# Team 8 ATAM

## Business Drivers

### Business Context

The AskMe.com discussion forum intends to provide its users the capability to start threads and sub-threads of conversations, upvote/downvote a conversation, view profile details, subscribe/create channels etc. The forum is to be accessible only to authenticated users.

### High-Level functional requirement

* Users from across countries should be able to use this forum for discussion purposes.
* The forum mandates user authentication before accessing any listed functionality.
  + Login

Allow users to log in using username and password.

* + Registration
    - Users will enter the desired username and password along with optional fields - first name and last name.
    - Specify if admin or not.
    - The system should notify the user if the username is available or not.
    - All the registering users will by default follow a system channel NewsUpdates.
* The users can create/edit/delete/subscribe to a channel (Future sprints)
* The users can create/edit/delete a post, comment or vote another thread or rate the same (depending upon his role).
  + View Post
    - Once the user clicks on the post (from the feed/channel page), the user should see the post details like - author, post creation date, channel name, post content etc.
    - View comments on the posts by users and add more comments.
  + Post Creation  
    Allow the user to create a post in the default channel created.   
    For post creation, the user will provide
    - the channel where the post is being posted
    - Content of the text
    - Hashtags - If the hashtag is not present in the system, the system should create the hashtag and store it
    - Profanity filtering - yes/no
* Feed should be visible to all users - logged in/anonymous. This will show latest post and top posts (as filtered by the algorithm/specified by the moderators)
  + Home screen  
    On the landing screen, the user should see the posts from the channel the user is subscribed to. For this sprint, there is only one channel.
* Any logged in user should be able to view his profile details and site related activities.
  + View Profile
    - Allow the user to access the user’s profile.
    - The user should be able to see - user’s name, last name, active since (date created), posts a user has posted, channels user is following.
    - Future sprints
      * View owned channels, interests, followers etc.
      * Admin can allocate roles and change user status.
      * Separate GUI exists for administrators and moderators
      * Moderators have the additional capabilities of deleting (post/comment), marking (useful/on fire) any post in the allotted topic.
* Users should have the ability to search for keywords on the website
  + Search  
    The user should be able to search through the posts and hashtags.

### High-Level quality attributes

* The forum requires robust handling of the concurrent database read and write.
* Real-time updates on the thread.
* The page load and refresh times should be low enough to go unnoticed
* The application and infrastructure should be robust and reliable enough to ensure very high availability.
* The pilot version of the forum should have scope to cater for further extensions.
* The application should have the support for frequent releases.
* It should be maintainable with zero to none annual downtime.

### Critical Requirements (Ranked 1 - Quality attributes most central to the system’s success)

* Time to market
* Performance
* Availability
* Security
* Modularity
* Extensibility
* Flexibility

### Architectural Drivers (Ranked 2 - Quality attributes that shape the architecture)

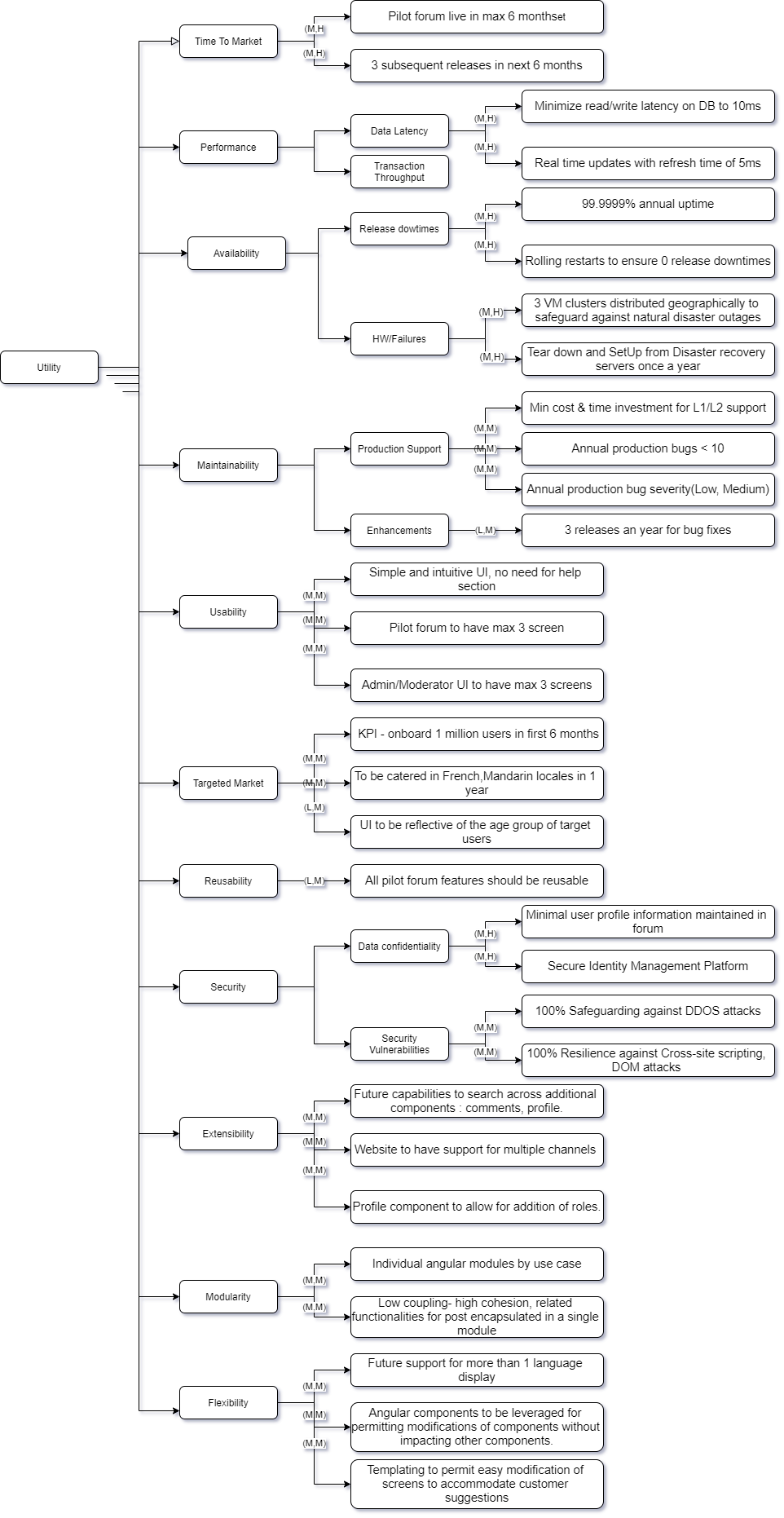
* Maintainability
* Usability
* Targeted market
* Reusability

## 

## Constraints

* Technical constraints
  + Platform: Support all browsers on the client side.
* Other systems with which the system must interact
  + No.
* Architectural styles
  + Client-server and MVC.

## Utility tree



**Scenario:**

**Use case scenario**

1. Remote user land on the feed page during peak period and page loads within 5 seconds.
2. Users request for all sub threads and should receive it within 2 seconds
3. Search threads, subthreads, posts, profiles within 1 second

**Growth scenario**

1. Add a new data server to reduce latency in scenario 1 to 2.5 seconds within 1 person-week.
2. Add pagination beyond a certain count of posts/comments on the page.
3. Add auto language translation feature to support more than one language without affecting system performance
4. Add new data server(reserved instances) on public CPS to avoid the load on existing private cloud instances
5. Add a load balancer to reroute requests to the reserved cloud as well as a private cloud in a scheduled pattern
6. Migrate data to distribute databases without affecting existing data or functionality of the system

**Exploratory scenario**

1. Half of the servers go down during normal operation without affecting overall system availability
2. Increase the load on the system, and check the overall system availability.
3. Systems robustness to DDOS attacks.
4. Assess the response time from the other regions and validate with the provided response time.
5. Check systems resilience against various security vulnerabilities such as SQL Injection, Cross-site scripting attack, DOM attack

# Design Patterns Implemented

1. Observer

2. Memento

3. Facade

4. Mediator

5. Command Pattern

6. Filter

7. Flyweight and Factory

8. Builder

9. Template

10. Singleton

11. Factory

12. Visitor

13. State

14. Strategy

15. Proxy

16. Adapter

17. Prototype

18. Decorator

19. Iterator

20. Interpreter

# 

# 1. Observer

**Justification**

**Problem:**

We have to display the functionalities of the application when the user logs in or logs out. To facilitate that, reload the webpage again in order to reflect available functionality in the navigation bar.

**Solution:**

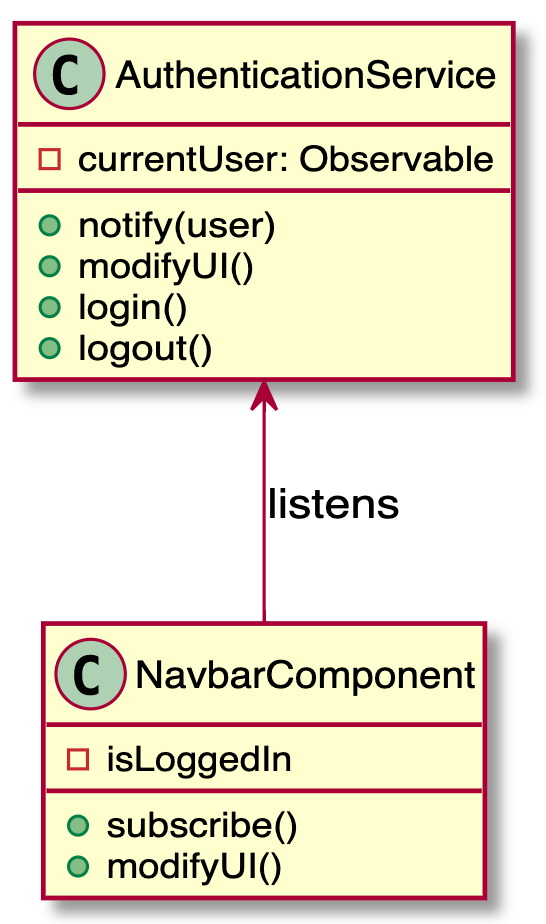
Instead of a refreshing mechanism, we can make use of the Observer pattern. Whenever the user logs in or logs out it generates an event and notifies its subscribers. Upon getting the event from the notifier the Navigation Bar Component changes its visible functionalities.

**Implementation**

The code is implemented in the frontend. Whenever the user logs in or logs out the event is triggered to all its subscribers.

We have the *AuthenticationService* class which has an *Observable* object. *NavbarComponent* class subscribes to the *Observable object* from *AuthenticationService* class. When the user logs in or logs out, it notifies the *NavbarComponent* and it changes the visible functionalities accordingly in the UI without refreshing the page.

**Class Diagram**



# 2. Memento

**Justification**

**Problem:**

We need to give the user an option to reset the signup form or to save the signup form so that he can undo the last saved state.

**Solution:**

Here, we can make use of the Memento pattern. Fist state of Signup Form is saved so that the user can reset the form. With it, we can also provide options for the user to save the form.

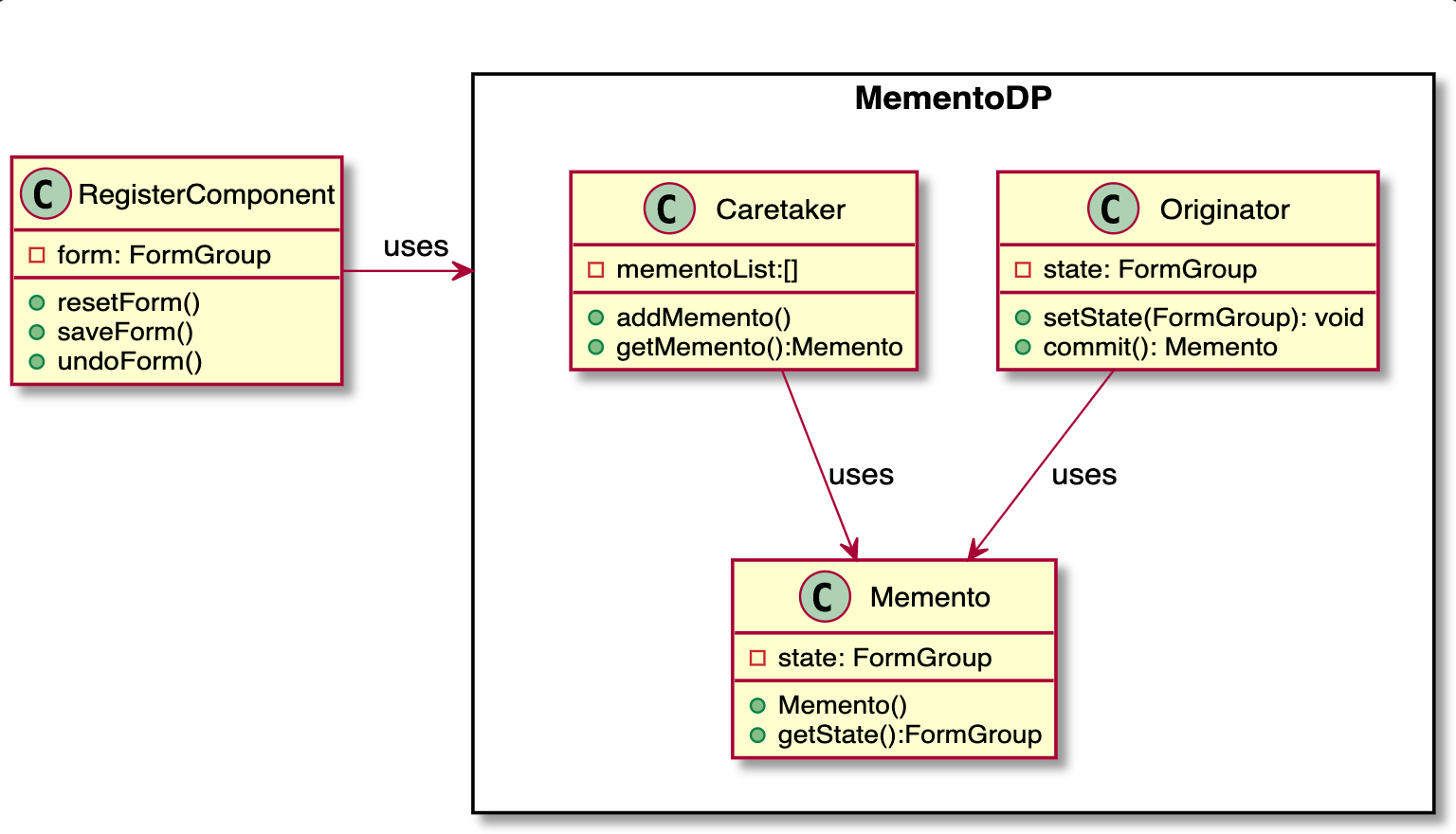
**Implementation**

We have created 3 classes

1. *Memento* -> stores the state of the signup form
2. *Originator* -> Creates the memento object
3. *Caretaker* -> Keeps the track of all the Mementos created

The initial state of the form with some default values is stored using Memento Design Pattern. Whenever the user clicks on the reset button the initial state is restored.

**Class Diagram**



# 3. Facade

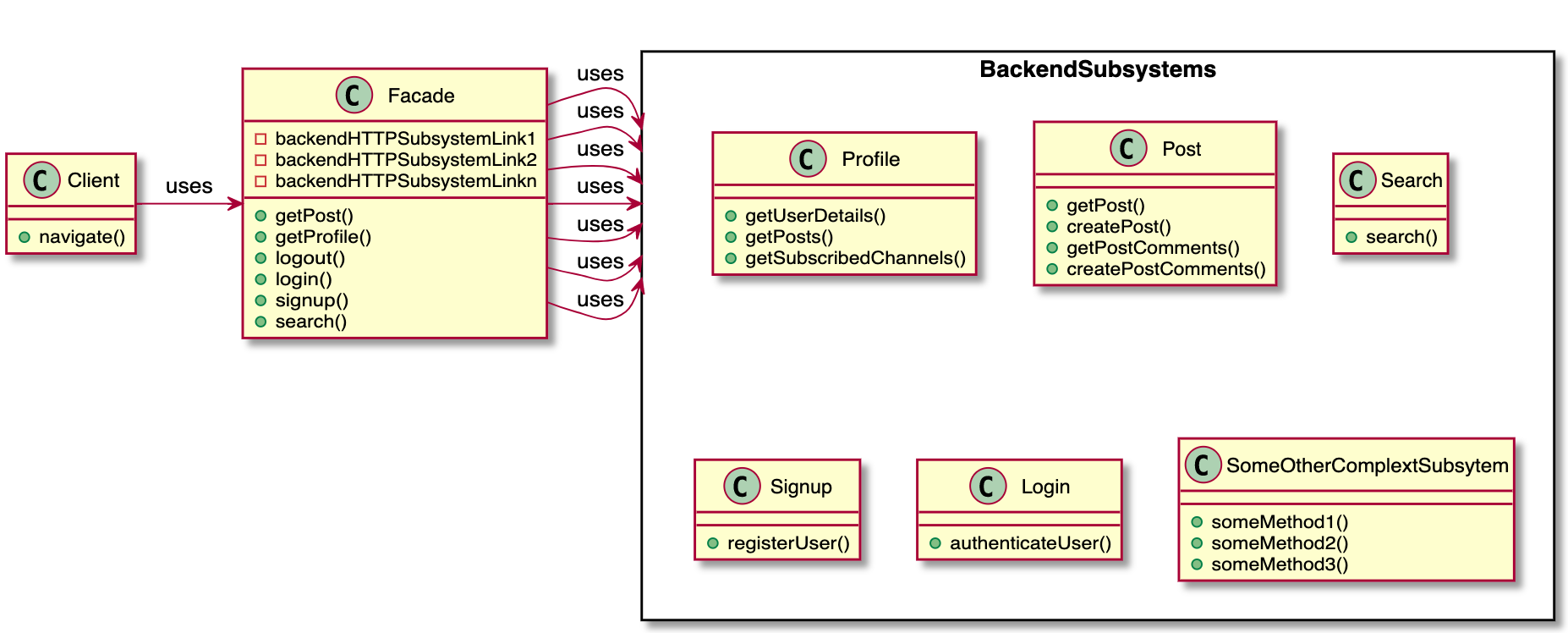
**Justification**

We use the Facade pattern when we need to have a limited but straightforward interface to a complex subsystem. Here, we want the users to use the application without knowing how the internal subsystems communicate and bring out the necessary results in the UI. To achieve this we are using Navigation Bar in the UI. Using the navigation bar, the user can access the functionalities implemented by the system.

**Implementation**

The design pattern is implemented in the frontend, it collaborates the complex subsystems in the backend to facilitate the expected behavior. Using the navigation bar, the user can access the functionalities implemented by the system such as view profile, create post, search, login, logout, etc.

**Class Diagram**



# 

# 4. Mediator

**Justification**

**Problem:**

We have to display a post created by different users to all users in the channel.

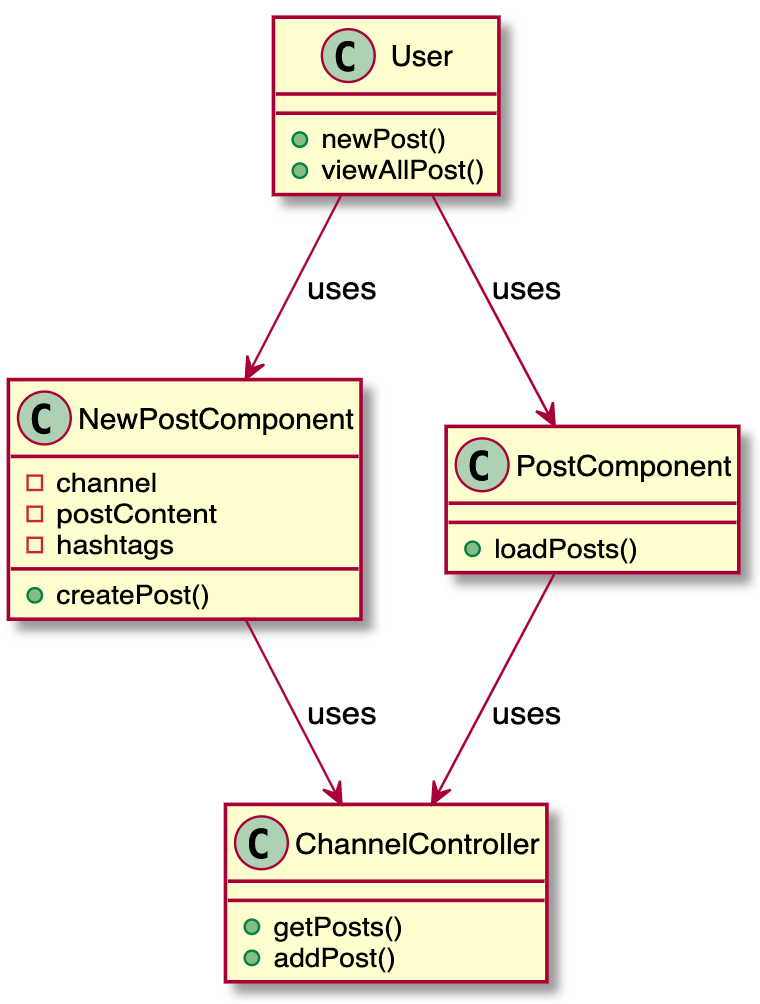
**Solution:**

Here, we can make use of the mediator pattern. Multiple users can create posts on a channel and it is the responsibility of the channel to show the posts to all users. Instead of communicating posts to all the users, user will only communicate with the channel and channel then will display the posts to all the users. As a result, the **channel** will act as a **mediator** between the users.

**Implementation**

The client will create a post in the channel using the interface of *ChannelController* for a Channel with method *addPost*. It will take input as the username, post contents, channel where the post will be posted. For viewing the posts the client uses interface of *ChannelController* specifically the method *getPosts* to get all the posts added by all users.

**Class Diagram**



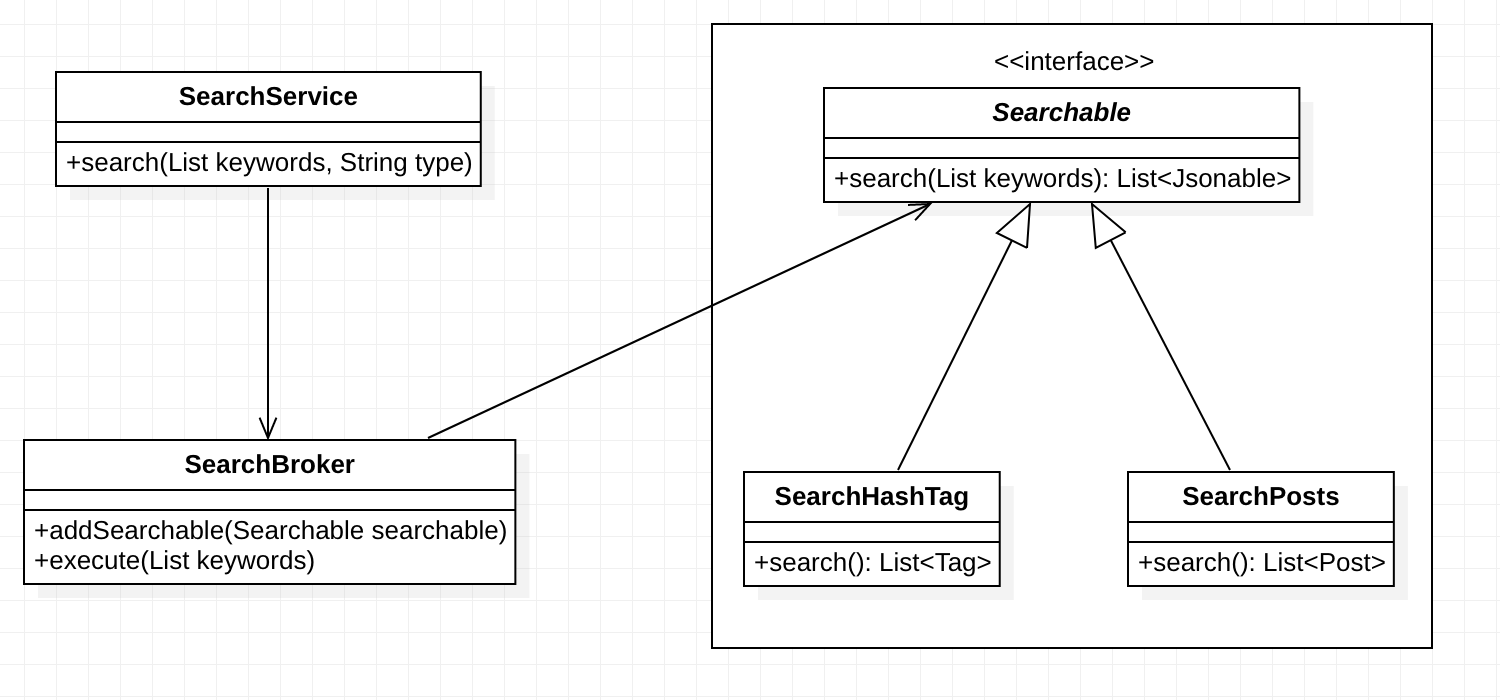
**Alternative example (Mediator):** all the controller classes act as a mediator between the clients and the business layer. All the calls to the business layer, let’s say - models and services, have to go through the Controller.

# 5. Command

## Justification

In the forum, we support three kinds of searches, including searching by posts, or by hashtags, or both. We use the command pattern to implement this requirement. Each search request serves as a command containing keywords and search type from the client side. Then we add SearchHashTag or SearchPosts or both to the SearchBroker based on the search type that the user expects. When the searchables are resolved and added, SearchBroker will execute the search in sequence and return the result.

## Class Diagram

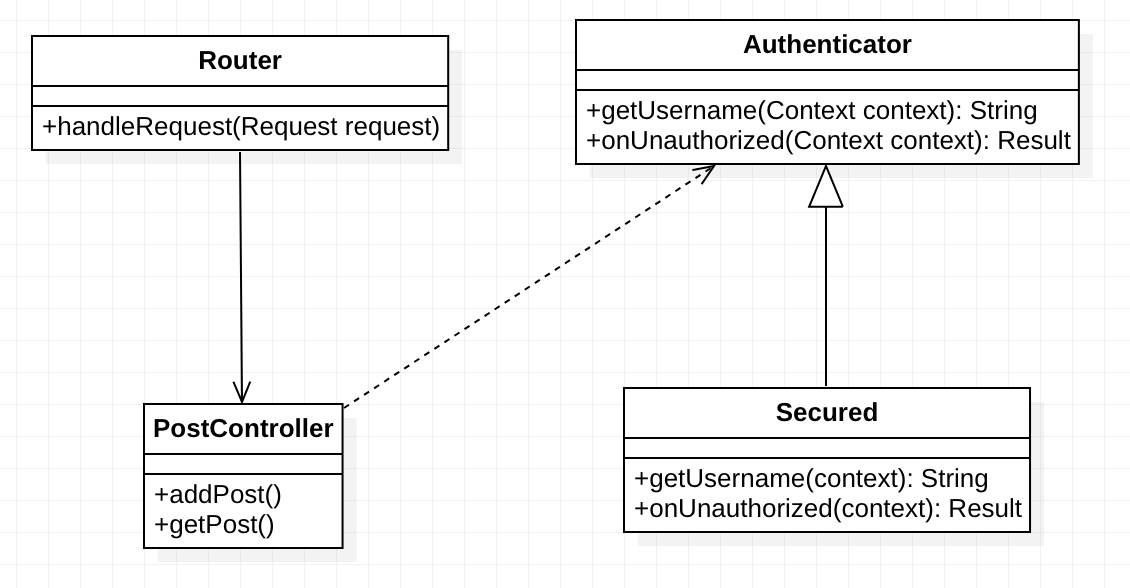


# 6. Filter and Pipeline

## Justification

We use the filter pattern to filter out unauthorized requests and only pass those valid requests to corresponding controllers. To be specific, we build a Secured middleware, which inherits from Play’s Authenticator class, as a filter using if a valid user id appears in the request header as the criteria. Basically, every incoming request to the server will go through the Secured middleware/filter first, and only those authenticated requests will be pipelined into controllers.

## Class Diagram

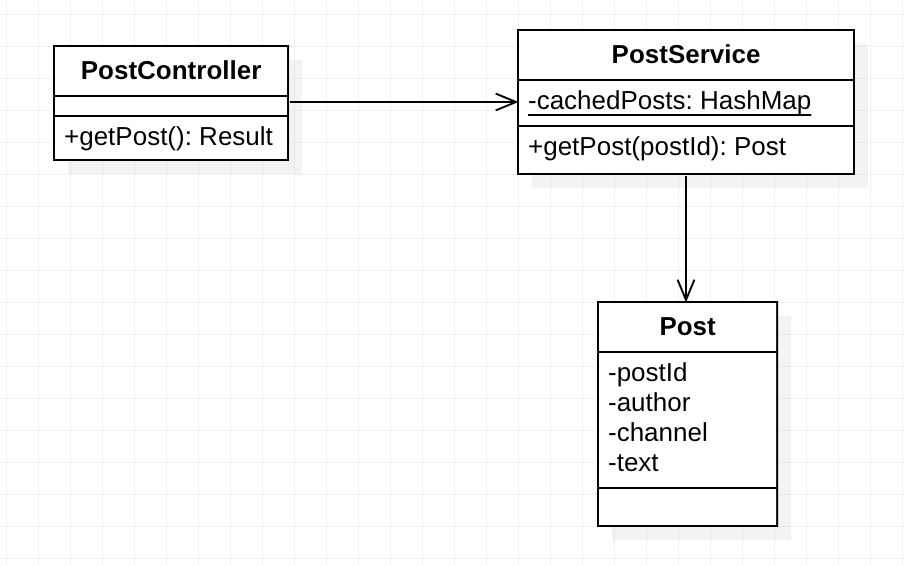


# 7. Flyweight and Factory

## Justification

To reduce the IO footprint of database access, we use the flyweight pattern to cache posts in memory. To be specific, PostService caches posts in a HashMap keyed with their Id. When retrieving a post, we first check if it is cached in the hash map, otherwise we will fire a query in the database. Alongside with Flyweight, we also use the factory pattern.

## Class Diagram

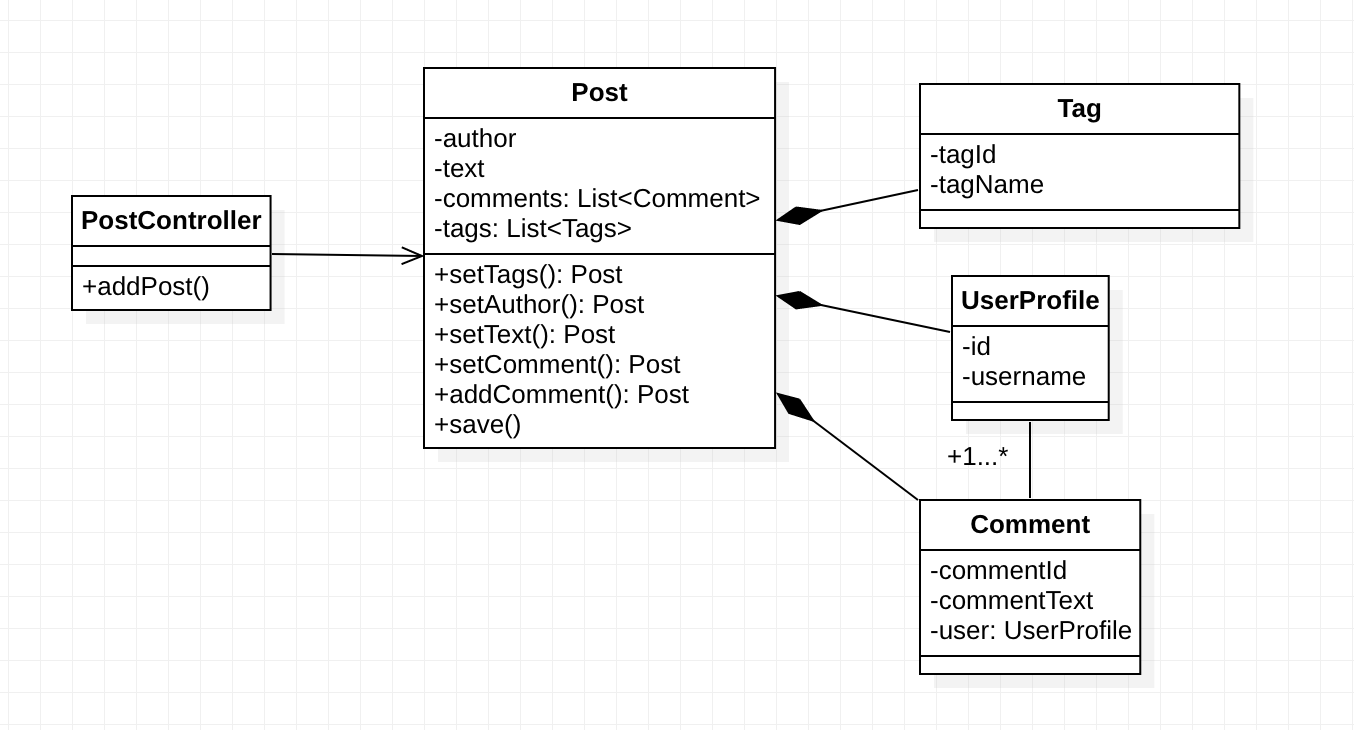


# 8. Builder

## Justification

When creating a post on the server side, we need to set its text content, author info, tags, and comments, and save it to the database in the end. In this case, it is a perfect fit for the builder pattern. Also, we make all the setters in the Post model return the identity so that we can chain the operations until save.

## Class Diagram



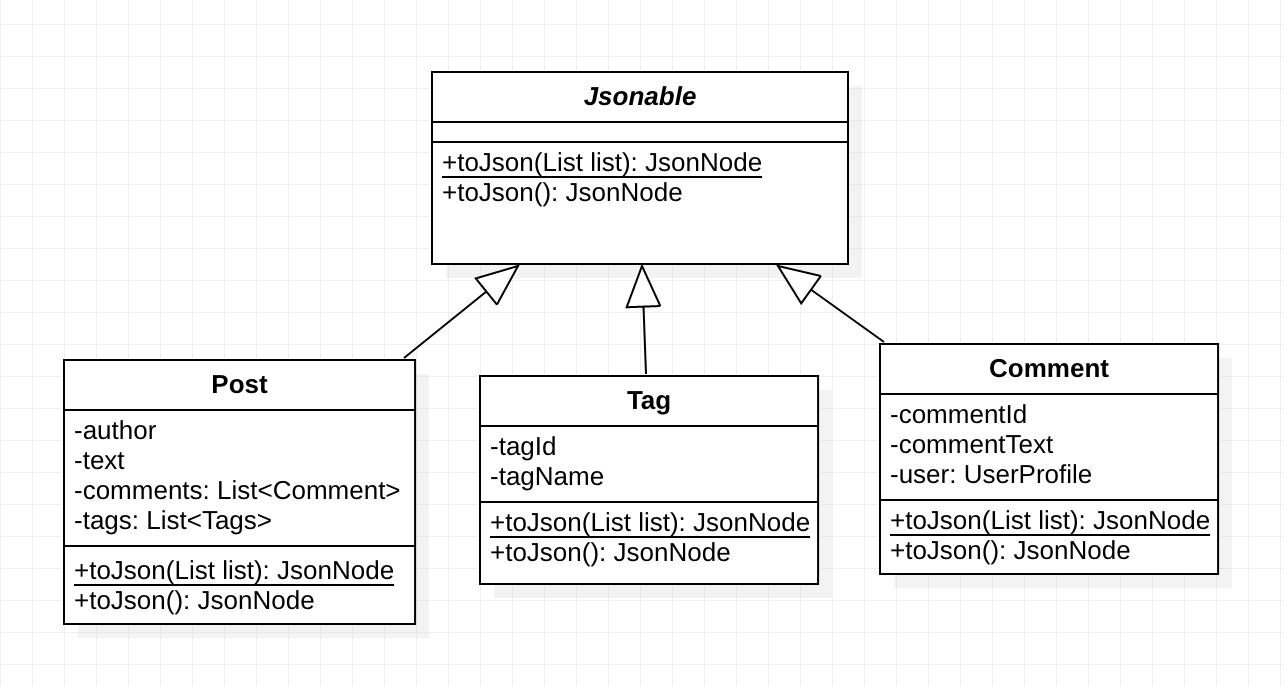
# 9. Template

## Justification

On the backend, models should be serialized into JSON format to send as responses to the frontend. This is a common behavior across the entire backend. So we choose the Template Pattern to encapsulate such a behavior into the Jsonable interface.

The interface defines a toJson method, which converts the instance into JsonNode and should be implemented by the concrete classes, and a toJson static method, which converts a list of instances into JsonNode. Currently, Post, Tag, and Comment models all conform to this interface.

## Class Diagram

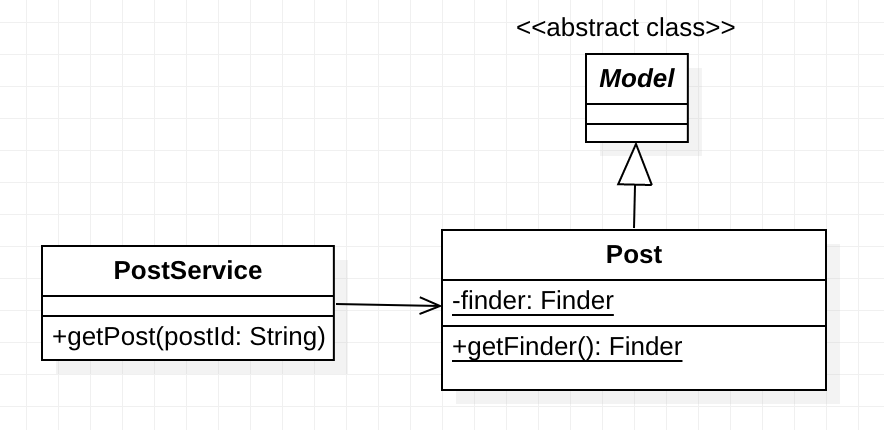


# 10. Singleton

## Justification

Each model in the backend has a static Finder instance to do queries in the database through Ebean. We make these Finder instances singleton in each model. For example, in the Post model, we expose a getFinder method to return the private static finder instance. In the PostService class, when getting a post by its id, it will invoke the getFinder method to build and execute a query.

## Class Diagram



# 11. Factory

## Problem:

While creating the user, the client shouldn’t need to know which class to call in order to create an admin user or a regular user. Also, if a requirement arises to create a third type of user, no code changes should be required at the client side.

## Solution:

We implemented the factory design pattern which allows the subclass to decide the type of object to create. Client doesn’t need to know about it. Also, for adding another type of user, just userFactory code needs to modified.

## Implementation:

We have implemented an interface called User. Admin class and Regular user class inherits it. We have created a UserFactory class that decides which type of user needs to be created at runtime. Whenever a request is received by SignupController(client) to create a new User, it asks the SignupService to create a user. SignupService calls the UserFactory to get the user object depending on the type requested by the controller.

## Class diagram:

# 

Sprint 2 Design Patterns

# 1. Factory

## Problem:

While creating a user object, the client shouldn’t need to know which class to call in order to create an admin user or a regular user. Also, if a requirement arises to create a third type of user, no code changes should be required at the client side.

## Solution:

We implemented the factory design pattern which allow the subclass to decide the type of object to create. Client doesn’t need to know about it. Also, for adding another type of user, just userFactory code needs to be modified.

## Implementation:

We have implemented an interface called User. Admin class and Regular user class inherits it. We have created a UserFactory class that decides which type of user needs to be created at runtime. Whenever a request is received by SignupController(client) to create a new User, it asks the SignupService to create a user. SignupService calls the UserFactory to get the user object depending on the type requested by the controller.

## Class diagram:

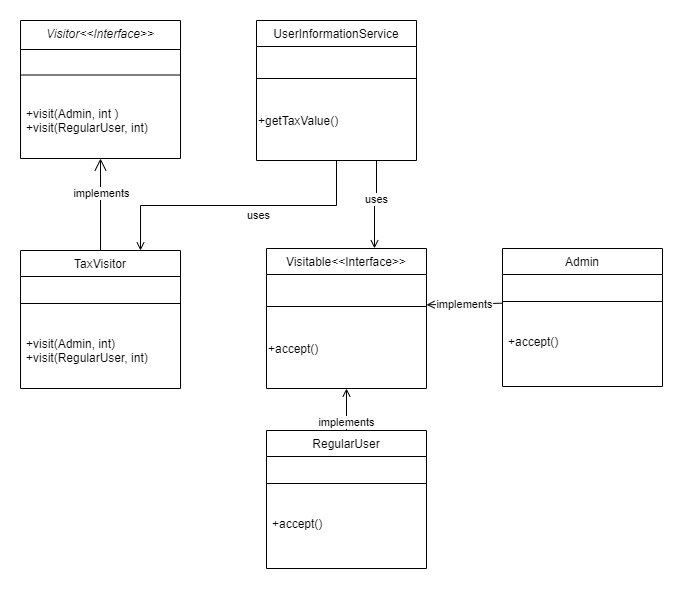
# 

# 2. Visitor pattern

**Justification**

The visitor pattern is used when similar action(but different algo) is required for different types of objects. In our current implementation of the payment system, action was to calculate tax on seperate objects i.e the tax calculation algorithm varies as the object varies. Instead of adding tax calculation methods in these objects we made use of visitor pattern which allow us to calculate tax once the object accepts the visitor.

**Class Diagram**

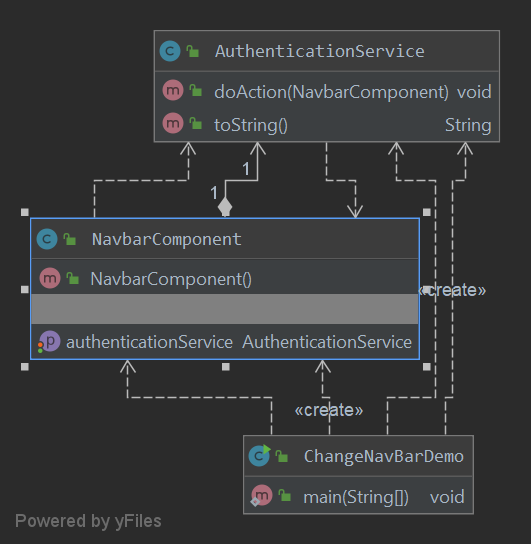


# 3. State pattern

**Justification**

The State of the currentUser authentication maintained in the AuthenticationService whereas the context object NavbarComponent keeps a reference to the AuthenticationService object. The behaviour of the Context object (NavbarComponent) changes as per the change in the logged in user state in the AuthenticationService i.e the Navigation Bar will change depending upon the fact that user is logged in or not.

**Class Diagram**

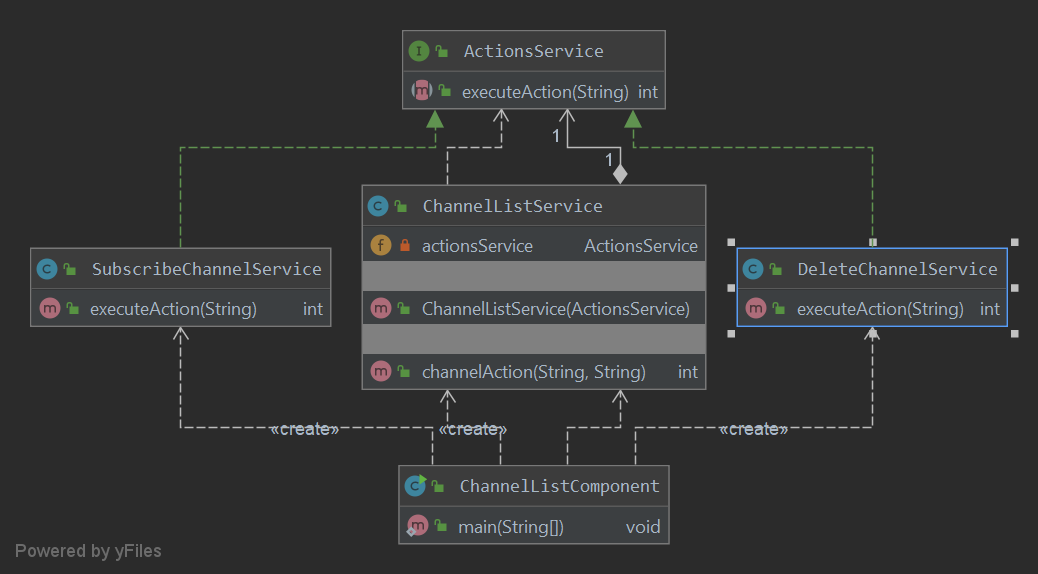


# 4. Strategy pattern

**Justification**

The Channel page allows the user to select either subscribe or delete channel options. Based upon the user selection action, the SubscribeChannelSerice or the DeleteChannelService is called appropriately.

**Class Diagram**

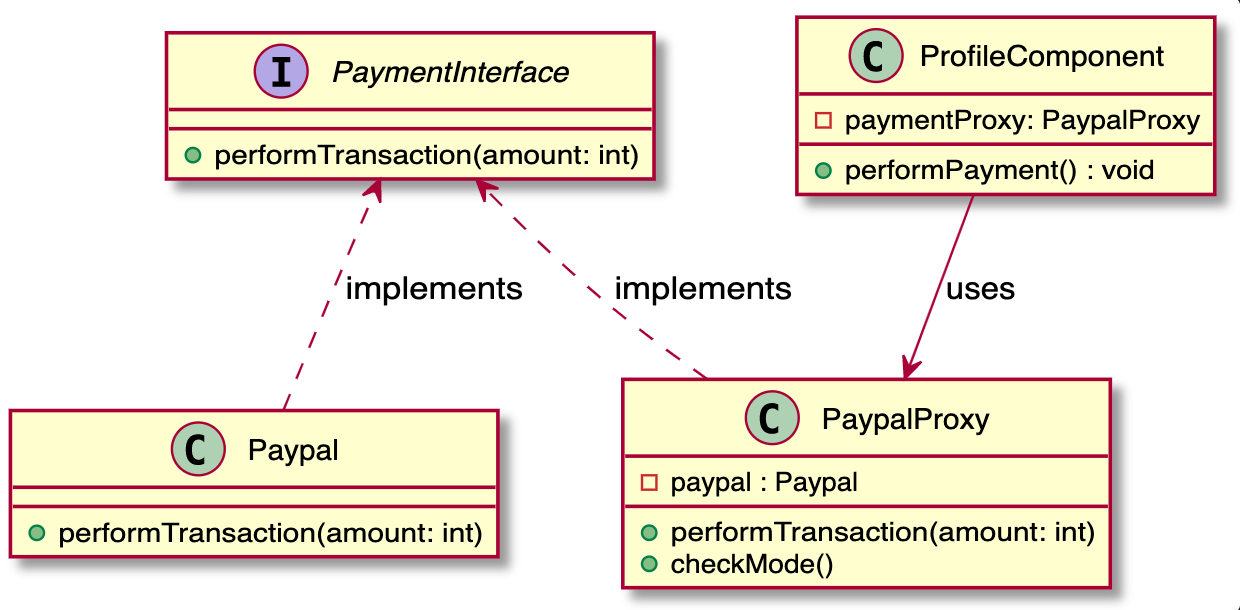


# 5. Proxy Design Pattern

**Justification:**

Proxy pattern perfectly fits in the payment processing. We create one more handle for the PayPal that does the mode checking as well, if the mode is development, the payment is not done but if it is in production mode, payment is delegated to Paypal service through Proxy.

**Class Diagram**

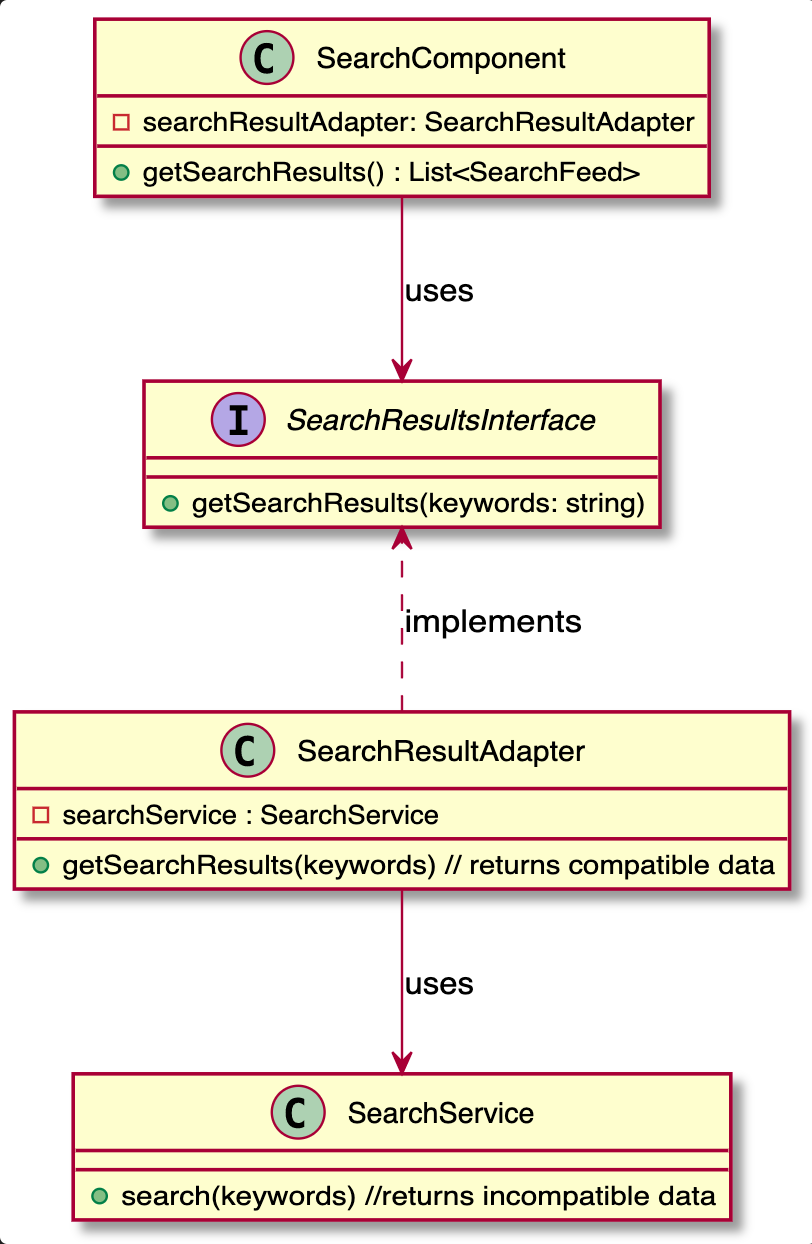


# 6. Adapter Pattern

**Justification:**

We get data from the backend for the user’s keyword search. API call is made and the data is received in JSON format. In the Json data, we have date strings, but the frontend expects dates in the Date object format. So to convert the incompatible interface of SearchService’s search method. We are making use of the adapter pattern. We use the SearchResultsAdapter class to use the legacy code and then convert the data in compatible format as expected by frontend

**Class Diagram:**

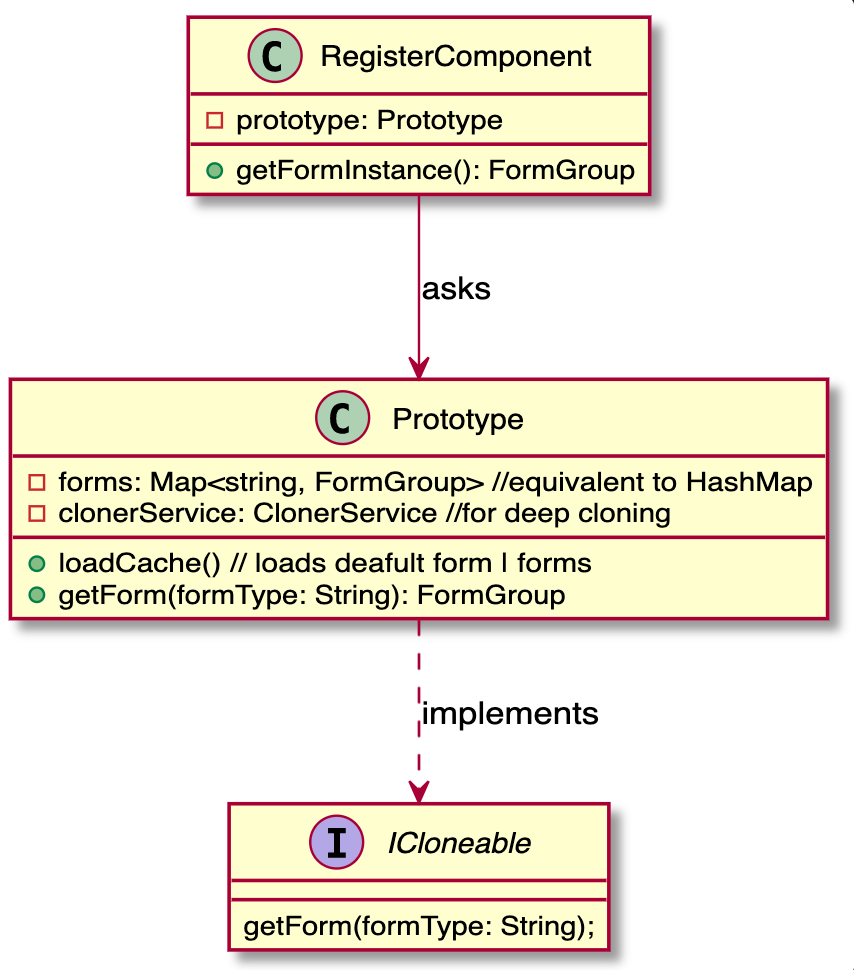


# 7. Prototype Design Pattern

**Justification:**

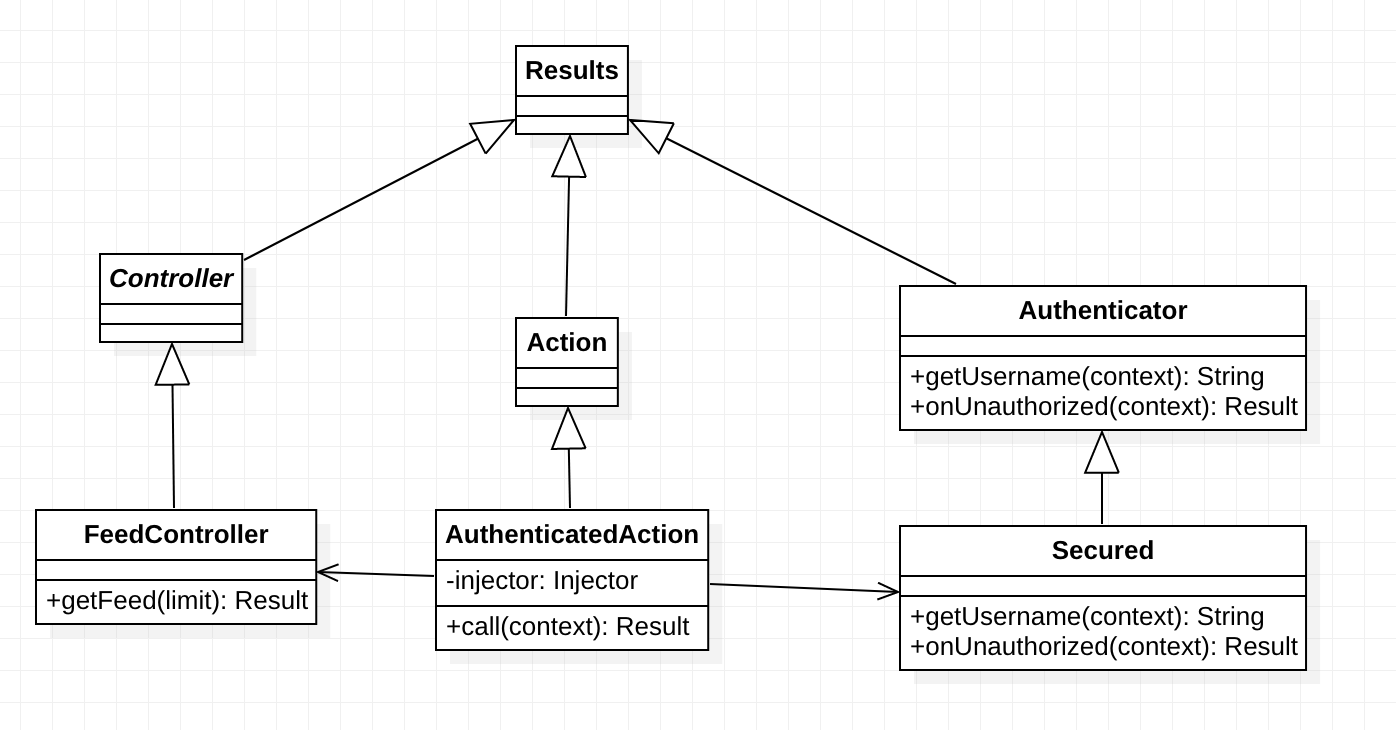
Consider the registration form creation as a costly process. We can avoid the process of creation of forms from scratch by simply cloning the created form whenever is required. we can create different types of forms and store it in Map and return those forms stored in map whenever queried.

**Class Diagram:**



# 8. Decorator Pattern

**Class Diagram**

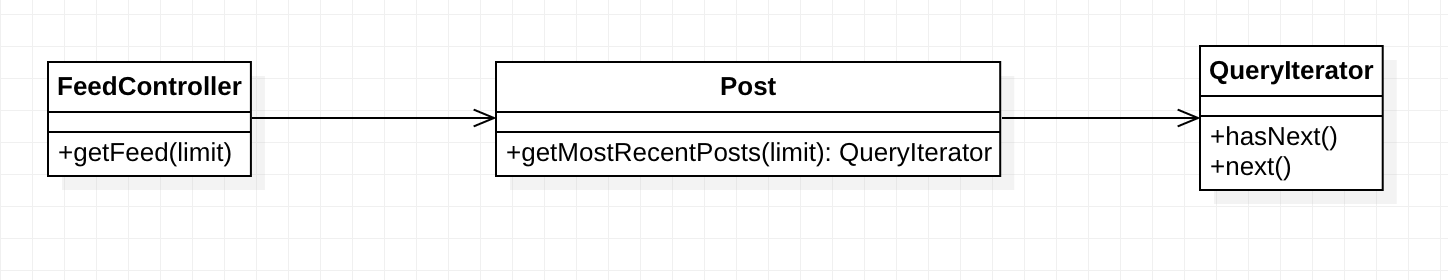


**Justification:**

Decorator pattern is used to enhance the original function without changing its code. Many APIs in the system require that the user who initiates the request is already logged in. Here we use the decorator pattern to enhance the behavior of each controller at runtime by injecting the Secured class.

# 9. Iterator Pattern

**Class Diagram**



**Justification**

Iterator pattern is used to sequentially access the elements of a collection. In the Post model, we expose the getMostRecentPosts(limit) API that returns a QueryIterator which is a built-in iterator class in Play and Ebean. When the FeedController tries to retrieve the most recent posts, it will loop through the iterator and visit elements until reaching the limit.

# 10. Interpreter Pattern

**Justification**

Interpreter pattern is used to define a grammatical representation for a language and provides an interpreter to deal with this grammar. In our scenario we are using it to extract hashtags from a given string. PostService(Client) invokes an object of expression to interpret a given string. If the string has ‘#’ then it calls HashTagsInterpreterContext(engine) to parse the string and get tags.

**Class Diagram**

