GymApp.life

University of Waterloo

SE 464

Bo Peng
b24peng

Dongyu Zheng
d28zheng

Gautam Gupta
g7gupta

Yuezhou Gao
y238gao

Table of Contents

1 Architectural Design

In this document, we will outline the architectural description for GymApp.life. We will go over the functional and nonfunctional requirements, components, connectors, configurations, and architectural design of the system.

1.1 Functional Requirements

GymApp.life is a web and mobile solution to track and inspire workouts, diet, and progress. The main functional requirements are listed below.

1. Choose, view, and log a workout
2. View and log a diet
3. Upload and view media
4. View analytics
5. User profiles and gyms nearby

1.2 Non-functional Properties

To provide a seamless and excellent user experience, we have determined the following non-functional attributes to be critical to the product's success. In section 1.7.3, we explain how we satisfied our NFPs through key design decisions.

1. Dependability
    a. Backups and redundancy to avoid single point of failures
    b. Automatic failovers
    c. Offline data persistence on the mobile application
2. Portability
    a. App works on any modern web browser and mobile browser
    b. Data seamlessly syncs between the web and mobile app

1.3 Components

Our system can be broken down into four major high-level components.

User Profile: The user profile represents the user entity. Each user entity may only have one active workout program. The user profile has relationships with workout program, nutrition, and media entities. The user profile also contains attributes that represents the user's settings and information.

Workout Program: The workout program component encapsulates the system's functionality and data for creating, viewing, and logging workouts. Within the workout program component, there are exercises and workout logs.

Nutrition: Similar to the workout program component, the nutrition component represents the system's functionality to capture diets. Within the nutrition component, there are diet logs.

Media: The media entity is related to the user profile entity. This entity captures the functionality to import and display photos.

1.4 External Services

In addition to the components listed above, we also use some external services. All external services have been found to be extremely reliable in testing.

1.4.1 Facebook API

To provide a secure and seamless sign up and log in experience for users, we decided to leverage Facebook's token-based authentication system. Through using the user's Facebook user ID and authentication token, the user workflow is significantly simplified, as the user just needs one click to create an account or log in. We also are able to extract data from the user's Facebook information, such as their name and profile picture.

1.4.2 Google Maps API

The Google Maps integration is used to provide an in-app experience for users to view nearby gyms.

1.4.3 Waterloo Portal API

The Waterloo Portal integration is used to provide an in-app experience for users to view the busyness level of a University of Waterloo gym. We display the number of people connected to the wireless network in each gym. The Waterloo Open Data initiative can be found on GitHub.

1.5 Connectors

Software connectors are architectural blocks tasked with regulating interactions between components. In our system, relevant connectors include:

1. Uploading and retrieving media requires a connection with AWS S3 storage
2. Real-time data stream connection with Waterloo Portal API
3. Procedure calls amongst the four major entities (profile, workout program nutrition, media), as well as the routing logic (frontend API endpoint)
4. Database access and synchronization

5. Integration with external services such as Facebook, Google Maps, and Waterloo APIs
6. Offline data persistence with Redux, and consequently function calls between views and states
7. Load balancing between the EC2 instances

1.6 Configurations

1.6.1 Component Diagram

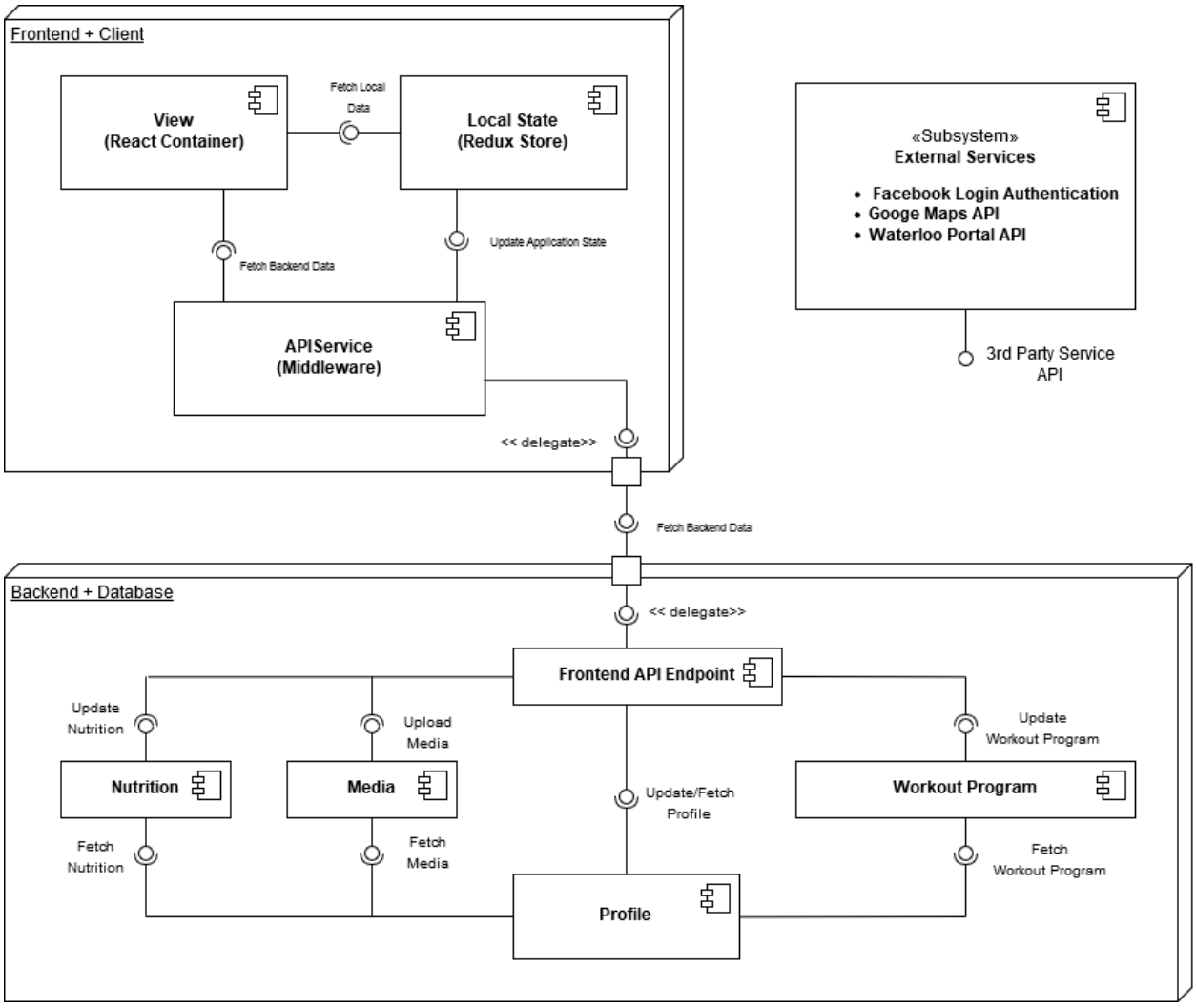

Figure 1: High level component diagram of the application's front and back ends

1.6.2 Physical Diagram

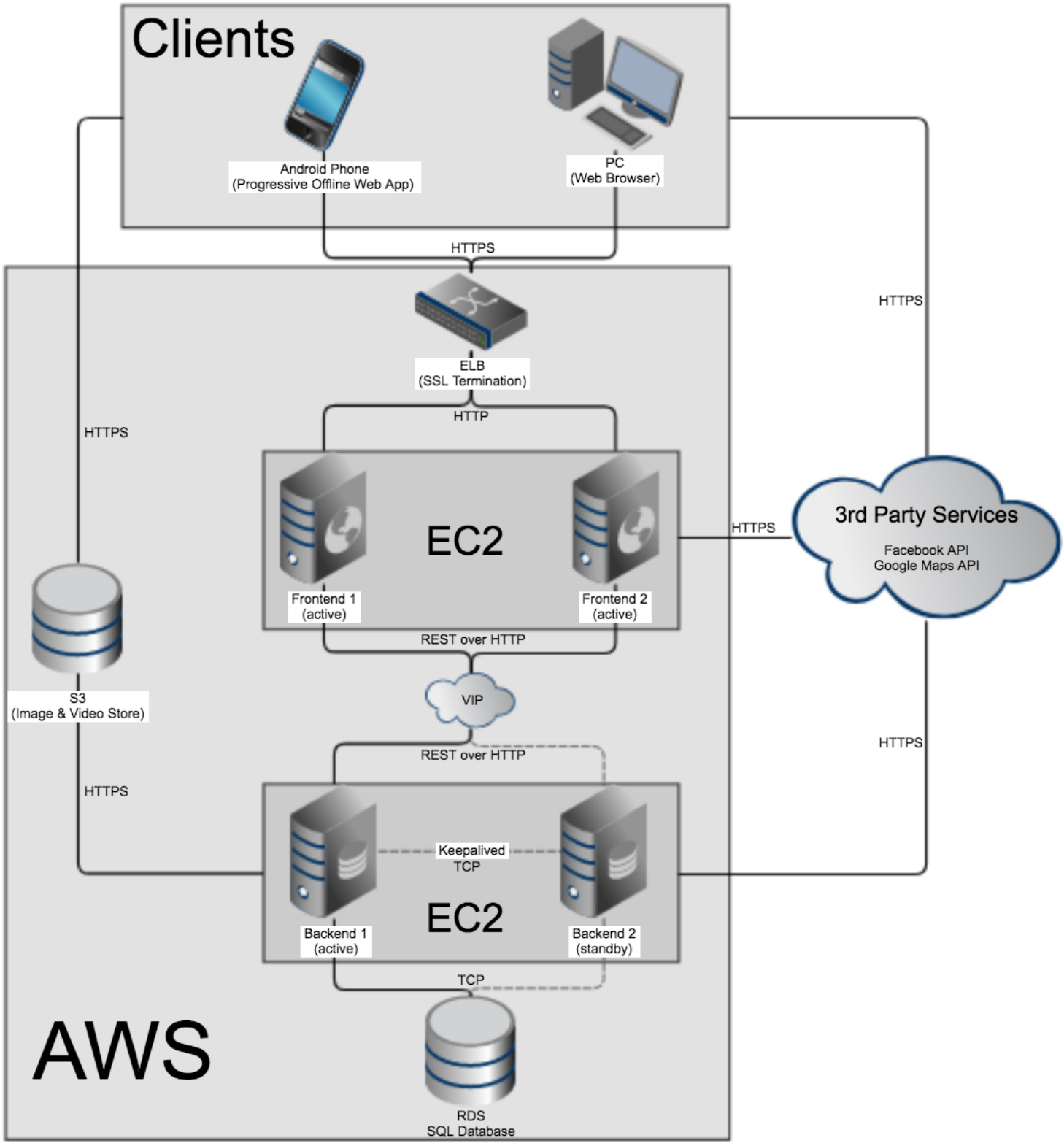

Figure 2: A visual of all the physical machines supporting each of the app's components

1.7 Architectural Design

Given the functional requirements and non-functional attributes listed above, we have determined a set of principal architectural decisions. In this section, we describe the architectural styles, provide explanations for key decisions and how we satisfied the NFPs.

1.7.1 Architectural styles

GymApp.life employs a combination of architectural styles:

1.7.1.1 Client-server (layered)

Table 1: Client-server chart

Layer	Client	Server
1	Phone / Desktop browser	Frontend
2	Frontend server	Backend server
3	Backend server	Database server

The 3rd party services (Facebook, Google Maps, Waterloo API) are considered as servers, with their clients as observed in figure 2.

1.7.1.2 Blackboard (shared memory)

S3 is a shared file store between the clients and backend servers. Both clients and backends read/write from/to S3.

1.7.1.3 Publish-subscribe / event-based (implicit invocation)

The frontend uses React which is implicit invocation based. Refer to section 2.2.3 for details.

1.7.2 Key architecture decisions

1.7.2.1 Why ELB?

Rather than deploying and managing our own load balancer, we opted to use ELB, which is provided and maintained automatically by Amazon. ELB is extremely reliable with very high throughput and provides many desirable features such as different balancing modes and health checking. It also provides SSL termination, which is a desirable feature in a layered web server architecture, with respect to efficiency and complexity.

1.7.2.2 Why RDS?

GymApp.life requires a database. RDS is a database service provided by Amazon. As with all of AWS's services, RDS is also extremely reliable. We used Amazon Aurora with PostgreSQL compatibility for our RDS database, which has amazingly high throughput. RDS also performs automatic snapshots for data backup.

1.7.2.3 Why Django?

We chose to use Django as our backend web framework. Django is a highly popular Python web framework used by organizations such as Instagram, Pinterest, and BitBucket. We chose to use Django because our backend developers are already highly experienced with Python and Django. Moreover, Django provides a superior out-of-the-box experience in terms of developer experience, community support, functionality, and reliability.

1.7.2.4 Why Keepalived?

Keepalived is a popular industry-wide solution for high-availability and load balancing. Keepalived can be used to perform health checks between our backend servers and elect a new master and execute scripts in failover scenarios. One of the team members had extensive experience using Keepalived for this exact purpose and therefore, we chose to use this software.

1.7.3 Satisfying NFPs

1.7.3.1 Dependability

Referencing figure 2, GymApp.life's core architecture employs two frontends, two backends, and Amazon's Elastic Load Balancer (ELB). Having two instances of frontends and backends help avoid single point of failures. ELB is provided and managed by Amazon and is therefore trusted to be highly-available.

Automatic failovers are achieved with the use of ELB for the frontends and a virtual IP + Keepalived for the backends. ELB is capable of detecting unhealthy nodes and automatically removes them from scheduling on failure; nodes are automatically re-added when they are determined to be healthy again. Keepalived is used to perform health checks within a group of 2+ backend nodes. There is one master backend active at a time, while the rest are on standby. The virtual IP, which cannot fail, is assigned to the master, and on failover, the newly elected master will assign the same virtual IP to itself. Thus, automatic failover is achieved for both the frontend and backend. Data backups are unnecessary as the frontend and backend servers are stateless, with all data stored in Amazon's provided Relational Data Store (RDS), which is trusted to be safe and automatically performs hourly snapshots.

Our servers are deployed across two availability zones. Since the servers are stateless and well decoupled, they can be scaled up at any point of time. Amazon has an SLA of 99.9%, which equates to a maximum of 8.76h of downtime per year. Adding more number of servers increases the number of 9's in the availability percentage and decreases downtime. However it comes at an added cost of maintaining more servers. The compromise between cost and availability depends on the revenue from the service. In our case, we chose to use two availability zones. Offline data persistence is explained in section 2.2.4.

1.7.3.2 Portability

Our app is a responsive application that utilises React on the frontend, and is responsive for both web and mobile browsers. The user logs into GymApp.life with a Facebook account, and can perform actions in any platform, whether mobile or web. Regardless of the platform, the user's information is stored in the database, so it can seamless sync between the web and mobile app. In terms of actual application execution, GymApp.life's backend runs on Python and the frontend runs on Javascript, which is portable on virtually every platform (Windows, MacOS, *nix). Keepalived is open-sourced and can be compiled for any platform.

2 Design

2.1 Backend Design

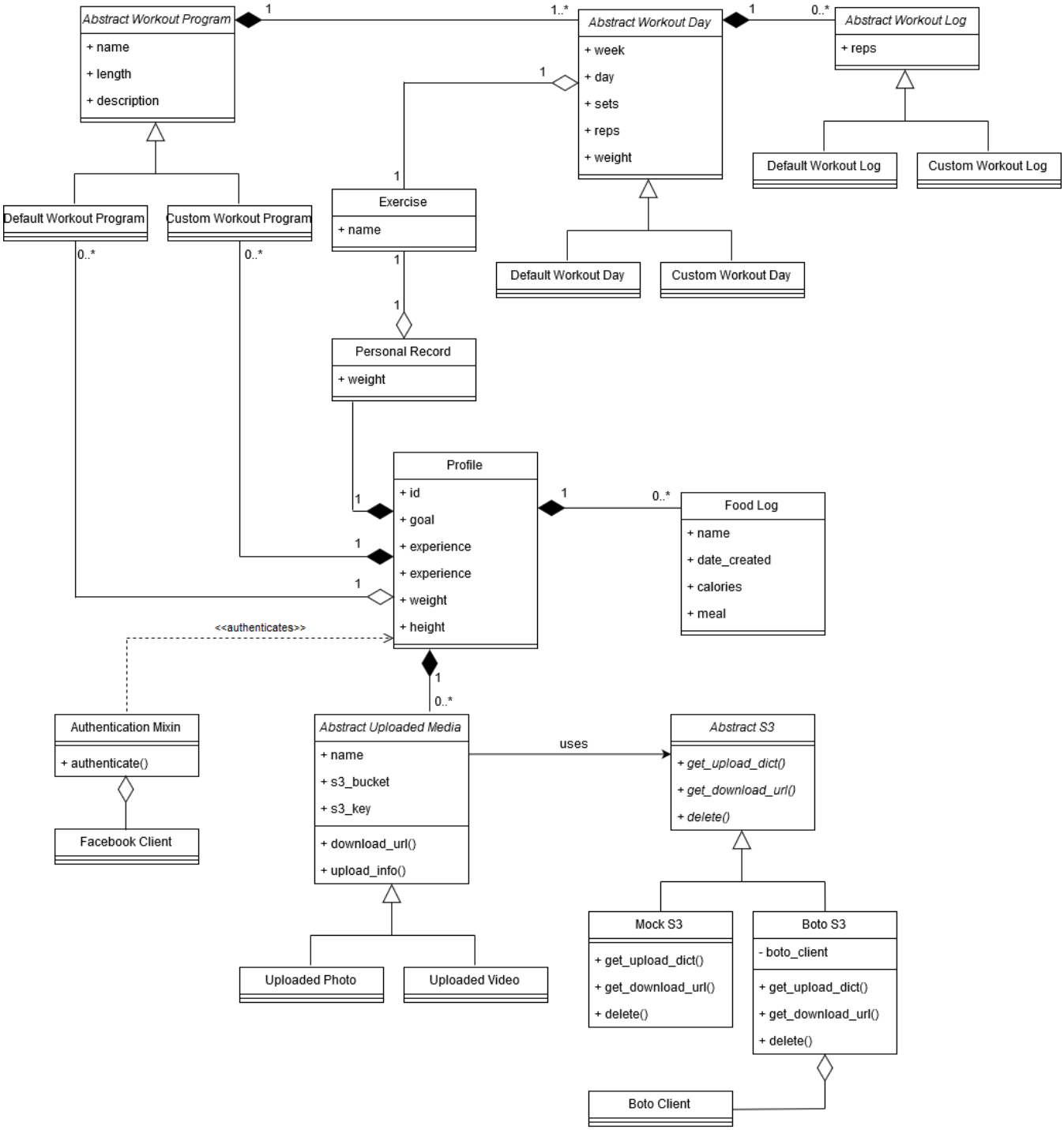

Figure 3: Backend and Database UML

Figure 3 above depicts a class diagram derived from our database schema. It maps closely to the backend and database subsystems detailed in figure 1.

As mentioned in the Components section, four major building blocks of the backend system are the profile, workout program, media, and nutrition components. The class diagram above further expands each component into its subclasses, and draws the relations between them.

2.1.1 Strategy Pattern

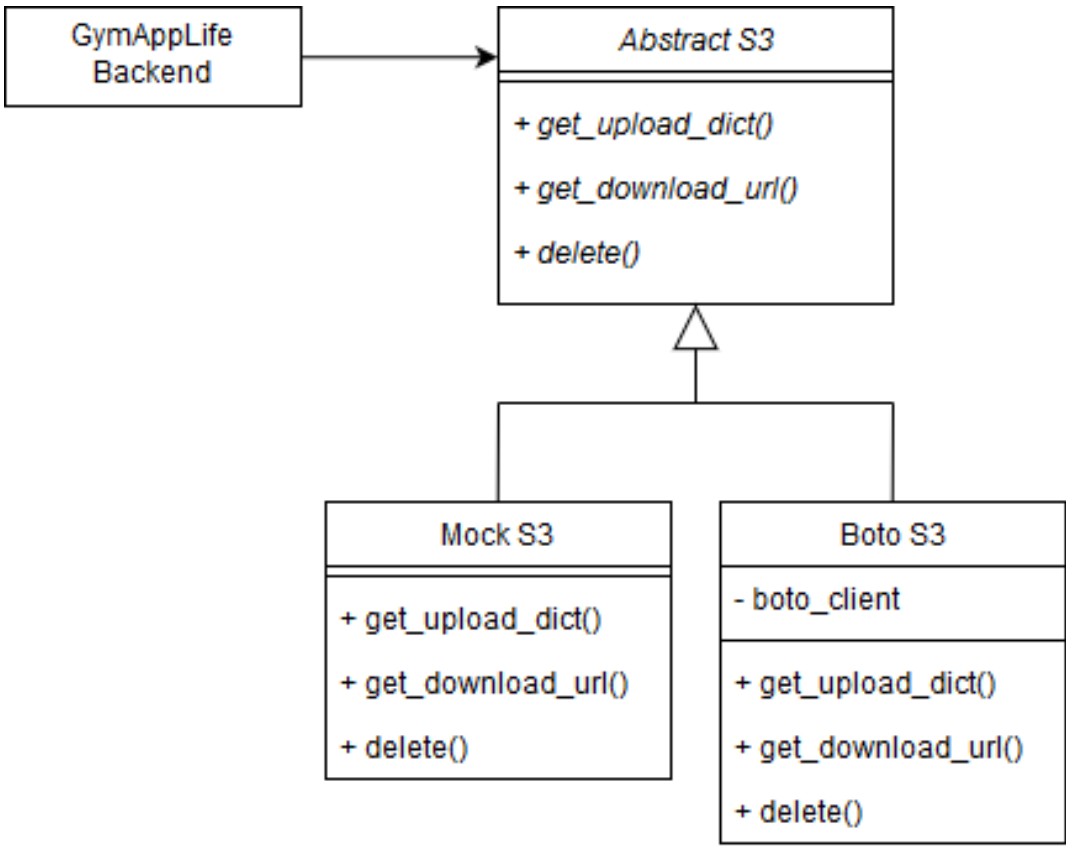

Figure 4: Strategy design pattern used in backend

The strategy pattern allows a program to swap between different concrete implementations of an abstract functionality at runtime. We leveraged this pattern in GymApp.life when building our connector to AWS S3. Specifically, when the backend server is running, it can be configured to run in debug or production modes which will swap the usages of Mock S3 and Boto S3.

The backend communicates with S3 through a S3 interface we wrote. In debug mode, Mock S3 is used so that function calls do not actually communicate S3 and return mocked data. While in production mode, Boto S3 is used to perform real API calls to AWS. We chose this pattern to enable our backend to easily switch between the mocked and real implementations.

2.1.2 Object Adapter Pattern

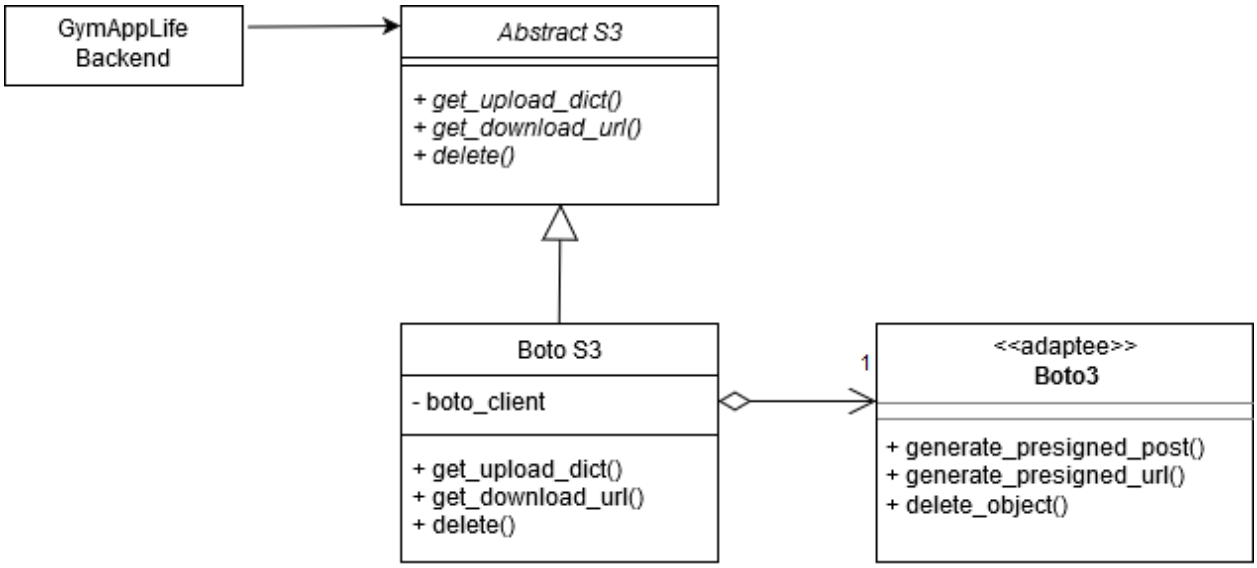

Figure 5: Object adapter pattern used in backend

AWS provides a Python client called Boto3 which is used interact with S3 resources. However, our backend developers do not program to Boto3's interface and through the adapter pattern, they are provided with a interface that abstracts away many concerns. Specifically, we used the object adapter pattern to wrap Boto3 in our own wrapper class. We chose the adapter pattern for the sole purpose of providing developers with a S3 interface more suitable than the one provided by Boto3.

2.1.3 Factory Method

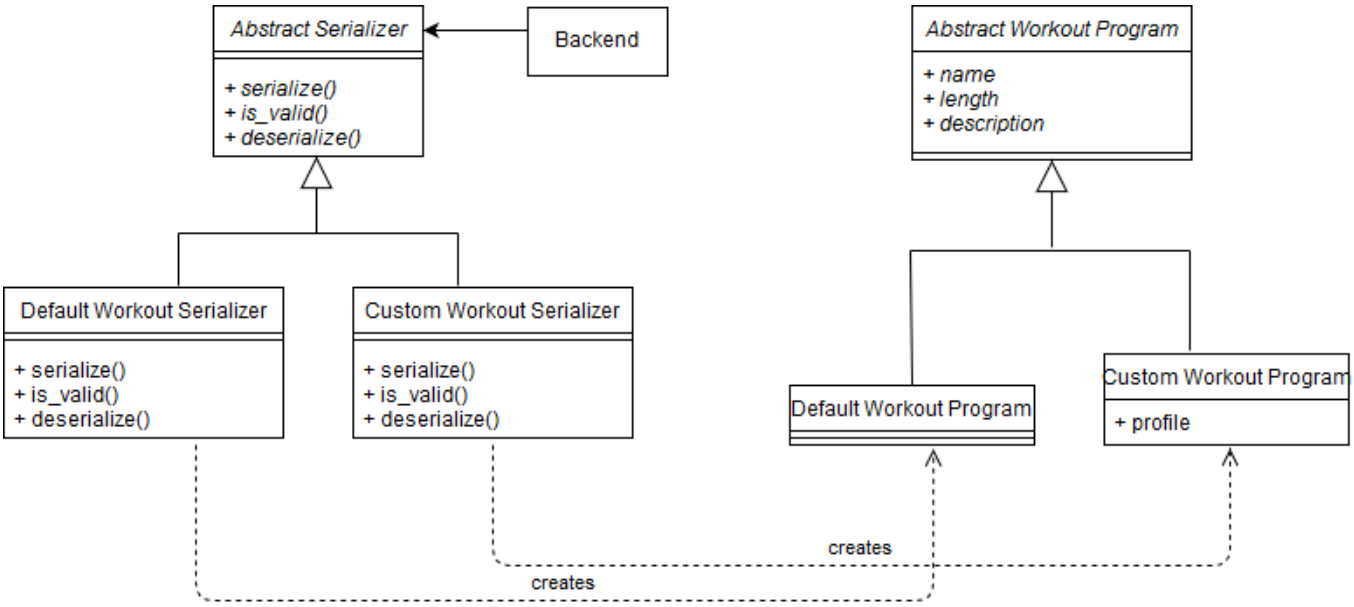

Figure 6: Factory Method used in backend

In order to transfer data between front and back ends, data structures need to be serialized and deserialized. Since many of our backend classes inherit from abstract base classes, we used the factory method design pattern to instantiate them from JSON strings. For example, our two types of workouts programs (default and custom) inherit from an abstract workout program class. They are respectively instantiated by two matching types of concrete serializer classes, both of which inherit from an abstract serializer class. This same pattern is used to instantiate many of our other backend classes, like Food Log and Uploaded Photos.

2.2 Frontend Design

Figure 7 below depicts a class diagram modelling our frontend React setup. It maps closely to the frontend and client subsystems detailed in figure 1.

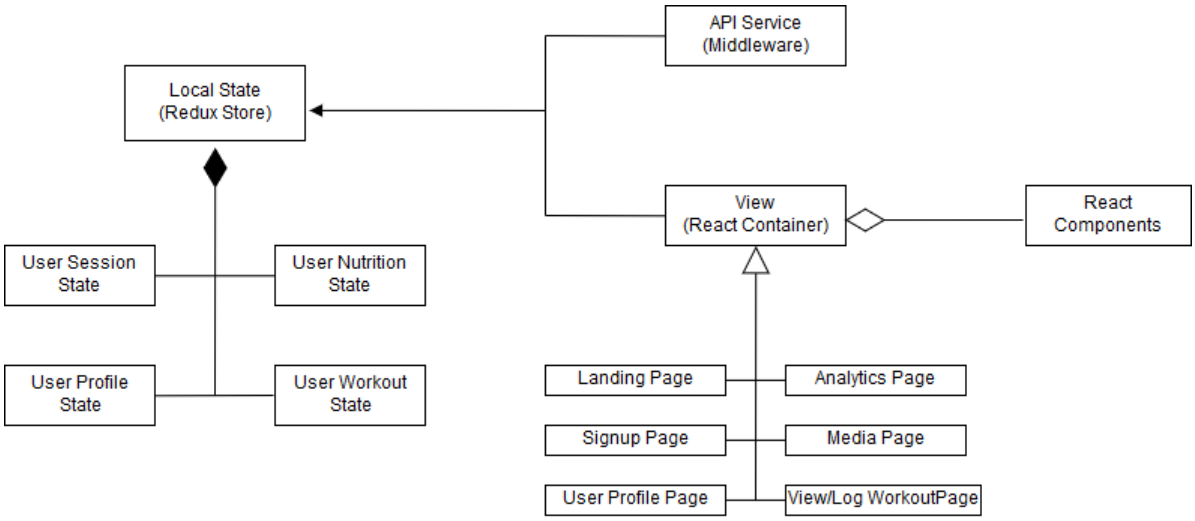

Figure 7: UML of frontend components

The frontend follows a very standard React + Redux setup. As a overview, it consists of raw React components, which are reusable HTML blocks that come together to form containers to be rendered in the browser. Because our app needs to function while offline, the frontend has its own notion of local state, used to populate and navigate among different pages. This state is called a Store, offered by Redux. Finally the API Service middleware serves to send and receive data from our backend server, and support asynchronous updates to the local state.

2.2.1 Composite Pattern

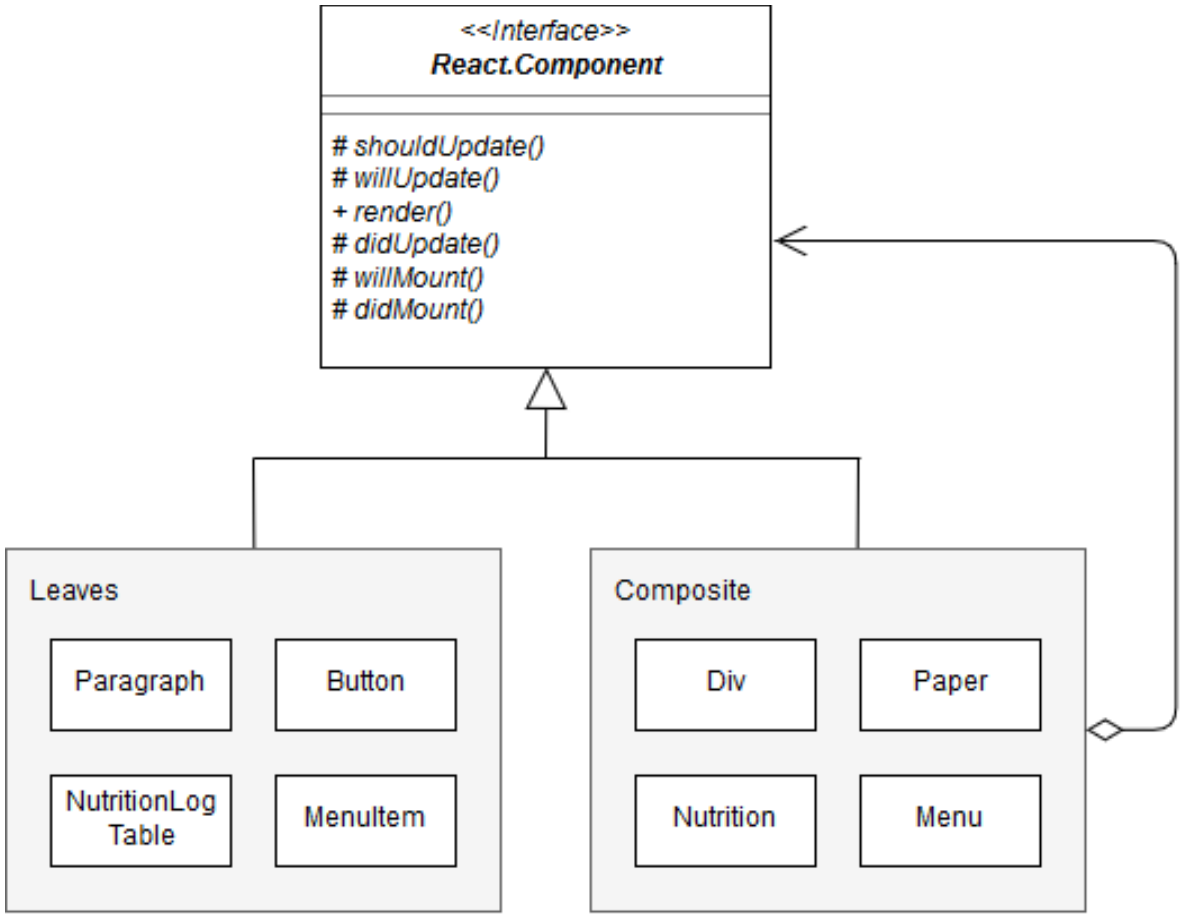

Figure 8: Composite design pattern used in React

The composite pattern is the most prominent design pattern used in web design. The DOM data structure of web pages are represented by a rendering tree built from composition. React makes this even more explicit with JSX classes that extend the React Component abstract base class. Examples of this pattern used in GymApp.life include nested and <Menu> classes with <p> and <MenuItem> tags as leaves. We chose this pattern because it exactly allows us to work with the DOM tree.

2.2.2 Decorator Pattern

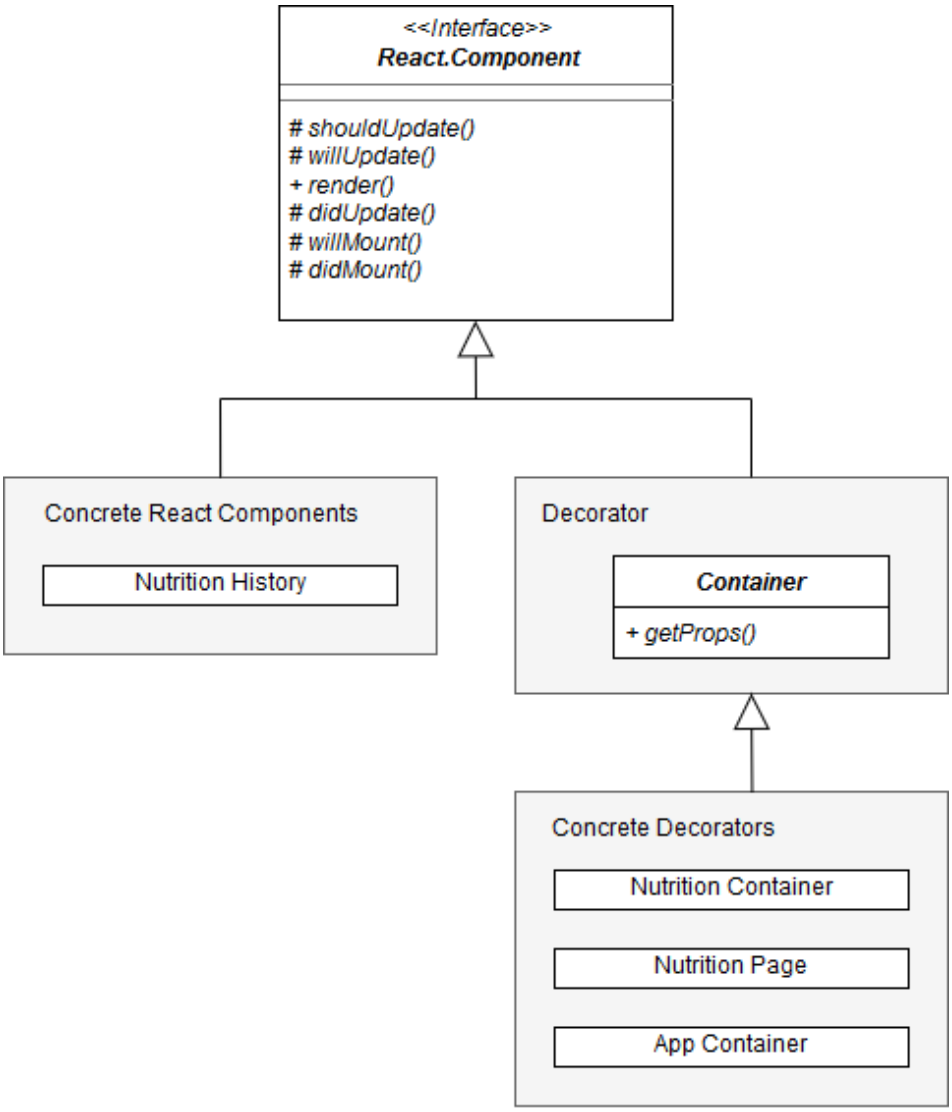

Figure 9: Decorator design pattern used in Redux

React also offers facilities to augment classes by wrapping them in “decorating” classes. Leaf classes of the DOM tree can be wrapped in a series of other react components to supply it with additional properties and functions. As an example, consider the Nutrition History component in GymApp.life. It is wrapped by the Nutrition Page component which gives it access to table data. This is further wrapped by the Nutrition Container, giving it access to the entire application state. We chose this pattern to help us modularize separate functionalities across all the wrapping layers.

2.2.3 Event Based Implicit Invocation

Javascript in web browsers inherently uses an implicit invocation architectural style. Specifically, it is event based and uses a hybrid bubble up and trickle down event dispatch mechanism. DOM components are aware of the existence of events, and specific handlers can be programmed into them. This is no different in React. For example, in GymApp.life, when “Log out” is clicked in the settings menu, the application state changes to set the loggedIn flag to false. The change in the flag is an event that eventually propagates to the listening App component, who handles it by returning the user to the home log in screen.

2.2.4 Frontend Framework Selection

We identified offline data persistence on mobile, ability for the app to work on a modern web/mobile browser, and seamless data sync between the frontend and backend as some of our non-functional requirements. This forms the basis of the selection of our frontend framework.

These requirements can be achieved by developing native apps for every platform we want to support or by developing a progressive web app (PWA) that works both on the web and mobile. Two approaches are discussed below.

1. Native apps: Developing native apps offer a very seamless experience for the user. However they are not platform agnostic and can be cumbersome to build, especially the offline functionality. Frameworks like React Native allow you to be platform agnostic between iOS and Android, but they does not support web.
2. Progressive Web Apps: The concept of PWA's support all of the non-functional requirements listed above. They are meant to live in the user's app drawer for easy access (only supported by Android at the moment). They offer close to native experience. PWA can be implemented by using React and Redux together.

We initially started with the approach of developing native apps using React Native and a separate web app for ease of access to the user. However, we soon realized this approach was not feasible given the time constraint and was proving to be cumbersome. We combined resources to work on one progressive web app to function on both desktop and mobile browsers, and to also support the aforementioned constraints.

React also makes use of some nice design patterns such as one direction data flow (eg. using Redux) and dependency injection. With dependency injection, React components receive dependencies via props. Using the concept of PropTypes and factory design pattern, dependencies can be validated while being passed down the tree.

2.3 Data Flow Design

2.3.1 User onboarding workflow

When the user accesses GymApp.life for the first time, the user can create an account via Facebook signup. Their basic information such as their name, email, and profile picture are captured and used to populate their GymApp.life profile. Next, the user would be prompted to answer questions relating to their fitness goals and lifestyle. This completes the signup workflow.

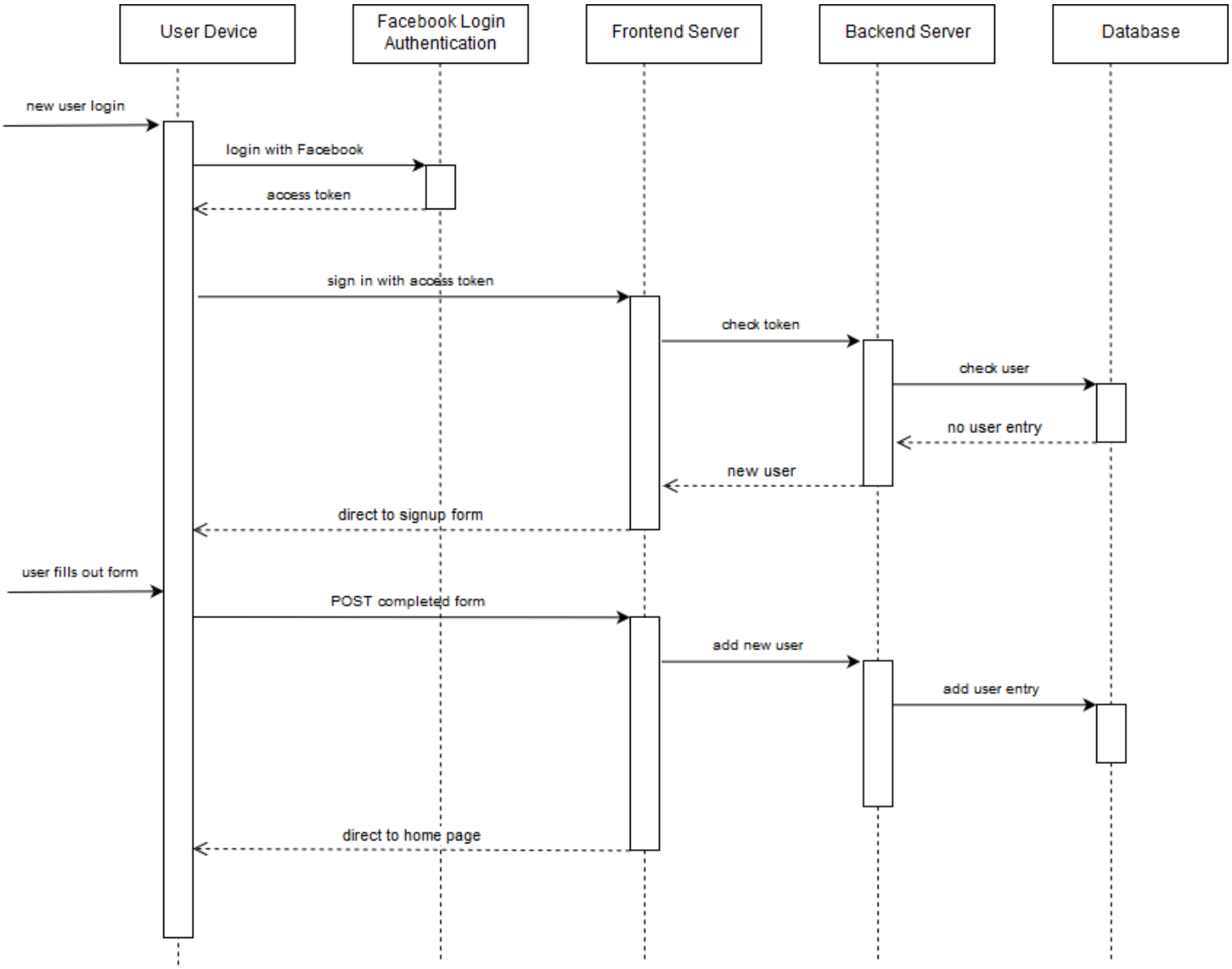

Figure 10: Signup scenario sequence diagram

2.3.2 Analytics workflow

After the user has entered multiple workout or nutrition logs, they can go to the “Analytics” module. From this screen, they can view graphs of how they have progressed in terms of strength, weight, and nutrition.

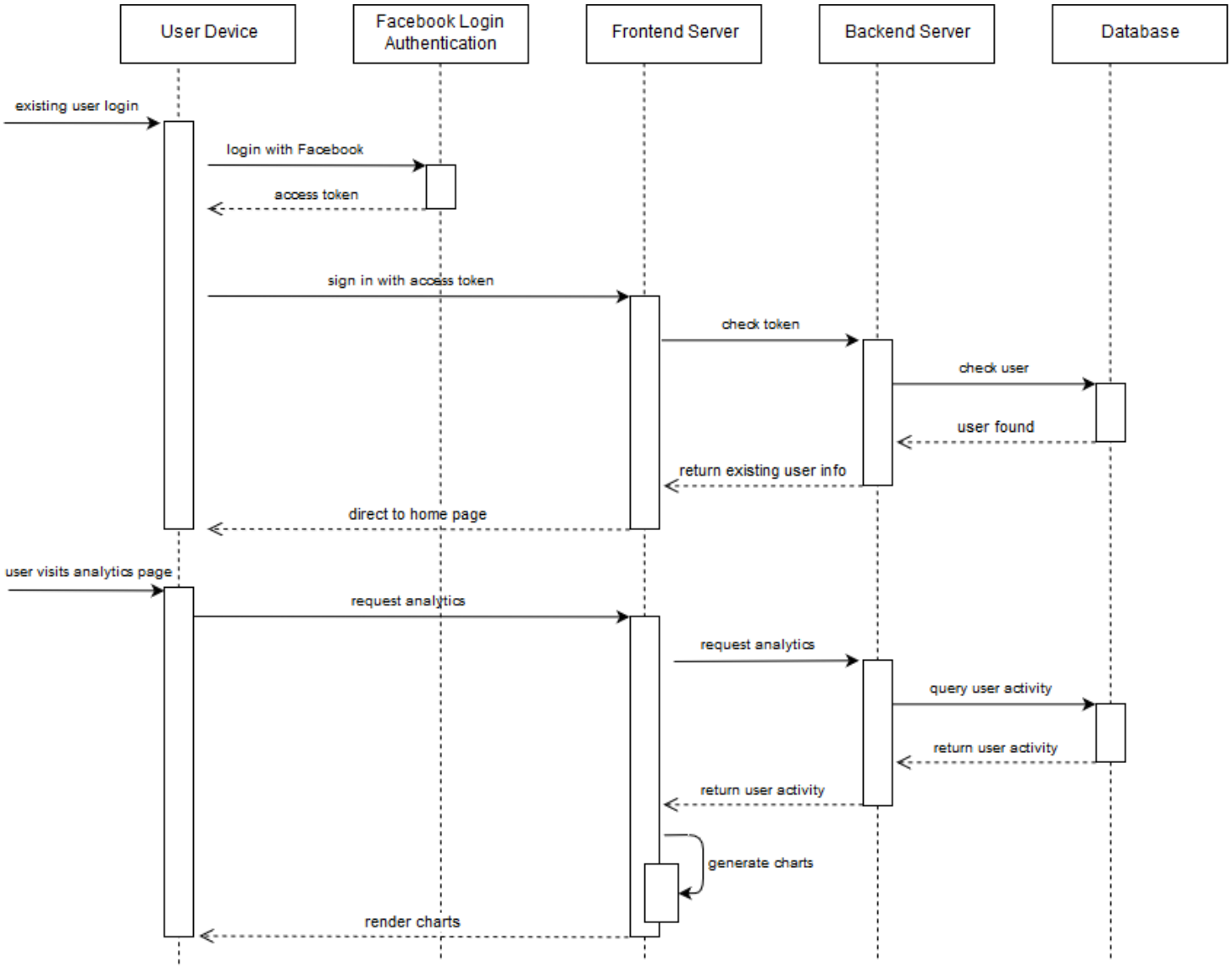

Figure 11: View analytics sequence diagram

2.4 Coupling

Our architecture is highly modular and decoupled to support future development and scalability. We have identified some scenarios below and explain how they scale with respect to coupling.

2.4.1 Adding new workout programs

We use the strategy design pattern for workout programs, so the user can easily switch between a default or custom workout program. The addition of a new workout can be represented as a Abstract Workout Program, which results in a smooth integration with the existing views and other components.

2.4.2 Backend vs. frontend

Our frontend views are completely decoupled from the backend. To communication with the backend, we use RESTful APIs to enable statelessness, and development of new frontend views can occur independently of the backend. This design enables high degrees of scalability.

2.4.3 New browser / device support

Since modern web browsers are omnipresent, our progressive web app can support any modern browser. GymApp.life is decoupled from any native bindings of platforms. In addition, we offer functionality that are usually associated with native apps such as offline support.

3 Participation Journal

Table 2: Participation journal

Name	Contribution
Bo	Project management, mockup designs, documentation, DB schemas
Dongyu	Database, backend, API, infrastructure, related documentation
Gautam	Frontend, related documentation
Yuezhou	Infrastructure, frontend, related documentation