NHL Fantasy League

Project Report

Masters Of Engineering - Electrical and Computer Engineering
ECE-651 Foundations Of Software Engineering

Group 1:

Sneh Patel	sneh.patel@uwaterloo.ca
Harshitkumar Patel	hv6patel@uwaterloo.ca
Mohit Gupta	m52gupta@uwaterloo.ca
Prateeti Deb Chaudhuri	pdebchau@uwaterloo.ca
Priya Patel	pb4patel@uwaterloo.ca

1. Introduction

1.1 Overview

Fantasy sports applications allow users to build custom leagues in which they compete with other users as well as, custom teams of real players that play professional sports (within a certain organization). The application assigns points based on the statistics generated by these professional players or teams in real life. Our application tries to implement one such sport, hockey which is played within the National Hockey League (NHL) organization. Our fantasy app is a web-based application designed for players who want to play small-scale fantasy hockey. They can play fantasy hockey without worrying about a whole bunch of rules that regular fantasy apps mandate. This fantasy app does not require too much maintenance on the user's part as required for regular high-scaled fantasy applications. Implementing this as a web-application was the best option because of the amount of flexibility it offers within a small time-frame given to build this project. The development team took all options into account and thus, decided to move forward with the web-application approach to maximize the quality of the product for the time-frame given to deliver this product.

1.2 Functional Properties

=> User Registration
=> Authentication & Encryption
=> User Operations (edit, display, etc.)
=> League Operations (create, join, display, etc.)
=> Team Operations (select, delete, etc.)
=> Scoring (apply points to users based on their team)
=> Real Scores (get real scores from the web)

1.3 Non-Functional Properties

=> Security: the application is secure and no unauthorized user is allowed to access other users' data (users can only access their personal data, not anyone else's).

=> Data Persistence: data of user should be available and stored whenever the user requests for it.

=> Reusability: Back-end and front-end are loosely coupled. We can use both for another application.

2. Architecture

2.1 Overview

The architecture of the system generally describes the behavior of the system. It incorporates the basic design decisions made by the software developers. The fantasy application is composed of a front-end implemented, a back-end, a database, and an API responsible for fetching real data from the web. This application incorporates various architectural patterns which have many decision implications behind them.

2.2 General Architecture-Layered Pattern

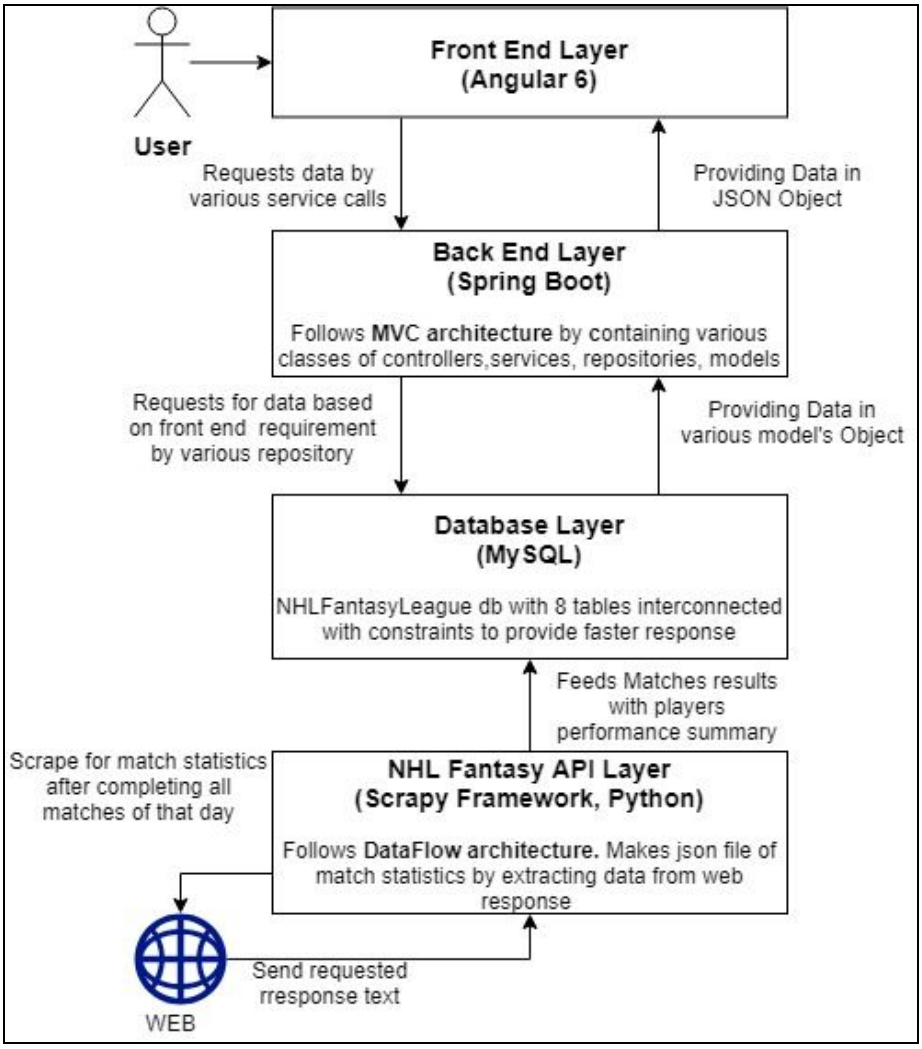


Figure 1.0: The different layers of our application

The fantasy application follows a layered architecture (as seen in Figure 1.0) consisting of four layers named front-end, back-end, data layer, and the API layer. Each layer has different architectures based on functionalities they provide.

This architecture made a lot of sense to use because a typical web application tends to use this architecture to allow for flexibility, maintainability, and scalability. This architecture is very simple to understand and allows the developers to precisely know the location of each component without rigorously searching for it in the codebase when it becomes very large and complex. This allows for increased maintainability. The layered architecture also provides flexibility and scalability. This allows future development of any new features easily without changing much of the general architectural structure of the system.

2.2.1. Frontend Layer (Event-based Architecture)

This is the layer that the user directly interacts with. It is implemented using Angular 6 with the Angular CLI tool. The developers decided to use this framework in the front-end because Angular 6 is cross-platform and allows an app to be used in various computing platforms and development environments. For example, Angular 6 can have backend support in different programming languages like Java, C++, etc. despite being written in TypeScript. Internally, Angular 6 uses other architectural patterns which in turn, provide great benefits to the system. More specifically, the reason why the developers selected Angular 6 is that of its utilization of the event-based architectural pattern as seen in Figure 2.0 below.

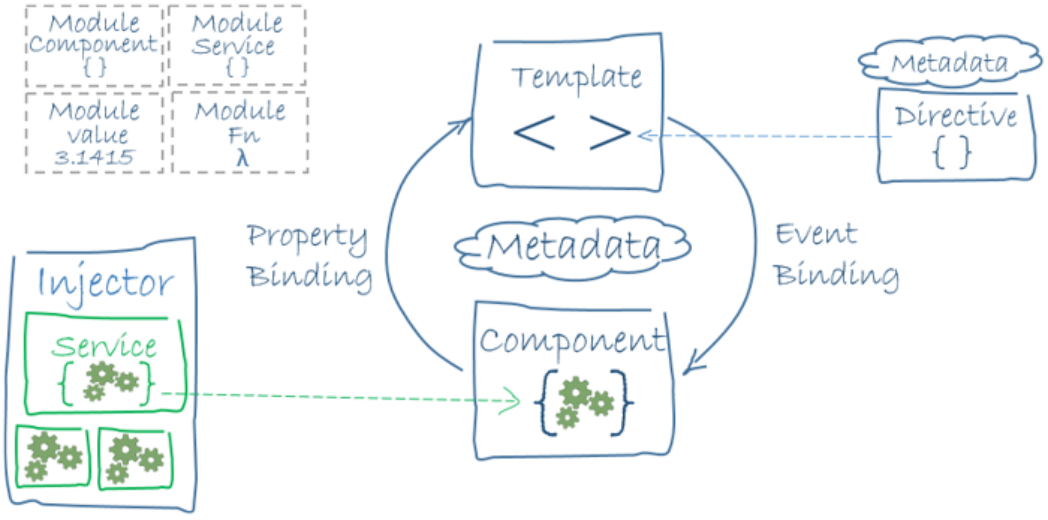


Figure 2.0: An angular component and its template are bind to each other based on property binding and event binding.

This allows for a dynamic view which constantly updates whenever properties within the component change and the properties of the component change whenever an event from the view

occurs. The benefits this provided to the developers is that it lowered the complexity of the code and made it more reusable by having reusable components.

2.2.2. Backend Layer (MVC architecture)

This is the layer that is responsible for implementing the business logic of the application. It is implemented using Java EE in particular, the Spring Boot framework. The developers decided to use this framework in the back-end because it uses the MVC architectural pattern. See Figure 3.0 for a high-level view of the MVC pattern.

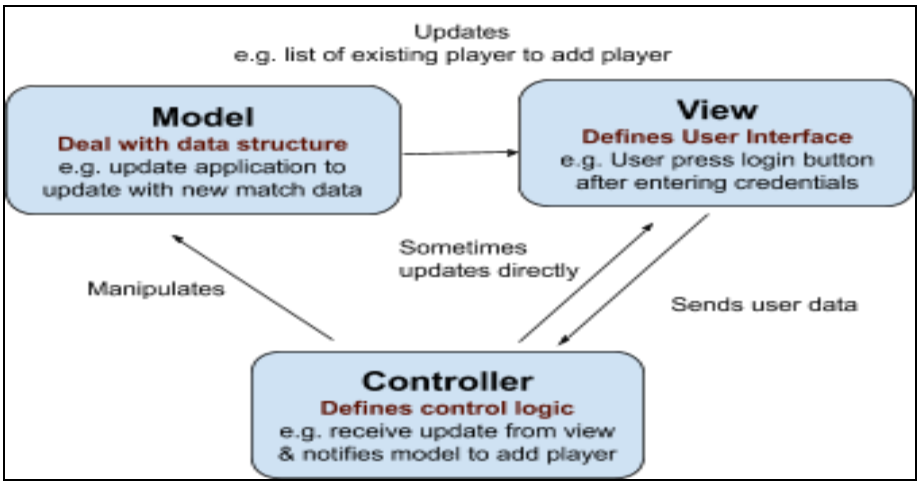


Figure 3.0: MVC divides a system into three main parts.

The main intent of the developers was to separate the user interface from the rest of the application and hence, a framework that utilizes MVC is the most applicable pattern for this requirement because it allows for separation of concerns. The model is responsible for managing the program data and its integrity. The view and controller are responsible for providing the user with a way to interact with this data. In order to separate the front-end with the back-end, the MVC architectural pattern was the ideal choice. Due to the loose coupling between the model, view, and controller, there is no dependency between these components. So, the application is fully functional even without the view because it has no access to the model or the controller. MVC allows supports parallel development. Multiple developers can work together in different modules without conflicting with each other. This helps in a faster development process given the time constraints. Figure 4.0 depicts the Spring MVC architecture.

In this application, the dispatcher servlet receives HTTP requests from a user and chooses the controller using the handler mapper component, then the controller processes this request by calling the appropriate service. The service returns a model/view object to the dispatcher servlet. Subsequently, the dispatcher servlet passes the model object to the view (i.e. the front-end

implemented in Angular 6 for this application) using the resolver component as seen in Figure 4.0.

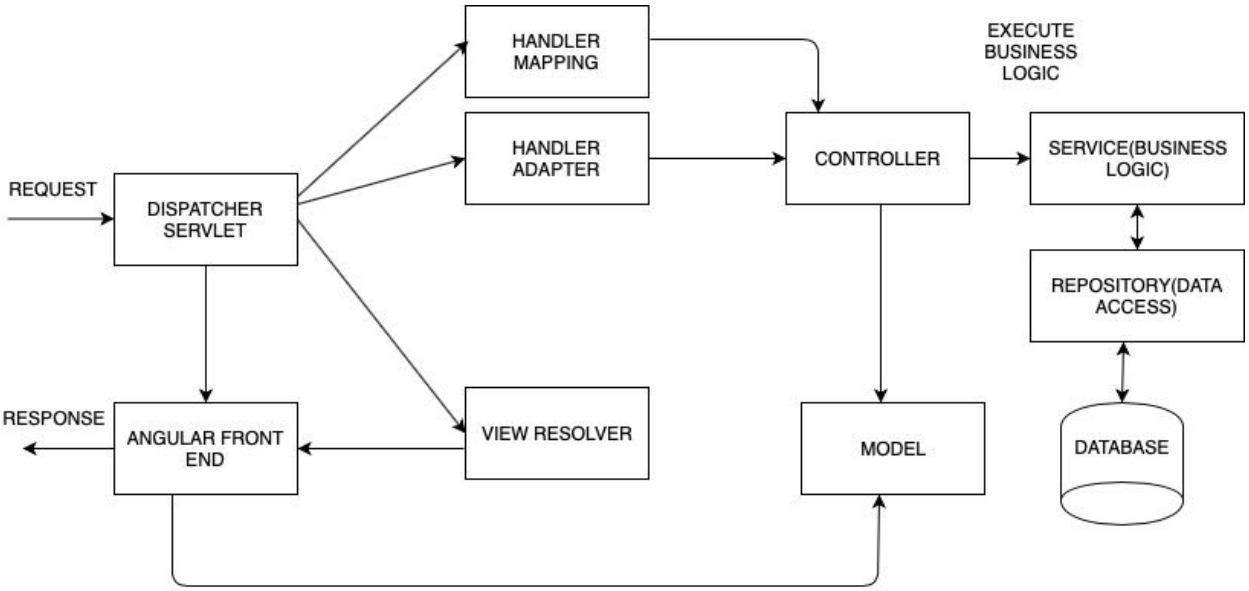


Figure 4.0: Spring MVC Architecture

2.2.3. Database & NHL Fantasy API Layers

2.2.3.a. Database

The application uses a MySQL database to store all data. This is one of the major components of the application because, by itself, it composes the database layer of the application.

2.2.3.b. NHL Fantasy API (Data-flow Architecture)

The NHL fantasy API itself composes the API layer of the application. It is responsible for feeding data to the database layer. It uses the Scrapy framework in Python and follows dataflow architecture. The data flow in the API is controlled by the execution engine.

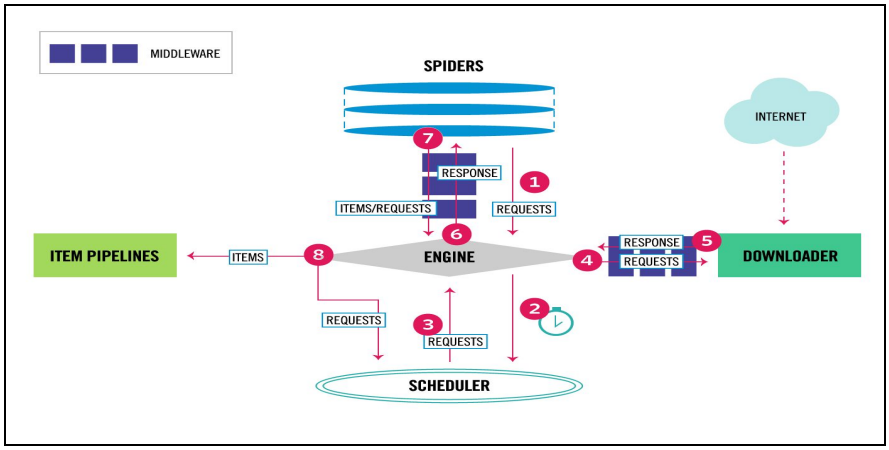


Figure 5.0: Data Flow Architecture in the API

The engine as seen in Figure 5.0, gets the initial requests to crawl match results with the score summary from the `getResultForDelta` spider. It schedules the requests in the scheduler and asks for the next requests to crawl. The scheduler returns the requests to the engine and sends them to the downloader. Once the requested page finishes downloading, the downloader generates a response (with that page) and sends it to the engine. The engine then receives the response from the downloader and sends it to the spider for processing. The spider subsequently processes the response and returns the scraped items and the new requests to the engine. Afterwards, the engine sends the processed items to the `GetMatchResultsScraperPipeline` pipeline. Finally, the processed requests are sent to the scheduler where the next possible requests to crawl are queried. This process repeats from the beginning until there are no more requests from the scheduler. In this approach, the data enters into the system and then flows through the modules one at a time until they are assigned to some final destination (output to JSON file). It has low coupling between filters and flexibility by supporting both sequential and parallel execution which justified the reason for choosing this architecture in the system.

3. System Design

As mentioned before, the system is composed of many layers in which there exists many components. These components are broken down by other components that form the low-level design of the overall system (i.e. the classes). In each layer, certain design patterns are used to make the code more reusable and certainly, modular.

3.1. Front-End

3.1.1. Components

3.1.1.a. Angular Component

The angular component controls a "patch" of screen called a view (or template-see below). Angular components are the most basic UI building block of an Angular app. An Angular app contains a tree of such components. These components are very reusable and can be injected into different parts of the views which is one of the major reasons as to why the decision to use Angular on the front-end was made. This application contains many types of components such as the `UserComponent`, `LeagueComponent`, `PlayerListComponent`, etc. See Figure 6.0 for a list of such components.

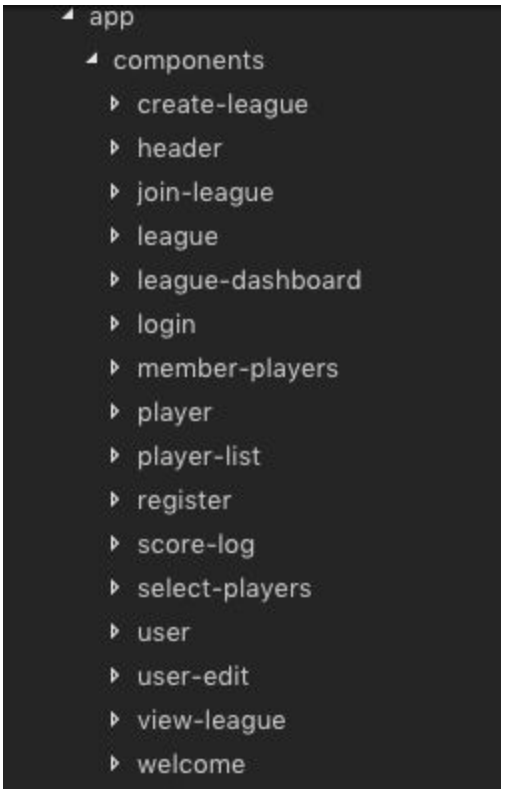


Figure 6.0: Application components (front-end).

3.1.1.b. Angular Service

Angular components shouldn't fetch or save data directly and they certainly shouldn't knowingly present fake data. They should focus on presenting data and delegate data access to a service. This fetching of data is done directly via Angular services. Angular services are called and invoked whenever a component needs data from a back-end. Each component has their own services which they rely on to get this data. The application contains various such services to handle fetching of data (see Figure 7.0).

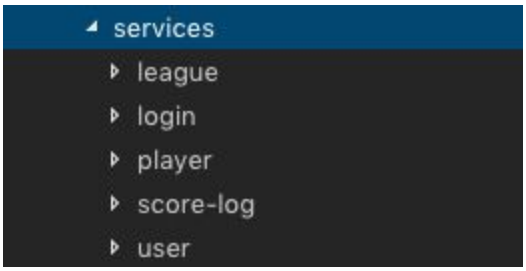


Figure 7.0: Angular services (front-end).

3.1.3.c. Angular Template

The angular template is the known as the final view of the application. This is what the user interacts with and sees. This template is directly attached to one of the angular components and thus, regulated by its component. Changes to the templates always go through their delegated components.

3.1.2. Design Patterns

3.1.2.a. Observer Pattern

Angular uses the observer pattern to implement its HTTP client requests. This allows for flexible and reusable services and components. The observer pattern also makes the data fetch process much simpler and more efficient. The angular service responsible for fetching data returns an observable in the form of a data stream (using async pipe-see Figure 8.1). The stream activates only when there is an attached subscriber or observer to it (making sure unnecessary requests are not made to the back-end). Whenever a component needs data from a particular service API, the component will invoke a method to “subscribe” to the observable returned by the service’s API (see Figure 8.0) . Subscribing or observing makes the whole process of requesting data much simpler because no more callback functions!

```
this.userService.getUser(usr).subscribe(data => {
  this.user = data;
});
```

Figure 8.0: Component subscribes to the observable returned by `getUser()` method.

```
getUser(usr){

  let tempObj = {
    "userid": usr.userid,
  }

  return this.http.post('/api/user', tempObj).
    pipe(map((response: Response) => response.json()));
}
```

Figure 8.1: Service API fetches data and returns a data stream observable ready to be observed.

3.1.2.b. Dependency Injection

Angular uses the dependency injection pattern to inject services into angular components that require them. This allows for applications to be more flexible, efficient, robust, testable and reusable. Dependency injection makes it possible to eliminate, or at least reduce, a components unnecessary dependencies. A component is vulnerable to change in its dependencies. If a dependency changes, the component might have to adapt to these changes. Reducing a components dependencies typically makes it easier to reuse in a different context. The fact that dependencies can be injected and therefore configured externally, increases the reusability of that component. Hence, this is another reason why Angular was the goto design choice for the front-end. See Figure 9.0 as to how dependency injection is used within the application.

```
export class UserComponent implements OnInit {
  user: User;
  id: number;
  inactive: boolean = true;
  editButton: boolean = false;
  submitButton: boolean = true;
  joinLeagueForm: boolean = false;
  joinLeagueButton: boolean = true;

  constructor(private userService: UserService,
      private route: ActivatedRoute,
          private router: Router) { }
```

```
@Injectable({
  providedIn: 'root'
})
export class UserService {

  constructor(private http: Http) { }
```

Figure 9.0: The `UserService` class is imported in constructor of `UserComponent` to perform function declared in `UserService` class.

3.2. Back-End

3.2.1. Components

The back-end of the system consists of several components including the controllers, services, entity classes, and the Dao layer. Each of these components interacts with each other in some form. Mainly, they follow the layered architecture. Figure 10.0 shows the low-level UML diagrams of each of these components and their relationships with other components.

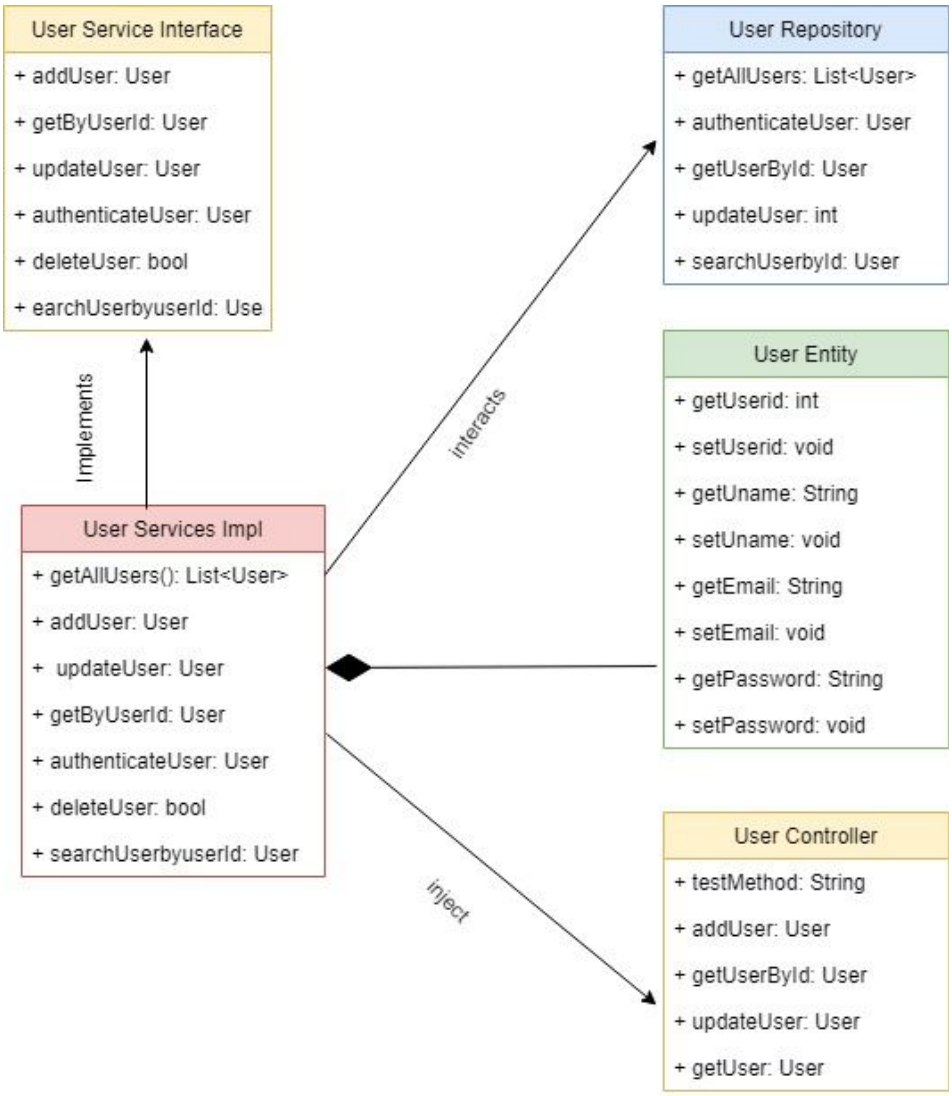


Figure 10.0: This UML diagram depicts the role of the user component within the application. Other components in the application also have a very similar UML structure.

3.2.1.a. Controllers

The controllers are directly connected to the dispatcher servlet which is responsible for dispatching responses to the front-end and also mapping requests to the controllers. The controllers contain services that are auto-wired which is Spring framework's annotation for injecting dependencies. The controllers take request bodies and process them in order to feed the corresponding services that they interact with. For example, `UserController` receives calls from its handler via GET/POST requests from the Angular front-end. The `UserController` delegates further processing to the `UserService`. It then renders the response and sends it off to the front-end.

3.2.1.b. Services

The services are responsible for interacting with the controllers and getting data objects to work with. The main business logic is executed in the this layer. The service is also responsible for passing this data using an entity class (see below) to the DAO layer (see below). The data passed back from the DAO layer is unpacked from its entity object and processed. Later it is passed down to the controller calling the service. For example, in the application, we have the `UserController` that requests the `UserService` whenever any of its methods get invoked by the handler. The service methods are always accessed via interfaces so that they are not accessible directly to the client. `UserService` then packages the data from the controller into an `User` entity class to provide to its DAO counterpart, the `UserRepository`. The `UserRepository` then returns data from the database in the form of the `User` entity class. `UserService` unpacks the data and provides it to the `UserController`.

3.2.1.c. Entity Classes

The entity class acts as a "package" between the back-end and the database. The entity classes are responsible for encapsulating data into objects. These objects are passed into the DAO layer and they eventually end up in the database (the data portion). The data is accessed via getter methods.

3.2.1.d. DAO Layer (Data Access Object)

The DAO layer is the front-line mechanism for interacting with the database. The service accesses this component via a `Repository` call. The DAO layer in the codebase is known as repositories. As mentioned, the entity object encapsulates details of persistence layer and provides CRUD interface for a single entity. The DAO is responsible for handling details of the persistence mechanism (see Figure 11.0).

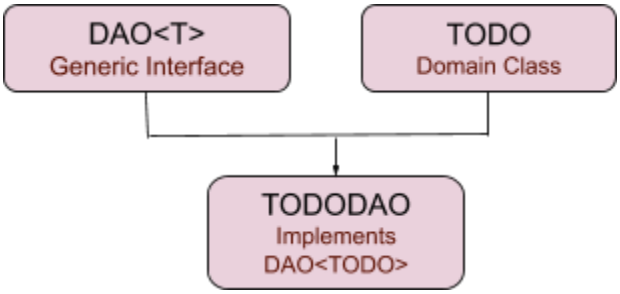


Figure 11.0: High-level diagram of the DAO layer structure.

3.2.2. Design Patterns

3.2.2.a Dependency Injection

The Spring Boot framework is an open source framework which provides infrastructure support for developing enterprise Java applications. By using Spring Boot, developers do not need to care about the infrastructure of the application. They can focus on the business logic of the application rather than on the configuration and infrastructure of the application. All the configuration, infrastructure, either Java-based or XML based are both handled by Spring Boot. Spring Boot uses the dependency injection design pattern (see Figure 12.0).

Dependency injection, allows dependencies to be injected into objects as previously mentioned. In the application, dependency injection is used to decouple objects in case an object needs to be changed, there need not be more changes required. Similar to the Angular dependency injection, Spring provides a similar approach. Dependencies (i.e. services) are autowired using the @autowired annotation instead of being instantiated.

The application uses dependency injection in the following way:

```
@RestController
public class LeageMemberController {

    @Autowired
    LeagueService leagueService;

    @Autowired
    LeagueMemberService leagueMemberService;

    @Autowired
    UserService userService;

    @Autowired
    HockeyPlayerStatsArchiveService hockeyPlayerStatsArchiveService;
```

Figure 12.0: Use of dependency injection in the back-end.

```
@Service("LeagueService")
public class LeagueServiceImpl implements LeagueService{

@Service("UserService")
public class UserServiceImpl implements UserService{
```

Figure 12.1: Use of dependency injection in the back-end.

All microservices use the @Service annotation so that Spring knows that it needs to create an object (an implementation) of that service (see Figure 12.1).

3.3. NHL Fantasy API (Design Patterns)

The application has its own API that collects real data like player statistics from the web to provide critical data involving points. The following patterns were used to implement this API:

3.3.1 Singleton Pattern

The singleton pattern is used to run a one-time python script to load data in the database. It fetches the data from the JSON file and loads the player information and the NHL match schedules in the database (see Figure 13.0, 13.1). The singleton pattern was used to avoid redundant entries of the constant player set into the database each time the application runs.

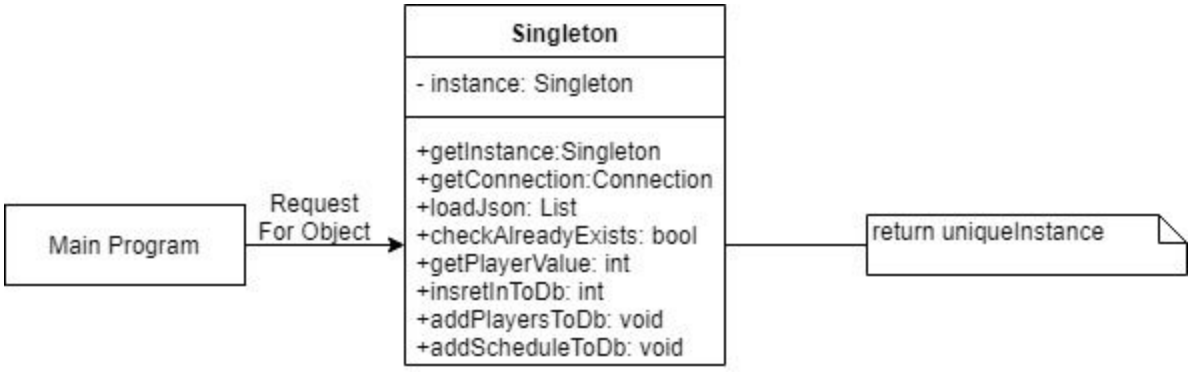


Figure 13.0: UML diagram depicts the singleton instance responsible for adding players only once.

```
class Singleton:
    __instance = None
    @staticmethod
    def getInstance():
        if Singleton.__instance == None:
            Singleton()
        return Singleton.__instance
    def __init__(self):
        if Singleton.__instance != None:
            raise Exception()
        else:
            Singleton.__instance = self
def addPlayersToDb(self,fileName,dbPassword)
def addScheduleToDb(self,fileName,dbPassword)
```

Figure 13.1: Code snippet singleton implementation.

3.3.2 Visitor Pattern

Statistics are fetched on a daily basis based on daily matches. To avoid constantly requesting data from real matches to calculate points and append them to the user score via internal calculations (within the application), the Visitor pattern is used. It allows for flexibility in case of a new point criterion. Whenever the app needs to add or remove a point scoring criterion, the visitor pattern will make the code more easier to modify. Figure 14.0 perfectly illustrates this pattern within the application context. There are three different visitors intended for adding different points based on goals, assists and goalie saves/goals against. The objects or classes that depend on keeping track of these statistics (namely, `LeagueMember` and `HockeyPlayerStats`) accept these three visitors. Each visitor visits each `LeagueMember` and `HockeyPlayerStats` objects and adds points points to them. These points are then reflected (via the application business logic) into the users' scores.

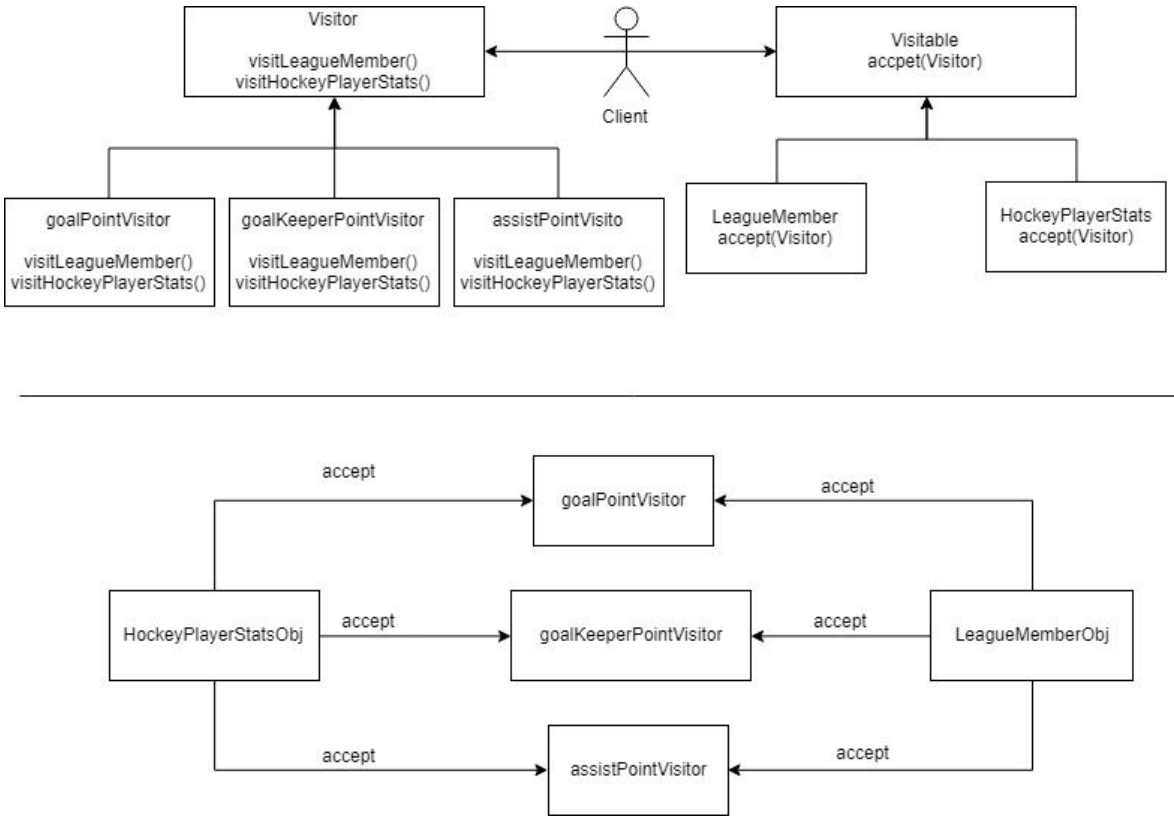


Figure 14.0: Low-level diagram depicting the usage of the visitor pattern to award points to users based on real statistics.

3.4. Database Design

Figure 15.0 shows the database schema of the application. It consists of seven tables: `User`, `League`, `HockeyPlayer`, `MatchSchedule`, `LeagueMember`, `MemberTeam`, `MemberTeamArchive`, `HockeyPlayerStatArchive`. The tables are mapped together with different mappings. The `User` table stores the user data with *userid* as a primary key. The `League` table contains the leagues created by the users with *leagueid* as primary key and *creatorid* as foreign key with a many-to-one relationship. The `LeagueMember` table has data about the created and joined leagues. The `MatchSchedule` table has the information about upcoming matches. The `HockeyPlayer` table contains the player data and is further mapped to `MemberTeam`, `MemberTeamArchive` and `HockeyPlayerStatArchive` tables.

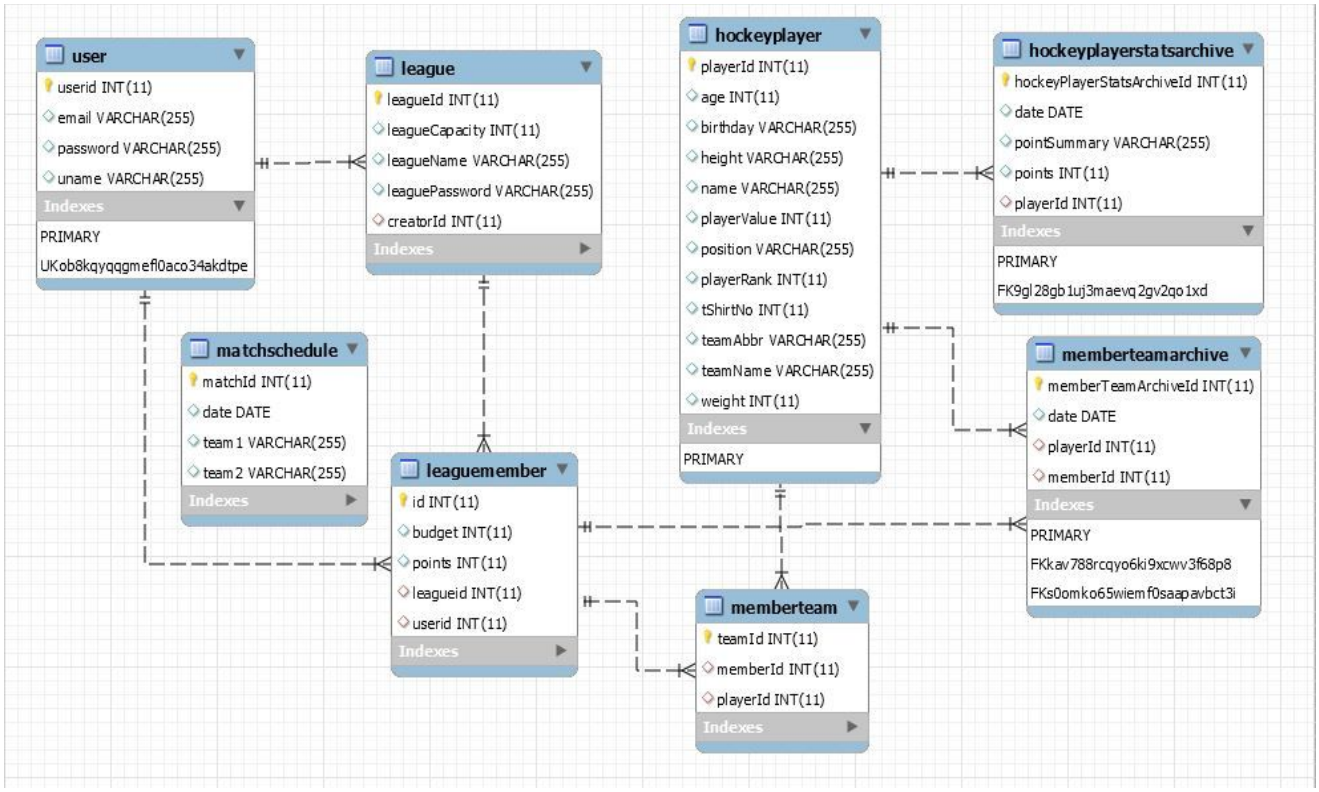
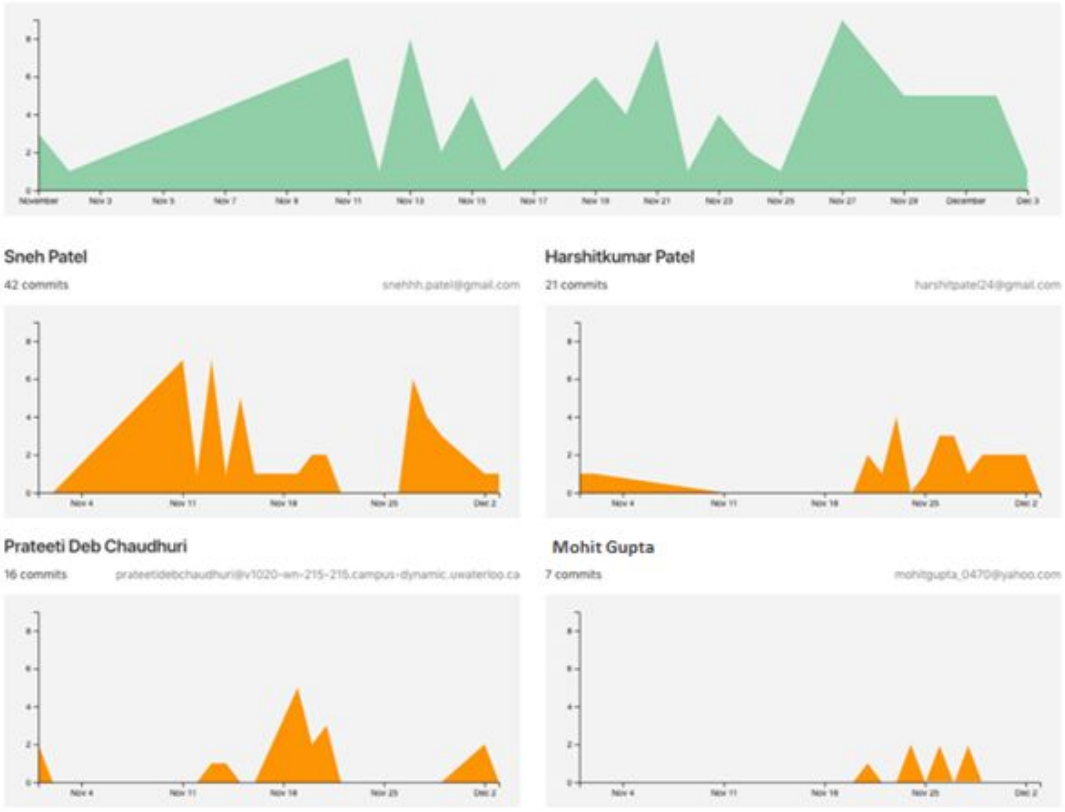


Figure 15.0: Database schema.

Participation Journal



Total commits to the Git repository. Note, this is just a rough overview of all the commits made by each group member (group members may have also collaborated with other group members for a particular feature). See the table below for the role of each group member on the parts of the application.

Part of System	**Group Members that Contributed**
Front-end	Sneh: welcome, login, register, create league, select players, user dashboard, league dashboard, player displays, points summary, **all** the services Mohit: session, user dashboard, league dashboard, header component. Prateeti: session
Back-end	Harshit: **all** player related controllers/services, **all** match related controllers/services Prateeti: encryption related services, entity classes, DAO layers, LeagueMember controller/service Mohit: show created leagues and joined leagues services Sneh: user controller/service, league controller/service
NHL-API	Harshit
UI	Harshit, Prateeti, Priya, Sneh, Mohit
Documents	All group members participated in this.