CS319

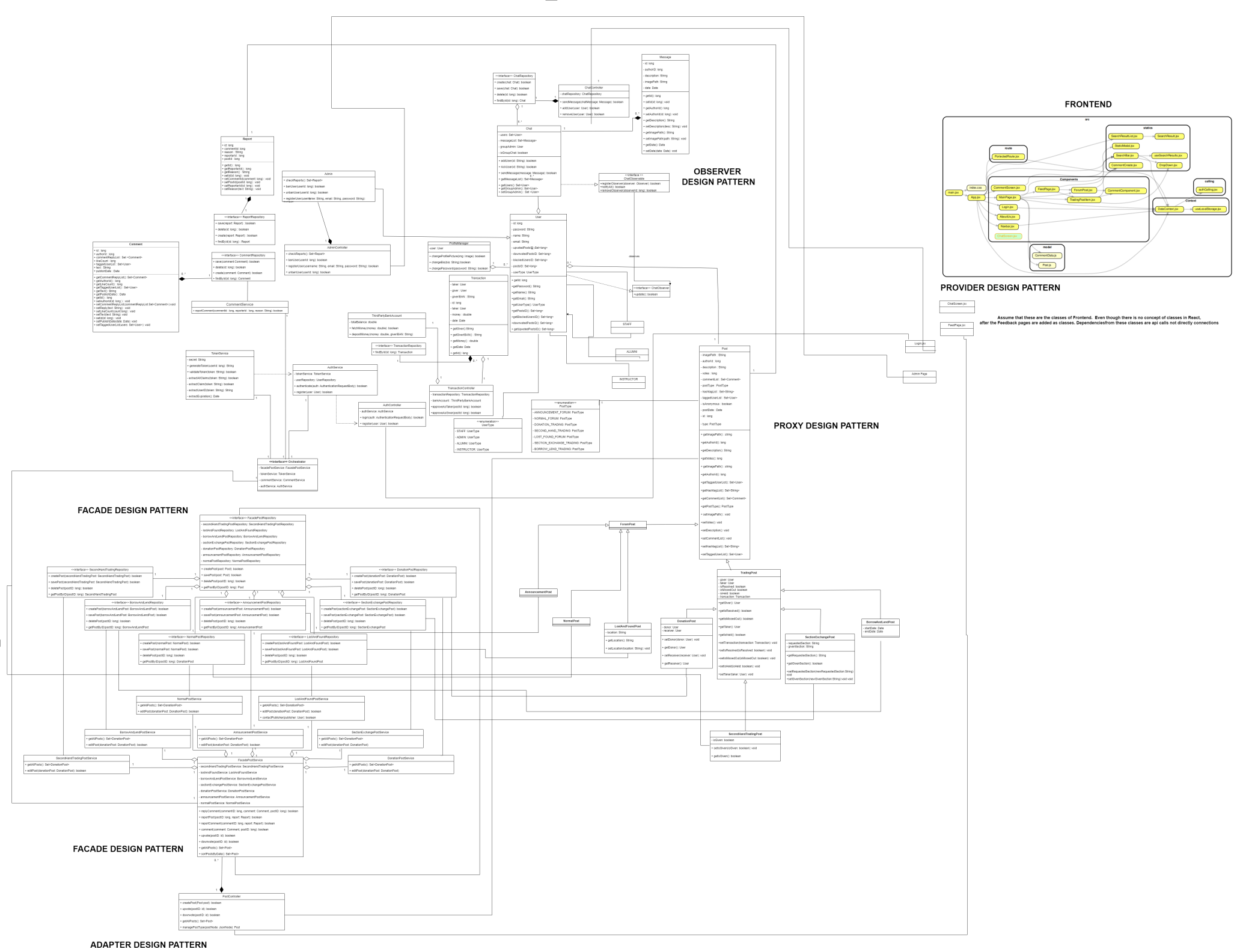
Deliverable 5

BILSYNC

Team 8

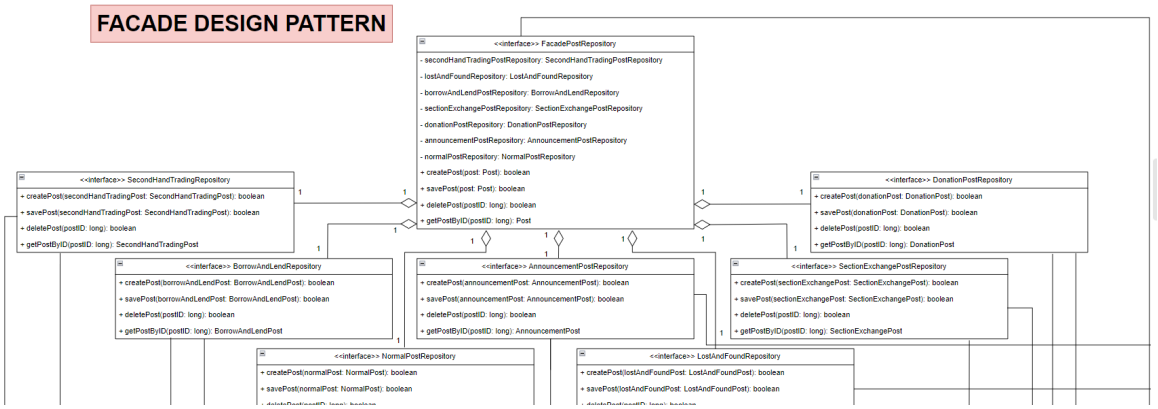
Section 1

Tuna Saygın - 22102566
Hüseyin Burhan Tabak - 22102516
Ahmet Tarık Uçur - 22102946
Işıl Özgü - 22102276
Kanan Zeynalov - 22101007


OBSERVER DESIGN PATTERN
PROXY DESIGN PATTERN
FACADE DESIGN PATTERN
FACADE DESIGN PATTERN
ADAPTER DESIGN PATTERN
FRONTEND
PROVIDER DESIGN PATTERN
Assume that these are the classes of Frontend. Even though there is no concept of classes in React, after the Feedback pages are added as classes. Dependenciesfrom these classes are api calls not directly connections

DESIGN PATTERNS USED IN CLASS DIAGRAM

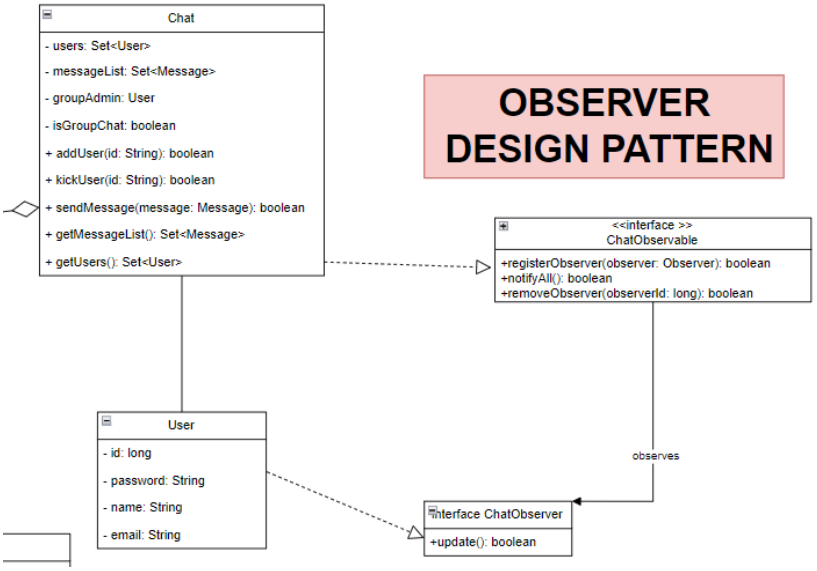
Facade Design Pattern



For implementing the Facade design pattern, we defined a high-level interface FacadePostRepository, FacadePostService, and Orchestrator. The FacadePostRepository and FacadePostService are utilized to define common functions that each Service or Repository for a post type should have. In addition, the Orchestrator interface acts as a centralized controller that gets called to check and control the workflow of the program as it controls which functions get called according to the return values of other functions.

Since interacting with our website consists of many consecutive functionalities, managing these interactions requires a deep understanding of each subsystem and the functions utilized by that subsystem each teammate needs to know about that subsystem. Considering the fact that we have limited time and resources we had to utilize some methodologies which would allow us to abstractify these subsystems to make them easier to use and follow the principle of least knowledge. This is why we decided to utilize a Facade design pattern that acts as a centralized controller for the “Repository” and “Service” types of classes while defining a higher-level interface.

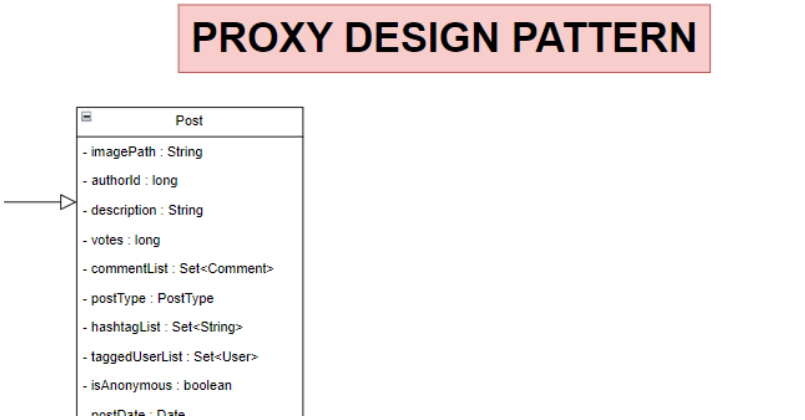
Observer Design Pattern



To manage the chat subsystem, we utilized classes and interfaces such as ChatObservable, ChatObserver, and the concrete classes that implement these methods, Chat and User, respectively. This way, the interfaces defined common functions that every implementer must-have for the chatting system to work properly. What we mean by this is whenever a message is sent, every participant of the chat should be updated, and it should be possible to become a participator in a chat and leave a chat by invoking the “subscribe” and “unsubscribe” functions, respectively.

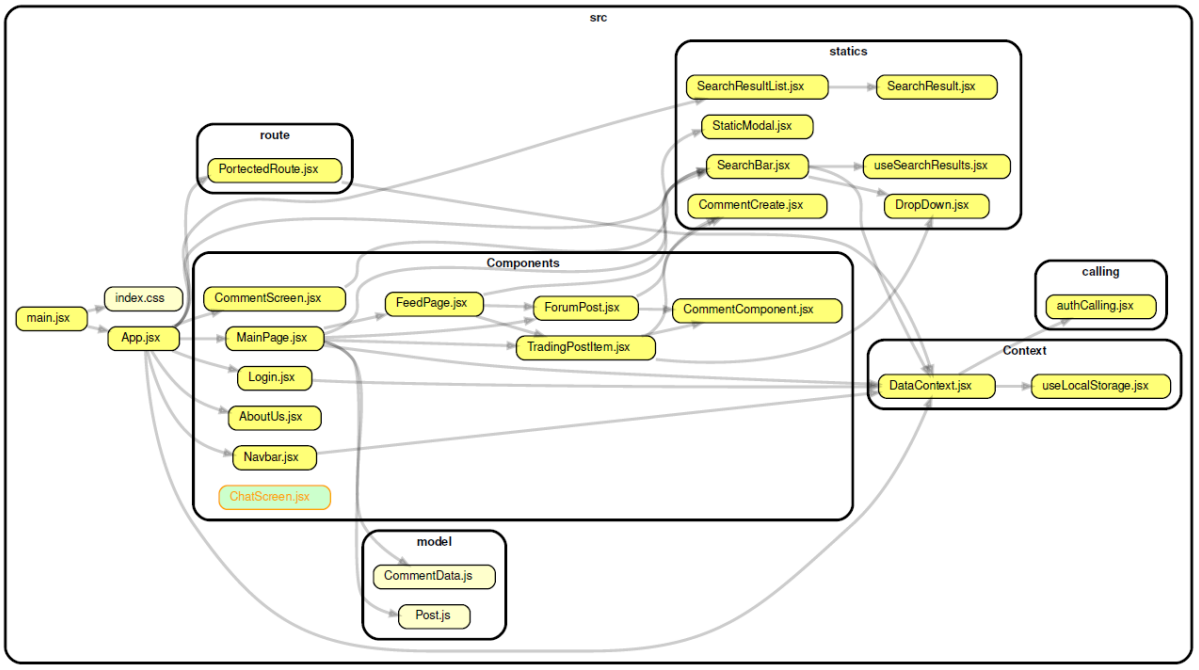
We utilized the observer designer pattern since the presence of the “Subject” and “Observers,” which are resembled by the “Chat” and “Participants,” correspond to a one-to-many relationship, and since our users can be the participants of multiple chats and a chat can have multiple participants, i.e., users, this design pattern is a solution to define common behaviors for the objects that either belong to the “Subject” or “Observers” category as well as separating their functionalities in interfaces.

Proxy Design Pattern



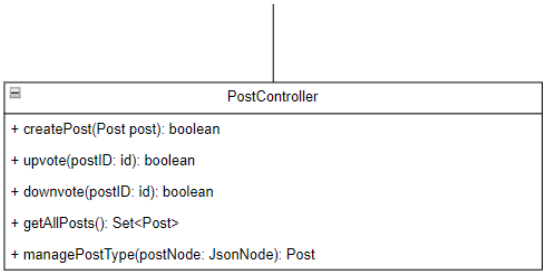
To optimize performance and not utilize the images unless they are to be shown as images, we utilized an imagePath for the classes that require an image. This way, the images won't be passed around unnecessarily, and performance won't be affected as much.

Provider Design Pattern



We used a provider design pattern in React to expose data among multiple children classes without giving parameters. Provider pattern is used by dataContext and useContext hooks.
Above given diagram is the current representation of our front end, which is not fully ready yet. The current representation of our UI is visually depicted using React-specific tools, as traditional Object-Oriented Programming UML diagrams may not precisely capture the nuances of React's component-based and declarative nature. Our decision to utilize online React tools for diagramming is due to the fact that React's architecture and relationships are better conveyed through specialized visualization tools.

Adapter Design Pattern



ADAPTER DESIGN PATTERN

While managing posts, we are getting posts as JSON, but to make them usable in our system, we change it back to Post type by using a glue code in the managePostType function. The adapter design pattern allows us to integrate library functions and makes them compatible with our program.