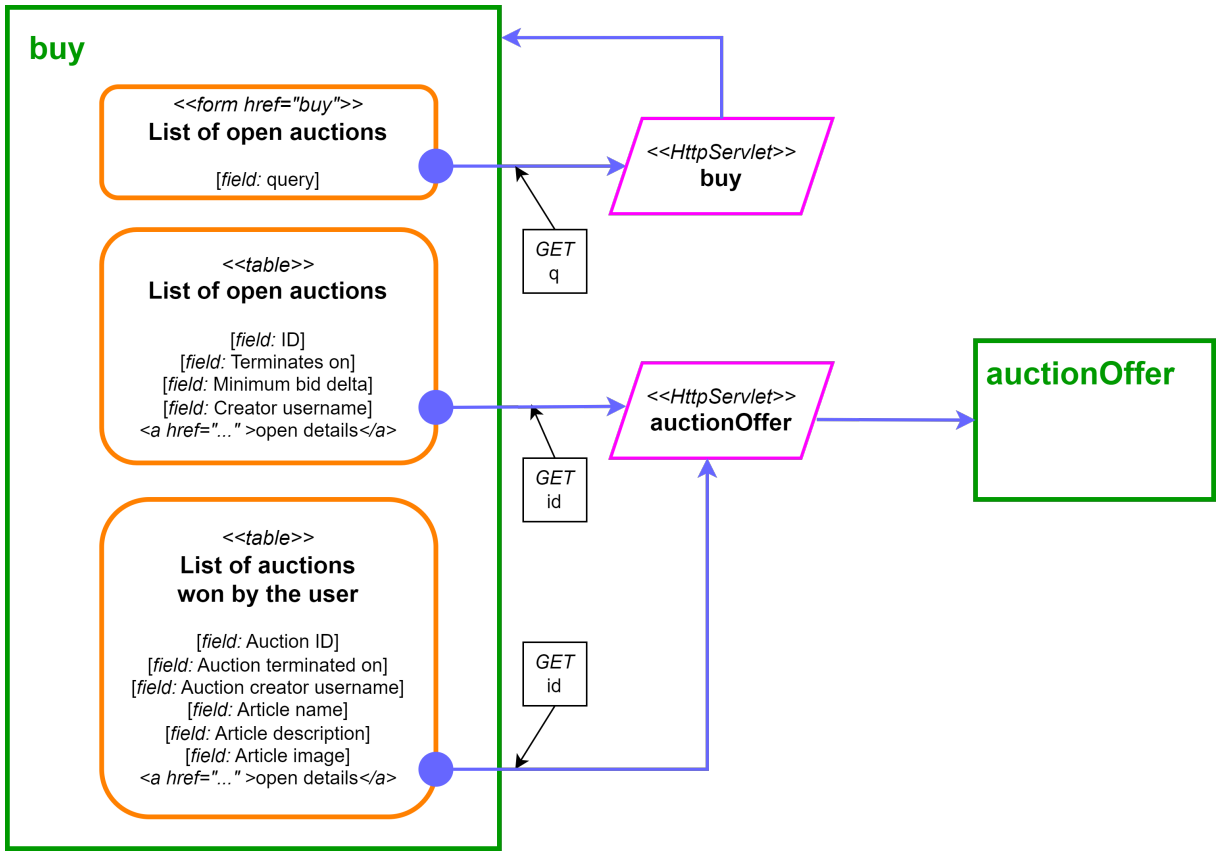
2.3.4 **Auction without articles** Every auction must have at least one article. This trigger must be temporarily disabled during the auction creation transaction, and the same *AFTER UPDATE* check is present on the articles table. Since the DBMS in use does not allow disabling triggers, a variable is required.

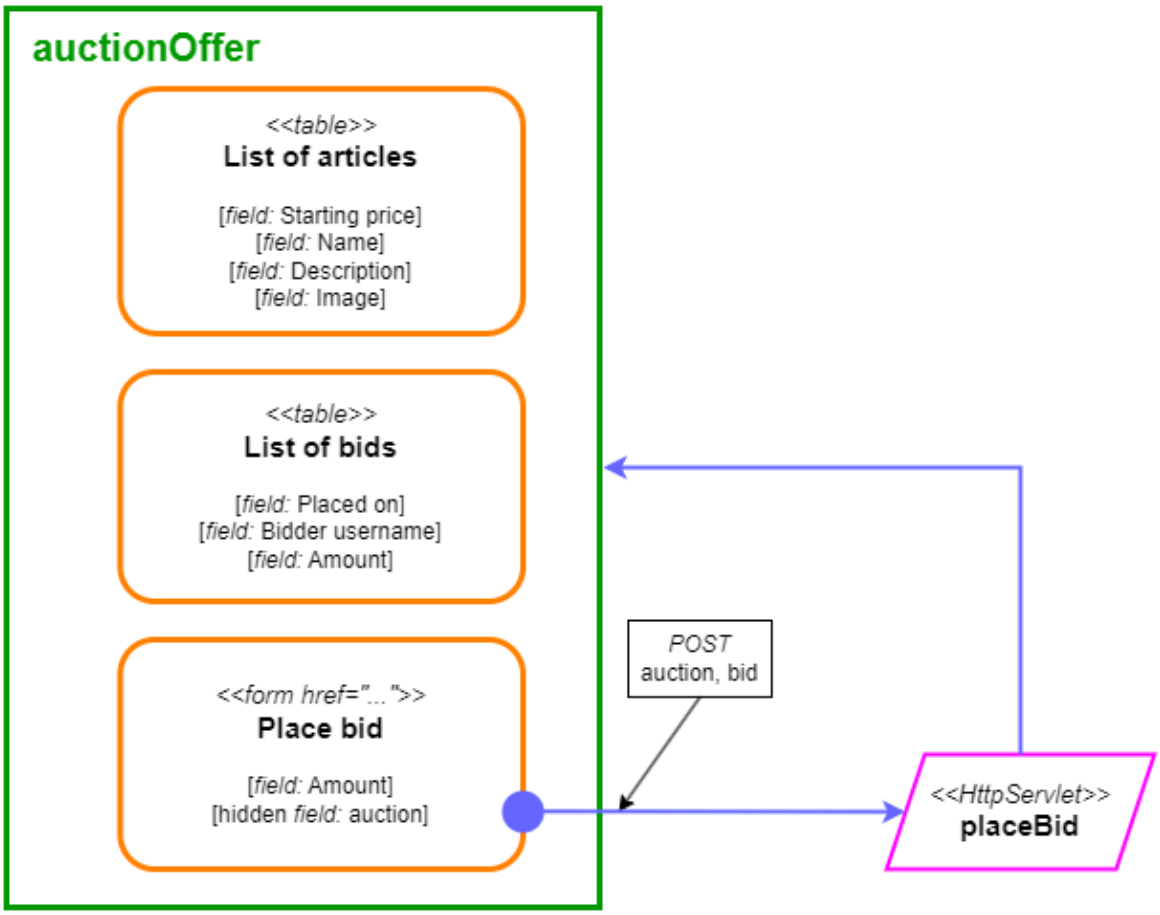
```
DELIMITER //
 CREATE TRIGGER `new_auction_without_articles` AFTER INSERT ON `auctions`
     FOR EACH ROW BEGIN
  IF (@AUCTION_TRIGGER_DISABLED IS NULL OR @AUCTION_TRIGGER_DISABLED!=1)
      AND EXISTS (
   SELECT id FROM auctions
   EXCEPT
   SELECT au.id FROM auctions AS au
   JOIN articles AS ar
   ON au.id = ar.auction_id
  ) THEN
   SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'Each auction must have at least one article';
 END IF;
END//
DELIMITER ;
```

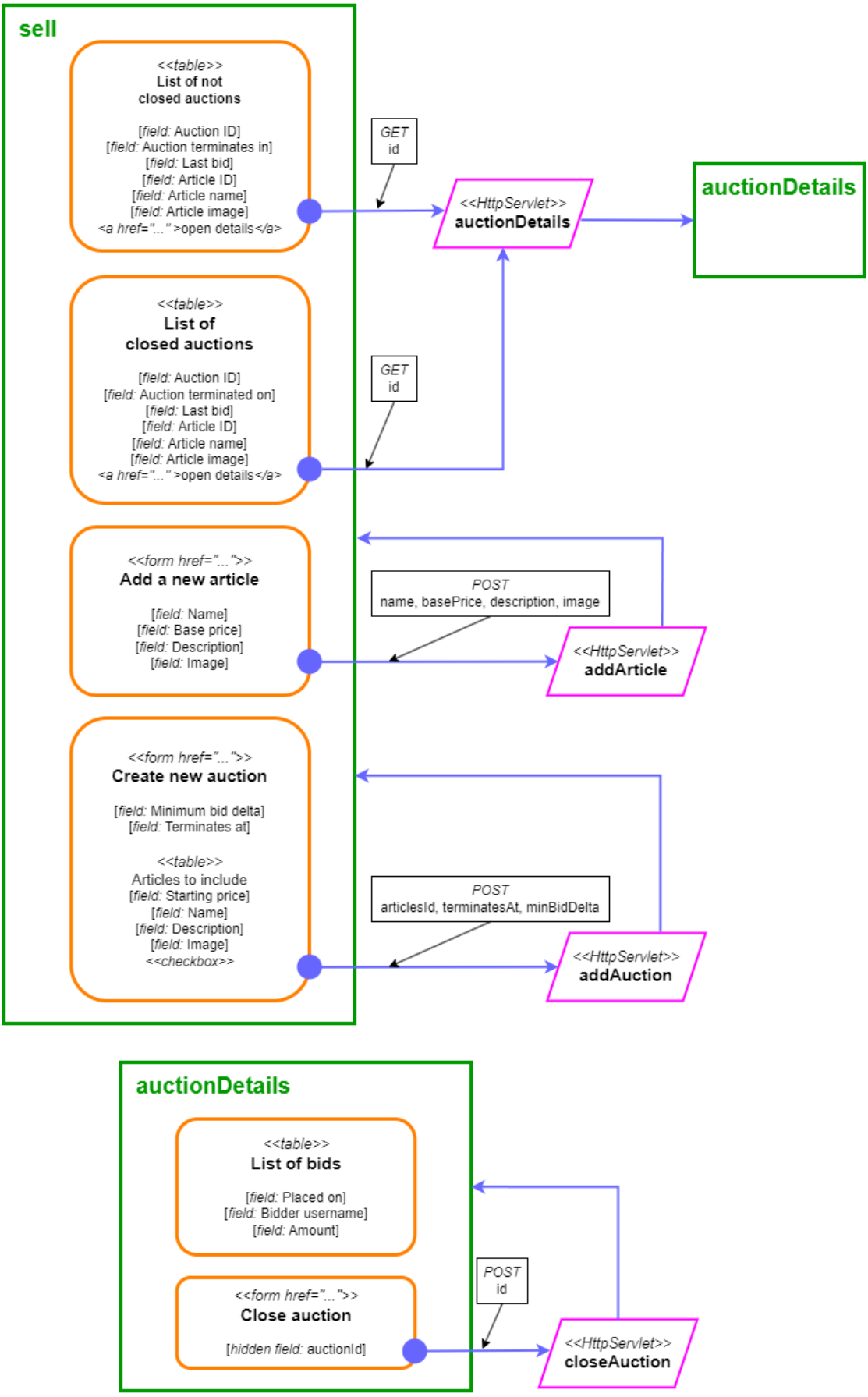
3 Application Design for the Pure HTML Version

3.1 Servlet Structure



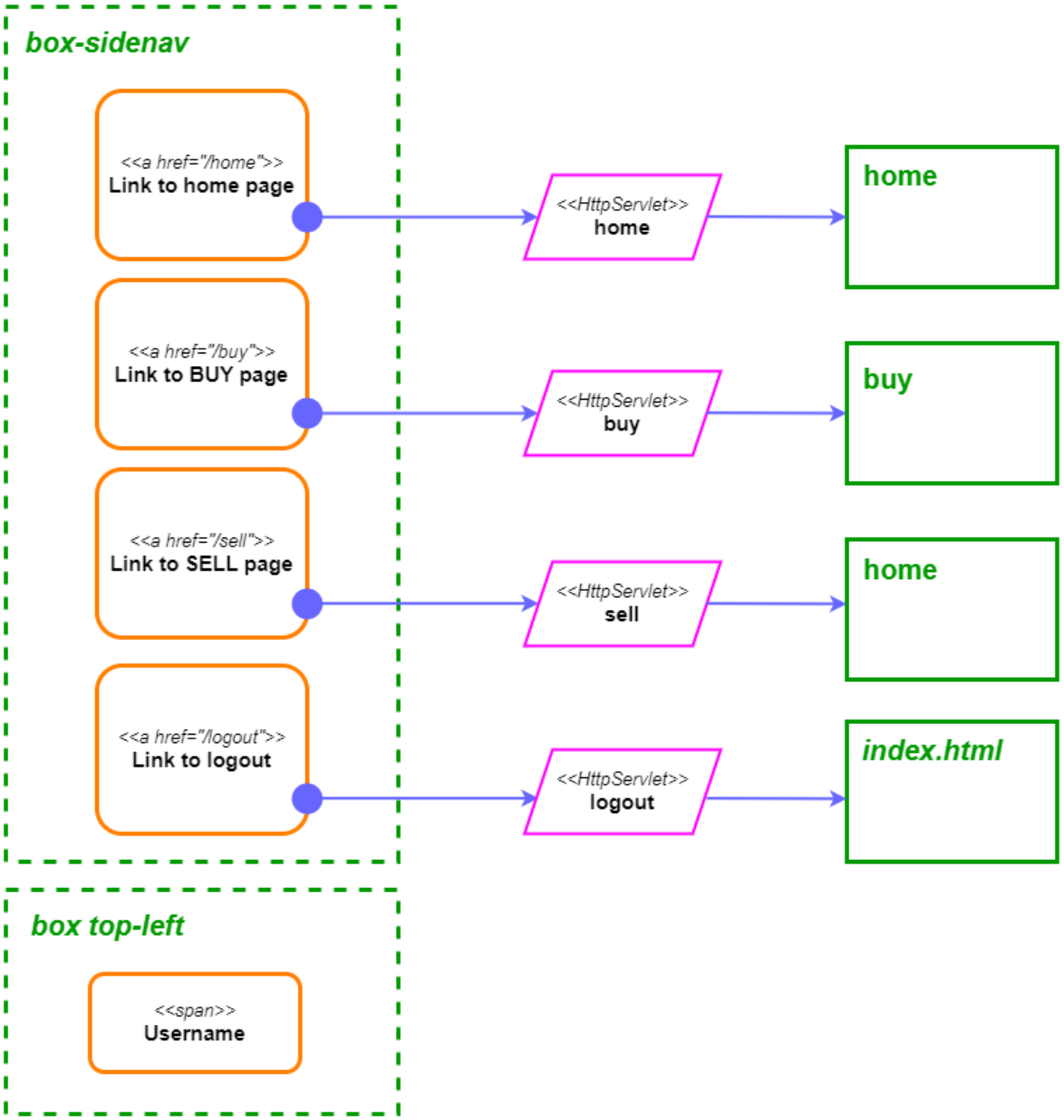

buy
<<form href="buy">>
List of open auctions
[field: query]
<<table>>
List of open auctions
[field: ID]
[field: Terminates on]
[field: Minimum bid delta]
[field: Creator username]
open details
<<table>>
List of auctions won by the user
[field: Auction ID]
[field: Auction terminated on]
[field: Auction creator username]
[field: Article name]
[field: Article description]
[field: Article image]
open details
<<HttpServlet>>
buy
<<HttpServlet>>
auctionOffer
auctionOffer
GET
q
GET
id
GET
id


auctionOffer
<<table>>
List of articles
[field: Starting price]
[field: Name]
[field: Description]
[field: Image]
<<table>>
List of bids
[field: Placed on]
[field: Bidder username]
[field: Amount]
<<form href="...">>
Place bid
[field: Amount]
[hidden field: auction]
POST
auction, bid
<<HttpServlet>>
placeBid


sell
<<table>>
List of not closed auctions
[field: Auction ID]
[field: Auction terminates in]
[field: Last bid]
[field: Article ID]
[field: Article name]
[field: Article image]
open details
GET
id
<<HttpServlet>>
auctionDetails
auctionDetails
<<table>>
List of closed auctions
[field: Auction ID]
[field: Auction terminated on]
[field: Last bid]
[field: Article ID]
[field: Article name]
[field: Article image]
open details
GET
id
<<form href="...">>
Add a new article
[field: Name]
[field: Base price]
[field: Description]
[field: Image]
POST
name, basePrice, description, image
<<HttpServlet>>
addArticle
<<form href="...">>
Create new auction
[field: Minimum bid delta]
[field: Terminates at]
<<table>>
Articles to include
[field: Starting price]
[field: Name]
[field: Description]
[field: Image]
<<checkbox>>
POST
articlesId, terminatesAt, minBidDelta
<<HttpServlet>>
addAuction
auctionDetails
<<table>>
List of bids
[field: Placed on]
[field: Bidder username]
[field: Amount]
<<form href="...">>
Close auction
[hidden field: auctionId]
POST
id
<<HttpServlet>>
closeAuction

3.2 Navigation Elements

To improve the usability of the application, a navigation menu is added to all pages, allowing users to reach the home, buy, and sell pages, and to log out. This element is positioned on the side using appropriate CSS definitions, and to include the HTML text, the *th:insert* functionality of Thymeleaf is used. This way, the same HTML code for displaying common elements is automatically inserted on all pages.



3.3 Access Control

All servlets (except for doLogin and logout) require that the *user* attribute in the session contains an object of type User. This requirement is both practical (the data of that object are needed) and functional (only authenticated users can use the application). To ensure this, a loginChecker servlet of type *Filter* is used. This implements a function that redirects the user to the login page if they are not authenticated.

4 Components for the Pure HTML Version

4.1 Database Access

To manage database connections, a class called ConnectionHandler has been implemented, with servlets using the *getConnection* and *returnConnection* methods. This class ensures that:

- there is no excessive number of open connections to the database,
- no connection stays open for too long without being used,
- servlets (when possible) obtain a connection from those that are ready (not yet closed) rather than creating a new one.

To achieve this, a consumer-producer pattern typical of multithreading programming is used, along with an asynchronous *Runnable* that manages the list of open connections. By using this utility class, the servlets do not need to open database connections during initialization and store them. For this reason, the functionality of DAO objects has been implemented through static methods in the Bean classes.

4.2 Data Beans

- **User**
- **Article**
- **Bid**
- **Auction**, with the addition of the subclasses **AuctionClosed** and **AuctionWithArticles**

4.3 DAO Functionality

To ensure greater robustness and facilitate debugging of the following functions, automatic tests have been implemented using the junit framework integrated with maven. This allows the correctness of the interaction with the database to be verified during the compilation phase. Although the work done is far from a "test-driven development" paradigm, also due to the stateful nature of the operations being tested, the simple tests performed ensure the correct functioning of a portion of the code.

4.3.1 User

- loadWithCreds(username, password)
- loadWithId(id)
- loadWithUsername(username)

4.3.2 Article

- getArticlesByAuction(auction_id)
- getArticlesByUser(user_id)
- createArticle(owner_user_id, base_price, name, description, imagePart)
- checkOwner(user_id, articles_id)

4.3.3 Bid

- getBidsPerAuction(auction_id)
- getBidsPerUser(user)
- placeBid(bidder_user_id, auction_id, amount)

4.3.4 Auction

- getAuctionFromId(id)
- getAuctions()
- getAuctionsPerUser(user_id)
- getAuctionsWonByUser(user_id)
- getAuctionsWithSearch(expression)
- getMultipleAuctionWithArticles(auctionList)
- closeAuction(auction_id)
- createAuction(creator_user_id, terminates_at, minimum_bid_delta, articles_id)

4.4 Servlets

4.4.1 Controllers

- doLogin
- home
- buy
- auctionOffer
- placeBid
- sell
- auctionDetails
- addArticle
- addAuction
- closeAuction
- logout

4.4.2 Filters

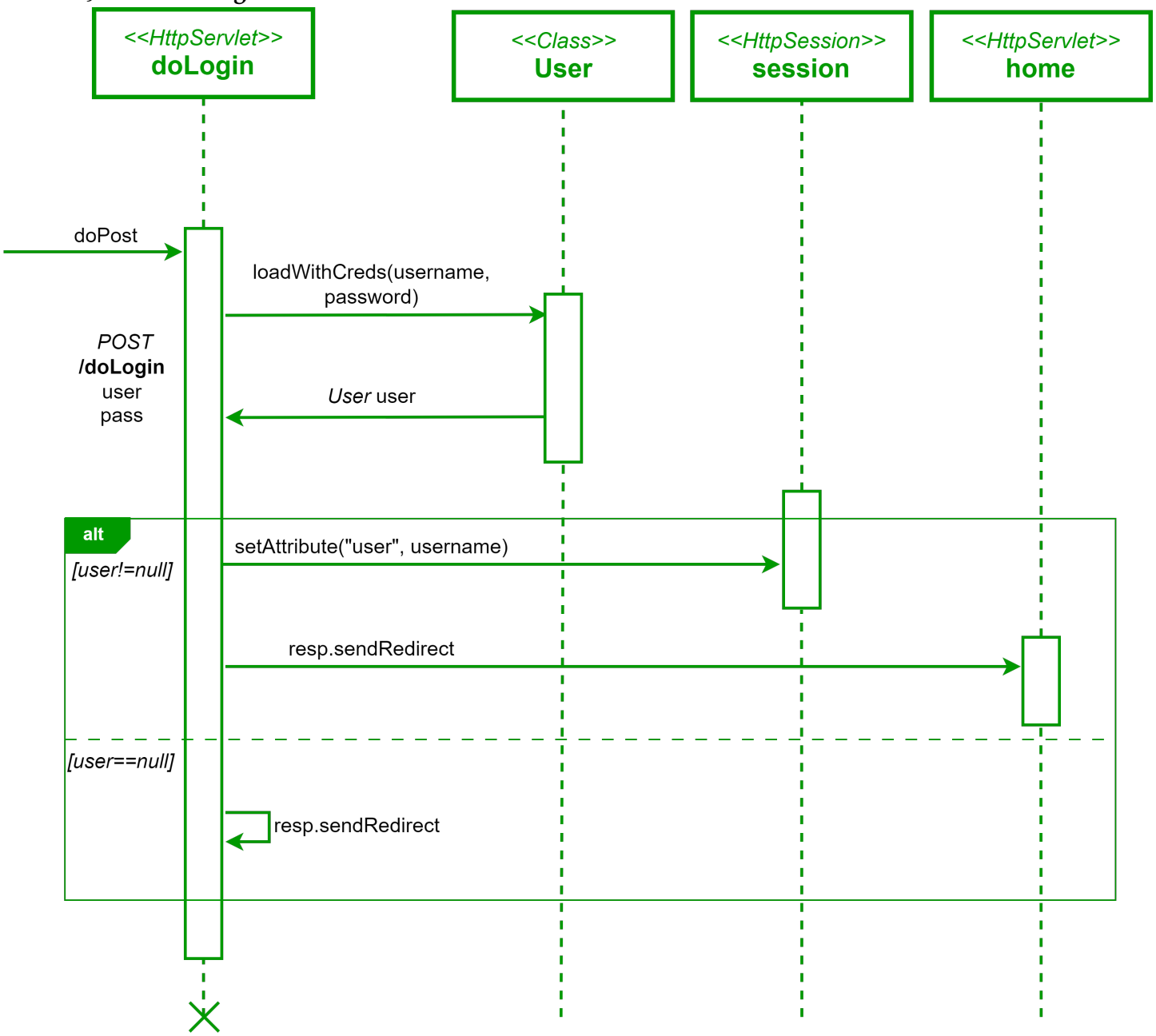
- loginChecker

4.5 Thymeleaf Template

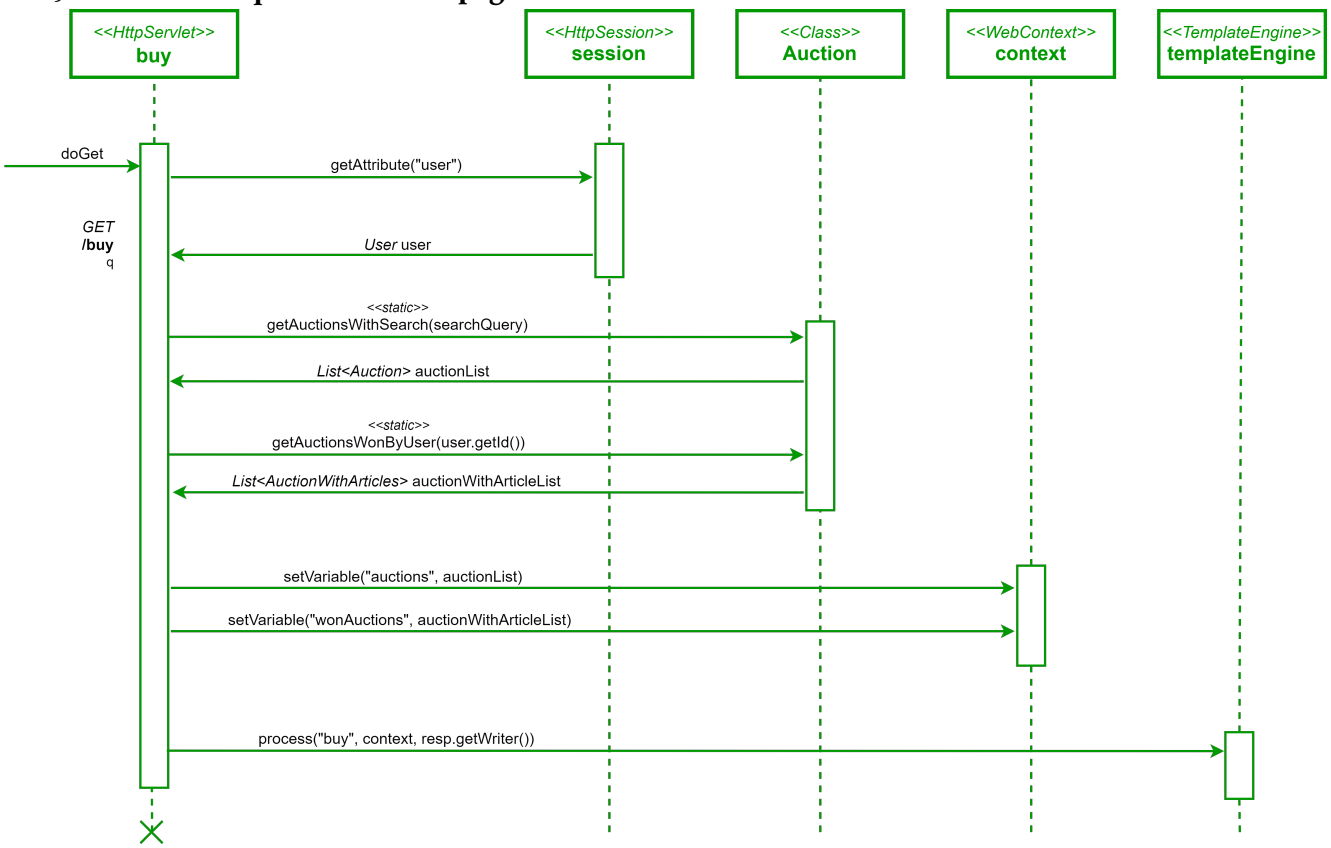
- index
- home
- buy
- auctionOffer
- sell
- auctionDetails
- header, this is a fragment

5 Events and Sequence Diagrams for the Pure HTML Version

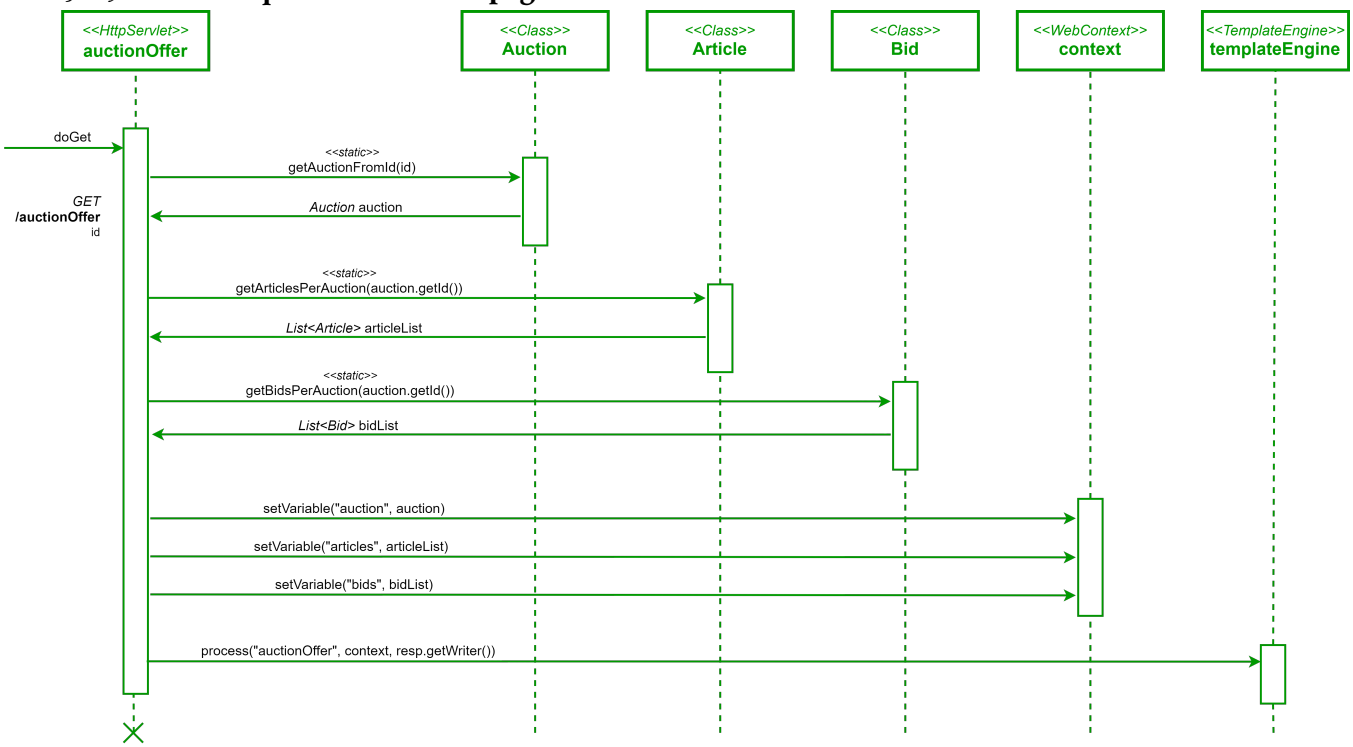
5.0.1 A user logs in



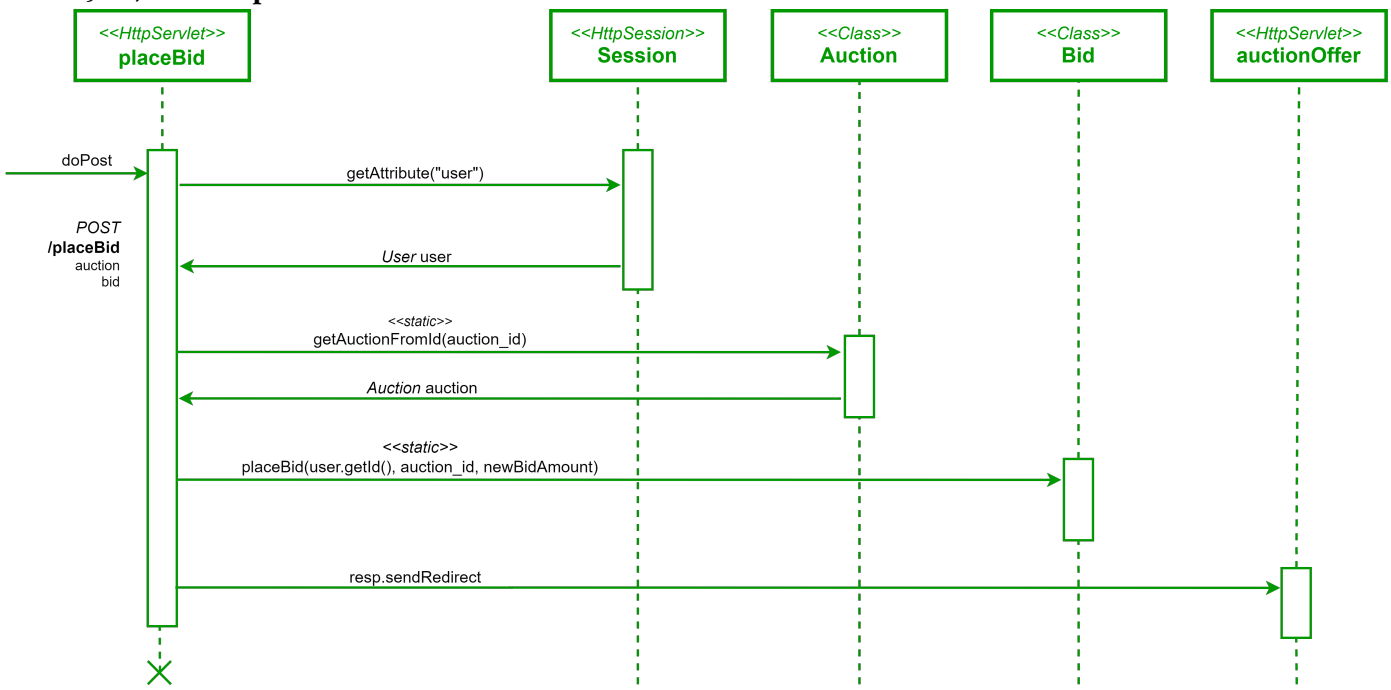
5.0.2 A user requests the BUY page



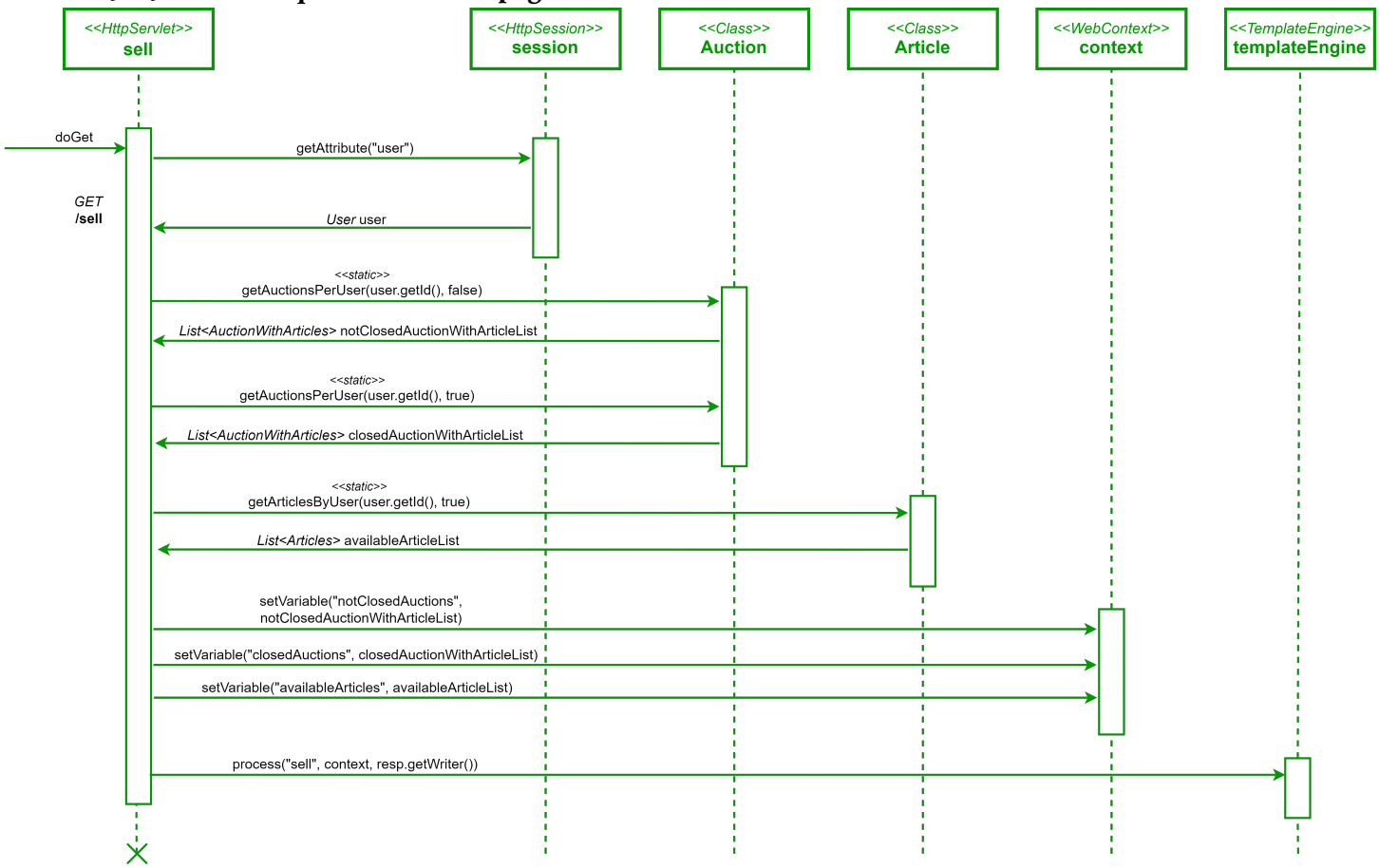
5.0.3 A user requests the details page of an auction



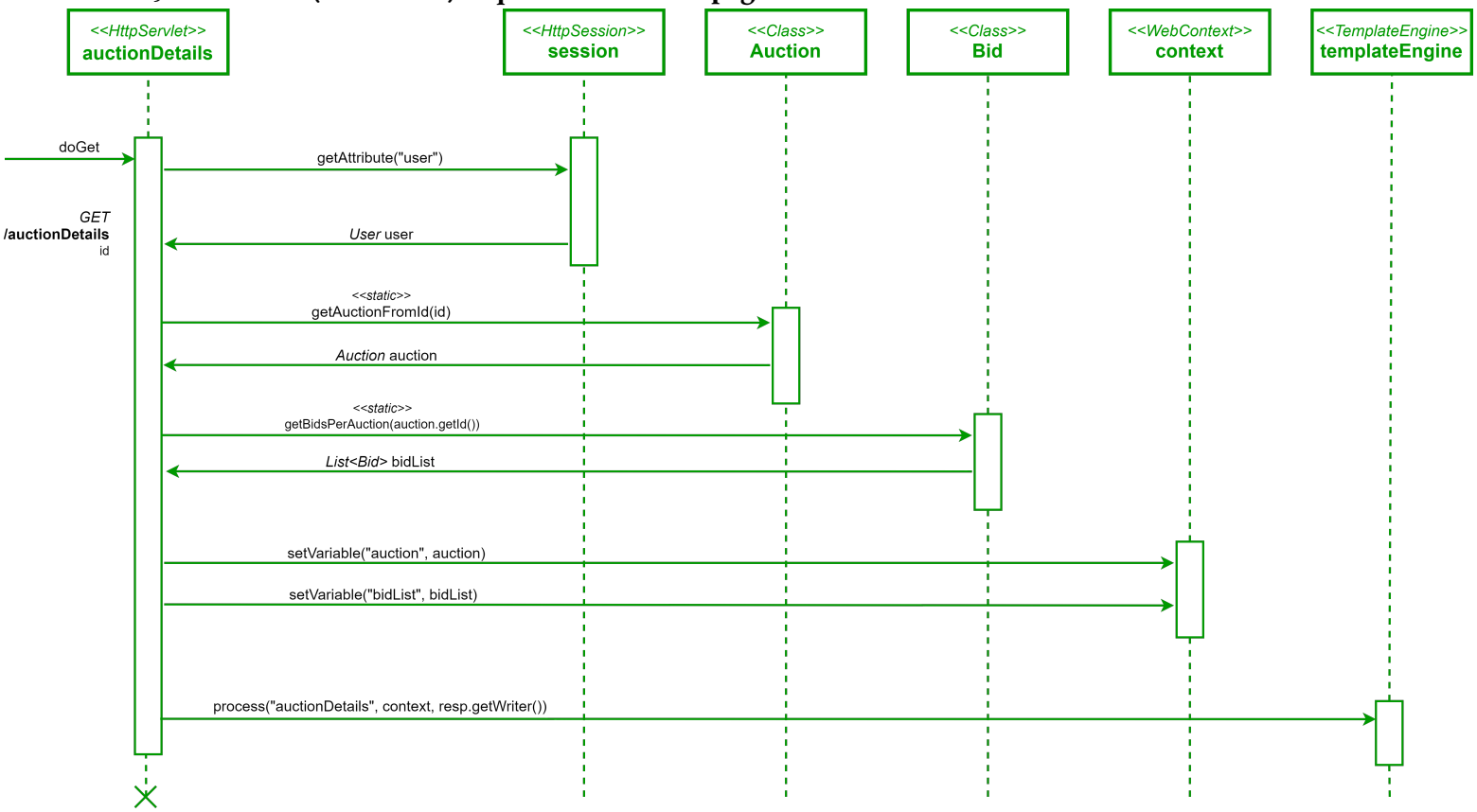
5.0.4 A user places a new bid



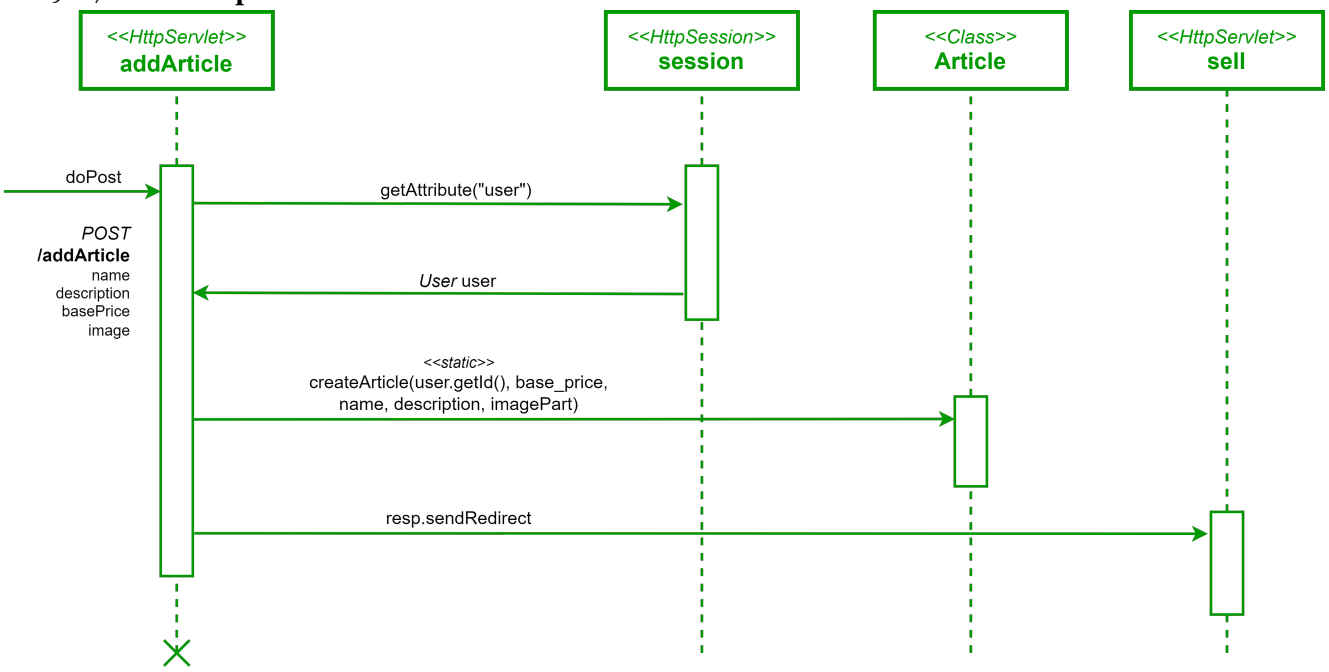
5.0.5 A user requests the SELL page



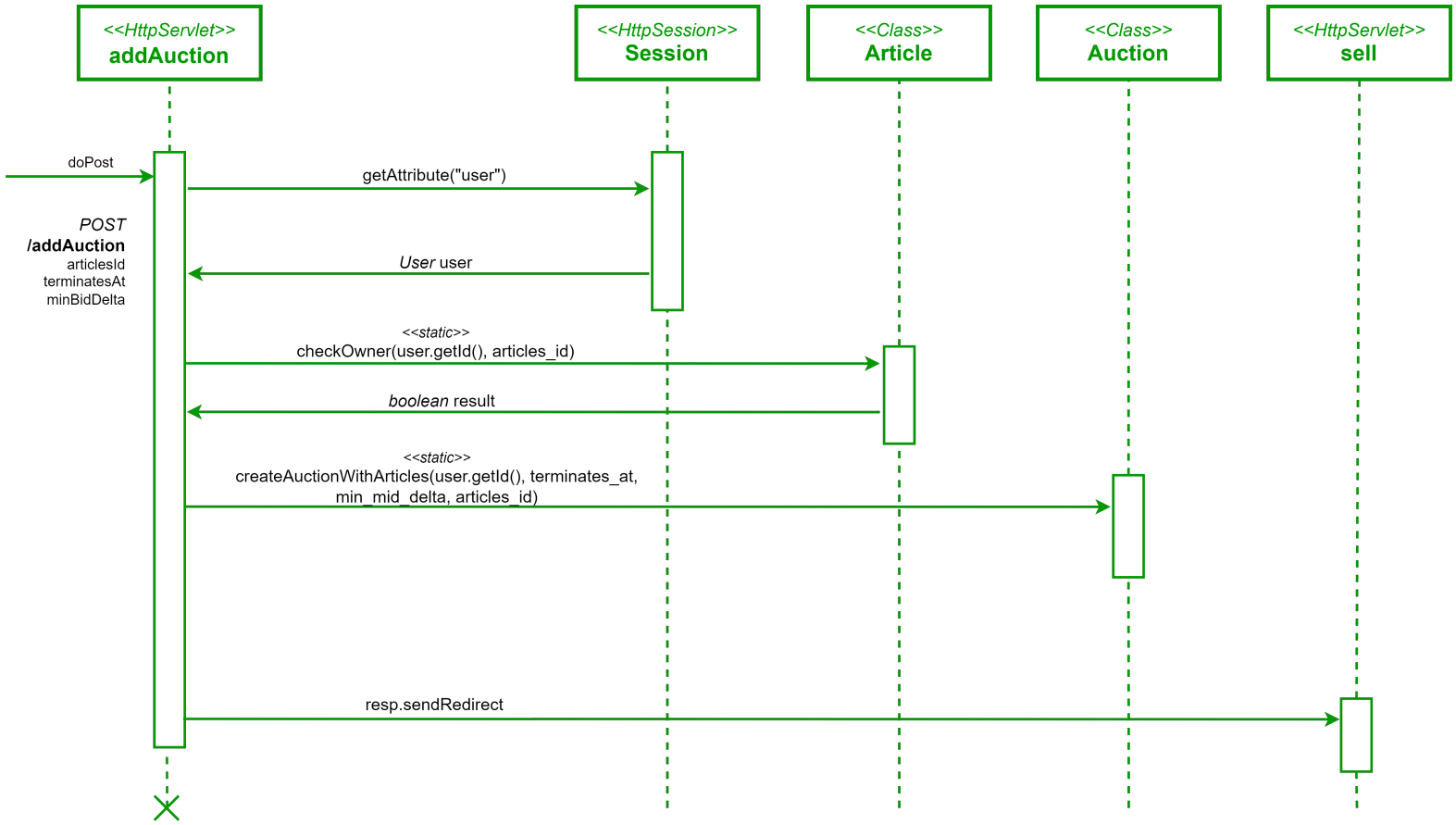
5.0.6 A user (the owner) requests the details page of their auction



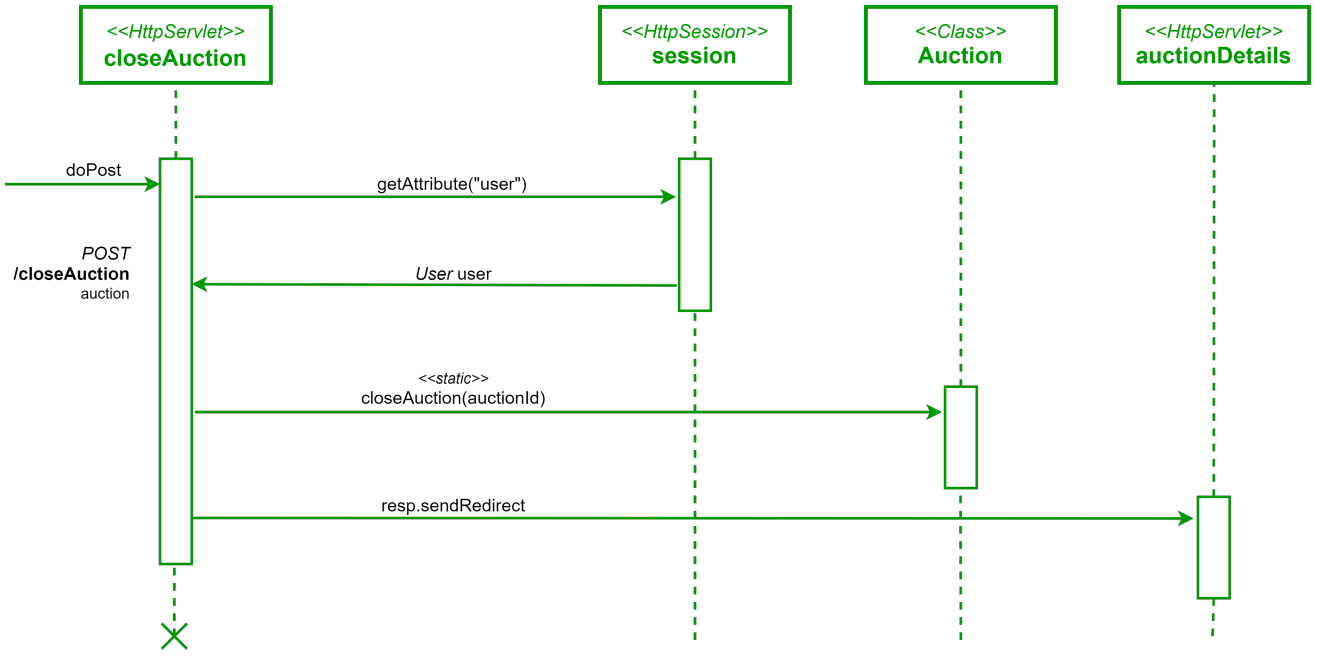
5.0.7 A user uploads a new article



5.0.8 A user creates a new auction

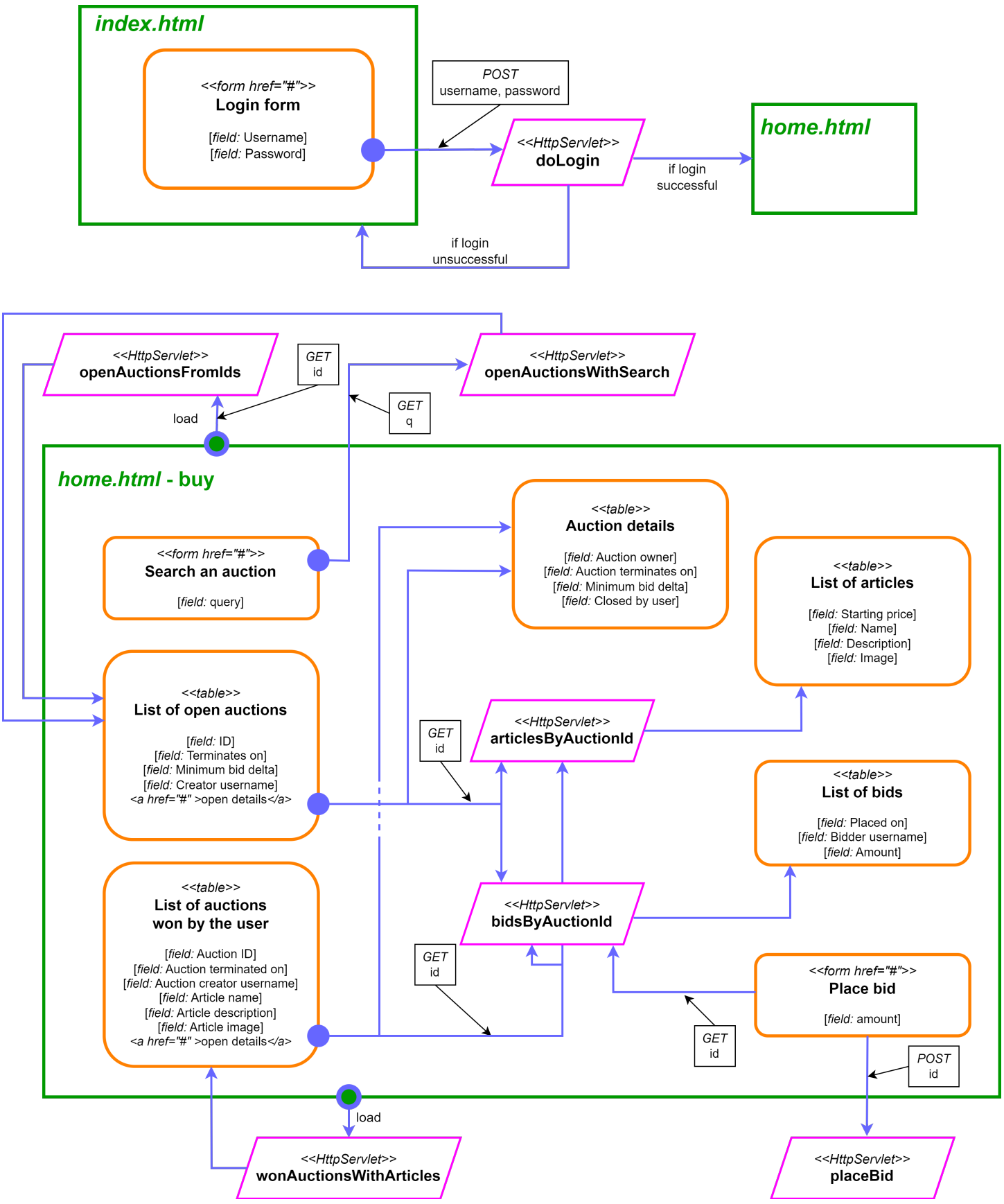


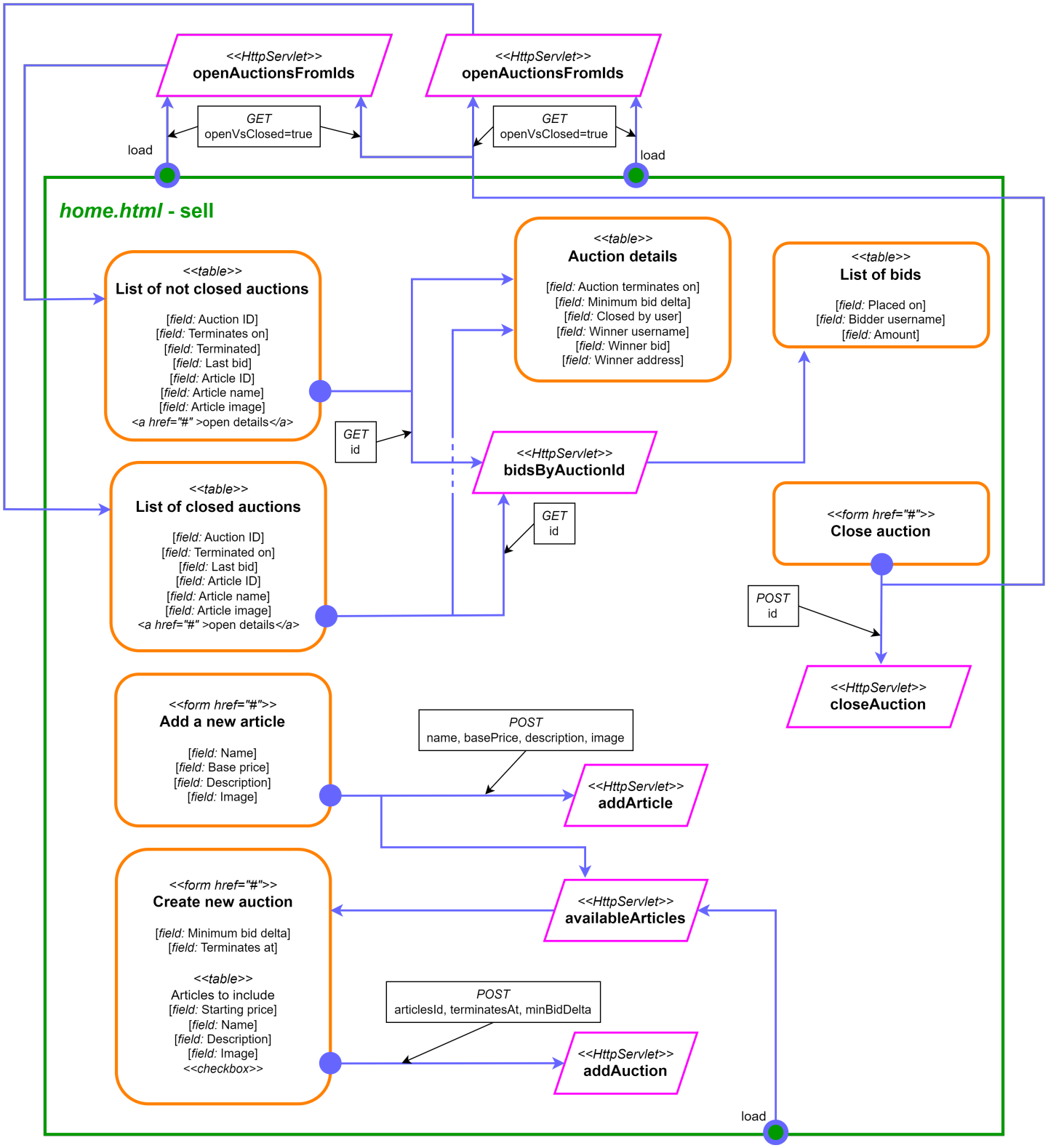
5.0.9 A user (the owner) closes a finished auction



6 Application Design for the JavaScript Version

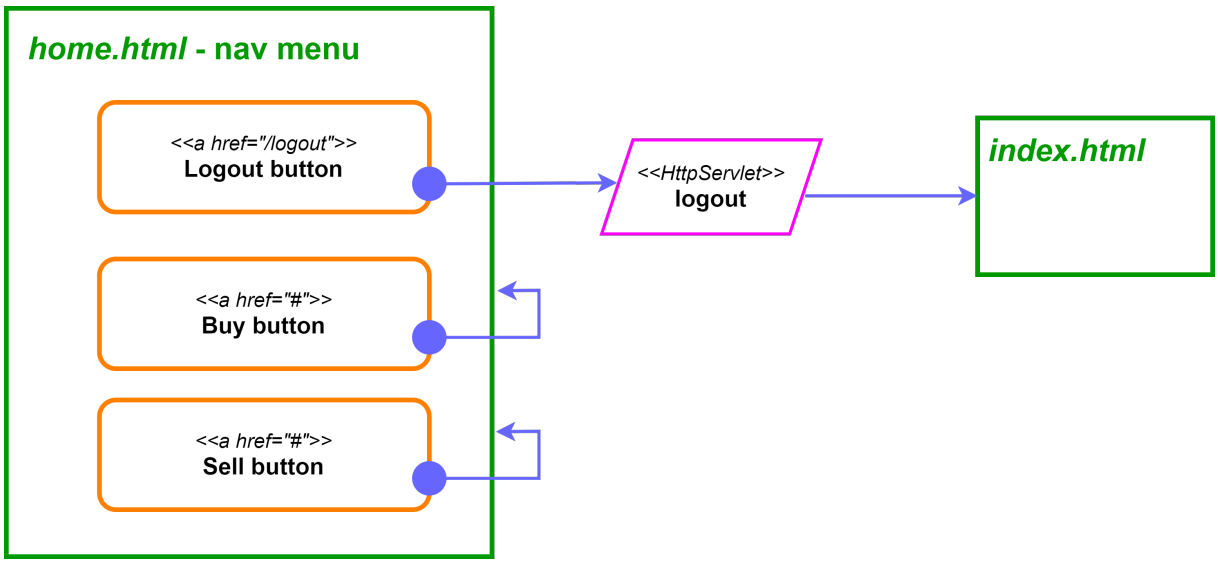
6.1 Page Structure




<<HttpServlet>>
openAuctionsFromIds
GET
openVsClosed=true
load
<<HttpServlet>>
openAuctionsFromIds
GET
openVsClosed=true
load
home.html - sell
<<table>>
List of not closed auctions
[field: Auction ID]
[field: Terminates on]
[field: Terminated]
[field: Last bid]
[field: Article ID]
[field: Article name]
[field: Article image]
open details
<<table>>
Auction details
[field: Auction terminates on]
[field: Minimum bid delta]
[field: Closed by user]
[field: Winner username]
[field: Winner bid]
[field: Winner address]
<<table>>
List of bids
[field: Placed on]
[field: Bidder username]
[field: Amount]
GET
id
<<HttpServlet>>
bidsByAuctionId
<<table>>
List of closed auctions
[field: Auction ID]
[field: Terminated on]
[field: Last bid]
[field: Article ID]
[field: Article name]
[field: Article image]
open details
GET
id
<<form href="#">>
Close auction
POST
id
<<HttpServlet>>
closeAuction
<<form href="#">>
Add a new article
[field: Name]
[field: Base price]
[field: Description]
[field: Image]
POST
name, basePrice, description, image
<<HttpServlet>>
addArticle
<<form href="#">>
Create new auction
[field: Minimum bid delta]
[field: Terminates at]
<<table>>
Articles to include
[field: Starting price]
[field: Name]
[field: Description]
[field: Image]
<<checkbox>>
<<HttpServlet>>
availableArticles
POST
articlesId, terminatesAt, minBidDelta
<<HttpServlet>>
addAuction
load

6.2 Navigation Elements

The navigation menu allows the user to log out and choose between the buying and selling functionalities. This element is positioned to the side using appropriate CSS definitions.



6.3 Access Control and Error Handling

All servlets (except for 'doLogin' and 'logout') require that the 'user' attribute in the session contains an object of type 'User': this requirement is both practical (the data of that object is necessary) and functional (only authenticated users can use the application). To ensure this, a 'loginChecker' servlet of type *Filter* is used. This implements a function that returns an error code *403-Forbidden* if the user is not authenticated. With this information, JavaScript utility functions for making connections check the response status and redirect the user to the login page if unauthorized.
When the data sent to the server is incorrect, a *status code 400* is returned. The callback functions handle this by extracting the error message from the response and displaying it to the user in the appropriate *div* elements.

7 Events, Actions, and Controllers for the JavaScript Version

7.1 Events and Actions

Client Side		Server Side	
Event	Action	Event	Action
index.html > login form > submit	Character validation > redirect	*POST* (user, pass)	Credential validation > load user
home.html > nav menu > click buy	Load buy items	*GET* (ids), *GET*	Load open auctions, load won auctions with articles
home.html > nav menu > click sell	Load sell items	*GET*	Load auctions with articles
home.html > nav menu > click logout	Reset username	*GET*	Close session
home.html buy > search form > submit	Update open auctions	*GET* (query)	Load auctions with search
home.html buy > open auctions list > click open details	Update detail	*GET* (id)	Load articles, load bids
home.html buy > won auctions list > click open details	Update detail	*GET* (id)	Load articles, load bids
home.html buy > auction detail > submit bid	Update bids	*POST* (auction, bid), *GET*	Update bids > load bids
home.html sell > non-closed auctions list > click open details	Update detail with close form	*GET* (id)	Load bids
home.html sell > closed auctions list > click open details	Update detail	*GET* (id)	Load bids
home.html sell > add article form > submit	Update available articles list	*POST* (name, desc, price, image)	Add article > load available articles
home.html sell > create auction form > submit	Update available articles, update non-closed auctions	*POST* (raise, date, article list), *GET*, *GET*	Create auction > load articles, load closed and non-closed auctions
home.html sell > close auction form > submit	Update auctions	*POST* (id), *GET*, *GET*	Close auction > load closed and non-closed auctions

7.2 Events and Controllers

Client Side		Server Side	
Event	Controller	Event	Controller
index.html > login form > submit	handleEvent > makeCall > update	*POST* (user, pass)	doLogin
home.html > nav menu > click buy	loadBuy > makeCall > initBuy	*GET* (ids), *GET*	auctionsFromIds, wonAuctionsWithArticles
home.html > nav menu > click sell	loadSell > makeCall > initSell	*GET*	auctionsForOwner, availableArticles
home.html > nav menu > click logout	logout	*GET*	logout
home.html buy > search form > submit	show > makeCall > update	*GET* (query)	openAuctionsWithSearch
home.html buy > open auctions list > click open details	show > makeCall > update	*GET* (id)	articlesByAuctionId, bidsByAuctionId
home.html buy > won auctions list > click open details	show > makeCall > update	*GET* (id)	articlesByAuctionId, bidsByAuctionId
home.html buy > auction detail > submit bid	handleBidPlacement > makeCall > bidsRefreshCallBack	*POST* (auction, bid), *GET*	placeBid > bidsByAuctionId
home.html sell > non-closed auctions list > click open details	show > makeCall > update	*GET* (id)	bidsByAuctionId
home.html sell > closed auctions list > click open details	show > makeCall > update	*GET* (id)	bidsByAuctionId
home.html sell > add article form > submit	handleEvent > makeCall > articlesRefresh	*POST* (name, desc, price, image)	addArticle > availableArticles
home.html sell > create auction form > submit	handleEvent > makeCall > show	*POST* (raise, date, article list), *GET*, *GET*	addAuction > availableArticles, auctionsForOwner
home.html sell > close auction form > submit	handleAuctionClosing > makeCall > auctionClosedUpdate	*POST* (id), *GET*, *GET*	closeAuction > auctionsForOwner

8 Components for the JavaScript Version

8.1 Database Access, Data Beans, and DAOs

For the components related to database access, data beans, and DAO functionality, please refer to the section dedicated to the HTML-only version, as these remain unchanged.

8.2 Servlets

8.2.1 Controllers

- doLogin
- auctionsFromIds
- openAuctionsWithSearch
- wonAuctionsWithArticles
- articlesByAuctionId
- bidsByAuctionId
- placeBid
- auctionsForOwner
- availableArticles
- addArticle
- addAuction
- closeAuction
- logout

8.2.2 Filters

- loginChecker

8.3 Client-Side Views

The components that manage a specific view generally consist of the following methods:

- **init**, used during page load;
- **show** and **handleEvent**, registered as event handlers, invoke `makeCall`;
- **callback**, invoked by `makeCall` when the server responds with the requested data, calls `update` when no errors occur;
- **update**, used to update the HTML page through the DOM.

The client-side JavaScript code is structured as follows:

8.3.1 utils.js Utility functions common to multiple components of the application:

- **logout**;
- various **makeCall** functions, depending on how the parameters should be passed to the server;
- **autoClicker**, a utility object with the `autoClick` method to click the first anchor in the provided DOM object;
- utility functions for working with **dates** and properly formatting a timestamp;
- **auctionDetailsShow** and **auctionWithArticlesCallback**, common functions for multiple components;
- functions to save auction data in the session **storage** and retrieve that information;
- **setLastAction**, to store the last performed action in permanent local storage;
- **BidsList**, a component that manages a list of bids for a given auction.

8.3.2 **home.js** *PageOrchestrator*, responsible for loading the buy/sell functionalities and constructing the objects that interact with the DOM. Key methods of this component include:

- init: retrieves the username, initializes event listeners in the navigation menu, fetches the last action, and determines whether to load the buy or sell functionalities;
- initBuy: constructs all the components for the buy functionality and initializes them, registering event listeners;
- initSell: constructs all the components for the sell functionality and initializes them, registering event listeners.

Each component of the buy and sell functionalities is associated with a constructor, so that all the necessary functions for that component become methods of the object that will be created. Here are the constructors for each:

8.3.3 **buy.js**

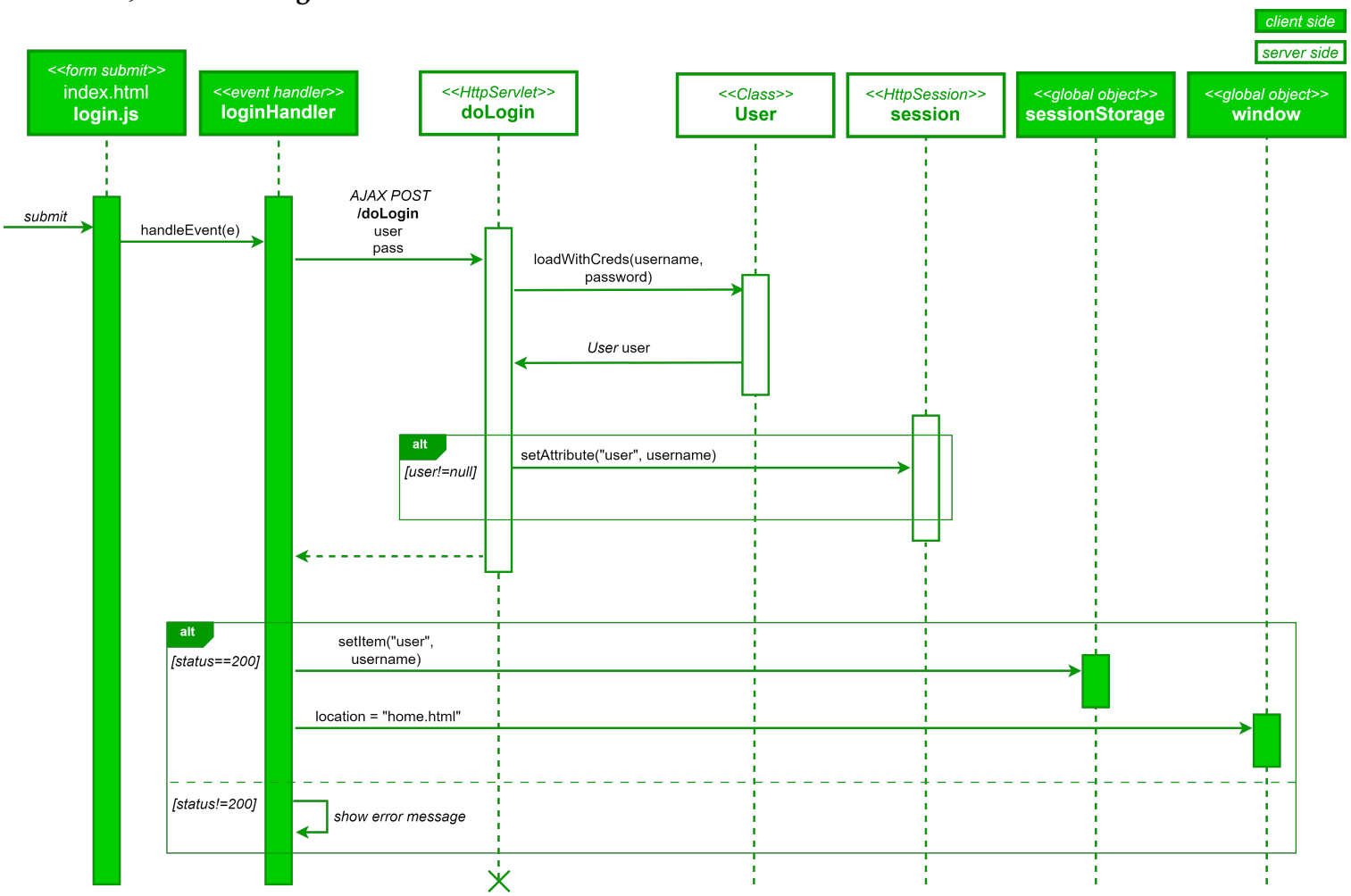
- **saveAuctionAsVisited**, a utility function;
- **OpenAuctionsWithSearch**, a table to search through open auctions;
- **ArticlesList**, a list of articles in the details of an auction;
- **AuctionDetails**, the outermost div of auction details, contains the form to place a bid;
- **WonAuctionsWithArticles**, a table of won auctions and the associated articles.

8.3.4 **sell.js**

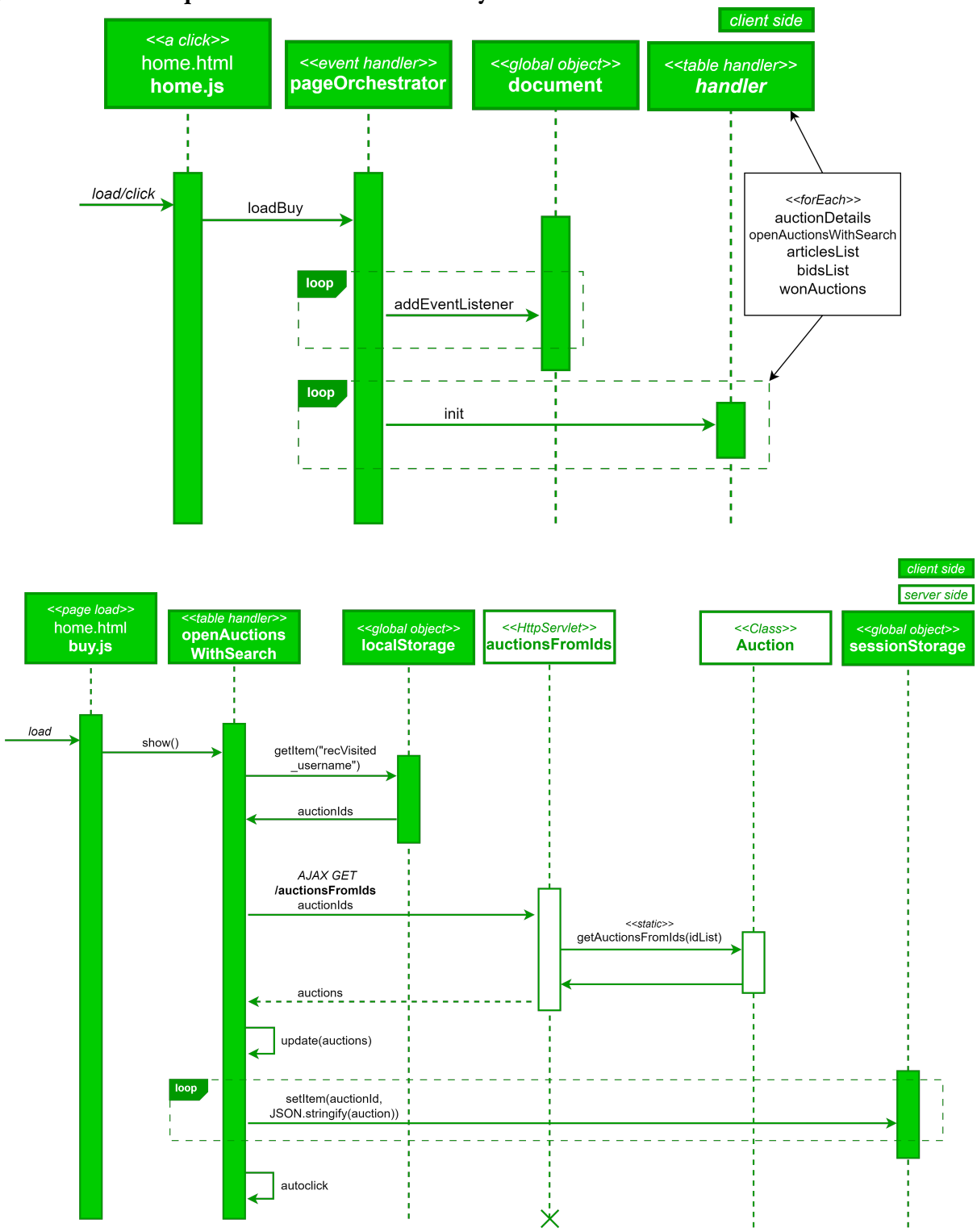
- **AuctionsWithArticles**, a list of auctions with their articles, used for both open and closed auctions;
- **AuctionDetailsOwner**, the outermost div of auction details, optionally contains the button to close the auction;
- **AddArticle**, a form for adding a new article;
- **AddAuction**, a form for creating an auction, includes the list of available articles.

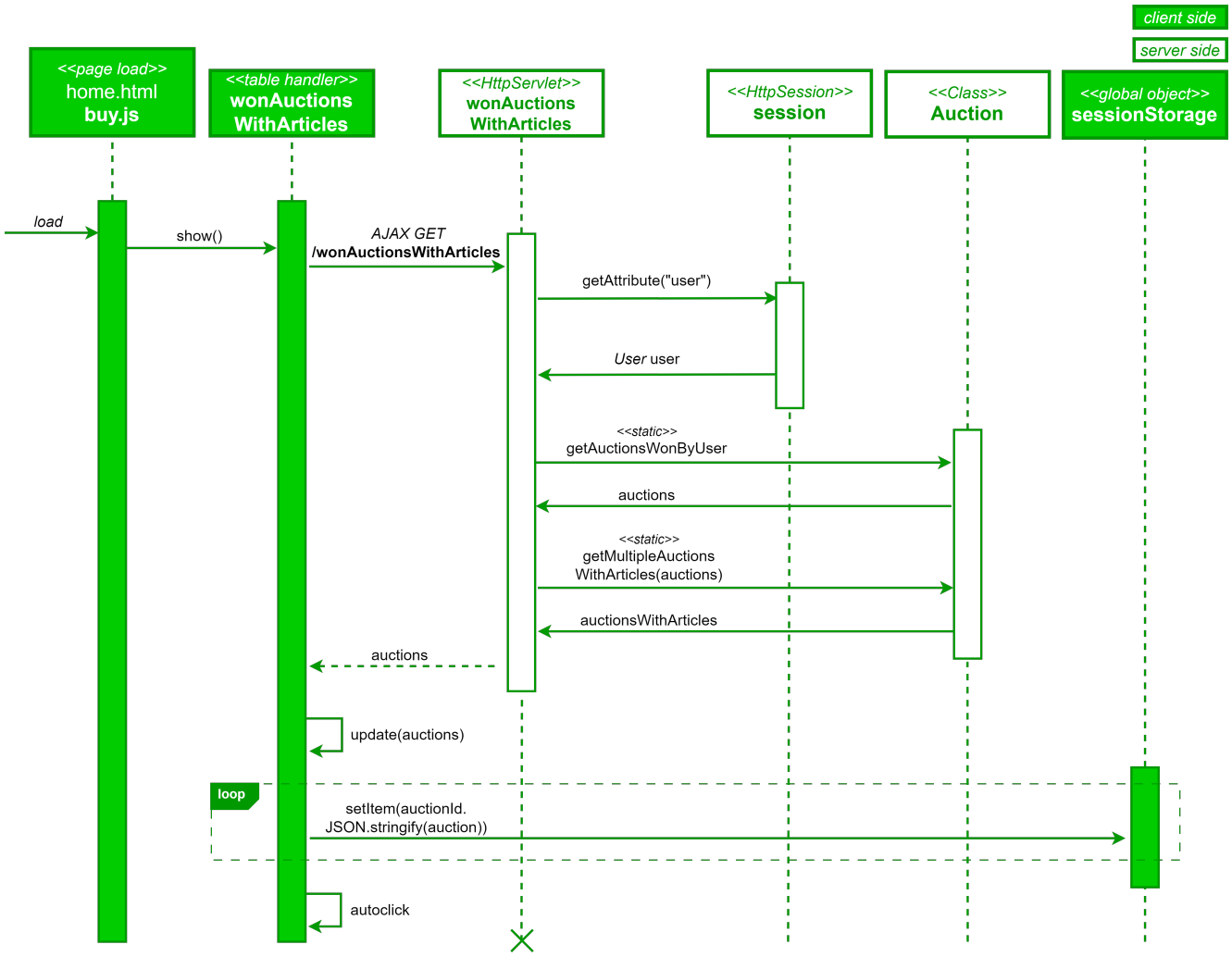
9 Events and Sequence Diagrams for the JavaScript Version

9.0.1 A user logs in

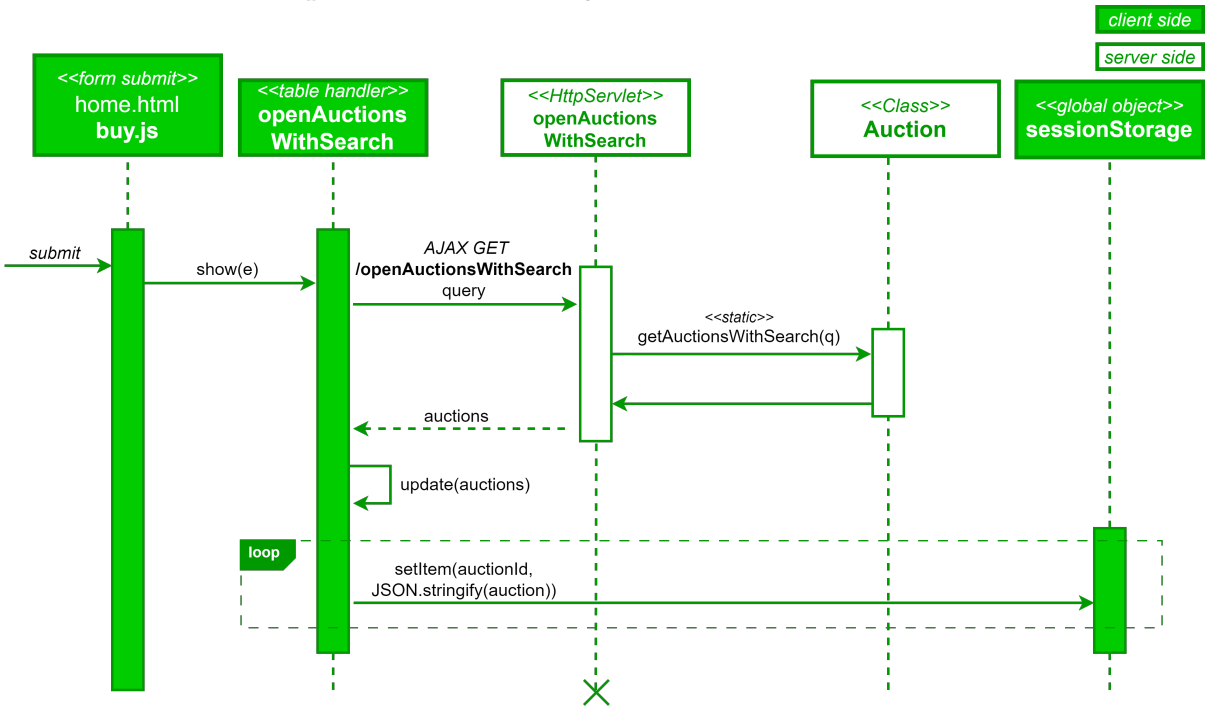


9.0.2 The user requests the BUY functionality

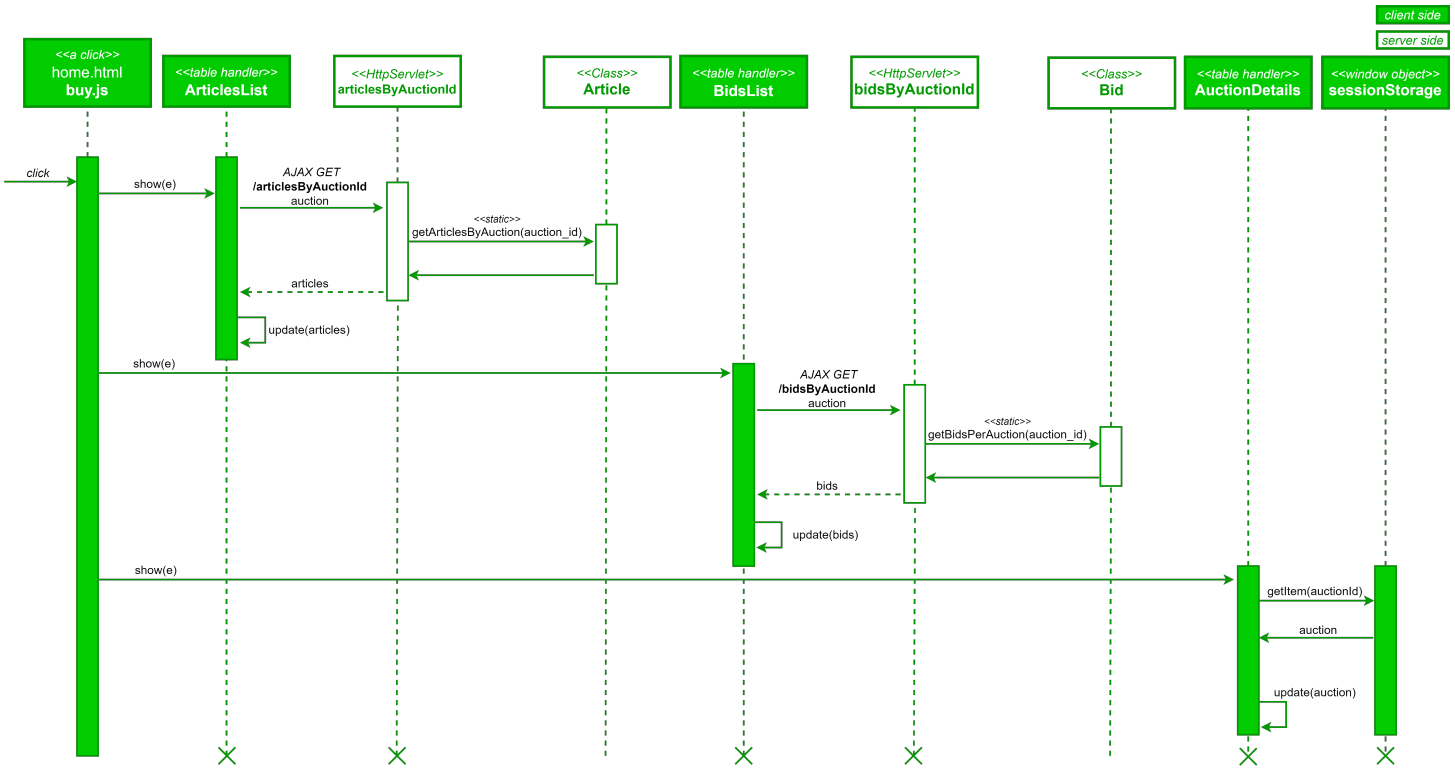




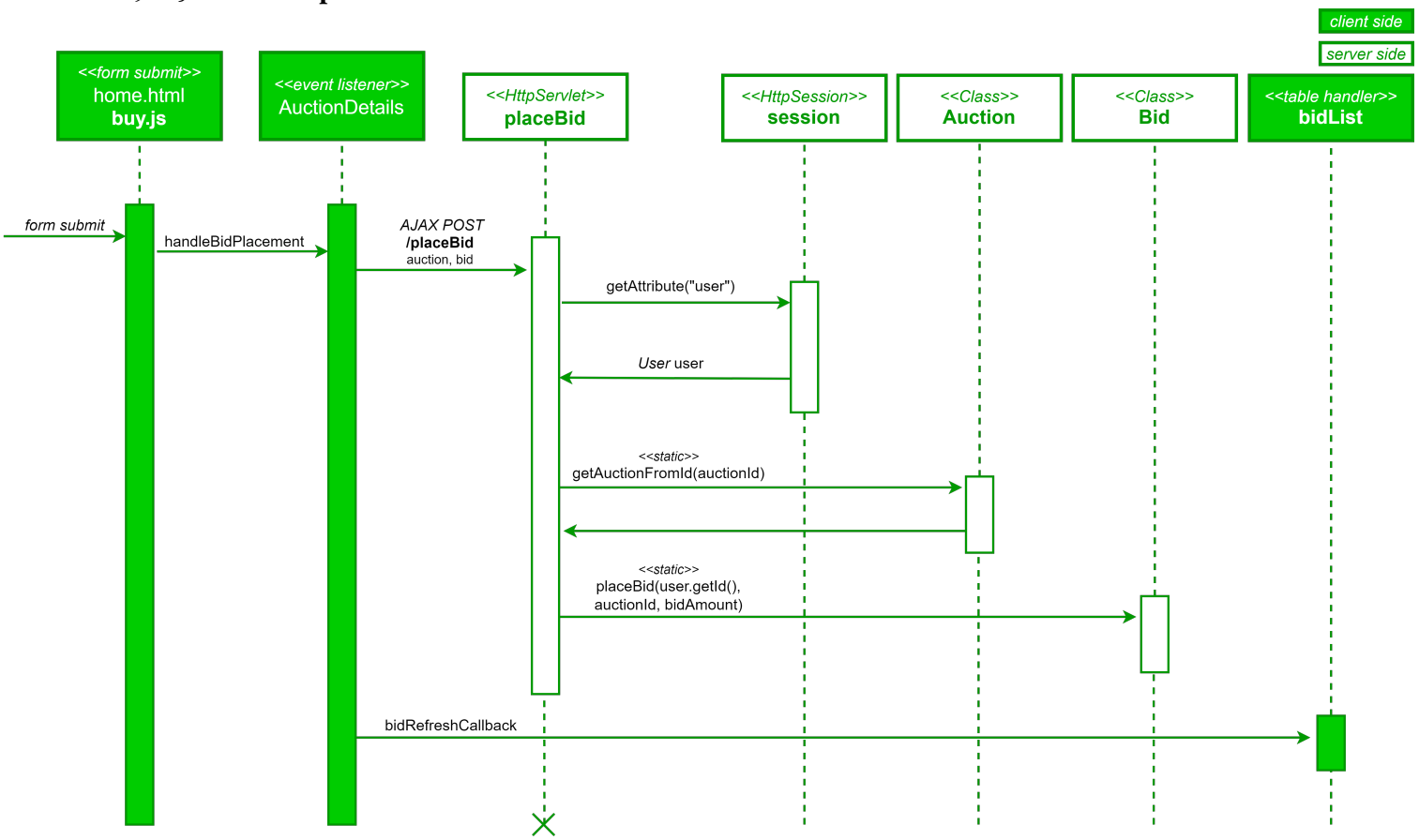
9.0.3 The user enters a query to search among the auctions



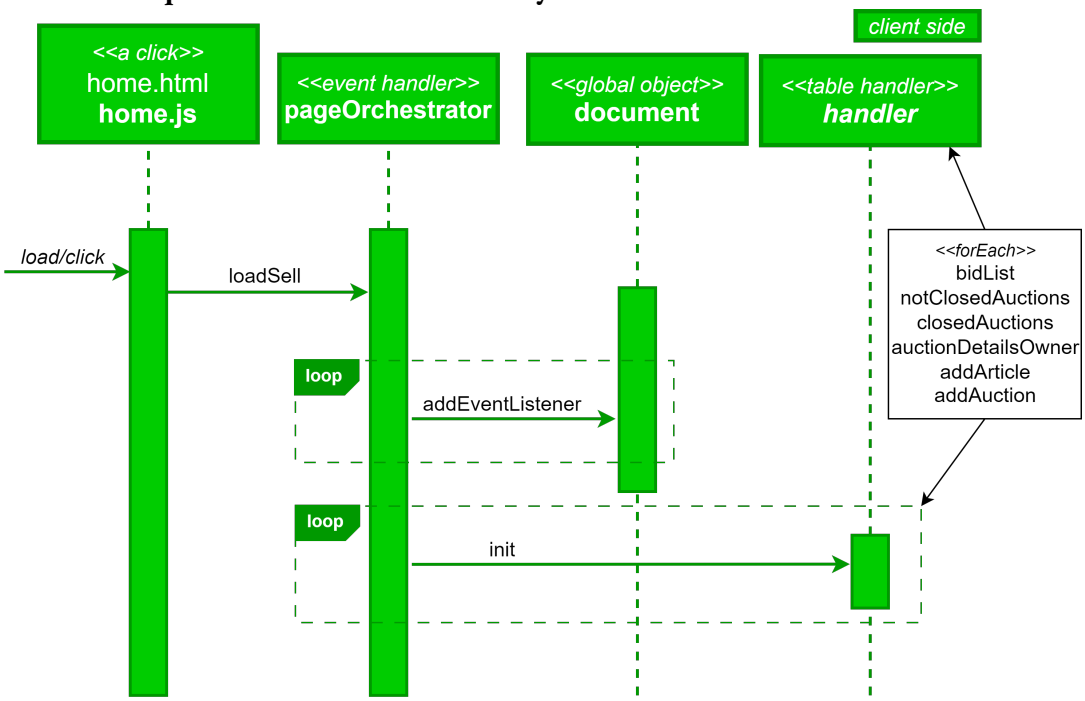
9.0.4 The user clicks to view the details of an auction



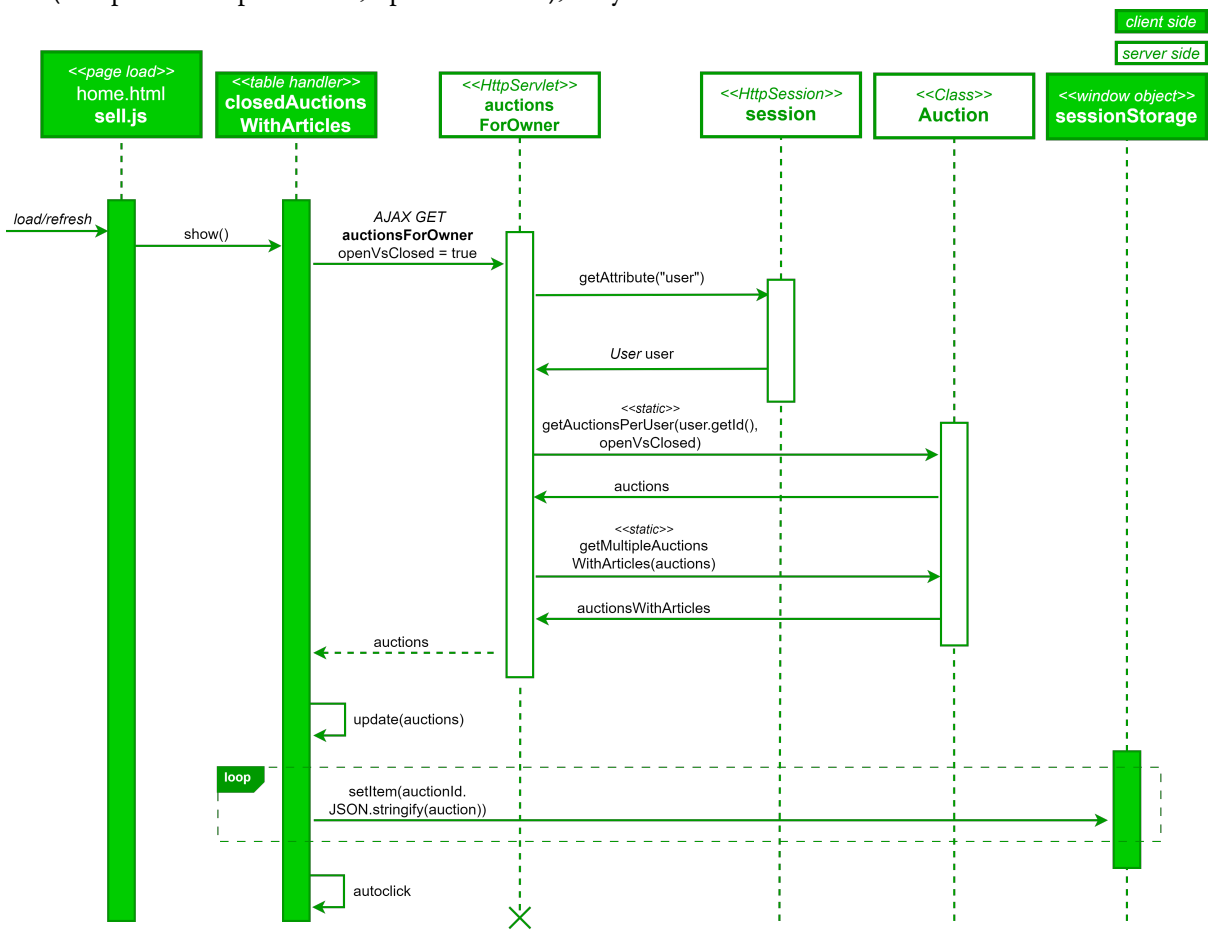
9.0.5 The user places a new bid



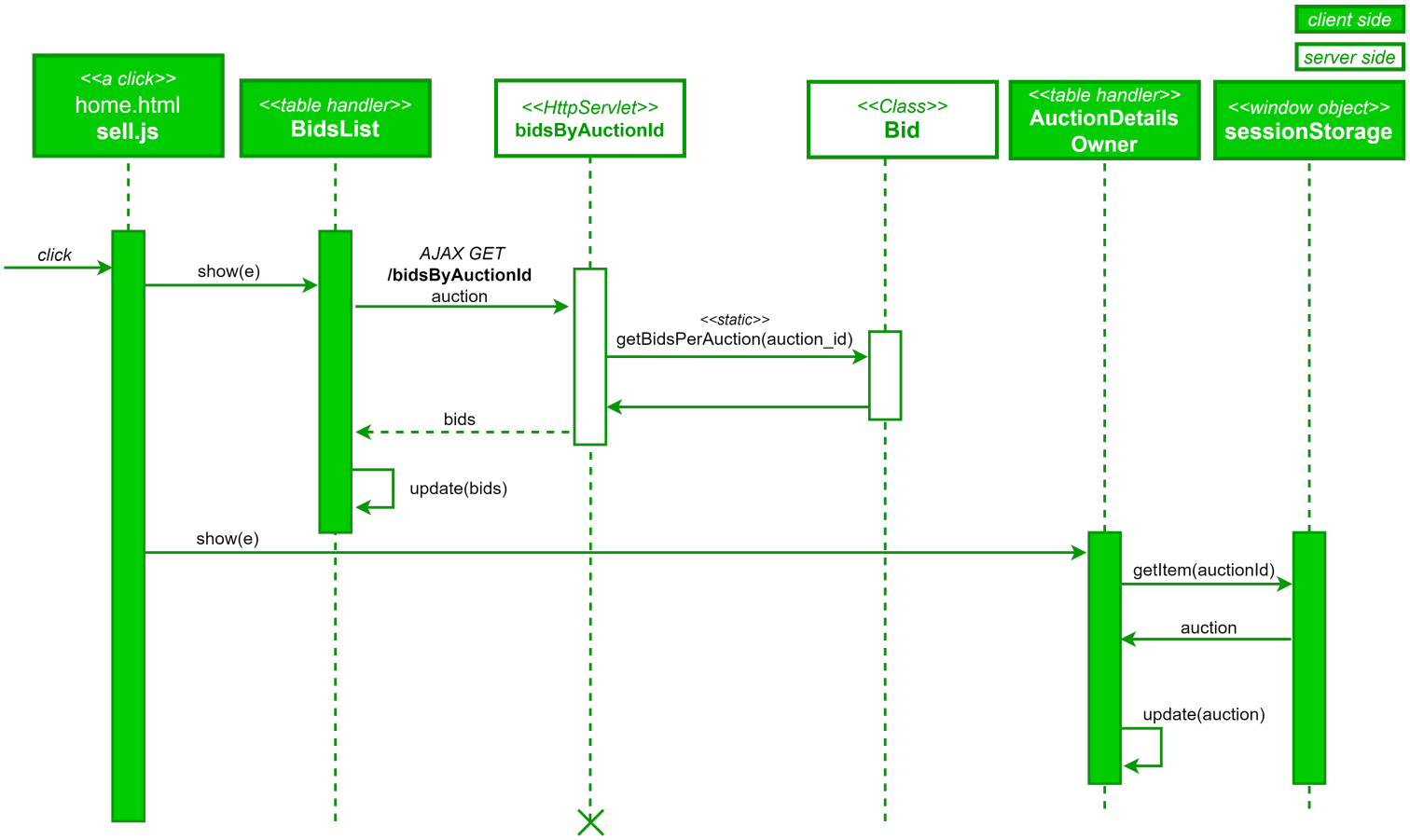
9.0.6 The user requests the SELL functionality



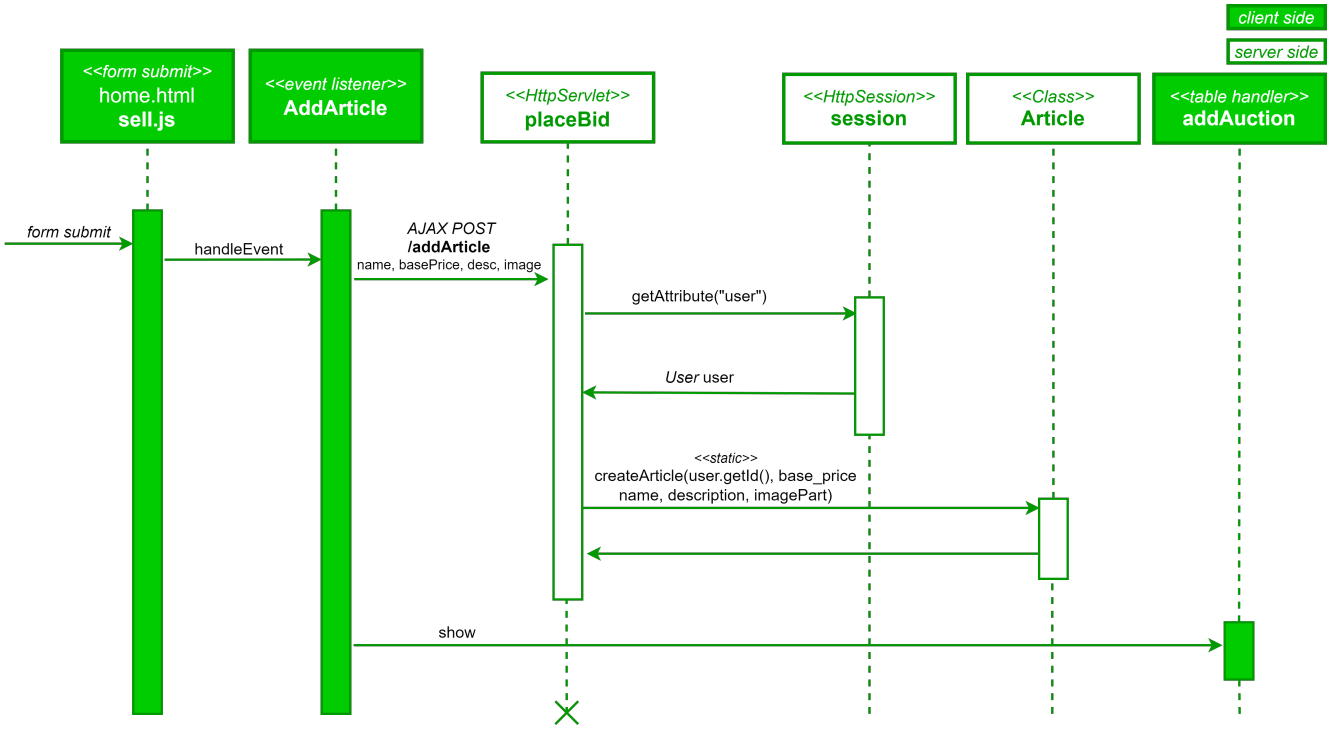
At the same time, both closed and open auctions are loaded. Since the sequence of operations is identical (except for one parameter, openVsClosed), only one of the two flows is shown here.



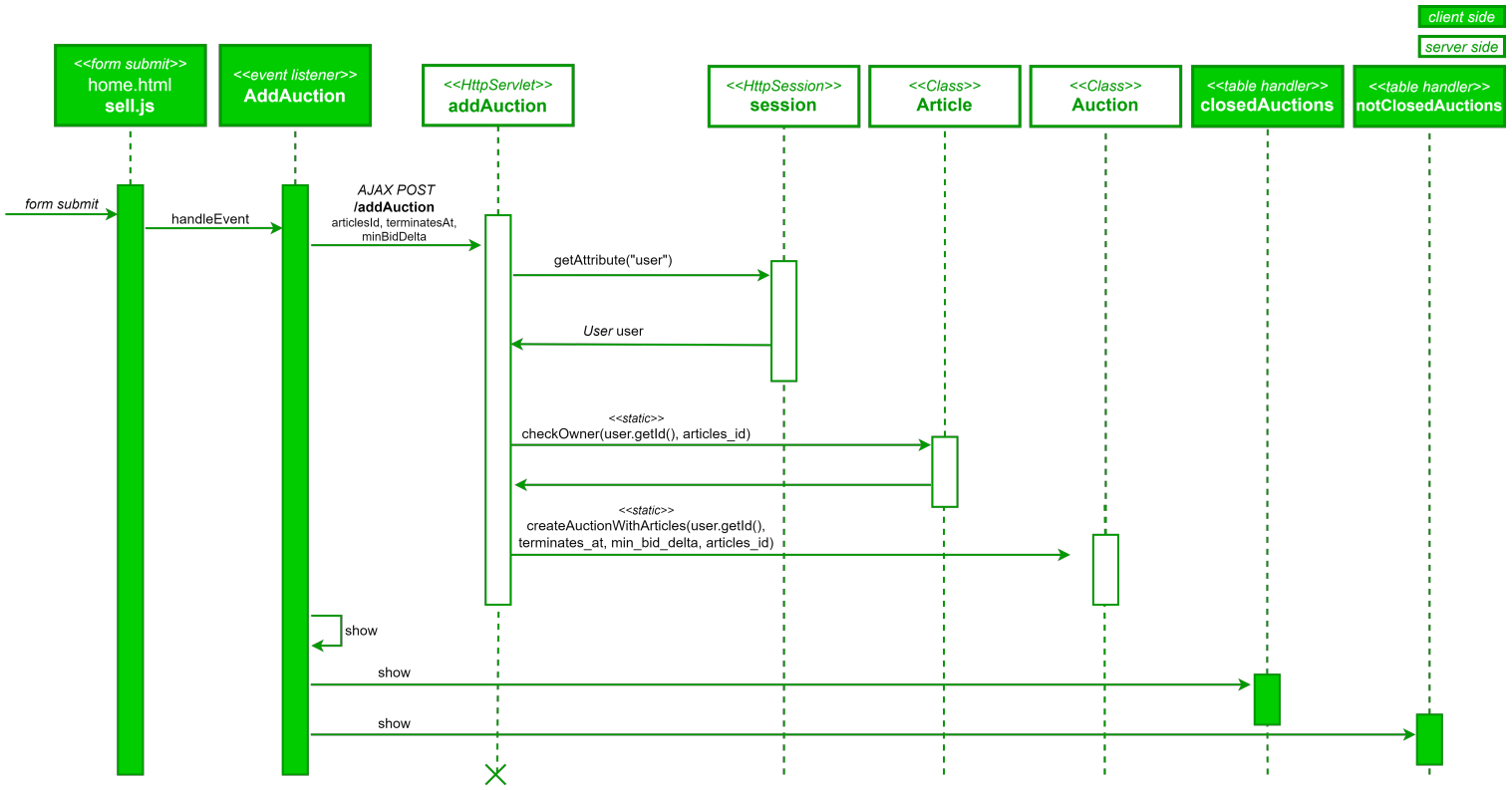
9.0.7 The user clicks to view the details of one of their auctions



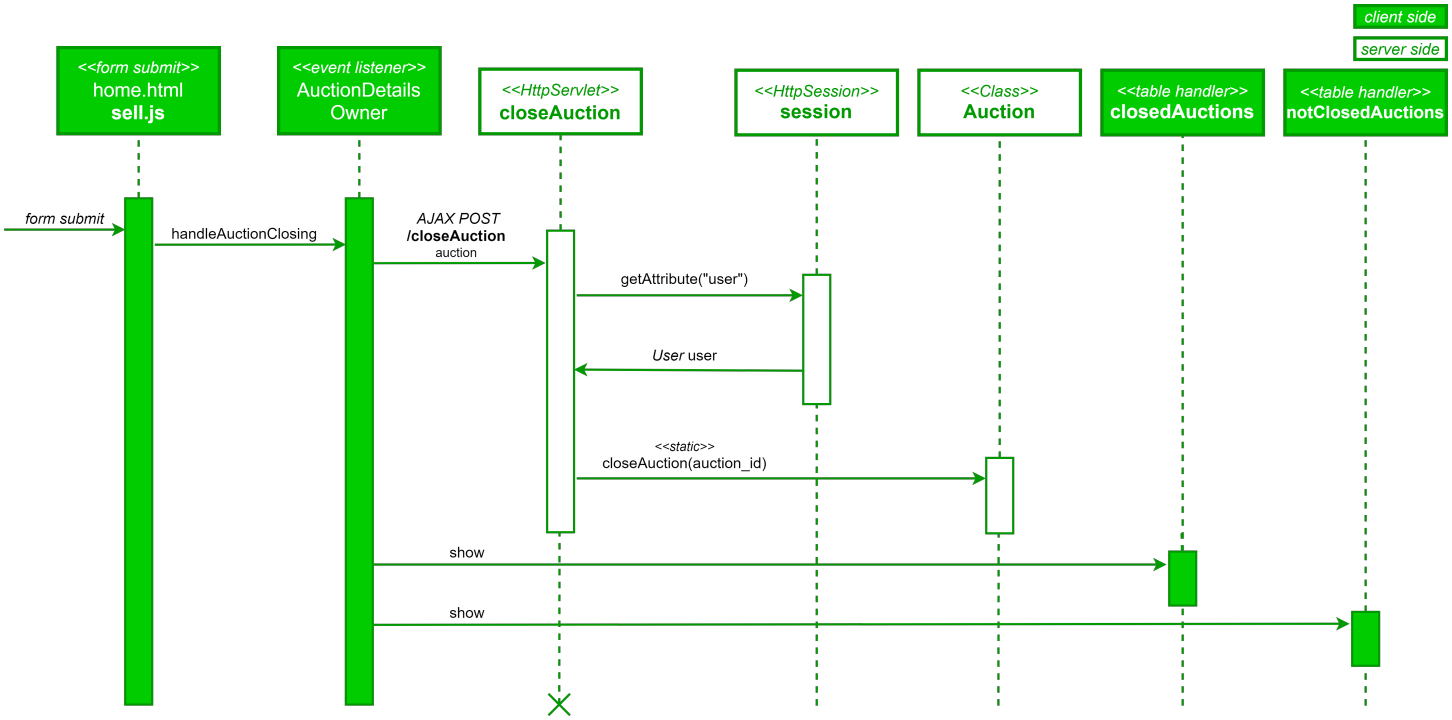
9.0.8 The user uploads a new article



9.0.9 The user creates a new auction



9.0.10 The user closes one of their completed auctions



10 Deployment in a Cluster with Docker and Kubernetes

For demonstration purposes, this project has been deployed in a Kubernetes (k8s) cluster to be publicly accessible on the web. This section outlines some of the key steps to achieve this goal, as well as some additional configurations required to support this type of deployment compared to a demo version.

10.1 Interactions with the Database

Interacting with a production database, as opposed to a local testing database, requires that:

- Connections do not remain open beyond a certain timeout, managed by the DBMS;
- Since opening connections is resource-intensive, connections should be requested as infrequently as possible and shared among servlets;
- It should be verified that the active connections have not been closed unexpectedly;
- The application must be robust against unexpected *SQLException*s thrown due to network errors;
- Database credentials should be stored as securely as possible, certainly not alongside the application code.

To meet the first three requirements, the connection management logic was delegated to a ConnectionHandler class, as detailed in the corresponding section. Regarding credentials, they should be read through environment variables, enabling integration with Kubernetes Secrets and standard access mechanisms for protected resources within the cluster.

10.2 Caching of Content

To minimize network traffic, Tomcat has been configured so that static content is cached locally in the browser.

For the pure HTML version, only the CSS stylesheet can be cached since all HTML pages contain dynamically generated content. In the JavaScript version, this can be optimized much further: all the JavaScript code, HTML markup, and CSS can be requested only on the first connection and cached by the browser. Additionally, the use of session storage allows auction information to be requested only when strictly necessary.

To properly instruct the browser on using the cache, some HTTP headers need to be set. This is handled by an *ExpiresFilter* provided by Apache Catalina. The configuration for this filter, which is inserted in the web.xml file, is as follows:

```
   <filter>
<filter-name>ExpiresFilter</filter-name>
<filter-class>org.apache.catalina.filters.ExpiresFilter</filter-class>
<init-param>
    <param-name>ExpiresDefault</param-name>
    <param-value>access plus 30 minutes</param-value>
</init-param>
<init-param>
    <param-name>ExpiresByType application/json</param-name>
    <param-value>access plus 0 seconds</param-value>
</init-param>
   </filter>
   <filter-mapping>
<filter-name>ExpiresFilter</filter-name>
```

```
<url-pattern>*</url-pattern>
<dispatcher>REQUEST</dispatcher>
  </filter-mapping>
```

10.3 Protection Against Brute-Force Attempts

There are several configurations to protect the application from malicious access attempts. To integrate with the existing architecture in the used cluster, logs are utilized as an interface: each access attempt is logged in the Tomcat logs so that this information can be analyzed by an intrusion prevention system such as Crowdsec. Within a Kubernetes cluster, logs must be written to stdout so that they can be consulted and made available to other pods. Therefore, it is necessary to modify Tomcat's configuration to add stdout output to the default logging pipeline (which writes to file).

10.4 Protection of Passwords

It is always good practice not to store user credentials in plaintext, so the *SHA*1 hashing algorithm is used to store passwords in the database. This is slightly more robust than *MD*5 (and other even weaker algorithms) while still having an acceptable computational cost. Using more complex algorithms such as *SHA*256 or *SHA*512 would make the application more vulnerable to denial-of-service attacks on machines with lower computational power, and is deemed excessive for demonstration purposes.

10.5 Session Distribution Across Multiple Nodes

Within a k8s cluster, HTTP requests are managed by load-balancing mechanisms (partially provided by an external load balancer, partially by Kubernetes Services) and reach a different Tomcat process each time. To ensure the user is properly recognized, the session content used by the servlets must be replicated across each instance. Tomcat provides clustering functionality to support this need, allowing configuration of how each Tomcat process can reach others on different nodes and how replication should occur. For this demonstration, each time a modification is made to the session, the delta is sent to all nodes.

By using the k8s StatefulSet resource to manage the pods, it is possible to address other containers in the set using Kubernetes' internal DNS resolution. The DNS name format is as follows: *nomePod*–*N.nomeSet.namespace.svc.cluster.local*. This is part of the necessary configuration, specifying how to reach Pod 1:

```
# Addressing other pods using internal DNS resolution
<Member className="org.apache.catalina.tribes.membership.StaticMember"
port="4000"
host="tiw-project-0.tiw-project.default.svc.cluster.local" # Pod 1
uniqueId="{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}"/> # Random
```

Alternatively, it would have been possible to store session content in the DB, associating a random string with each client, stored in the browser's cookies, and manually implementing the mechanism to reconstruct the session from the cookie.

10.6 Building the Docker Image and Deployment

For the application to be deployed, it is necessary to "build" a Docker image that contains Tomcat, the Java code, and the resources (configurations are inserted into the container at runtime). To do

this, we use the base Tomcat image from DockerHub and add the .war file compiled by the IDE to this image. The following Dockerfile contains all the necessary instructions:

```
FROM tomcat:10.1.8
LABEL maintainer="dudoleitor@dudoleitor.com"
ADD tiw.war /usr/local/tomcat/webapps/ROOT.war
RUN chown 65210:65210 -R /usr/local/tomcat/
RUN mkdir /data
RUN chown 65210:65210 /data
EXPOSE 8080
USER 65210
CMD ["/usr/local/tomcat/bin/catalina.sh", "run"]
```

This image must then be uploaded to a Docker registry so that it can be downloaded by the cluster nodes. The operations for building the image, uploading it, and restarting the resources in the cluster are automated using Apache Maven.

To complete the deployment, the following steps are taken:

- Define the StatefulSet resource, which specifies the details of the containers (environment variables, mounted folders, hardware resource limits, privileges, etc.);
- Add a Secret with database access credentials;
- Add a ConfigMap for Tomcat configurations;
- Prepare a persistent volume to store files (e.g., item images);
- Define a Service to expose the pods to the reverse proxy of the cluster and ensure load balancing;
- Define the necessary Ingress resources so that the reverse proxy manages requests for the appropriate domain names;
- Add DNS records at the domain provider.