Aste Online

0. Informazioni generali

Realizzato da:	Paolo Brusa (codice persona: 10835369, matricola: 211863)
Data colloquio:	5/08/2025 Traccia 1
Software usati:	
	HeidiSQL
	Intellij IDEA
	Apache Tomcat
	Firefox
Librerie esterne:	
	Gson 2.10.1
	JSTL Core and Fmt

Documentazione versione pure HTML

1. Progettazione database e schema ER

Dopo un'attenta lettura e valutazione delle informazioni utile per definire e progettare una base di dati ho trovato 5 entità e 6 relazioni:
(**grassetto:** PRIMARY KEY; FK: FOREIGN KEY)
Entità:

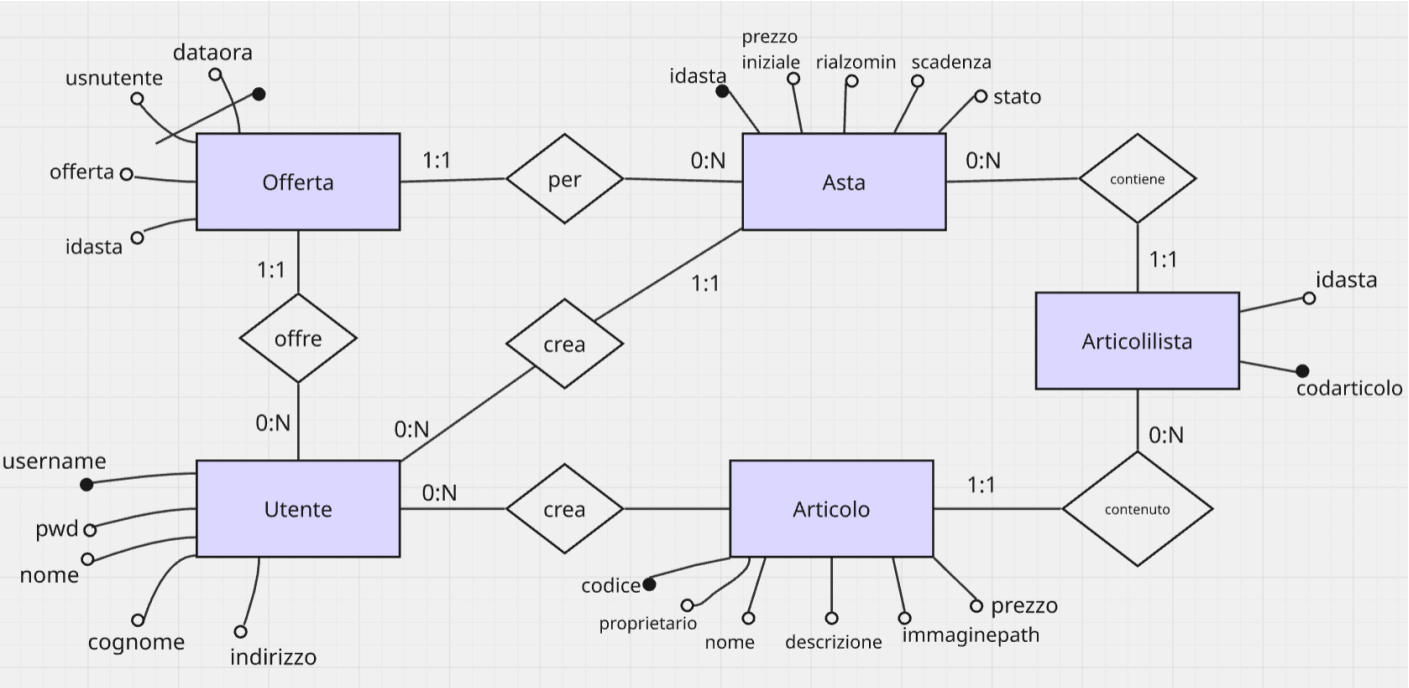
- Utente: **username**, pwd, nome, cognome, indirizzo
- Articolo: **codice**, proprietario(FK), nome, descrizione, immagine path, prezzo
- Asta: **idasta**, prezzoiniziale, rialzomin, scadenza, stato
- Offerta: **usnutente(FK)**, **dataora**, offerta, idasta(FK)

Serviva però qualcosa per relazionare gli articoli alle aste. Ci possono essere diverse opzioni tra cui salvare gli id dell’asta nell'articolo stesso oppure creare un nuovo record. Io ho optato per creare una tabella di supporto lasciando così le responsabilità separate nel database, aumentando la scalabilità e la manutenibilità. Così bisogna fare un join in più per determinate

query ma non lo considero un problema considerando che con le altre opzioni risultavano record molto più pesanti a livello di memoria.

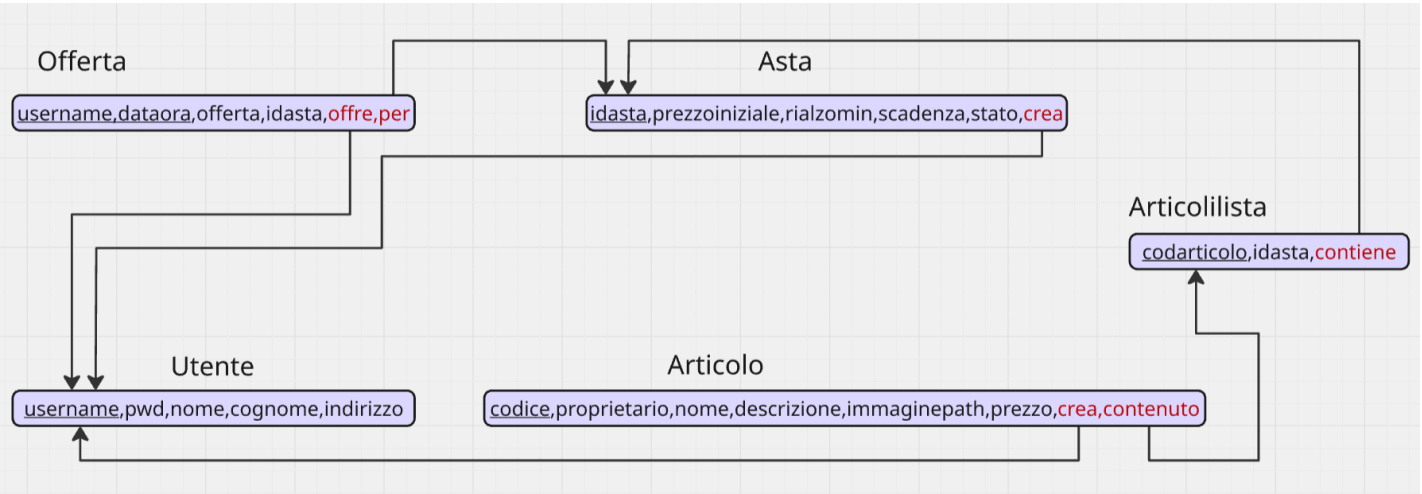
- Articolilista: **codarticolo(FK)**, idasta(FK)

Per fare un focus sulla scalabilità banalmente questo approccio con poche modifiche permette in futuro di scegliere che un articolo possa appartenere a più aste.



1.1 Da schema ER a concettuale

Il passaggio da ER a concettuale in questo progetto avviene in maniera molto intuitiva da che come abbiamo visto prima non ci sono relazioni molti a molti:



Ho aggiunto in rosso le relazioni presenti nello schema ER.
Dettagli generali utili:

Utente:

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default
1	**username**	**VARCHAR**	**50**	☐	☐	☐	**No default**
2	pwd	VARCHAR	50	☐	☐	☐	No default
3	nome	VARCHAR	50	☐	☐	☐	No default
4	cognome	VARCHAR	50	☐	☐	☐	No default
5	indirizzo	VARCHAR	100	☐	☐	☐	No default

Offerta:

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default
1	**usnutente**	**VARCHAR**	**50**	☐	☐	☐	**No default**
2	offertaprezzo	INT	11	☐	☐	☐	No default
3	idasta	INT	11	☐	☐	☐	No default
4	**dataora**	**DATETIME**		☐	☐	☐	**No default**

Asta:

#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default
1	**id**	**INT**	**11**	☐	☐	☐	**AUTO_INCREME...**
2	prezzoiniziale	INT	15	☐	☐	☐	No default
3	rialzomin	INT	11	☐	☐	☐	No default
4	scadenza	DATETIME		☐	☐	☐	No default
5	stato	ENUM	'attiva','chiusa'	☐	☐	☐	'attiva'

Articolilista:

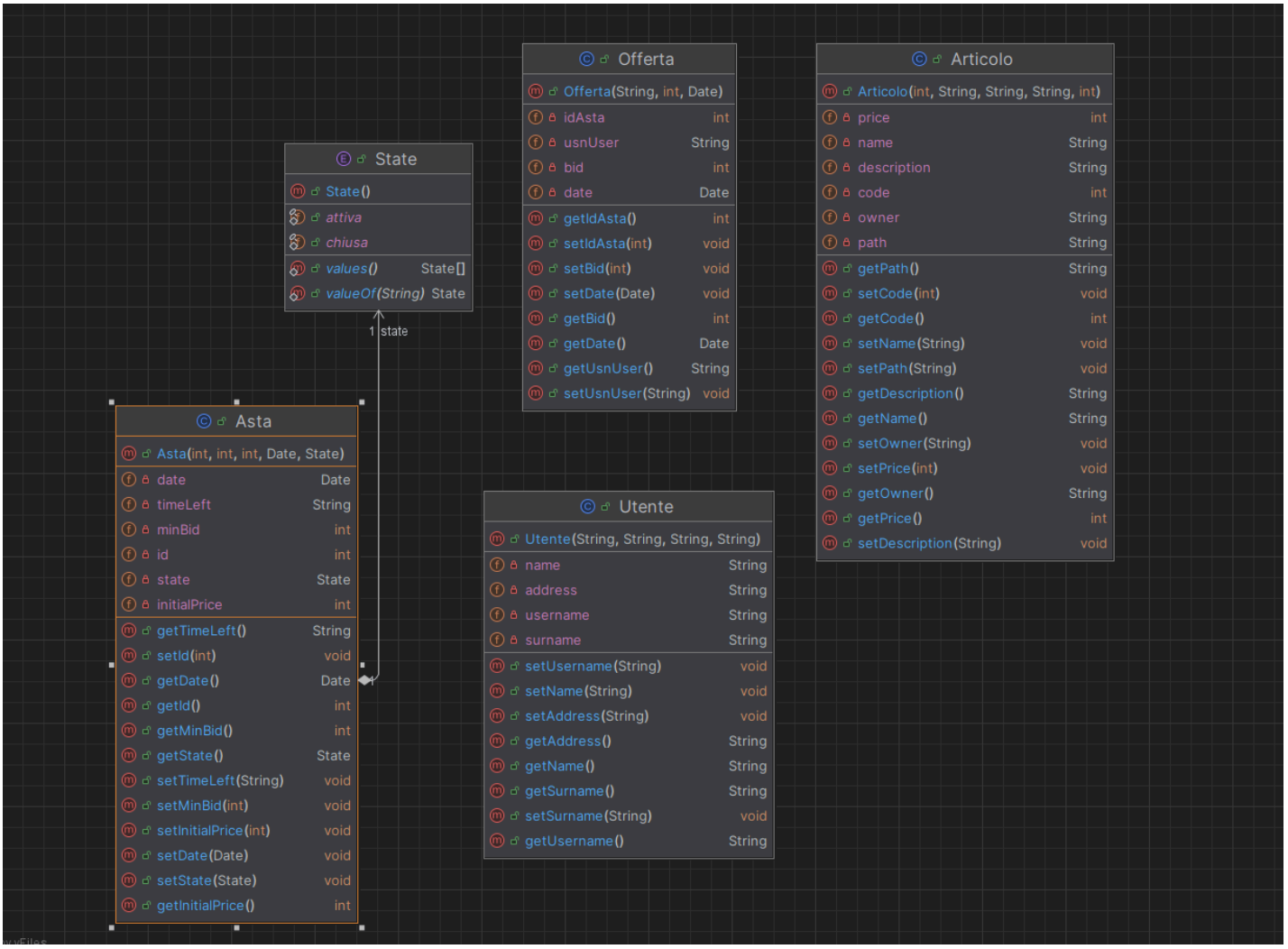
#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default
1	idasta	INT	11	☐	☐	☐	No default
2	**codarticolo**	**INT**	**11**	☐	☐	☐	**No default**

Articolo:

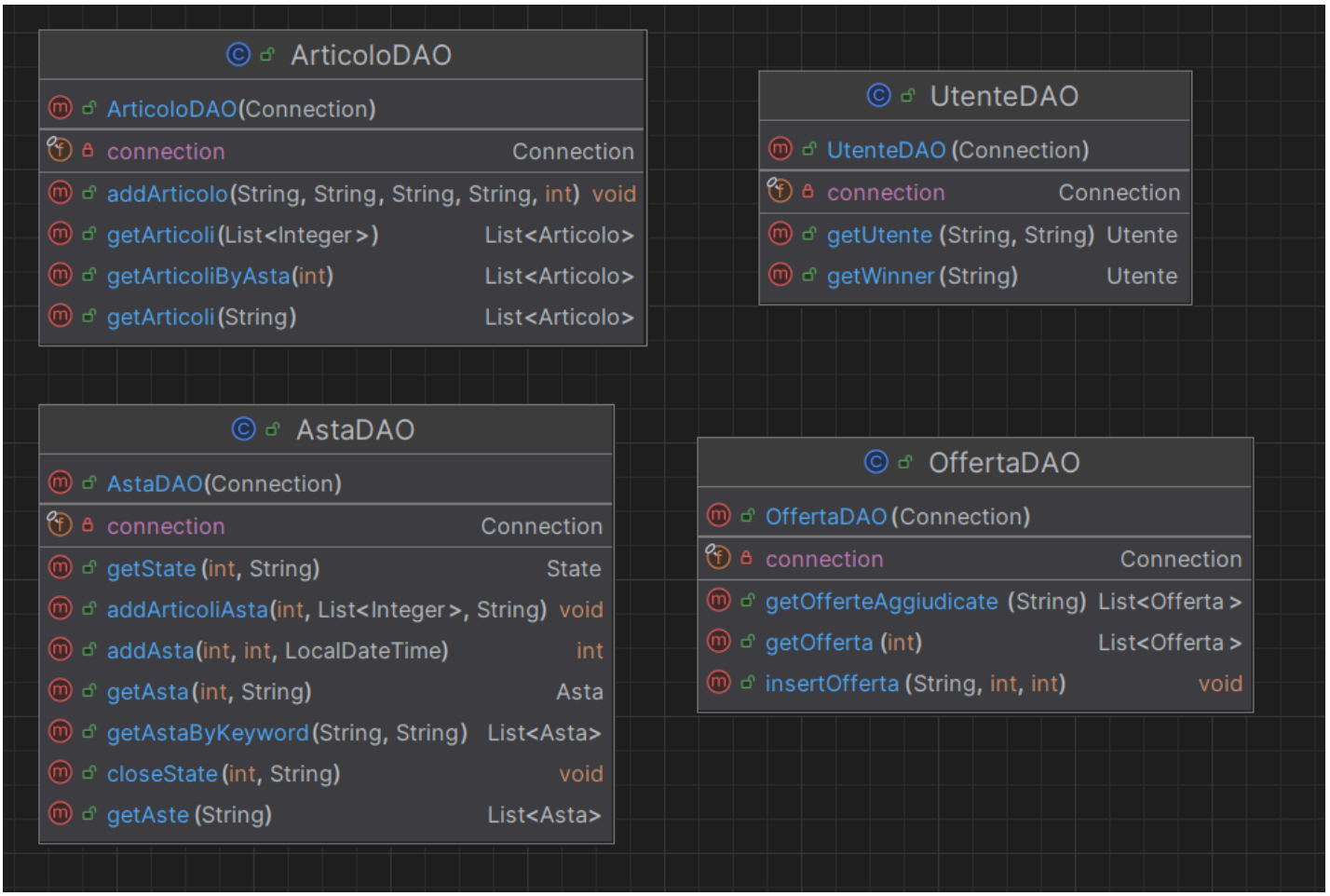
#	Name	Datatype	Length/Set	Unsigned	Allow NULL	Zerofill	Default
1	**codice**	**INT**	**11**	☐	☐	☐	**AUTO_INCREME...**
2	proprietario	VARCHAR	50	☐	☐	☐	No default
3	nome	VARCHAR	50	☐	☐	☐	No default
4	descrizione	VARCHAR	255	☐	☐	☐	No default
5	immaginepath	VARCHAR	255	☐	☐	☐	No default
6	prezzo	INT	11	☐	☐	☐	No default

1.2 Beans e DAO

Come abbiamo visto nel data base, ci sono dei vincoli da rispettare come ad esempio la lunghezza e la non nullità di certi dati i quali vengono verificati nelle servlet prima di essere passati con dovute precauzioni ai DAO che si occupano dei rispettivi Beans.
I Beans e i DAO servono per "nascondere" il database alla connessione separando inoltre cosi la logica applicativa da quella di accesso e storage. Un bean è una classe che rappresenta in modo esaustivo i dati di un oggetto presente nel database con alcuni accorgimenti. Per esempio nella classe Utente che serve per memorizzare dati dalla tabella omonima non ho messo nessun attributo inerente la password e non si può neanche usare un set o un get perché essa è giusto che sia privata nel db e deve rimanere tale. I DAO invece servono per interrogare il database, utilizzandoli deleghiamo così dalla servlet la logica di accesso nascondendo alla rete il database.



Nell'immagine soprastante viene visualizzato il diagramma UML dei Beans e vediamo appunto che hanno i propri attributi e rispettano quelli del database. In più sono presenti solo metodi di getter e setter i quali alcuni non servono neanche ma ho deciso di lasciarli per scalabilità in futuro.
Da notare che ci sono 3 peculiarità, la prima quella più evidente è che ho aggiunto una classe enum (State) di supporto. Essa serve per gestire lo stato della asta (aperta,chiusa) in modo tale da rendere il tutto più esplicito e avere la possibilità di fare determinati controlli nelle servlet ma soprattutto per memorizzare in modo consono lo stato dell'asta. La seconda aggiunta non presente nel db è che in asta c'è anche un attributo timeleft, esso serve per memorizzare il tempo rimanente dalla data di scadenza. Per calcolarlo mi servo di una classe di supporto definita in filterAndUtils che permette di effettuare il calcolo con il date.now() del server. Ho utilizzato il formato String permettendo così di gestirlo come json senza ulteriori conversioni e soprattutto inviando tutto con un'unica variabile. L'ultima cosa da evidenziare è l'attributo path in articolo. Esso non memorizza proprio il path dell'immagine ma solo il nome con il corrispettivo formato come è presente nel database, per accedere a il vero e proprio file lo combina con la dir corrispettiva nel fileSystem e lo preleva.



L’immagine soprastante invece rappresenta l’UML dei DAO. Come possiamo notare in ognuno di essi è presente l’attributo connection che serve per creare una connessione con il db nel momento della creazione nelle Servlet. Per preservare l'atomicità nel DAO vengono chiamate singole operazioni al db rendendo così evidentemente che non c’è possibilità di fallire operazioni oltre a quella effettuata gestita con un adeguato try catch.

```
public Utente getUtente(String username, String pwd) throws SQLException {  1 usage  paolobrusa
    Utente u = null;
    String query = "SELECT username, nome, cognome, indirizzo FROM Utente WHERE username = ? AND pwd = ?";
    ResultSet rs = null;
    PreparedStatement ps = null;
    try {
        ps = connection.prepareStatement(query);
        ps.setString( parameterIndex: 1, username);
        ps.setString( parameterIndex: 2, pwd);
        rs = ps.executeQuery();
        if (rs.next())
            u = new Utente(rs.getString( columnLabel: "username"), rs.getString( columnLabel: "nome"),rs.getString
```

Per prevenire invece un attacco SQL injection ho utilizzato il pattern con il prepared statement.
Esso infatti prevede, come si vede in figura, una sostituzione del parametro che serve alla query con un ‘?’ facendo così poi possiamo inserire i dati convertiti correttamenti con i metodi che offre PreparedStatement scongiurando così le injection di qualsiasi forma (da un accesso non autorizzato, alla eliminazione di tabella).

1.3 Controllers e Views

I controllers ovvero le Servlet che servono per effettuare le operazioni dell'utente sono il core dell'applicazione. Per crearle ho usato due pattern specifici: il Single Responsibility Principle (SRP) e il Post-Redirect-Get (PRG). Il primo definisce in modo chiaro la funzionalità della servlet e appunto permette di mantenere una singola responsabilità, ci sono due casi in cui penso di non averla rispettata ovvero in Offerte e in DettaglioAsta dove ho messo sia doGet che doPost (ovviamente il primo si occupa di lettura e il secondo di scrittura e soprattutto inerente allo stesso oggetto). Mentre il pattern PRG serve per garantire il corretto funzionamento dell'app web evitando situazioni spiacevoli all'utente che la utilizza (ad esempio rinvio di un form per colpa di un refresh della pagina da parte dell utente).
Ogni Servlet controlla una view ovvero un jsp. La servlet si occupa di inviare le informazioni e il jsp le organizza per visualizzarle in modo corretto.

Dalla specifica ho localizzato 6 view (ovvero 6 jsp) con le rispettive servlet integrate da un filtro e delle utils per qualche doPost.

- Login → login.jsp
- Homepage → homepage.jsp
- Vendo → vendo.jsp
- DettaglioAsta → dettaglioAsta.jsp
- Acquisto → acquisto.jsp
- Offerte → offerta.jsp

Essendo che in generale html non permette di natura di inviare dati tra una pagina e l'altra ho adottato una strategia per inviare gli errori evitando di "sporcare" il Url.

```
if (id == null || offerta == null || offerta.length() > 11) {
    request.getSession().setAttribute( s: "errorMessage",  o: "Un parametro è null, non è accettato")
    response.sendRedirect( location: request.getContextPath() + "/Offerta");
    return;
}
```

Con questa prima immagine possiamo vedere come vado a gestire gli errori in un doPost dove sono costretto a usare una redirect (nei doGet uso setAttribute con forward senza la sessione). L'unico parametro che non viene perso durante una redirect è appunto la sessione permettendo di passare cosi parametri, l'importante però logicamente è toglierlo immediatamente dopo appena viene utilizzato. Anche l'user logicamente viene memorizzare in questo modo permettendo di creare una sessione valida.

```
String errorMessage = (String) request.getSession().getAttribute( s: "errorMessage");
if (errorMessage != null) {
    request.getSession().removeAttribute( s: "errorMessage");
    request.setAttribute( s: "errorMessage", errorMessage);
}
```

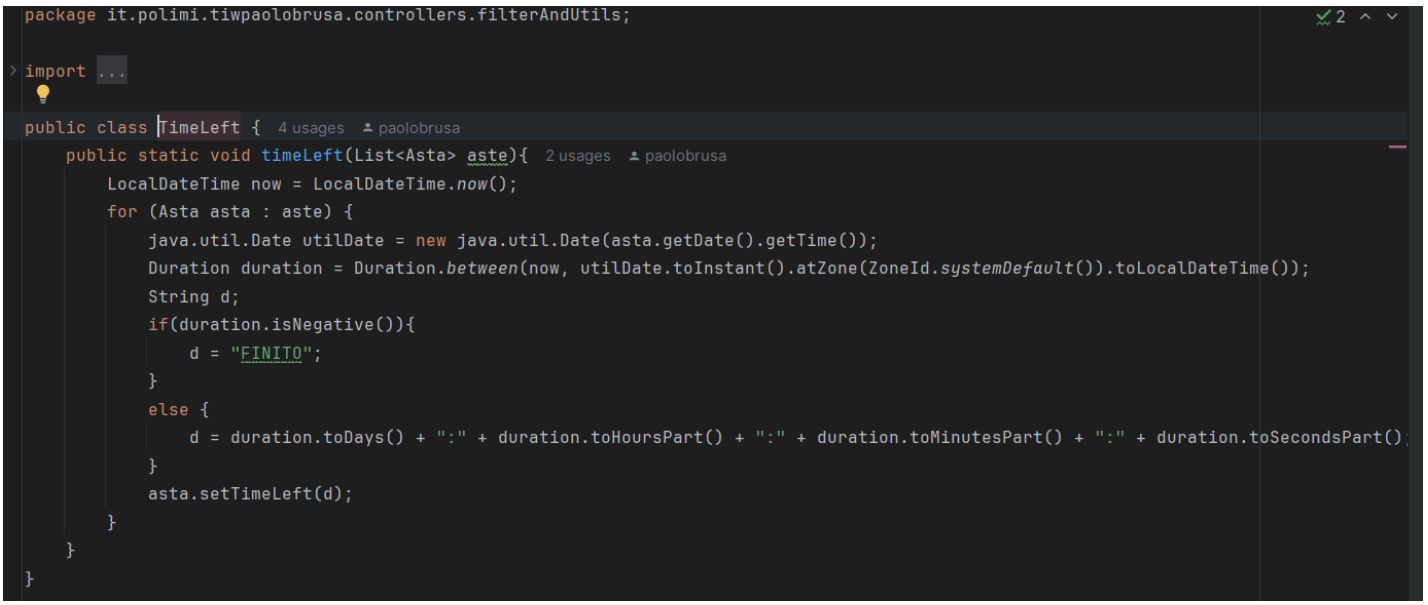
Questo è come viene gestita la visualizzazione e l'eliminazione dell'attributo.
Oltre a quello appena detto vorrei fare 3 accorgimenti su delle implementazioni utilizzate:

```java
package it.polimi.tiwpaolobrusa.controllers.filterAndUtils;

import ...

@WebFilter("/*")  paolobrusa
public class LoginFilter implements Filter {

    private static final String[] paths = {"/Homepage", "/Vendo", "/Logout", "/Dettaglio", "/AddArticolo",  1 usage
            "/CreateAsta", "/Acquisto", "/Offerta", "/css/style.css"};

    @Override  no usages  paolobrusa
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, S
        HttpServletRequest request  = (HttpServletRequest) servletRequest;
        HttpServletResponse response = (HttpServletResponse) servletResponse;
        String path = request.getRequestURI().substring(request.getContextPath().length());
        if ((path.equals("/Login") || path.equals("/css/style.css")) && request.getSession().getAttribute( s: "user") == null) {
            filterChain.doFilter(request, response);
        }
        else if (request.getSession().getAttribute( s: "user") != null && request.getSession( b: false) != null && (Arrays.asList(paths).
            filterChain.doFilter(request, response);
        }
        else if (request.getSession().getAttribute( s: "user") != null && request.getSession( b: false) != null) {
            response.sendRedirect( location: request.getContextPath() + "/Homepage");
        }
        else {
            response.sendRedirect( location: request.getContextPath() + "/Login");
        }
    }

}
```

Per evitare che un utente possa accedere al sito senza effettuare l'autenticazione ho aggiunto una classe WebFilter che permette appunto di filtrare tutti gli Url in base al fatto se una persona ha la sessione valida settata con un user…. Essendo il server l'unico che può settare la sessione (JSESSIONID) ovviamente siamo sicuri che se presente l'attributo user è perché è stato autenticato con successo. Dopo 30 minuti senza utilizzo la sessione scade in automatico anche se la persona non ha effettuato il logout.

```xml
</context-param>
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
<welcome-file-list>
    <welcome-file>Login</welcome-file>
</welcome-file-list>
```

Un altro accorgimento è la classe utils TimeLeft, essa permette di calcolare il tempo rimanente di un asta e salvarla in formato String permettendo una condivisione facilitata.



```java
package it.polimi.tiwpaolobrusa.controllers.filterAndUtils;

import ...

public class TimeLeft {  4 usages  paolobrusa
    public static void timeLeft(List<Asta> aste){  2 usages  paolobrusa
        LocalDateTime now = LocalDateTime.now();
        for (Asta asta : aste) {
            java.util.Date utilDate = new java.util.Date(asta.getDate().getTime());
            Duration duration = Duration.between(now, utilDate.toInstant().atZone(ZoneId.systemDefault()).toLocalDateTime());
            String d;
            if(duration.isNegative()){
                d = "FINITO";
            }
            else {
                d = duration.toDays() + ":" + duration.toHoursPart() + ":" + duration.toMinutesPart() + ":" + duration.toSecondsPart();
            }
            asta.setTimeLeft(d);
        }
    }
}
```

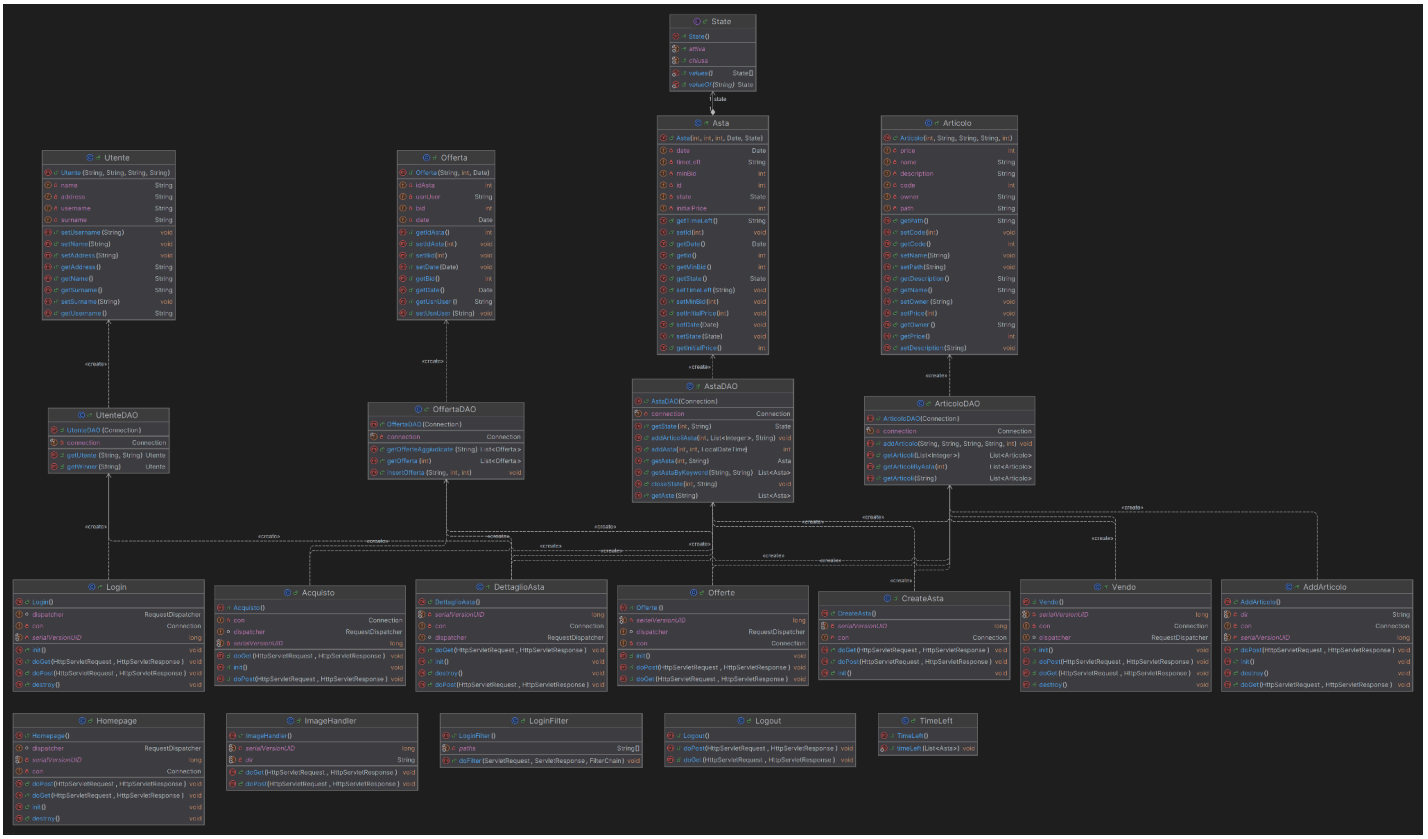
L'ultima implementazione che volevo mostrare era come ho gestito la scrittura delle immagini e la lettura. Ho utilizzato una cartella (o che crea se non esiste) sul File System permettendo di memorizzare le immagine fisiche cambiando il nome del file per renderlo univoco, il database memorizza solo il nome di quel file nell'attributo immaginepath. Il nome del file l'ho creato usando la classe UUID con il metodo .RandomUUID() che permette di generare codici alfa numerici del tipo xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx. Facendo così la probabilità che due nome file siano identici è considerabile nulla.

```
Part image = request.getPart( s: "immagine");
if(image == null || (!image.getContentType().equals("image/jpeg") && !image.getContentType().equals("image/png"))){
    request.getSession().setAttribute( s: "errorMessage",  o: "Immagine non valido");
    response.sendRedirect( location: request.getContextPath() + "/Vendo");
    return;
}
String path = UUID.randomUUID() + "." + image.getContentType().replace( target: "image/",  replacement: "");
File upload = new File(dir);
if (!upload.exists()) {
    boolean a = upload.mkdir();
    if (!a) {
        request.getSession().setAttribute( s: "errorMessage",  o: "Errore creazione cartella");
        response.sendRedirect( location: request.getContextPath() + "/Vendo");
        return;
    }
}
String filePath = dir + File.separator + path;
image.write(filePath);
ArticoloDAO aDAO = new ArticoloDAO(con);
try {
    aDAO.addArticolo(n, d, o, path, prezzo);
} catch (SQLException e) {
    request.getSession().setAttribute( s: "errorMessage", e.getCause().getMessage());
    response.sendRedirect( location: request.getContextPath() + "/Vendo");
    return;
}
```

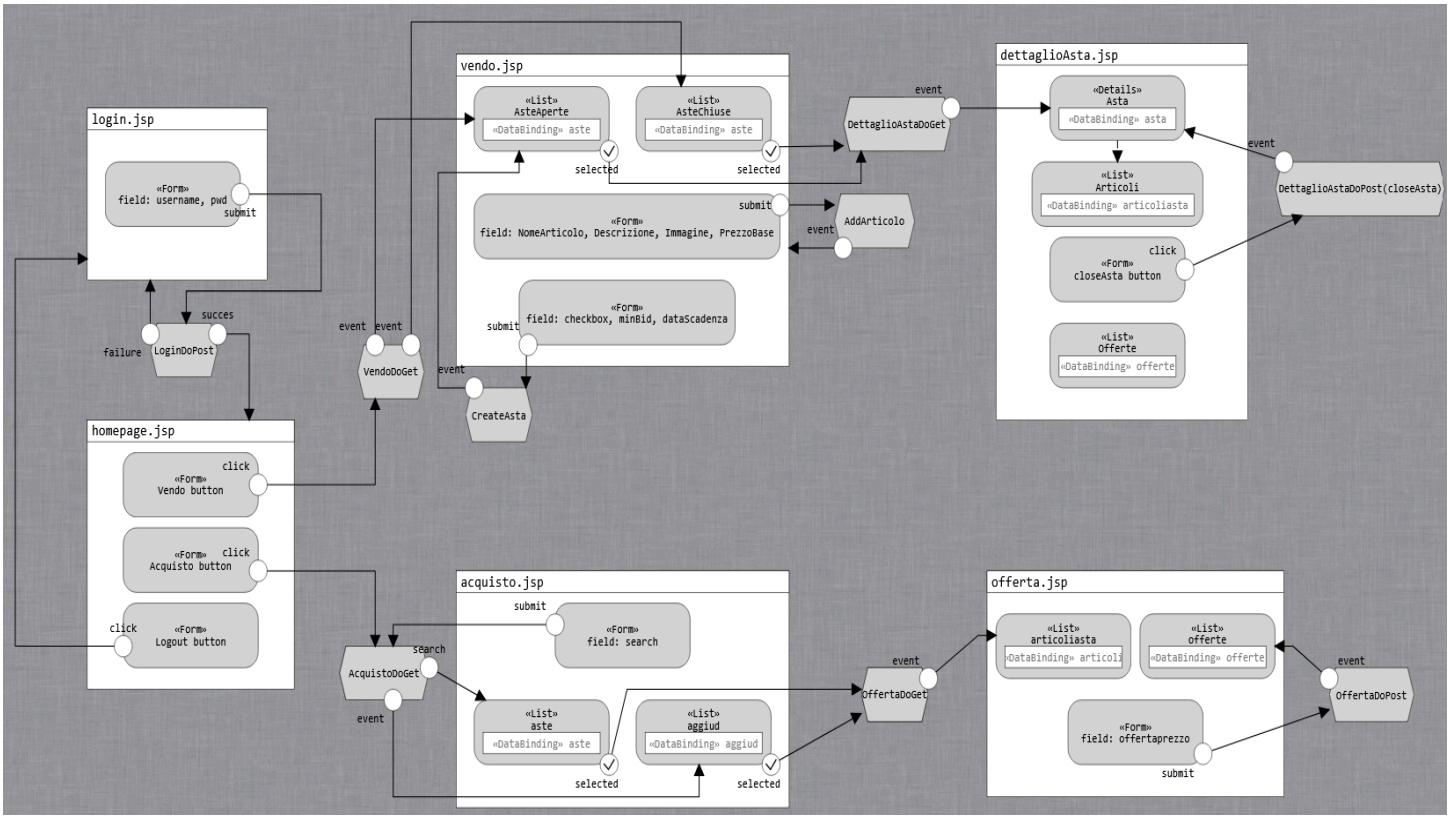
Per prelevare le immagini salvate e farle visualizzare ho optato per creare una servlet dedicata alla visualizzazione delle immagini (SRP). Ogni volta che si genera una tabella in un jsp se serve anche l'immagine semplicemente fa una richiesta intrinseca con l’url per ogni immagine, se dovesse dare errore (ovvero non riceve l’immagine) è il jsp che se ne occupa e restituisce al posto dell l’immagine una stringa “errore”.

```java
@WebServlet("/Image/*")  paolobrusa *
public class ImageHandler extends HttpServlet {
    @Serial  no usages
    private static final long serialVersionUID = 1L;
    private static final String dir = System.getProperty("user.home") + File.separator + "TIWImage";  1 usage

    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  paolobrusa *
        String path = req.getPathInfo();
        if(path == null || path.isEmpty()) {
            return;
        }
        String contentType = getServletContext().getMimeType(path.substring( beginIndex: 1));
        File image = new File(dir, path.substring( beginIndex: 1));
        if(!image.exists()){
            return;
        }
        if (contentType == null) {
            return;
        }
        Image image1 = ImageIO.read(image);
        int height = 75;
        int width = 75;
        image1 = image1.getScaledInstance(width, height, Image.SCALE_SMOOTH);
        BufferedImage imageb = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
        imageb.getGraphics().drawImage(image1,  x: 0,  y: 0,  observer: null);
        resp.setContentType(contentType);
        ImageIO.write(imageb, path.substring( beginIndex: path.lastIndexOf( ch: '.')+1), resp.getOutputStream());
    }
```

Per completezza, ho deciso di inserire anche un uml completo del server



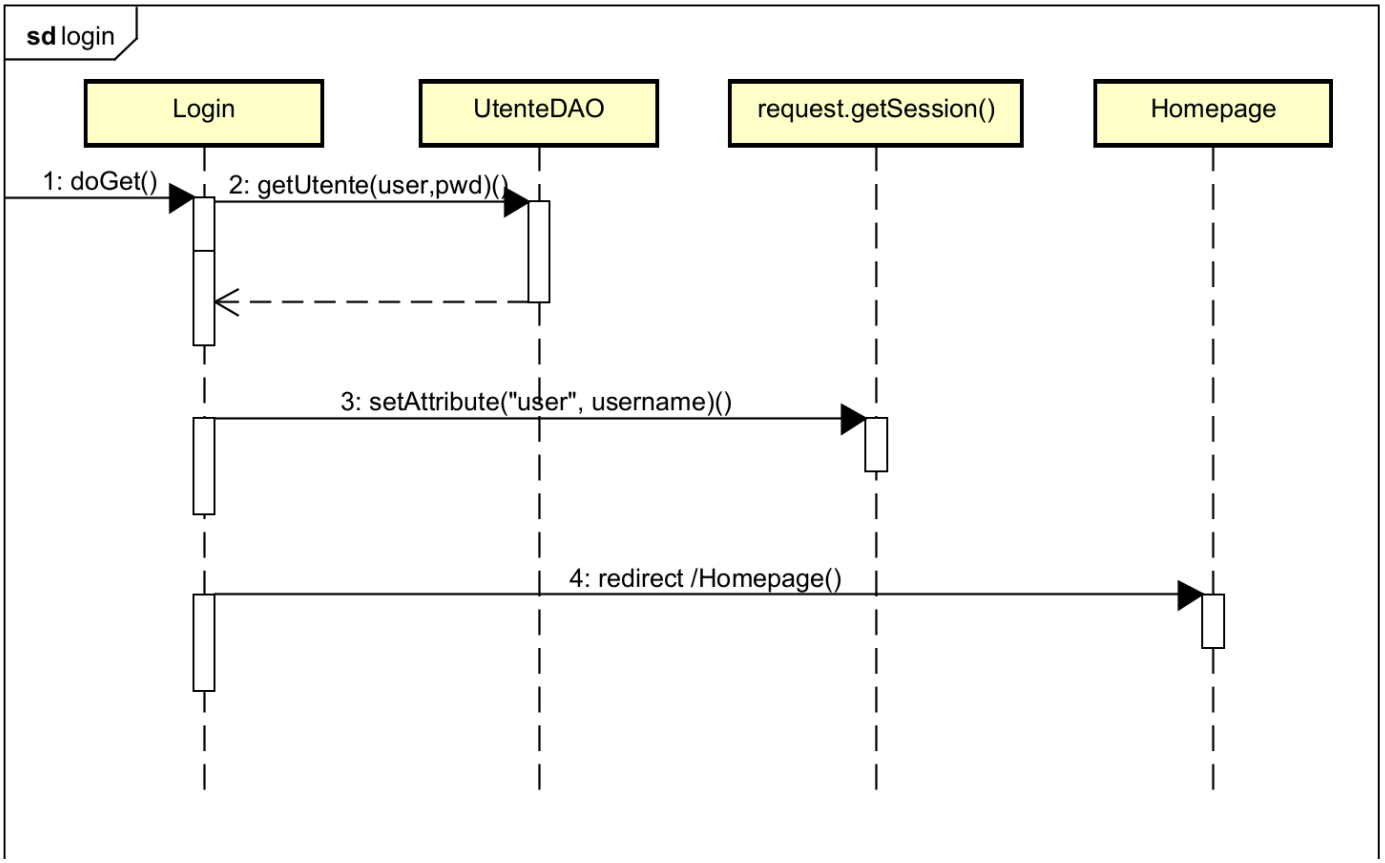
1.4 IFML Diagram



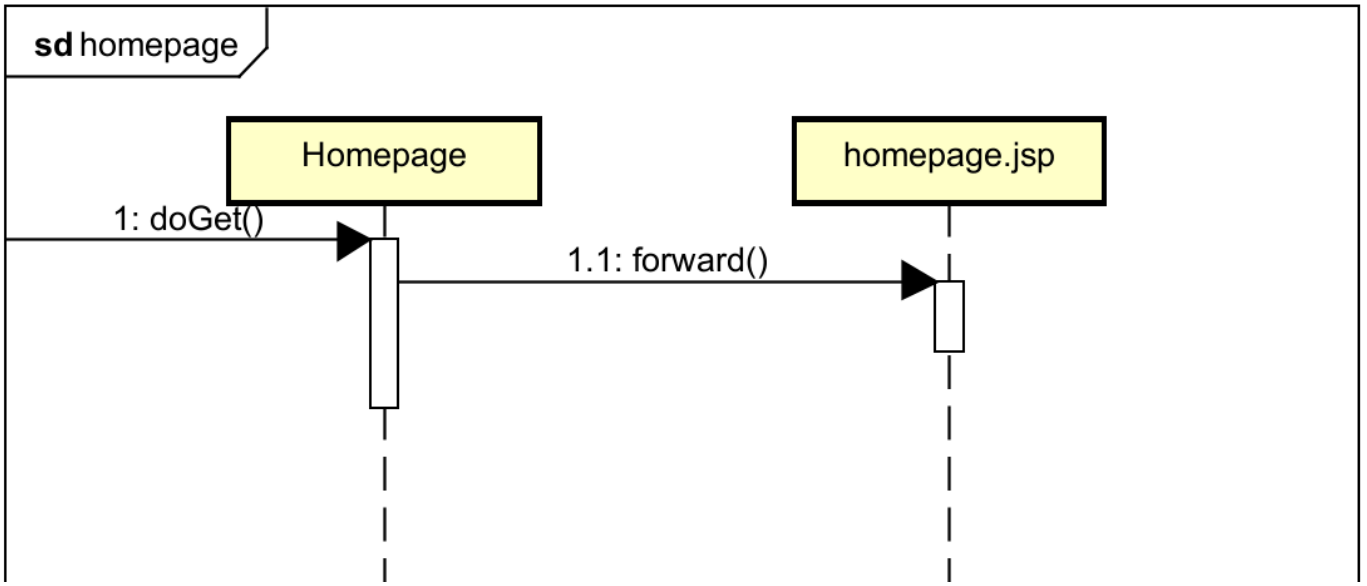
Ho trascurato di esplicitare i bottoni per tornare alla homepage nelle varie pagine.

1.5 Sequence Diagram

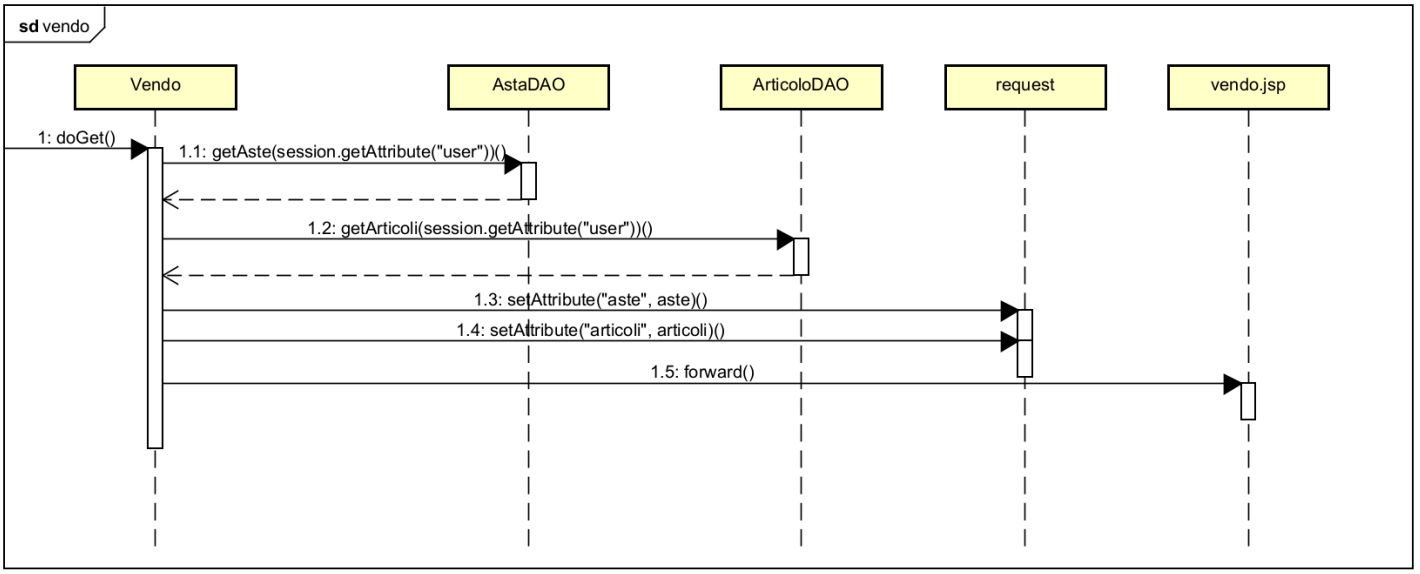
Il primo evento scatenato da un utente è l’accesso alla propria homepage attraverso la schermata di login (login.jsp).



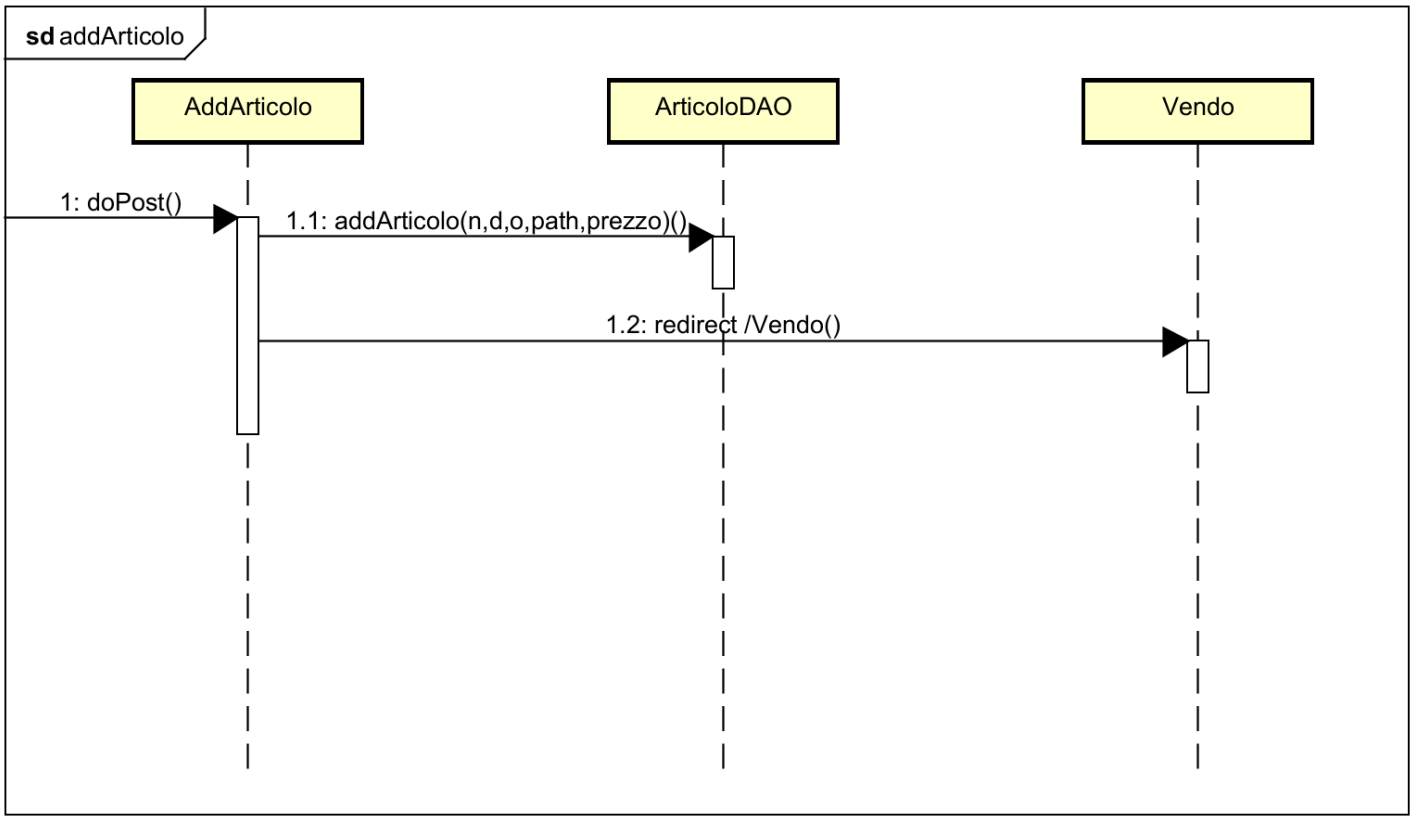
Una volta trovati nella Homepage la servlet invia i dati alla pagina (questa procedura la farò vedere solo per la homepage dato che è sempre uguale), tutti i bottoni per tornare alla homepage funzionano così.

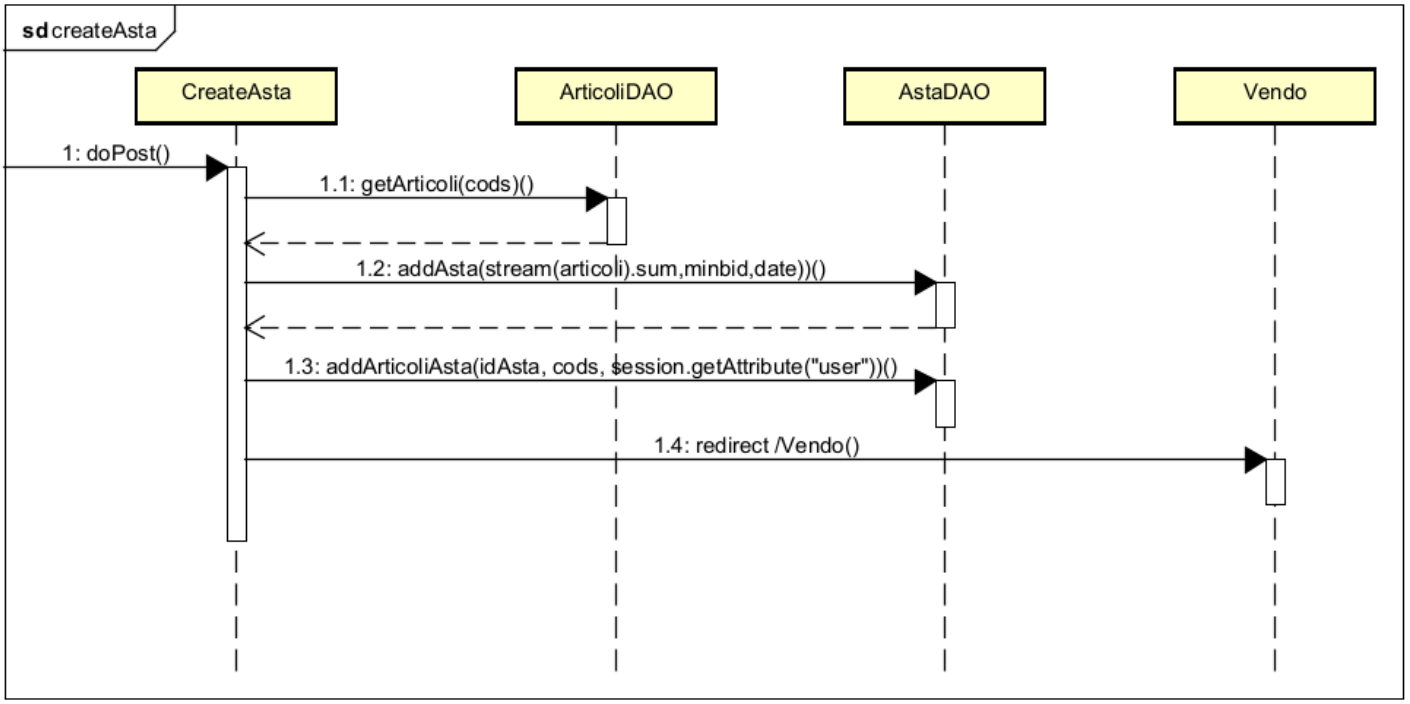


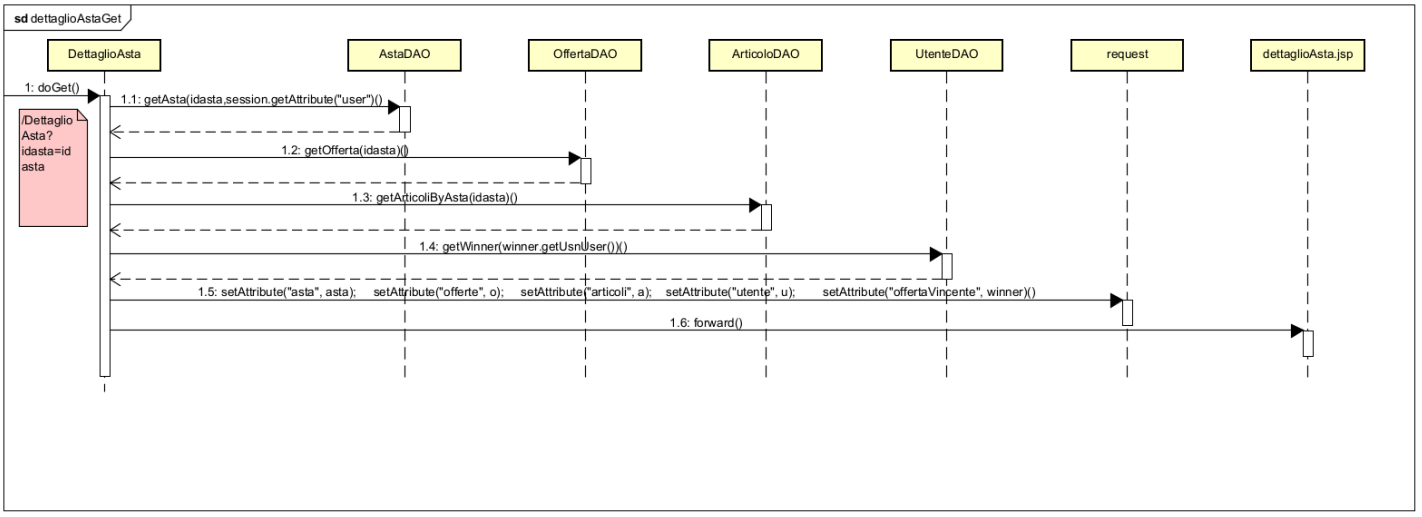
Dalla Homepage possiamo selezionare o Vendo o Acquisto:



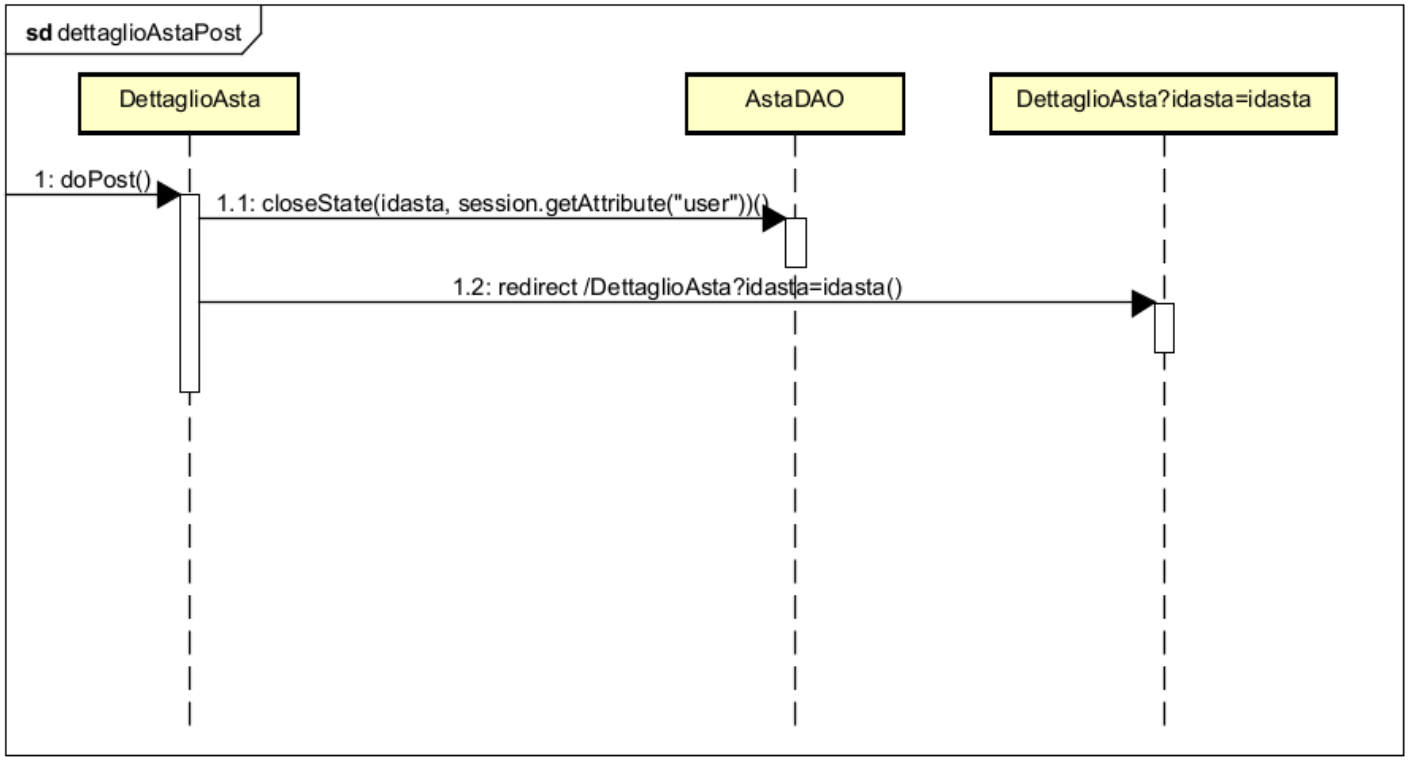
Dopo aver selezionato vendo come nell'immagine sopra, possiamo creare nuovi articoli oppure nuove aste e accedere se no a un asta che abbiamo già aperto o chiusa.



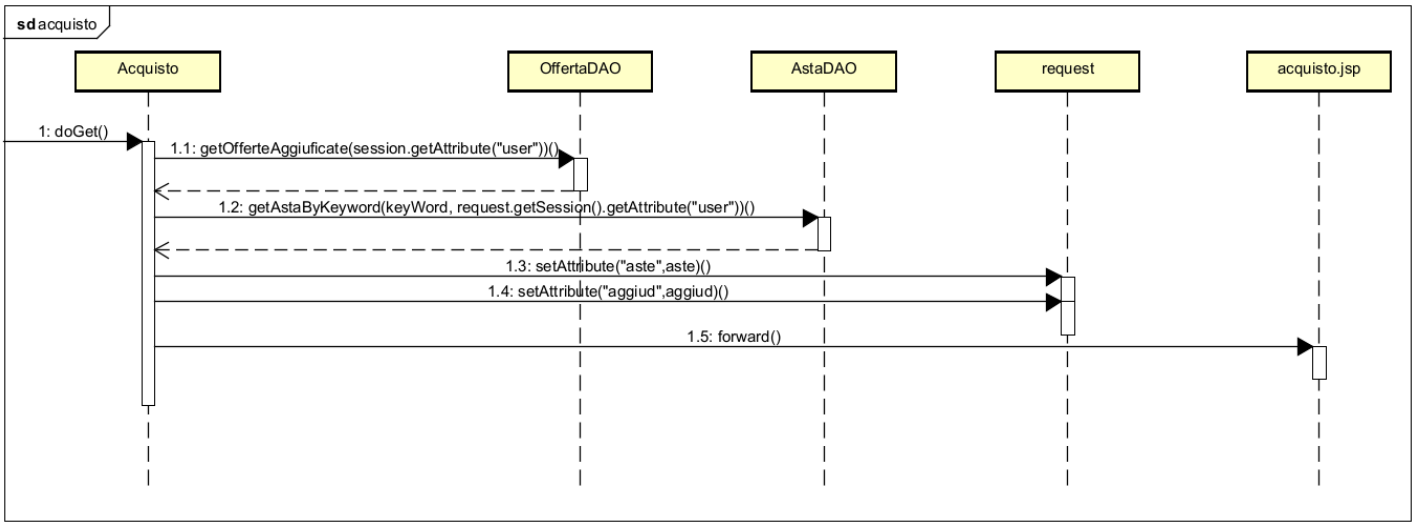




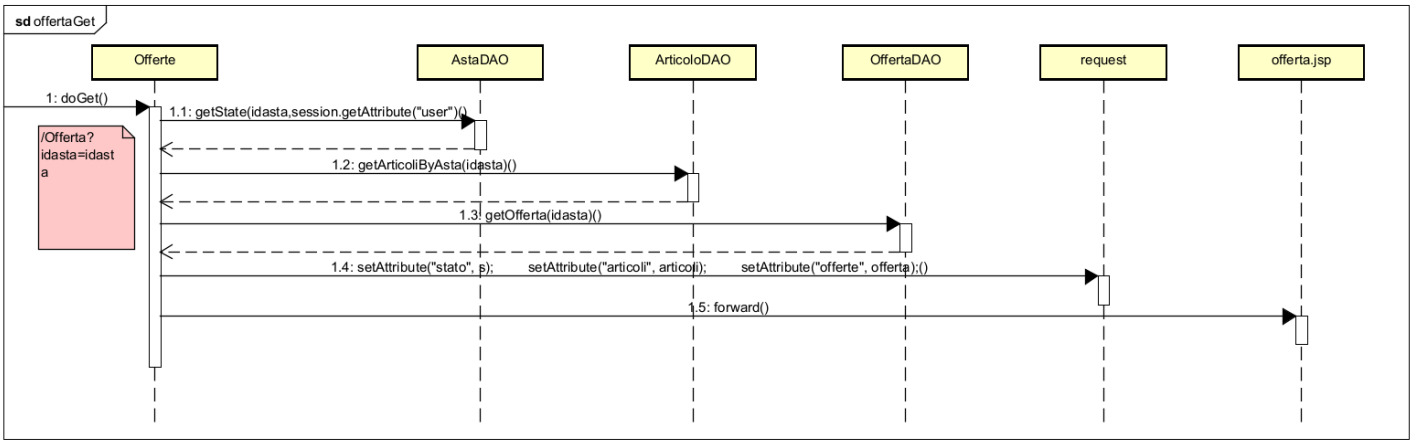
Da Dettaglio Asta si può usare il metodo post clickando su chiudi per chiudere un'asta che sia già scaduta.



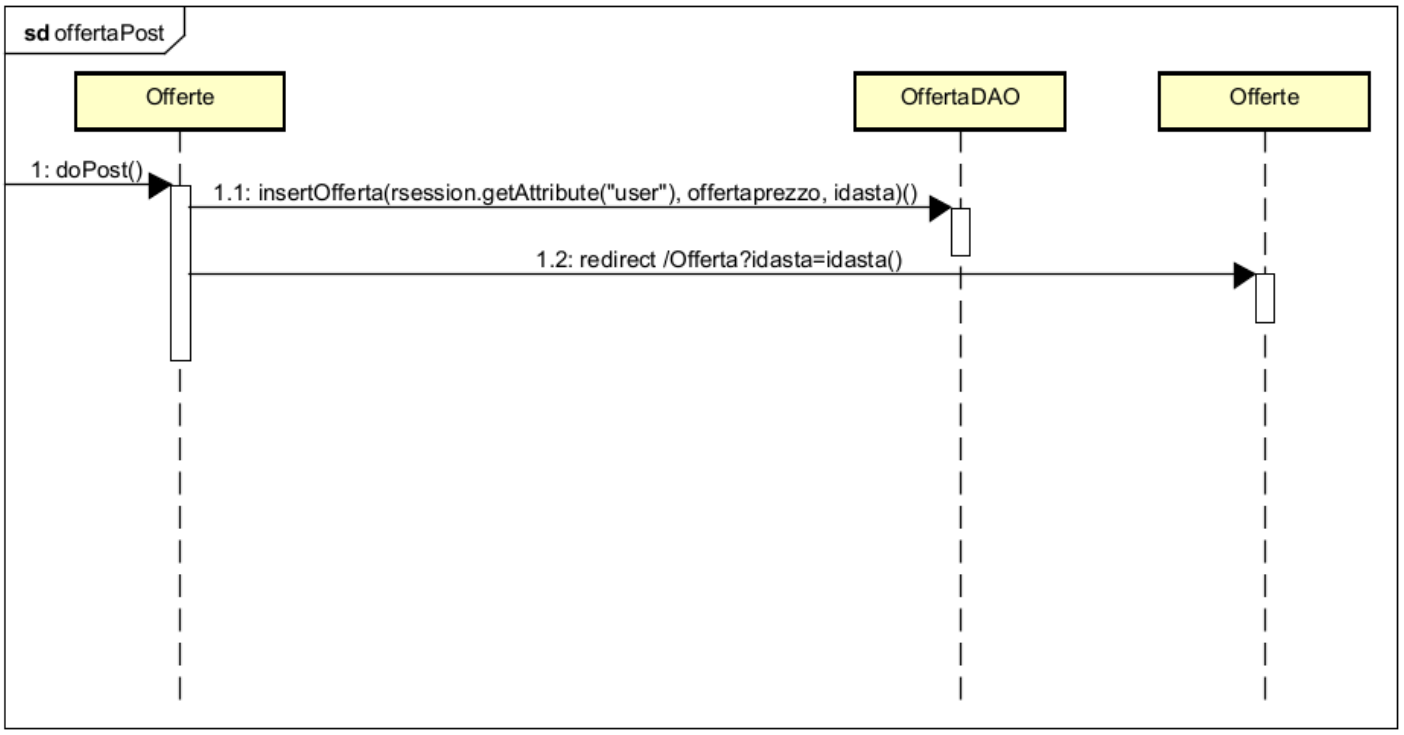
L'altra opzione era la pagina Acquisto, essa permette anche di ricercare aste da nome o descrizione. L'url corrispondente di ricerca sarebbe /Acquisto?search=keyword



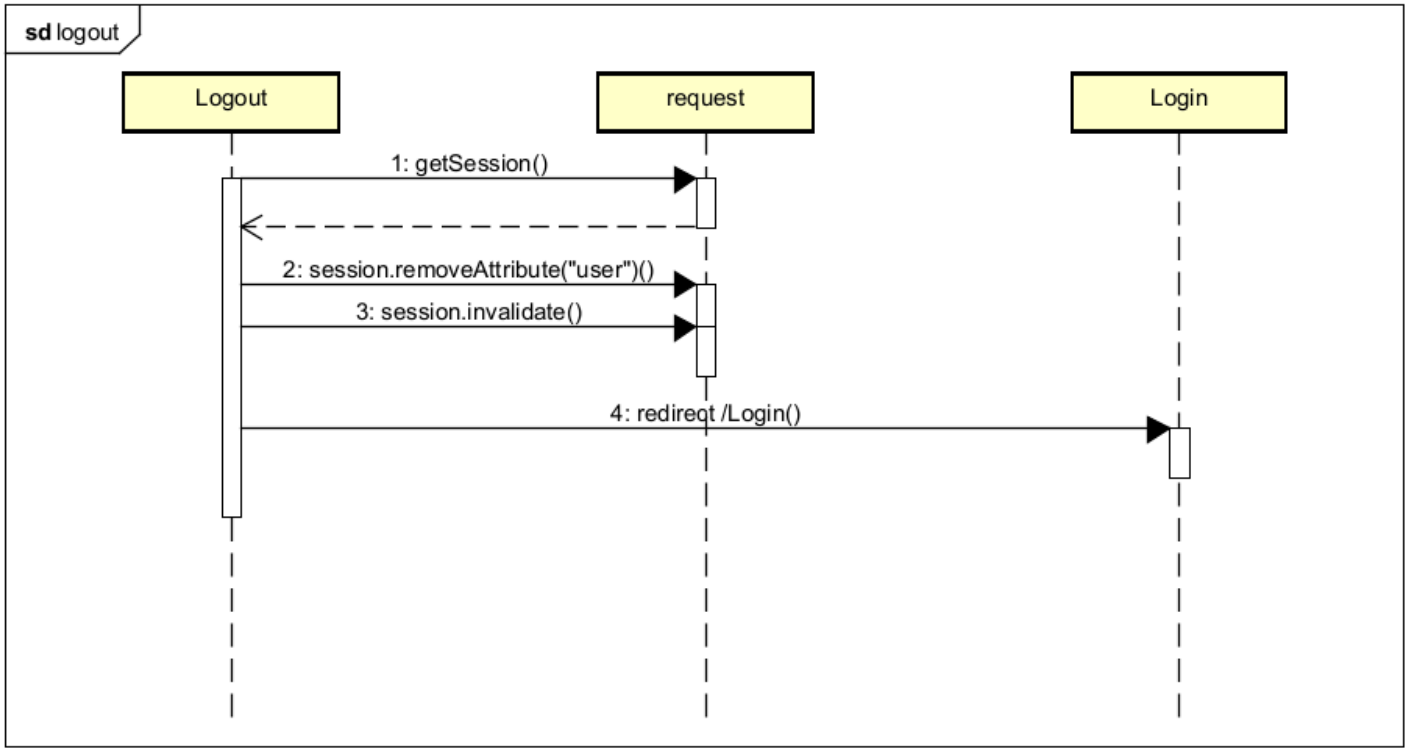
Da acquisto si puo poi andare a visualizzare l'asta con le offerte oppure le tue aggiudicazioni senza visualizzare il form per l'offerta

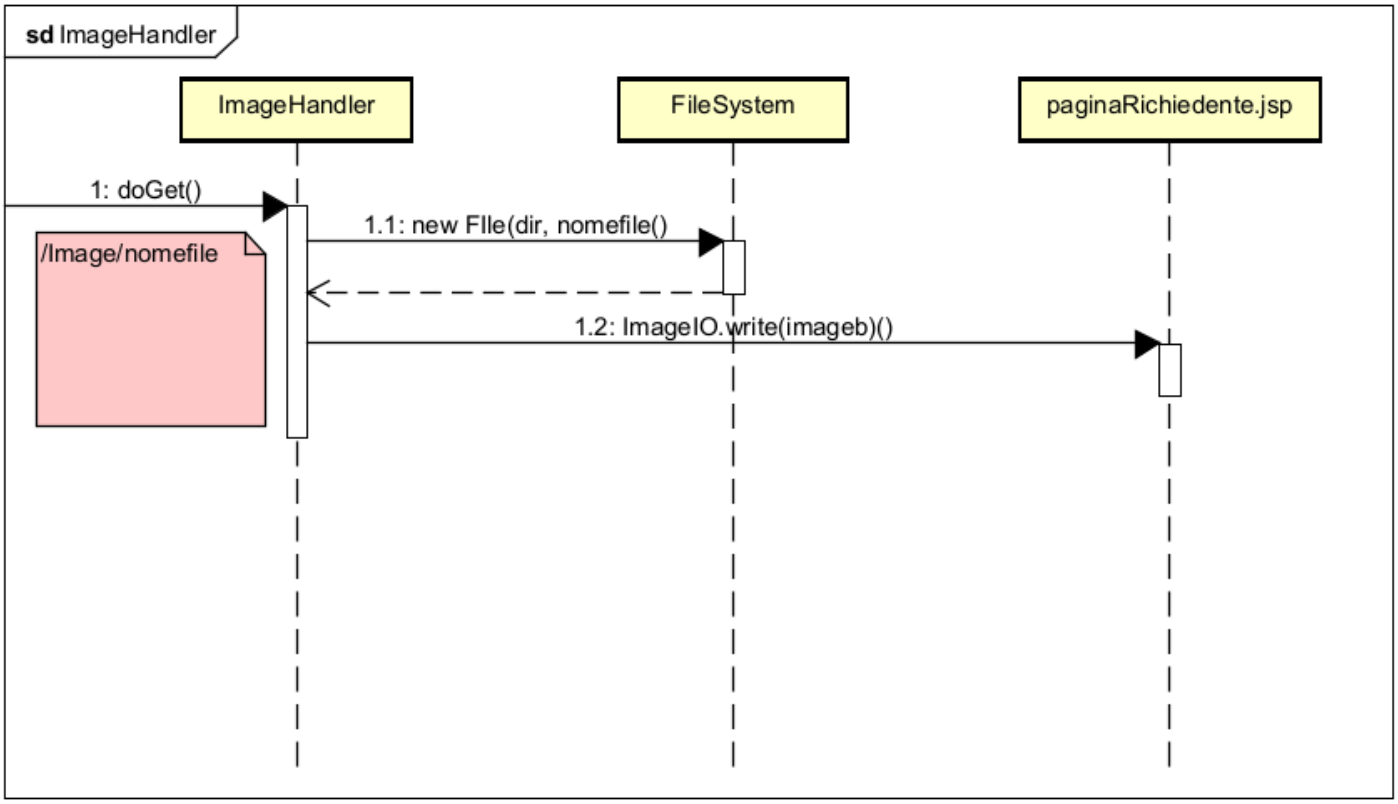


Come ho accennato prima si può anche effettuare un offerta con il doPost all'asta selezionata.

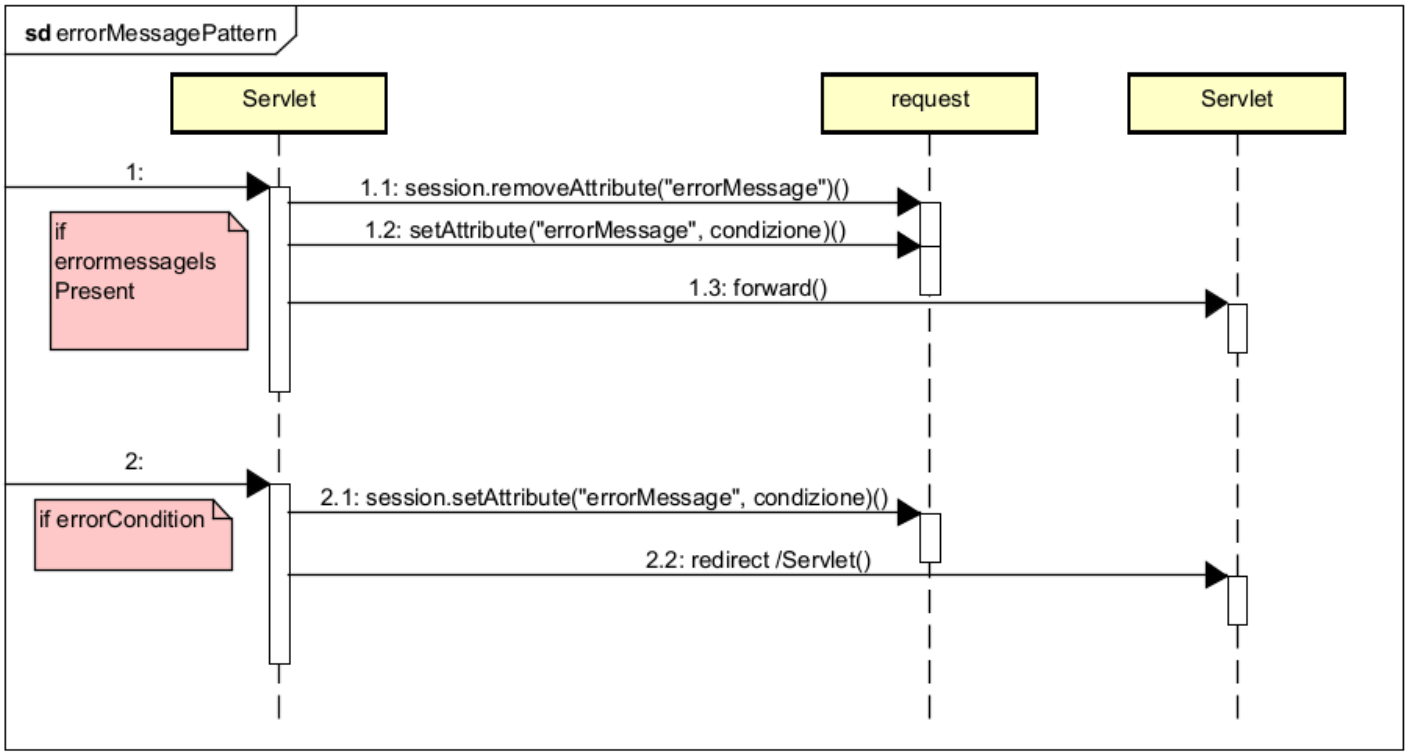


Oltre ai diagrammi del funzionamento effettivo dell'applicazione ne abbiamo alcuni che si possono definire di utilità come il logout e imagehandler





Nei diagrammi di rete inoltre non ho aggiunto gli errori ma perché li gestisco tutti nello stesso modo (come spiegato precedentemente) e ho fatto un diagramma generale separato



Documentazione versione JavaScript

2.1 Introduzione

A livello server nella versione viene cambiato solo l'invio dei dati i quali vengono prima formattati in stringa json e vengono inviati alla view attraverso la libreria Gson. E' stata aggiunta una servlet in più ovvero AsteVisitate la quale serve per inviare le aste che sono richieste all'apertura della homepage, nel prossimo paragrafo verrà spiegato meglio.
Un altra cosa che cambia a livello di server è la gestione degli errori infatti ora si limita a inviare un messaggio di successo o fallimento e sarà direttamente il javascript che si occupa della gestione dell errore mostrando il fallimento con il messaggio allegato per tot secondi.

2.2 Analisi modifica specifica

Modifiche dalla versione HTML non ci sono ma ci sono delle aggiunte tra cui avere l'applicazione web single page e memorizzare l'ultima azione dell'utente ovvero creare asta o visitare un asta nella sezione acquisto salvando il suo id. Per effettuare queste operazioni ho utilizzato il localStorage di javascript associato al currentUser che viene comunicato direttamente dal server, mentre per creare la single page ho usato un HTML template passato all'interno del javascript il quale attiva o disattiva/inserisce HTML all'interno.

2.3 Dettaglio Progetto JS

Il javascript è progettato per modellare il DOM che sono i template dell HTML.
Il Js è strutturato in modo tale da rispettare il pattern module e anche MVC e ho utilizzato una modalità di programmazione molto simile a Java. Ho creato un IIFE, il quale non espone così nulla all'esterno tranne la funzione init che inizializza tutti i moduli e i controller. I moduli creati sono:

StorageModule: che permette la gestione della data di scadenza delle informazioni memorizzate e verifica la disponibilità del localstorage e espone anche metodi per prelevare e scrivere dati (dati per salvare ultima azione o aste visitate).

UserData: permette di salvare le azione svolte attraverso lo StorageModule, sottolineo che ho impostato che le aste visitate al massimo 20, ma potenzialmente possiamo metterne fino a quando riesce a gestirne la memoria.

Ho scelto di usare il localStorage per una questione di capienza ma soprattutto per scambi di rete, se avessi scelto i cookies avrei dovuto portarli dietro a ogni chiamata/risposta.

ApiModule: gestisce i metodi get e post ajax passandogli un endpoint.

TemplateModule: preleva i template dal HTML e modifica i dati inviandoli nuovamente in modo dinamico con il supporto delle classi renderer che gestiscono cosa mostrare/disattivare. Oltretutto templatemodule sanitizza anche per evitare attacchi xss.

```
function sanitizer(text) : string { Show usages  paolobrusa
    const div : HTMLDivElement = document.createElement( tagName: 'div');
    div.textContent = text;
    return div.innerHTML;
}
```

In poche parole andiamo a forzare il tipo testuale al contenuto dinamico.

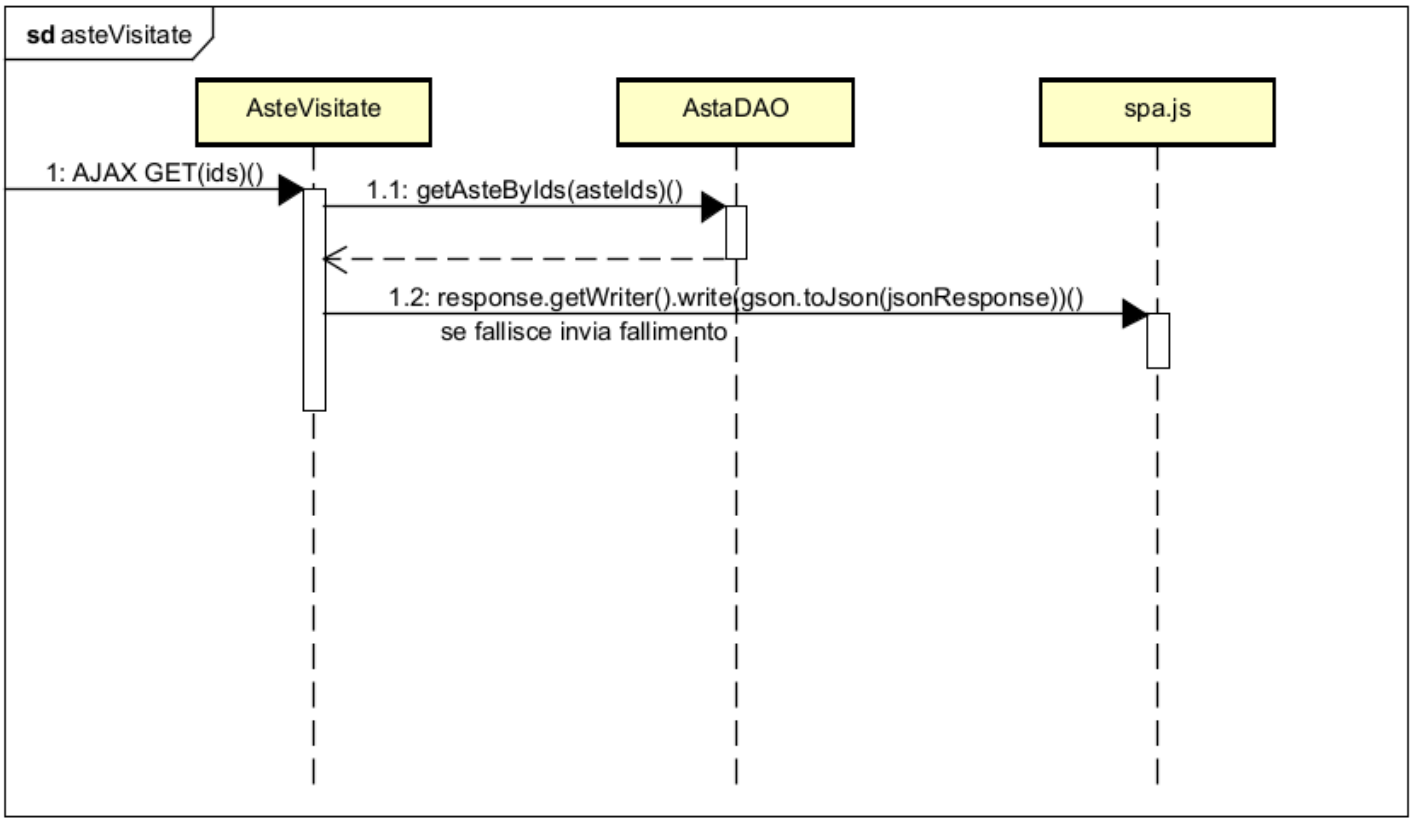
Poi abbiamo i controller che gestiscono le chiamate ajax per la sezione corrispondente, ogni controller si può dire che gestisce le pagine della versione HTML. Mi vorrei soffermare su due in particolare i quali servono nella versione single page.

NavigationController: serve per gestire le varie schermate e fare in modo quindi che attraverso i bottoni si attivano e disattivano i template corrispondenti.

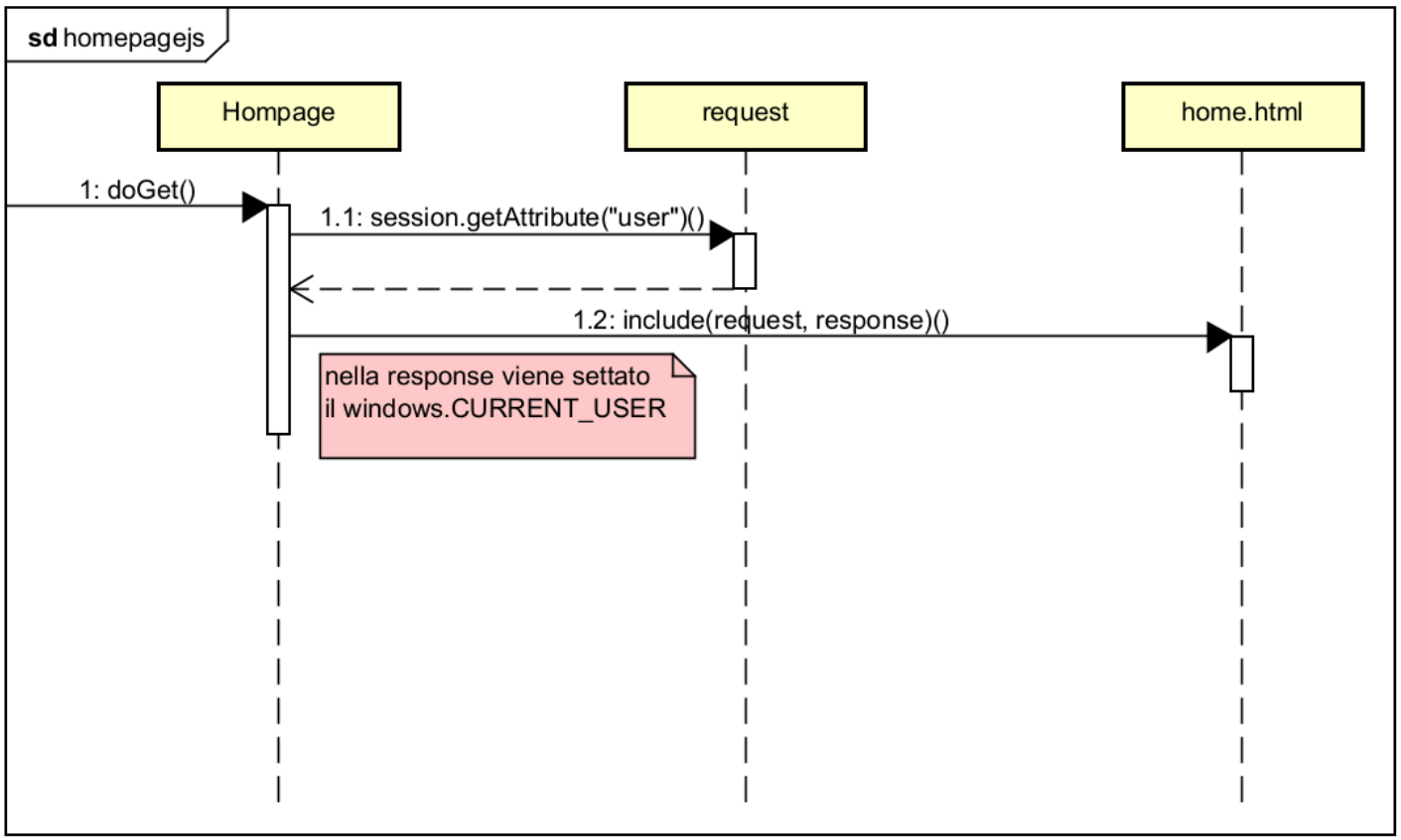
EventController: serve per centralizzare e inizializzare gli eventi click e submit facendo così possiamo gestirli in questa classe senza delegare alle classi corrispettive.

2.4 Sequence Diagram

I sequence diagram non cambiano in modo significativo a parte la gestione errori che come detto prima non esiste più e per il fatto che adesso non esistono più redirect ma soltanto invio di json.

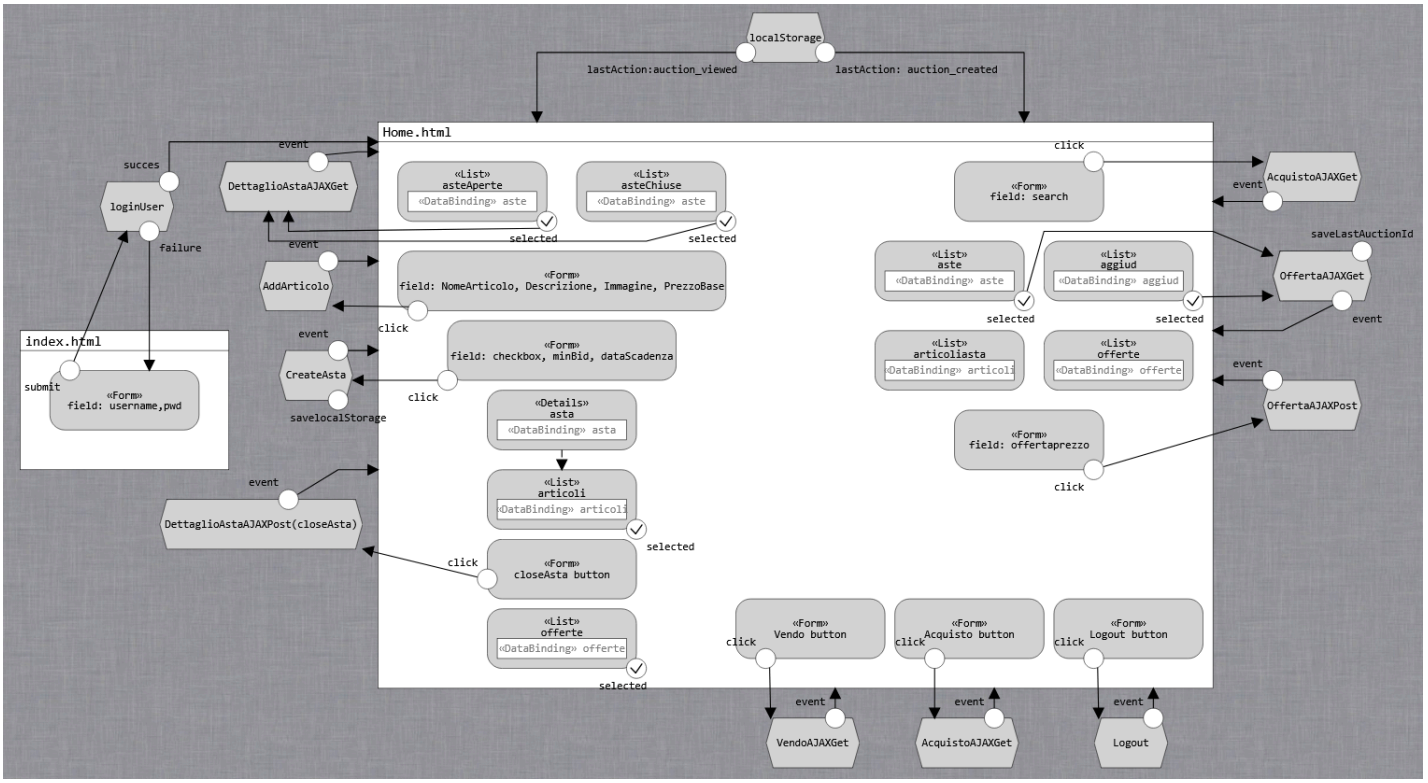


Questa è la nuova servlet che viene utilizzata quando viene chiamato Startup di spa.js che permette così di visualizzare le aste viste precedentemente oppure di visualizzare la pagina vendo. Gli altri sequence diagram non li riporto dato che sono uguali agli altri con solo la modifica della tipologia di chiamata e risposta. L'unica che conviene comunque analizzare è la servlet Homepage.



Come vediamo adesso la home page non si occupa di far visualizzare solo il HTML come prima che serviva solo per visualizzare il jsp, adesso prima di tutto richiede l'utente che poi andrà a scrivere direttamente nel html in modo tale che il js lo possa prelevare, infatti usa include per includere anche ciò che è stato aggiunto al HTML

2.5 IFML Diagram



La differenza tra quello pure HTML e questo JS sta nel fatto appunto che esiste una singola pagina la quale viene modificata ogni qual volta che viene effettuata un operazione.