Analysis and Design Document
Student: Şinca Mădălina
Group: 30433

Revision History

Date	Version	Description	Author
10/04/2018	1.0	Initial iteration of document	Șinca Mădălina
22/04/2018	1.2	Second iteration of document	Șinca Mădălina

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 4

1. Domain Model 4

2. Architectural Design 4

2.1 Conceptual Architecture 4
2.2 Package Design 4
2.3 Component and Deployment Diagrams 4

III. Elaboration – Iteration 1.2 4

1. Design Model 4

1.1 Dynamic Behavior 4
1.2 Class Design 4

2. Data Model 4

3. Unit Testing 4

IV. Elaboration – Iteration 2 4

1. Architectural Design Refinement 4

2. Design Model Refinement 4

V. Construction and Transition 5

1. System Testing 5

2. Future improvements 5

VI. Bibliography 5

I. Project Specification

The purpose of this document is to collect, analyze, and define high-level needs and features of the Arch Climbing Wall (ACW) Web Application. It focuses on the capabilities needed by the stakeholders and the target users, and why these needs exist. The details of how the ACW Web Application fulfills these needs are detailed in the use-case and supplementary specifications.

II. Elaboration – Iteration 1.1

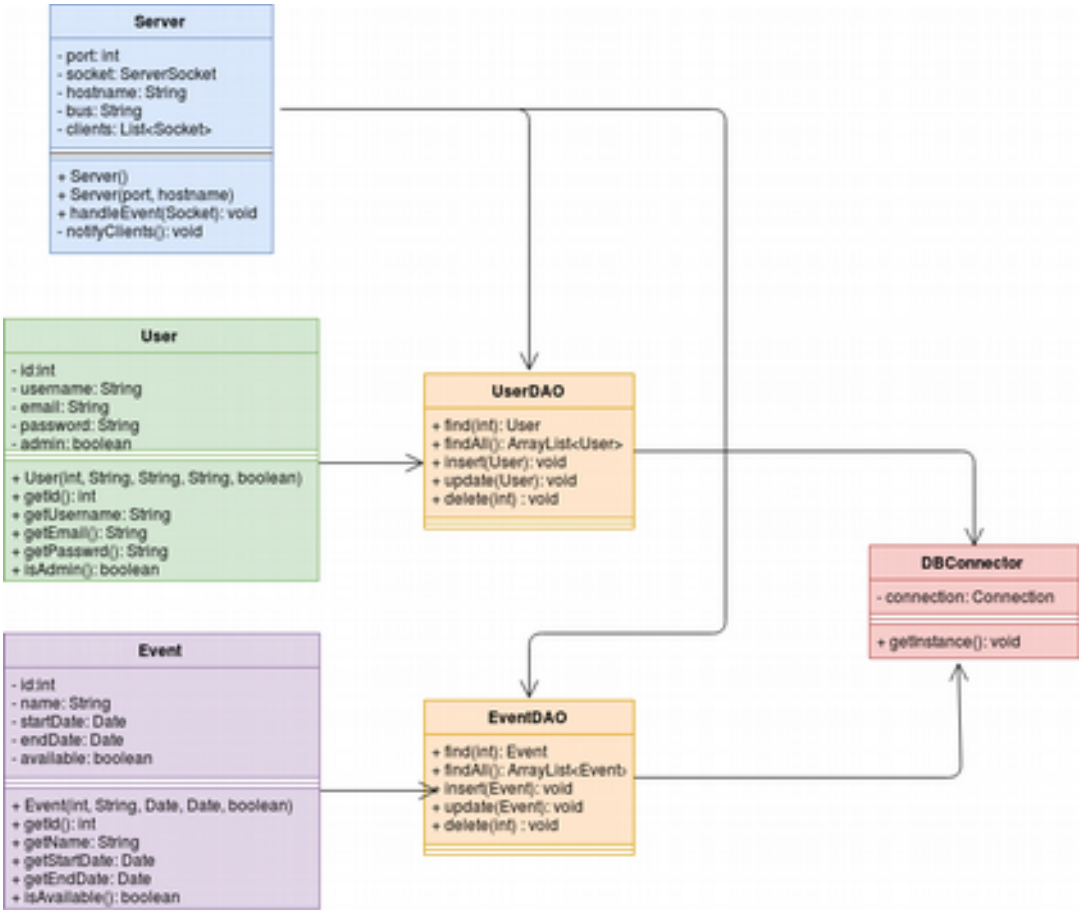
1. Domain Model

The Client will contain 1 model:

- *User – represents a user, either the active one or inactive one.*

The Server consists of the following models:

- *User – same as above;*
- *Event;*
- *Course;*



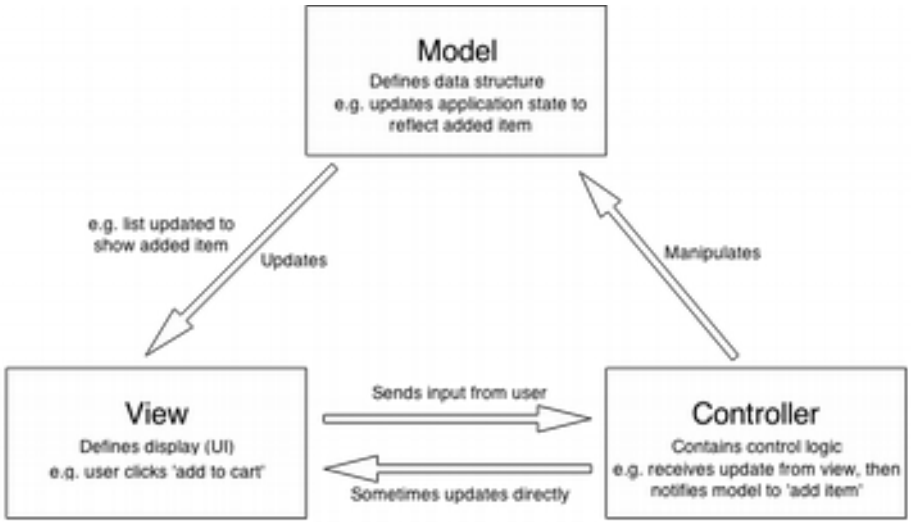
2. Architectural Design

2.1 Conceptual Architecture

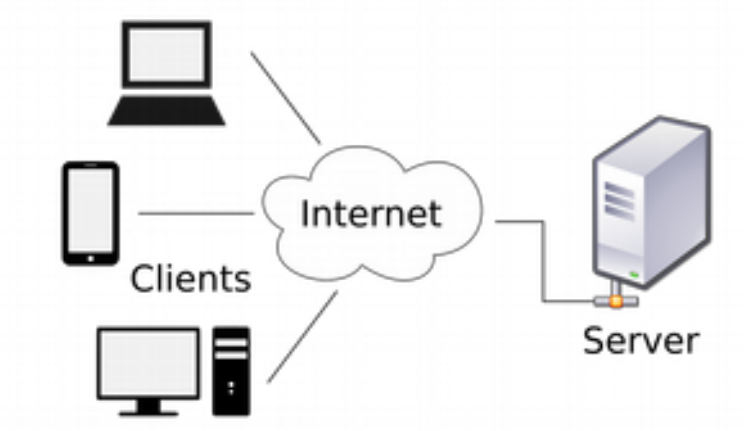
The system consists of two endpoints : the users and the server. The used architectural patterns are:

- Client – Server : The server sends request / response messages to each client at a given point in time and vice-versa. While communicating with all its clients, the server also listens for new connections;
- Event Bus : The server acts as a virtual bus. Every client listens for events while performing playback. When a client wants to perform an action (eg. pause, stop), it sends a message to the server telling it to place the message on the bus, and all the other clients are then notified.

Model View Controller (**MVC**) is a software **architecture** pattern, commonly used to implement user interfaces: it is therefore a popular choice for architecting web apps. In general, it separates out the application logic into three separate parts, promoting modularity and ease of collaboration and reuse.

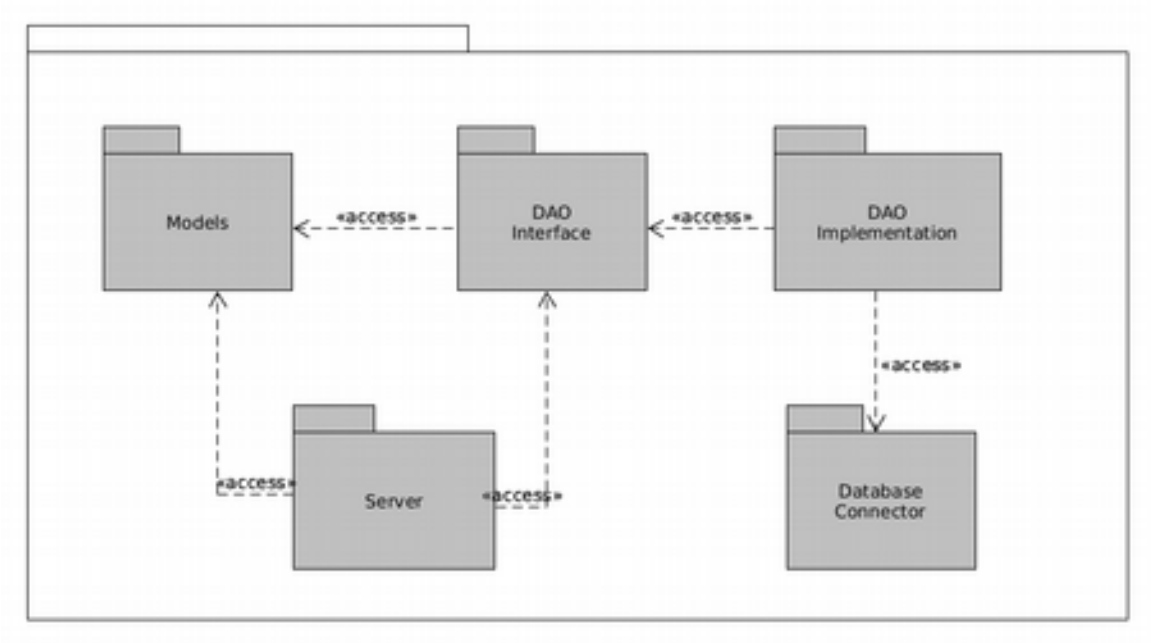


Client-server architecture (**client**/**server**) is a network **architecture** in which each computer or process on the network is either a **client** or a **server**. Servers are powerful computers or processes dedicated to managing disk drives (file servers), printers (print servers), or network traffic (network servers).

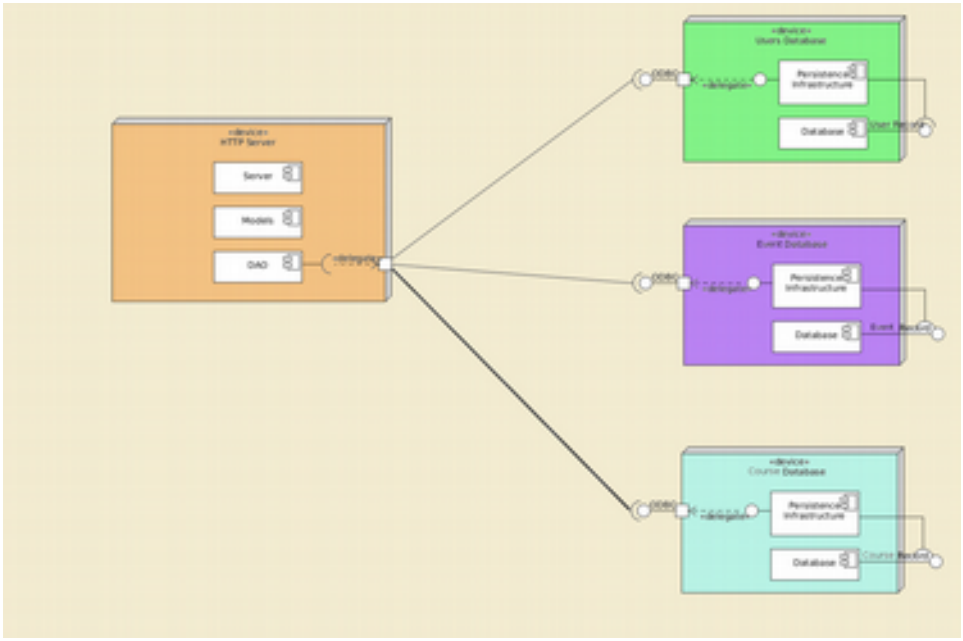


The described approach has been selected due to its simplicity and performance.

2.2 Package Design



2.3 Component and Deployment Diagrams



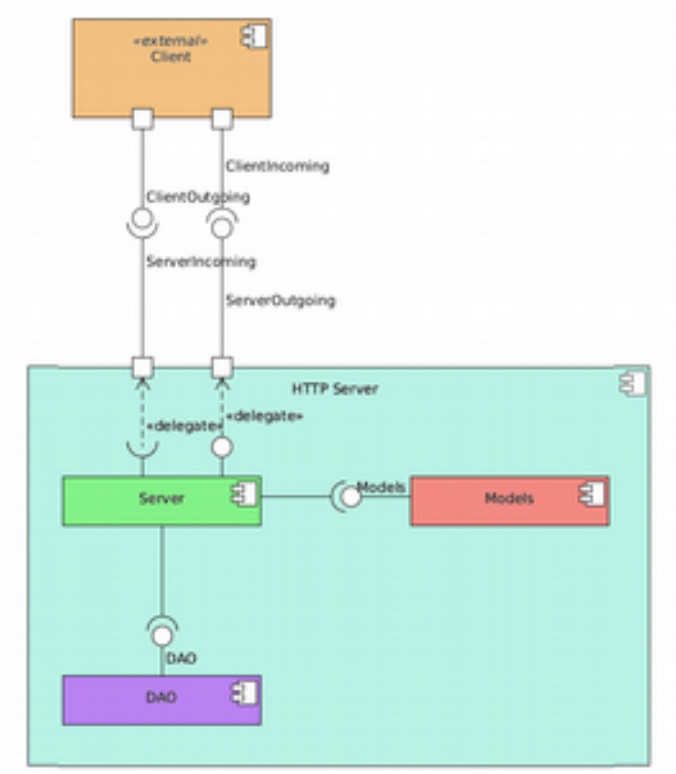
III. Elaboration – Iteration 1.2

1. Design Model

1.1 Dynamic Behavior

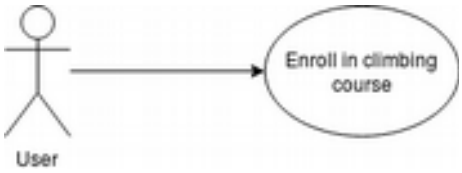
Course Enrollment Steps:

- The user logs in;
- The user queries through the available course options;
- The user selects the course he/she desires to enroll in (or wants to enroll their children in);
- The user presses the “enroll me” button at the bottom of the page, below the course

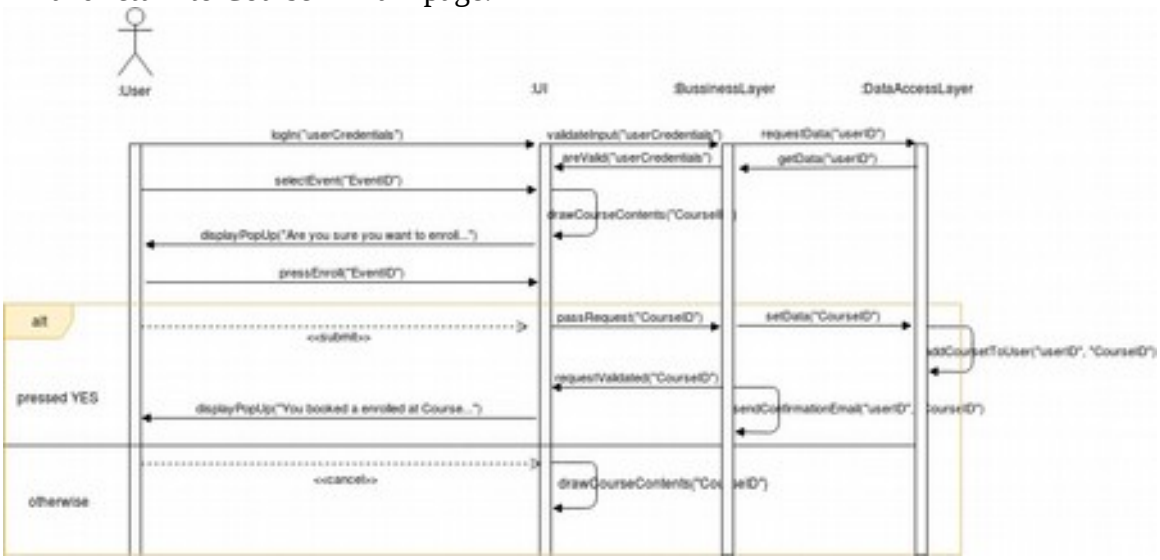


description;

- Pop-up window displaying “Are you sure you want to enroll to Course X?” - No, Yes;
- The user presses Yes;
- Pop-up displaying “You have just enrolled to Course X. You must have received an e-mail confirmation and Course X should now appear under the <<my courses>> section in your profile”;
- The user will receive an e-mail confirmation and the course they just enrolled in will appear under the “my courses” section in their profile;

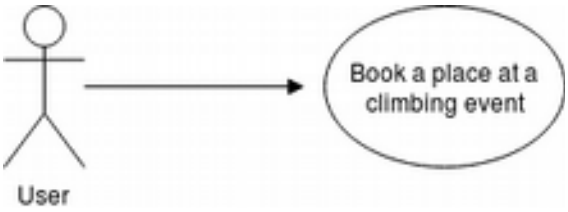


- ***Extensions:*** If the user presses the No button then the application cancels the enrollment and return to Course X main page.



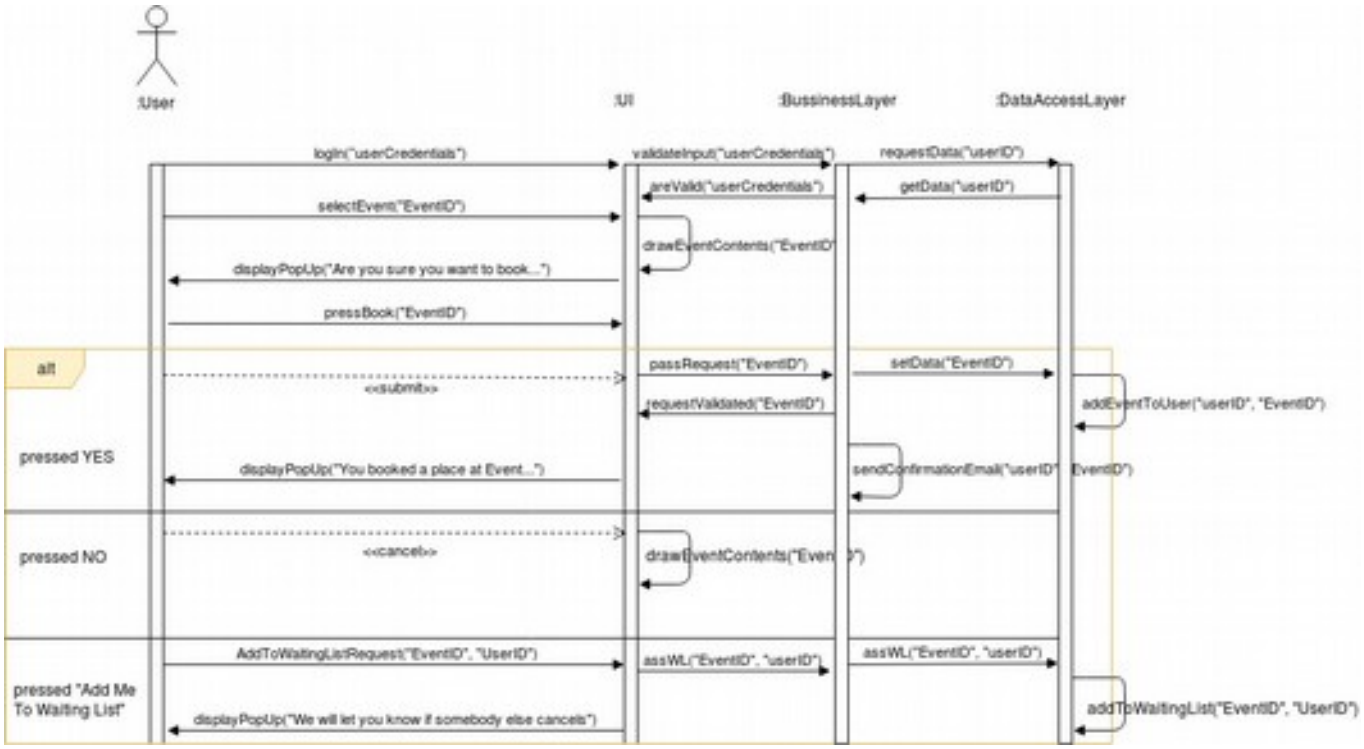
Event Booking Steps:

- User logs in;
- The user queries through the available events options;
- The user selects the event he/she desires to attend;
- The user presses the “Book A Place” button at the bottom of the page, below the event description;
- Pop-up window displaying “Are you sure you want to book a place at Event X?” - No, Yes;
- The user presses Yes;
- Pop-up displaying “You have just booked yourself a place for Event X. You must have received an e-mail confirmation and Event X should now appear under the <<my events>> section in your profile”;
- The user will receive an e-mail confirmation and the event they just booked will appear under the “my events” section in their profile;



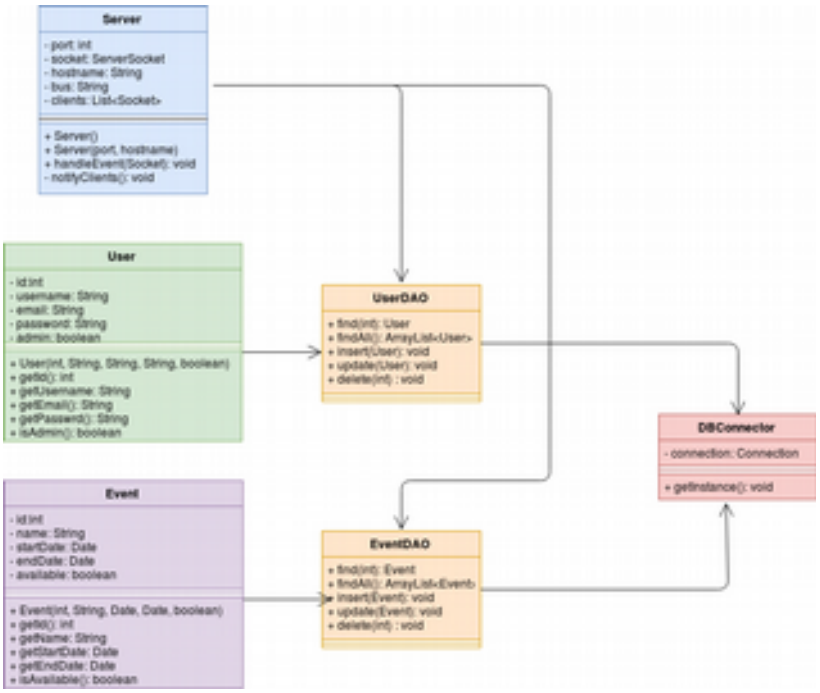
- ***Extensions:*** If there are no more places available at said event, then the user has the option o pressing “Let Me Know If a Place Becomes Available” button.

- If the user presses the No button then the application cancels the enrollment and return to Event X main page.

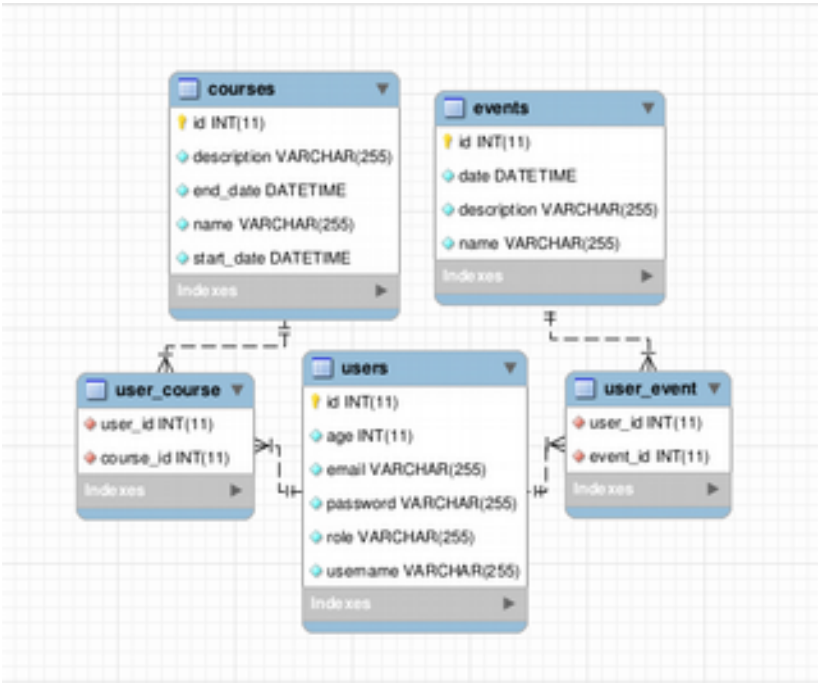


1.2 Class Design

The most important patterns used in this project’s creation are the Command, Observer and Factory. The Command pattern is used to add functionality to the UI’s buttons, the Observer pattern is useful for displaying data onto the UI, while the Factory pattern is used to create a connection of the back-end to the database. A conceptual class diagram can be seen below:



2. Data Model



The presented system relies on 3 data models: the User data, Course data and the Event data. (+ the

auxiliary connector tables). The User data encapsulates a user's username, email address and password, while the Event data model does so about the Event Credentials.

3. Unit Testing

The unit testing should consists of gradually testing small parts of the application such as **user connectivity** (log in, sing in), **uploading events** such as courses or **offers.** Each Service class will be tested individually (ideally by the use of Junit test & the dependency injections will be created using a mocking framework). A Junit test case is characterized by a known input and an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post-condition.

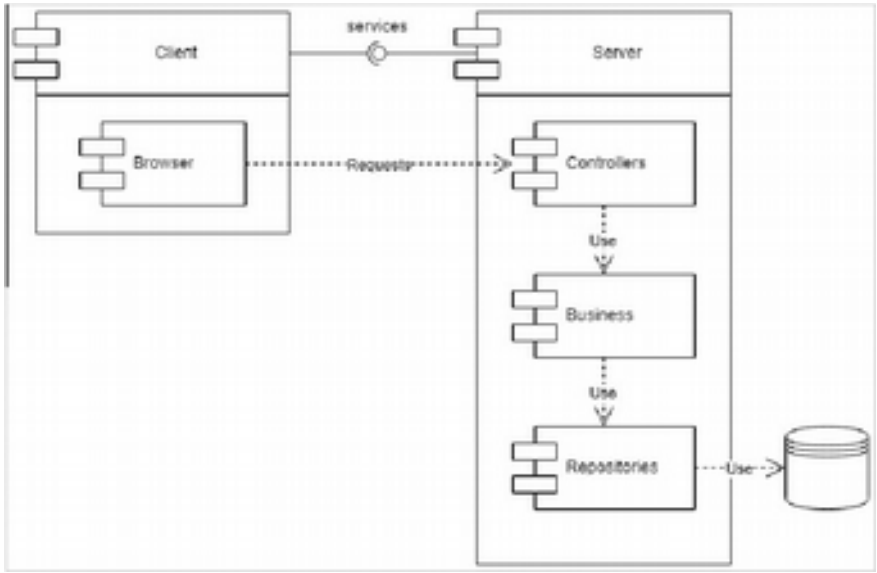
Example:

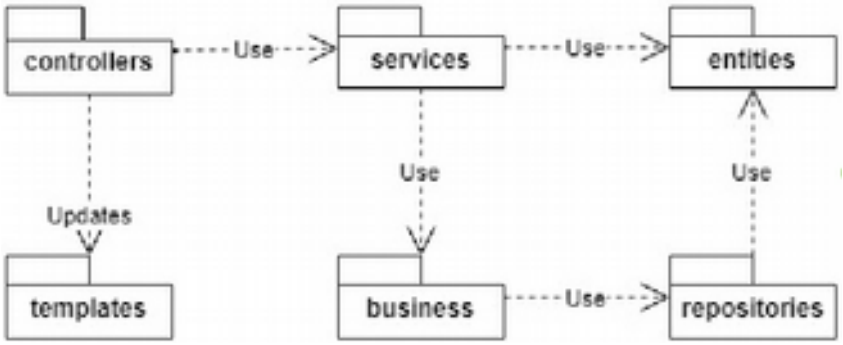
step 1: create a user with specific credentials e.g. id = 34;
step 2: create a climbing course with id = 18;
step 3: test the enrollment feature by making the new user enroll to the new course;
step 4: check whether the new user has been truly enrolled to the course and whether the course now contains the new user as a participant;

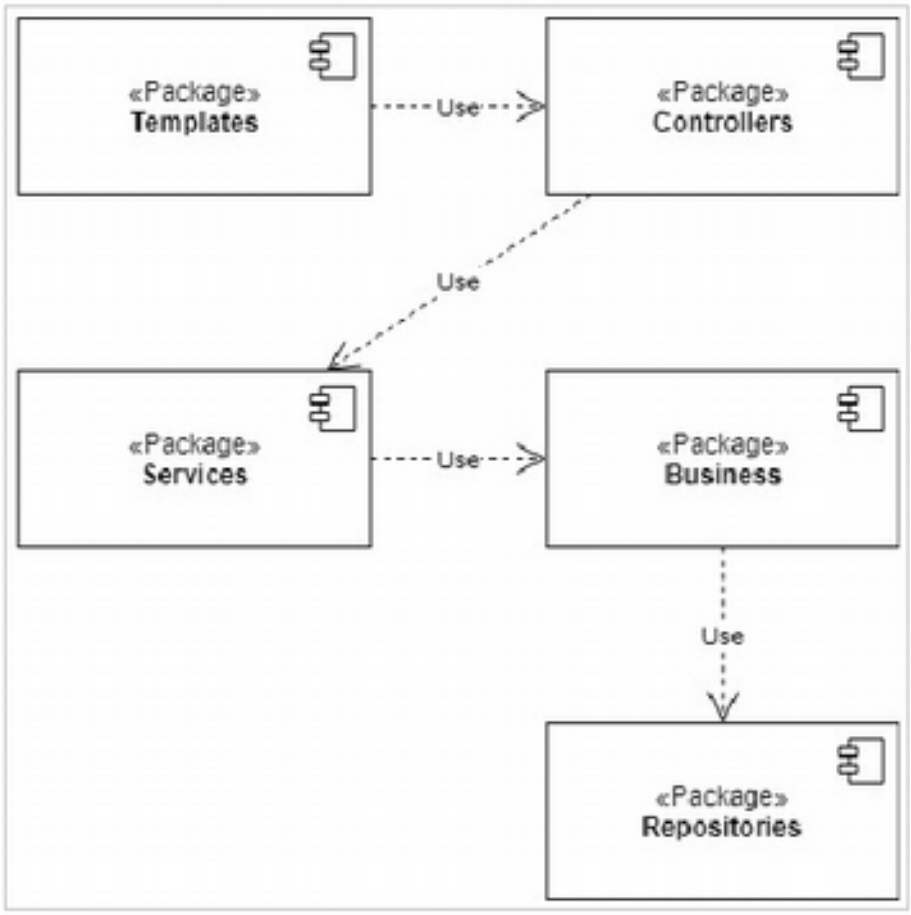
IV. Elaboration – Iteration 2

1. Architectural Design Refinement

Architectural changes were just minimal and were introduced to accommodate the workflow using the SpringMVC framework and the Thymeleaf templating language. The new **architectural diagram is presented in the figure below**. There is new business layer added between the controllers and repositories that handles the business functionalities. This is also highlighted in the **package diagram and in the component diagram**. In the latter two the services packages is also present, which acts as a moderator between the controllers and the business layer.



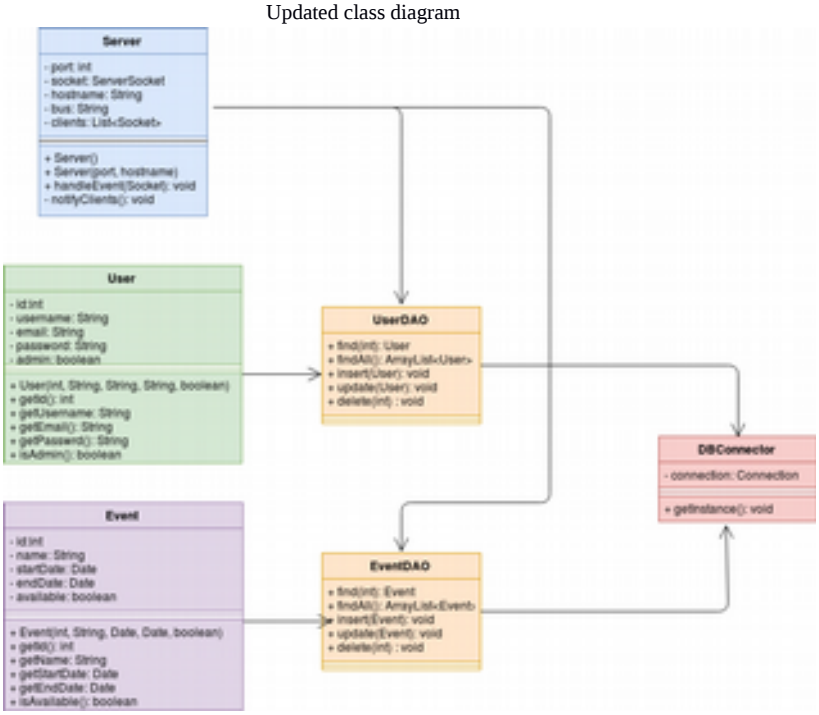




2. Design Model Refinement

The model of the back-end also underwent different changes to better suit the needs of the framework. The service layer, instead of being structured by the use-cases, is structured around the domain model and provides functionalities of creating, updating and reading domain model data. The domain model also changed: there is only one User class, and each User has a Role: either Admin or User. This change was introduced to adapt to the Spring Security, which needs an email/user name and a role for each user. This way, each role can access only functionalities associated to them. Since the domain model changed, the database schema also changes in accordance.

Updated class diagram



V. Construction and transition

1. System Testing

Integration testing was realized manually, with the server running and in a browser the functionalities being tested. The testing process consisted of trial-and-error attempts of manually looking for errors, such as access to restricted URLs for anonymous users and users with different roles. The testing was incremental, the first objective for each use case being the success scenario's working, and then the detecting of issues. When designing web-applications there is usually no way to test automatize the testing of the user interface, so manual testing is a large part of the validation process. The data sent by the user to the server is validated against incorrect data and is only accepted if it does not yield inconsistencies on the system and in the database.

2. Future improvements

- Better security protocol (e.g. Spring Security);
- User receives e-mail reminder when an event they want to attend is coming up;
- Implement the feature of related courses / articles;
- Implement the feature of monetary transaction in order for a user to be able to buy himself tickets to events or enrollment cards for courses;

- A more diverse account with the options to add / find friends, post on your timeline, on your profile etc.

The system's implementation could also be improved by reducing the coupling in the business layer, by using more web-services for the implementation of the features. A major benefit would consist in migrating the system to a REST API, where the server would receive and send JSON objects that represent domain objects and Data Transfer Objects, and the client would implement the visualization. This way the client could have a much more responsive and modern UI.

VI. Bibliography

M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee and R. Stafford, Patterns of Enterprise Application Architecture, Addison Wesley, 2002.

https://javabrains.thinkific.com/courses/springboot-quickstar

https://www.thymeleaf.org/doc/tutorials/2.1/thymeleafspring.html

http://www.baeldung.com/spring-email

https://spring.io/guides/gs/serving-web-content/#scratch