Pump-Pals

Ariya Seng, Carlos Negretes, Kelvin Tims, Michael Liam Sinclair, S M Raisul Alam Bhuiyan,

Tom Tran

College of Science & Technology, University of South Carolina Upstate

CSCI U540: Software Engineering

Dr. Ashok Gupta

December 10th, 2023

Table of Context

I. Abstract

II. Business Case

III. Requirements

A. Purpose

B. Scope

C. Functional Requirements

D. Non-functional Requirements

IV. System Management

A. System Modeling

1. Overview

2. Backend

3. Frontend

B. Data Management

V. Programming Analysis

VI. Testing & Use Cases

VII. Appendices

A. Appendix 1: Hours Spent

B. Appendix 2: Requirements Checklist

C. Appendix 3: Program Code and Executable

D. Appendix 4: Project Status Reports

E. Appendix 5: Test Cases

VIII. References

Abstract

In the world of personal fitness, there is alway an ever-evolving demand for more accessible products that many of our end users can enjoy. When we first set our goal, we went beyond the possibilities of just a class project, and into the real world problems that exist in our niche market. What we've found is that there is a distinctive lack of an industry where we can collectively group together to not only form a community, but to also share and support each other. This makes us come up with the idea of Pump-Pals, a personal fitness and health application that is designed to give the end users the ability to communicate with each other, the ability to track their health, and the ability to succeed with their own personal ambitions. With crafted expertise utilizing Spring Boot, React, MongoDB, and many other resources, we were able to transform our application into a place where everyone who is interested in the personal fitness and care industry are able to enjoy. We chose to go this route as we believe that not only will it be immensely useful not in terms of class projects, but also in terms of real-world applications that can be utilized to its fullest potential.

Business Case

For many cases, the personal fitness and care industry has experienced a growth, driven in part by other industries, as well as increasing global awareness in wellness. On the other hand, there is an increase in the activities of the technological world, where one such technology known as the Internet has dominated for the past few decades, with rapid activities starting to prop up for every other industry. Hence the personal fitness and care industry has been utilizing the Internet to its potential, but there is an amassed amount of resources on the Internet that hasn't been utilized that much, especially on the World Wide Web. We believe that the personal fitness and care industry could use a lot of work on not only utilizing the Internet to its full potential, but also utilizing the services given to promote user interaction and create a community of passionate fitness enthusiasts.

For our group, we've chosen to first decide on the scope of the project to be our target audience who are primarily in the personal fitness and care industries. These include people such as fitness enthusiasts interested in utilizing our community engagement and progress tracking. Our application will have these following key services such as user connections, workout logging, user discovery, progress tracking, and refined user content. This will allow us to not only meet our target audience, but to also give us a clear direction as to our scope. Our scope will involve us primarily building an application with a full comprehensive feature list such as the one mentioned before so that we can expand our audience to a greater degree.

Requirements

I. 1.1 Purpose

A. The purpose of the application document is to outline the requirements for the development of a fitness-related application.

B. It aims to assist users in fitness-related progress, personal data, and making connections to other fitness enthusiasts.

II. 1.2 Scope

A. The fitness app will provide users with a platform to input and track their fitness data, personal information, and exercise routines.

B. Users will be able to follow and connect with other users.

C. Users will be able to post personal content visible to other users that they may interact with.

2. Functional Requirements

I. 2.1 User Registration and Profile

A. 2.1.1 User Registration

1. Users should be able to create an account by providing basic information (name, email, password).

B. 2.1.2 User Profile

1. Users can create and update their profiles with personal details, such as age, gender, weight, height, profile picture, and fitness goals.
2. Users can create posts that show up on their profile as well as the forum and their followers feed.

II. 2.2 Fitness Tracking

A. 2.2.1 Exercise Logging

1. Users can log their daily exercise routines including the exercise type, sets, and number of repetitions performed.
2. These workouts are publicly visible and show on the user's profile, their followers' feed, and the forum.

B. 2.2.2 Weight and Measurement Tracking

1. Users can track changes in their weight and body measurements over time.

III. 2.3 Connections

A. 2.3.1 Dashboard

1. Users can come to their dashboard and see recent posts and workouts from users they follow, as well as their own.

B. 2.3.2 Forum

1. Users can come to the forum and see recent posts and workouts from global users, as well as a list of recommended users for them to follow.

C. 2.3.3 User Profiles

1. Users can find other users' profile pages by searching their username or clicking a direct link.
2. The loaded user profile will display that individual's recent posts, workouts, user information, and personal statistics.

3. Non-functional Requirements

- 3.1 Performance
  - The app should load quickly and respond to user inputs in a timely manner.
  - The server should have sufficient capacity to handle concurrent user requests.

- 3.2 Security
    - User data, including personal information and fitness data, should be encrypted and securely stored.
    - Secure authentication methods should be implemented to protect user accounts.
- 3.3 User Interface
    - The user interface should be intuitive and user-friendly, catering to users of various fitness levels.
    - Design should be visually appealing and compatible with different screen sizes.
- 3.4 Compatibility
    - Compatibility with a range of devices and screen sizes should be ensured.
- 3.5 Maintenance and Support
    - Regular updates and maintenance should be provided to address bugs, security issues, and compatibility with new devices.

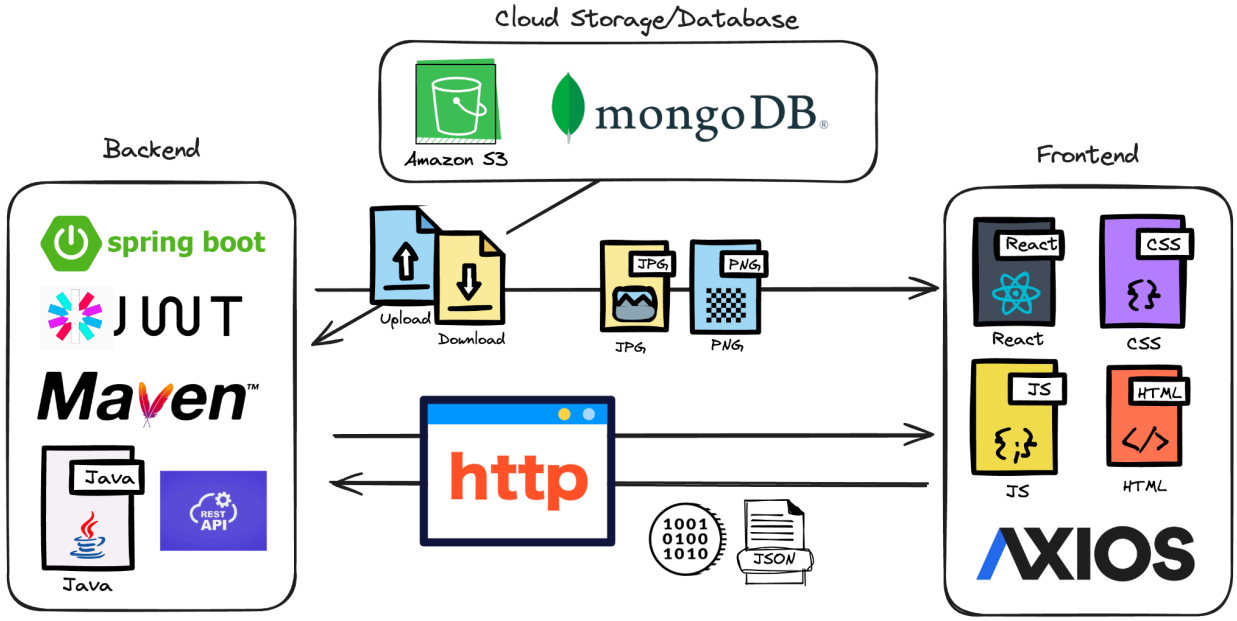
4. Technical Requirements

- 4.1 Platform
    - The app should be developed for any devices supporting HTML5.
- 4.2 Technology Stack
    - The app should be built using native technologies for optimal performance.
    - Backend services should be implemented using scalable and secure frameworks.
- 4.3 Data Storage
    - A robust and scalable database system should be implemented to store user data securely.

System Management

System Modeling

Overview



Pictured above is a high-level overview of the system. The database is a MongoDB cloud database that hosts user data and information, and the cloud storage is an AWS S3 bucket that contains user submitted media (profile pictures, post photos/gifs, etc). MongoDB was chosen for its ease of use and flexibility, since user submissions and profiles will not always be uniform. The data in MongoDB databases are JSON formatted, which is perfect for our backend and frontend, since that is primarily the form that information will be processed in. The AWS S3 bucket was also chosen for its accessibility as well as its higher level of security. The backend contains an access key and a secret key that belongs to an IAM service worker account on AWS. The service worker account has limited rights and its keys can be revoked or renewed at any

time, increasing data security.

Backend

The backend for our software is written in Java and utilizes Maven as a build automation and project management tool as well as a Spring Boot framework for its RAD (Rapid Application Development). Maven also provides dependency management, of which we use the following for the backend (found in the pom.xml file):

- AWS Java SDK
- MongoDB Connector
- Spring Security
- Lombok
- JWT API + JWT Jackson
- Spring Boot Devtools
- Jaxb API
- Commons IO

The backend's API is a REST API that is accessible using HTTP requests and it fulfills the following functions:

User Authentication Using JSON Web Tokens

JWTs are signed (by a secret key) tokens that contain the user's username, claims, and the token's expiration date. These tokens are stored in the user's local storage after successful authentication or account creation and expire after two hours. The tokens must be included in every request made to the API outside of account creation or authentication, and this makes sure

that not only is every endpoint protected, but the software knows exactly who is making the request. This not only improves security but also makes API requests easier since the username does not need to be globally tracked in the frontend, it can simply be gathered from the saved token.

Data Storage and Retrieval

All data that the user produces (input, likes, comments, follows, etc) must be sent to the API so that it can be stored in the S3 bucket or database. This is done using the AWS Java SDK and the MongoDB connector. All data pulled from both are saved in local memory in the backend until it is sent to the frontend, after which it is removed from volatile memory via Java's garbage collection. Data gathered from the database is applied to Java class objects that are modeled by the model classes (further on this later) from which they can easily be processed by the written Java code. Following processing, it is then reformatted to JSON format and sent to the frontend, where it is then recast to JavaScript objects to be further processed. User submitted media is uploaded and downloaded from the S3 bucket to the backend, saved to a Java object in local memory, sent as a byte array as an HTTP response when requested, then finally processed by the frontend and displayed to the user. As for data storage, it is a similar reversed process, though instead of HTTP GET requests made by the frontend, it is a POST/PUT method with the media/data in the body of the request (and of course the JWT in the header of the request so the application knows who is making it).

Serving the Frontend

The frontend will be discussed at a later point, but it is built on a React framework which

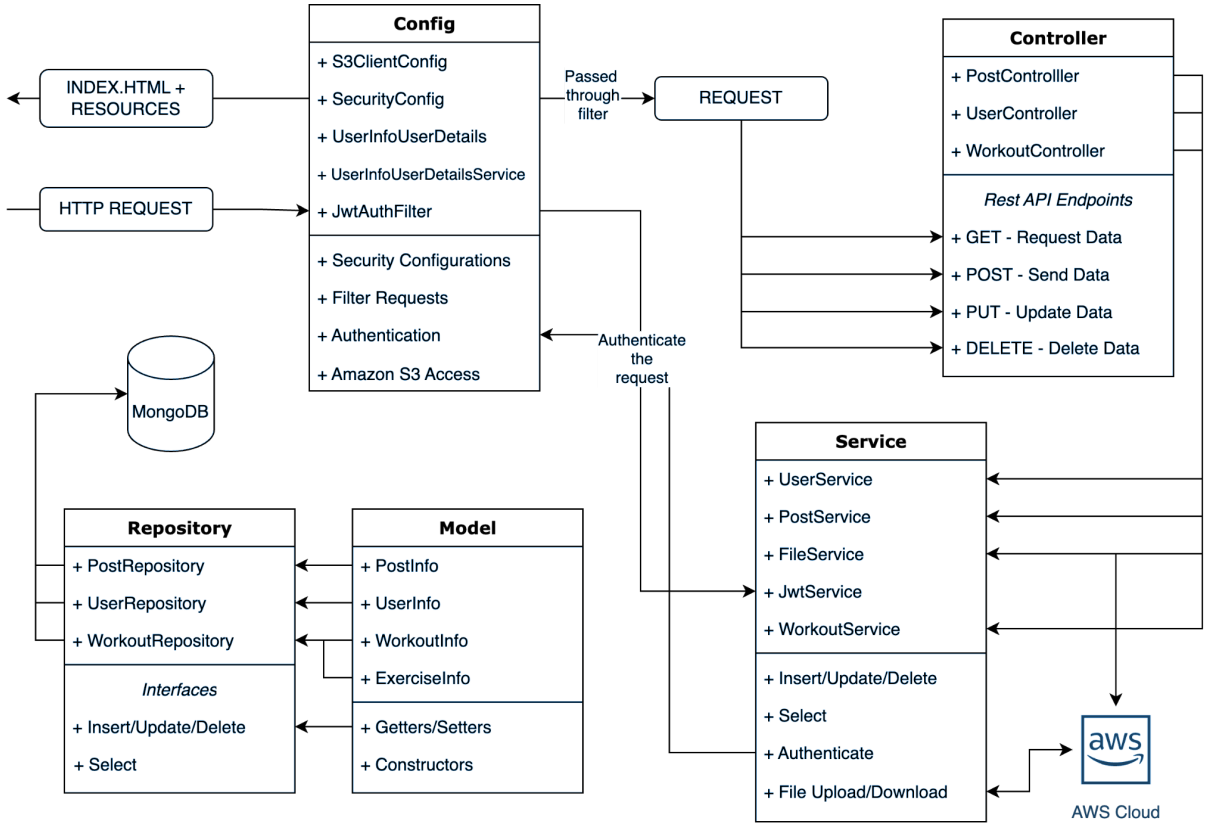
makes it a single static resource. The frontend is compiled using npm and stored in /src/resources/static and served to the user as the application is accessed through the user's browser.

Data Processing

All data received by the backend must be properly processed before it is stored or sent. The majority of this processing is carried out by each model's service class, which are the intermediary between the repository interfaces and controllers. Processing includes gathering the user's details from their token, filtering requests, adding and modifying incoming data, appending data to existing objects and entries, error handling, etc.

Security Configurations

Spring Security allows for customized security, and the backend utilizes this by only having certain endpoints public. As previously mentioned, only the account creation and user authentication endpoints are public, as well as the static resources that are to be served to the client. All other endpoints are private and return a 403 status code when unauthenticated users attempt to make a request.



Pictured above is a high-level overview of the backend architecture, as well as the typical path that requests take. The backend's structure is divided into five components: Configuration classes, Controller classes, Service Classes, Model classes, and Repository interfaces. The table below provides a brief description of each as well as their functions.

Category	**Description**	**Functions**
Configuration Classes	Configure the use and implementation of dependencies.	Configure the AWS S3 connector as well as define the security policy, password encoder, authentication provider, request filter, and user details service that is used for authentication
Controller Classes	Rest controllers that contain methods mapped to specific	Receive incoming authenticated requests and do

	endpoints. The HTTP request invokes the appropriate method and supplies the correct parameters, if needed. Once the method completes its function the proper data is then sent to the frontend. Utilizes a REST API.	exactly what the request is for. These controllers have the function that the name implies, they control the functionality of the application and manage requests. These classes do not process data on their own, and call on service classes to do this for them. All incoming/outgoing data passes through the controllers.
Service Classes	Carry out the operations that the controller calls for and provide the services the application seeks to serve.	Process incoming data passed by its respective controller, handle file uploads and downloads, interact with the repository interfaces for saving/updating/deleting data stored in the database, and handle exceptions.
Repository Interfaces	Access the database, each repository interface only accesses its respective collection (ex: UserRepository for User collection). Cannot process information and can only save, update, and delete data.	Used by the service classes to access the database. They not only save, update, and delete data as per requested by the client, but also retrieve data as necessary for computations and operations as required by the service classes.
Model Classes	Used by the MongoDB connector to attach retrieved data to so that it may be used as an object. Also used by controllers to map incoming data to Java objects, these are the general interfaces for interacting with data.	Allow for the interaction and processing of data, as the application is object-oriented. Incoming request bodies are automatically applied to Java objects and are able to be immediately processed. These classes also model the database structure, since all documents take the form of the model classes.

With these categories defined, it becomes much easier to understand the path that

requests take. As previously mentioned, all requests to private endpoints must include a valid token in the request header and must pass through an authentication filter. The moment a request is received it is met by the JwtAuthFilter class which parses the request for a token string and passes it to the JwtService classes for validation. Once the request is validated, it will typically take the following path:

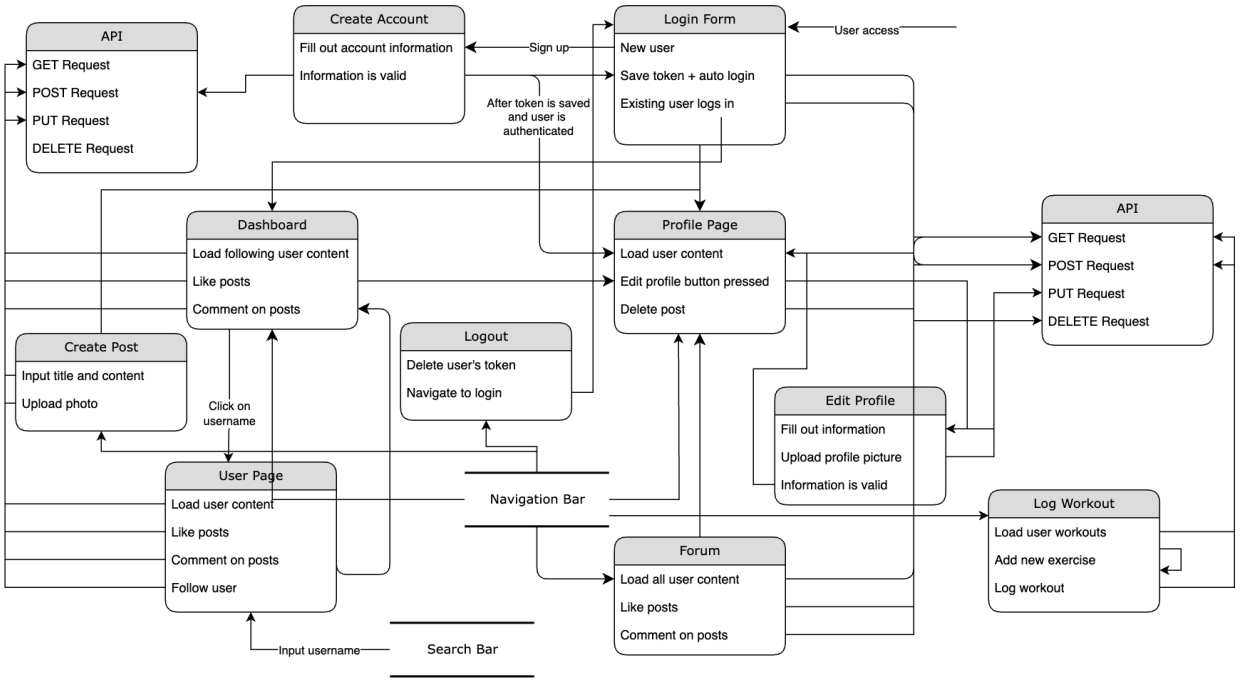
1. The request will be met with its corresponding endpoint in one of the controller classes.
2. The data will automatically be applied to a Java object as defined in the model classes if required.
3. The controller class will read the data in the request and either:
    a. Respond with an error code because the request is missing information or contains incorrect information
    b. Process the data as requested
4. The controller will call the service class if needed to retrieve data, upload data, update data, or delete data based on the request type.
5. The service class will read the data and perform the requested operation, and call a repository interface if needed.
6. The repository interface will interact with the database by inserting, updating, deleting, or selecting data as requested. (If a file is to be uploaded or downloaded then the file service class will do this on its own).
7. The repository will return the result of its operation to the service class, from which the service class will then further perform operations until the method is fully performed.
8. The service class will then return its result to the controller class which will then return an HTTP response to the client frontend.

Below is a basic table of HTTP request types and their functions. These request types communicate to the application what type of request is going to be carried out.

Type	**Description**
GET	Request data from the server
POST	Send data to the server
PUT	Update data on the server
DELETE	Delete data from the server

This finishes the summary on the backend's architecture, as well as the paths that requests will typically take. Further details and information on the implementation of this model can be found under the programming analysis section of this document.

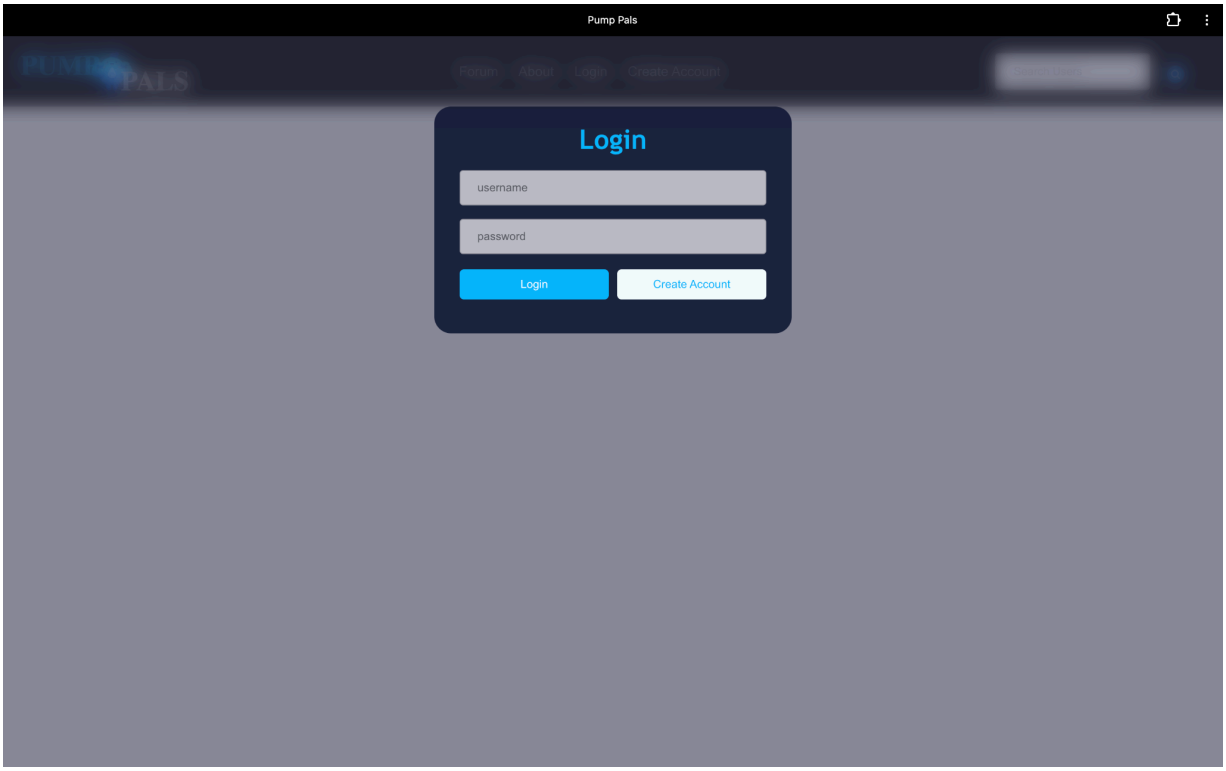
Frontend

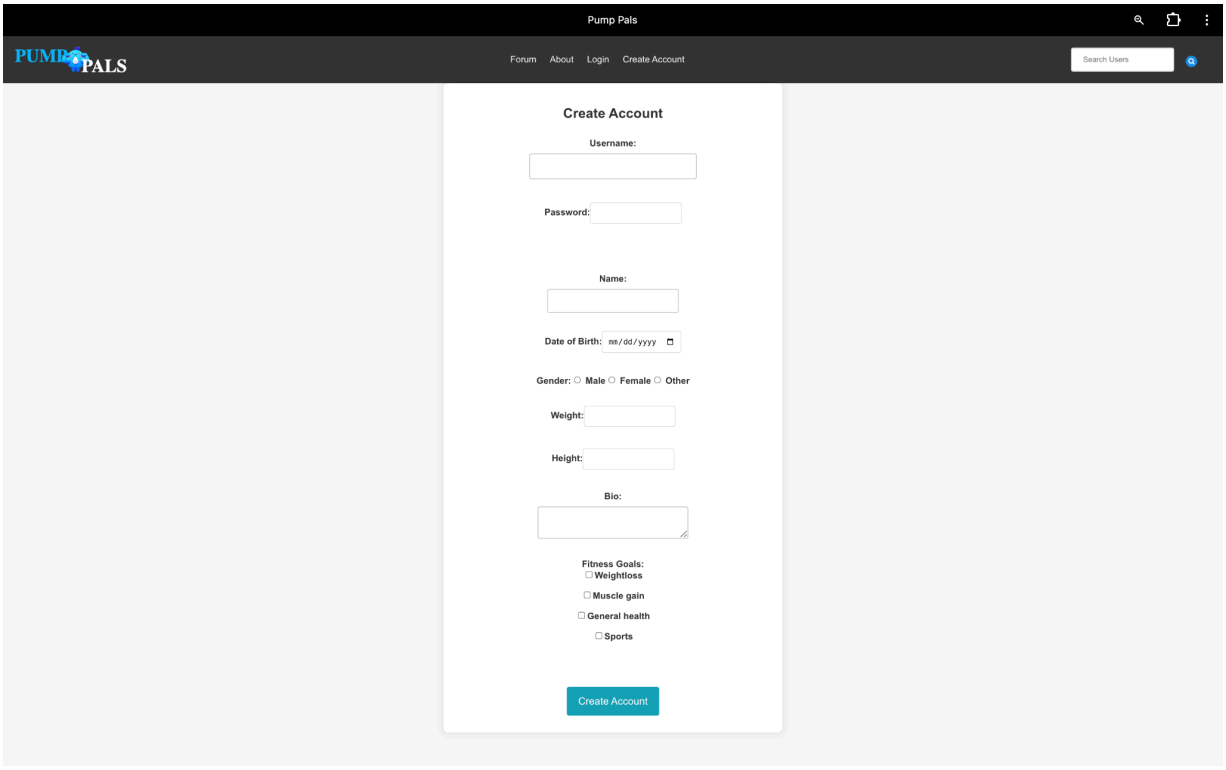


Pictured above is the general architecture for the frontend, or user interface, of the application, as well as the flow of user's experience. The application contains several pages that the user may interact with, beginning with the login page. Each page outside of the login page requires authentication to access, so if no valid token is found in the user's local storage then the application will route to the login page. The API objects in the diagram represent the backend, with a list of each type of HTTP request that that page may make to the backend. Each page that loads user data displays this data in the form of cards that are rendered with each object of information. The table below goes into detail on the information that each card contains, as well as the information that is presented to the user.

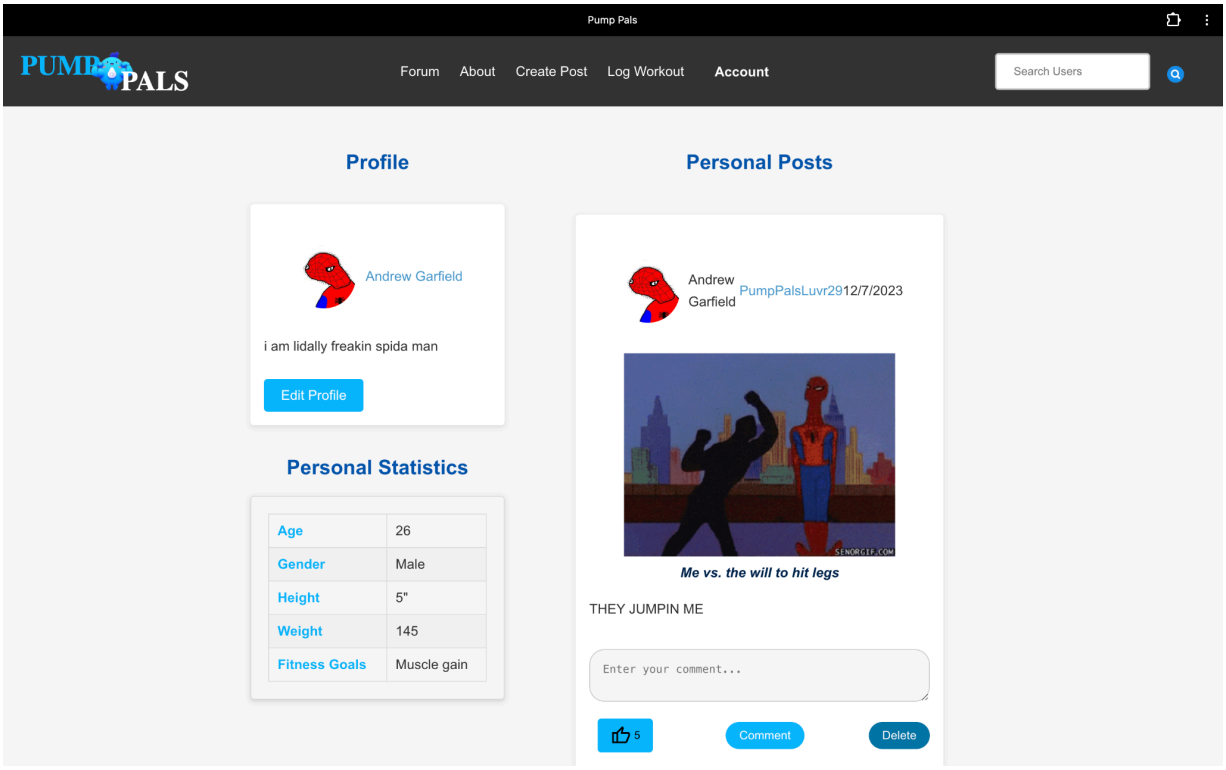
Type	**Visible to the user**	**Not visible to the user**
User Card	Name, bio, isFollowing, profile picture	Username, followers
Post Card	Profile picture, name, username, upload date, content, likes, comments, commenters, picture, title	Likers, hasPicture, postId
Stat Card	Age, height, weight, fitness goals, gender	Date of birth
Workout Card	Date, name, exercises	Username, workoutId

Graphic examples of these user, post, stat, and workout cards may be found in the following table on the dashboard, forum, user, and other similar pages. Each page with user content will display a list of these cards. More information on these cards will be covered in a later section. The table below further discusses each page's function:

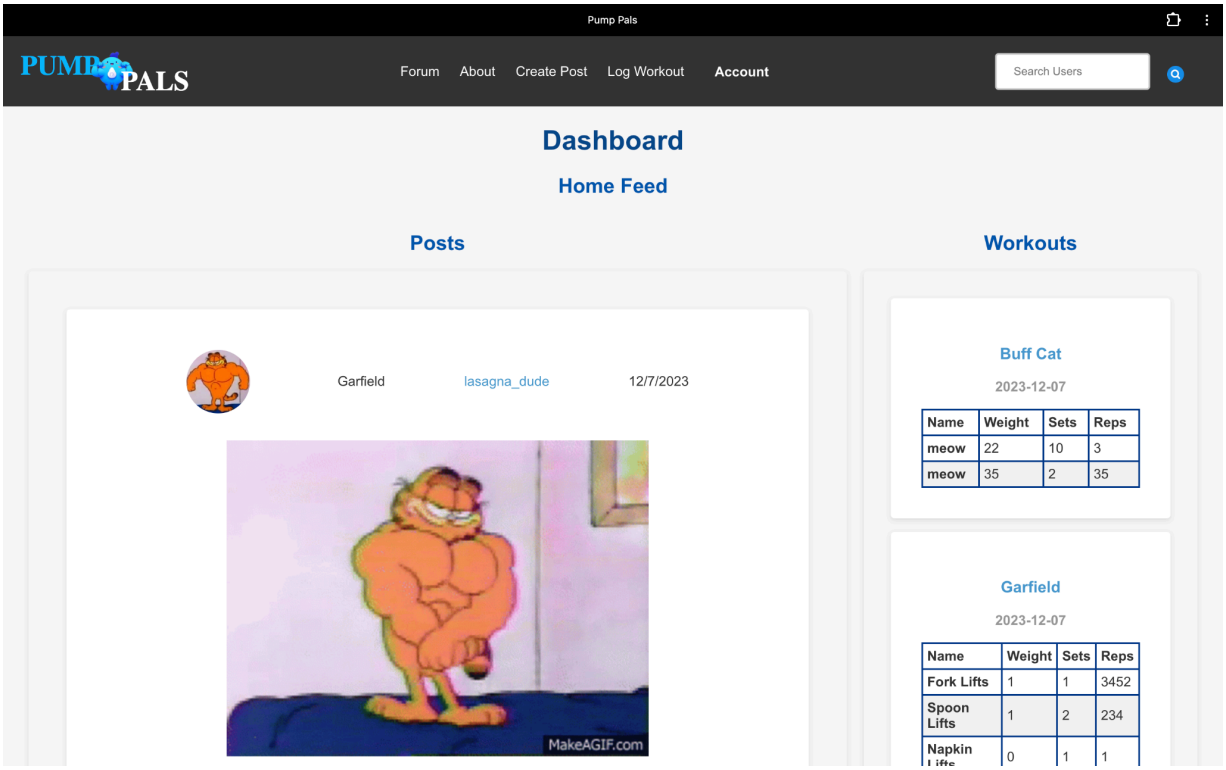
Page	Description
	
Login Form	Authenticate the user and save the returned token to local storage. If a valid token is found in local storage then the login page is bypassed. Tokens are refreshed every thirty minutes as the user interacts with the site so that the user is not automatically signed out after the token expires. This form also contains a link to the create account form.



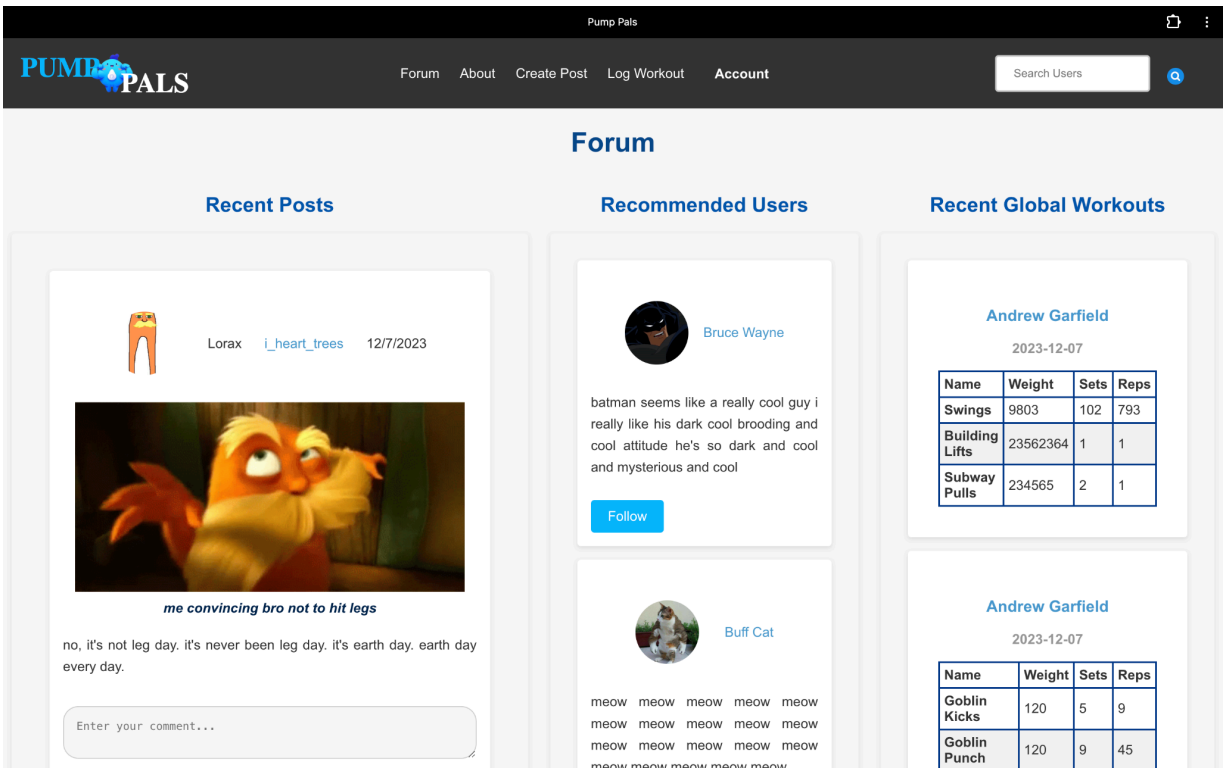
Create Account	Create the user's account. This page will record all information and statistics the user inputs such as their age, height, weight, etc. The user may also only use a username that is not already taken. This is also where a user submits their password to be saved and encrypted by the backend.



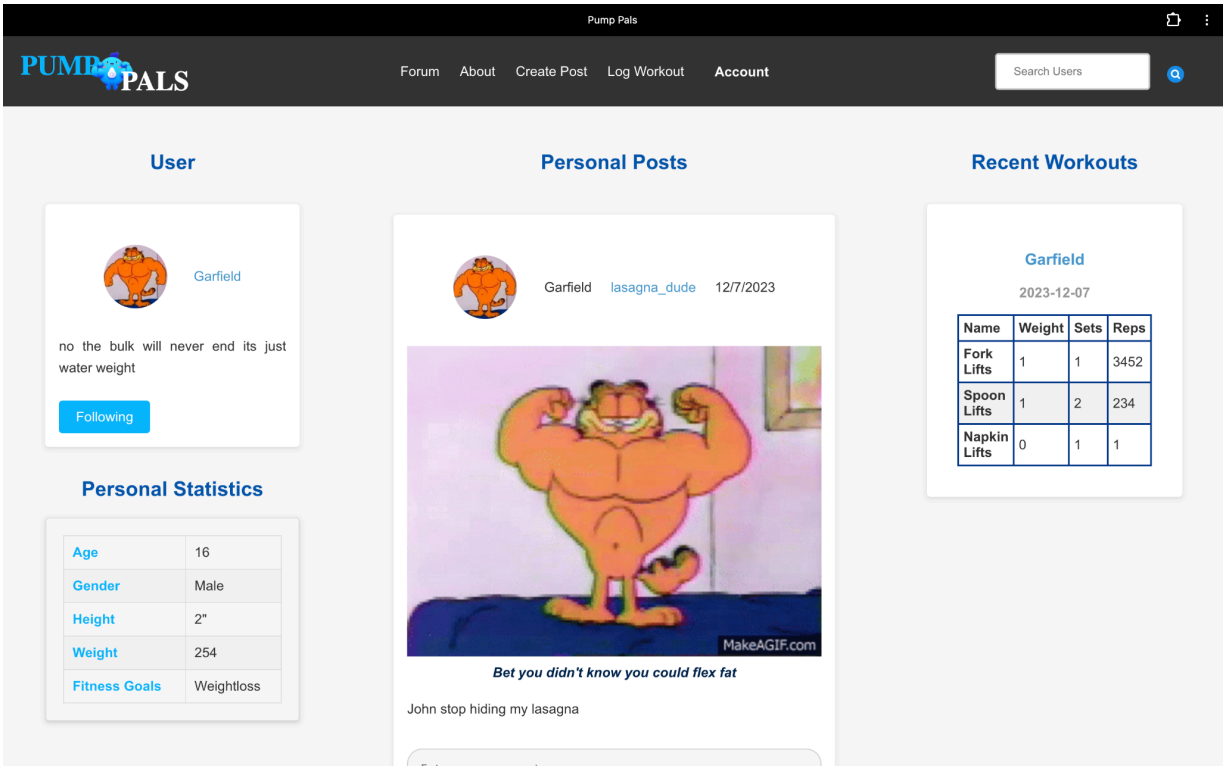
Profile	Only displays the user's profile which contains their user, stat, and post cards. This will contain an "Edit Account" button that the user may press to edit their account, and every post of theirs will have a delete button if they wish to delete it.



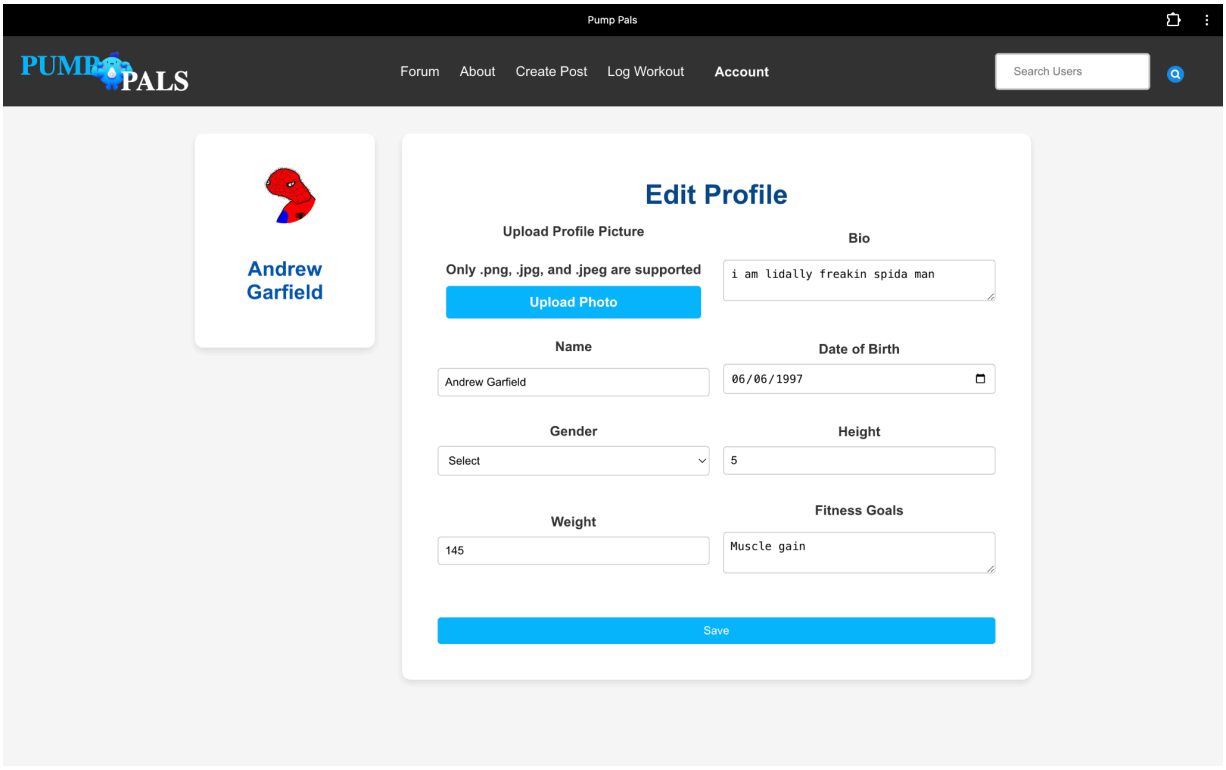
Dashboard	Only displays user content from users that the current user follows as well as themselves. This content includes post and workout cards. If the user does not currently follow anyone, then a message will show on the screen suggesting the user check out the forum to discover other users.



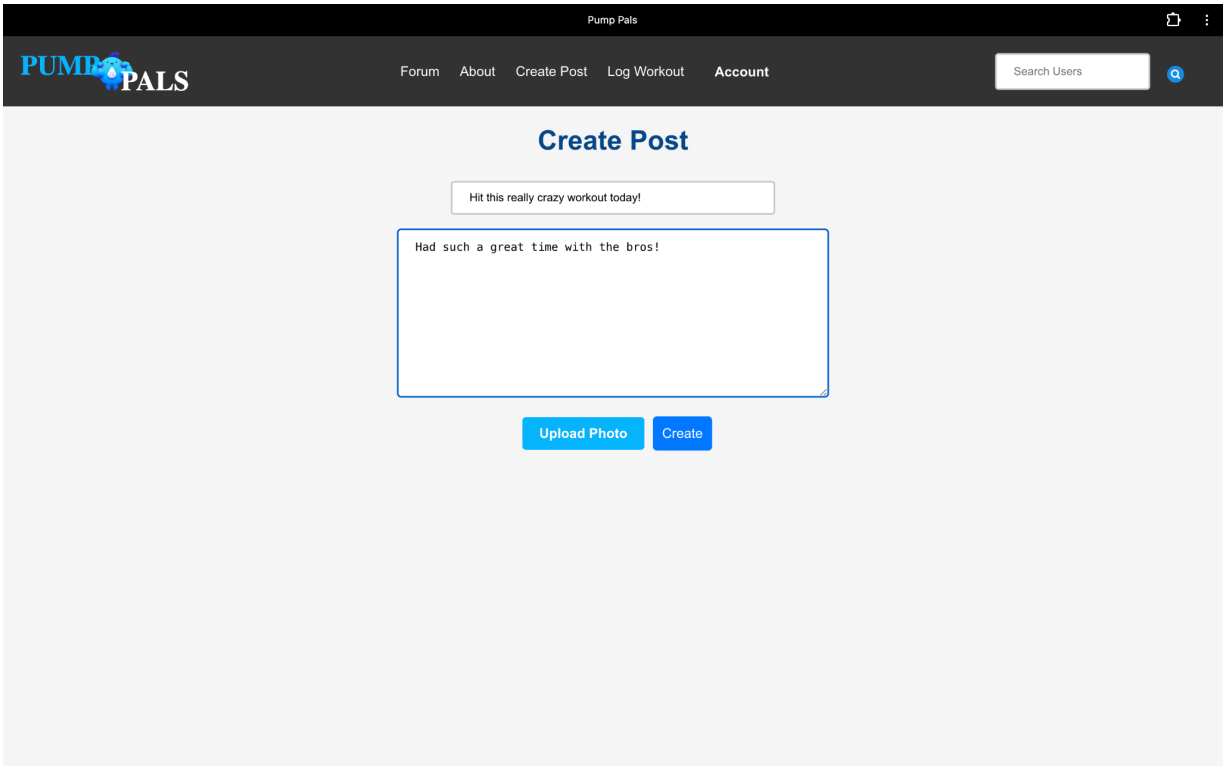
Forum	This page displays all user content including recent post, workout, and user cards. User's will find it easy to discover other users here and follow them directly from the forum page. Each card contains a link to that user's profile.



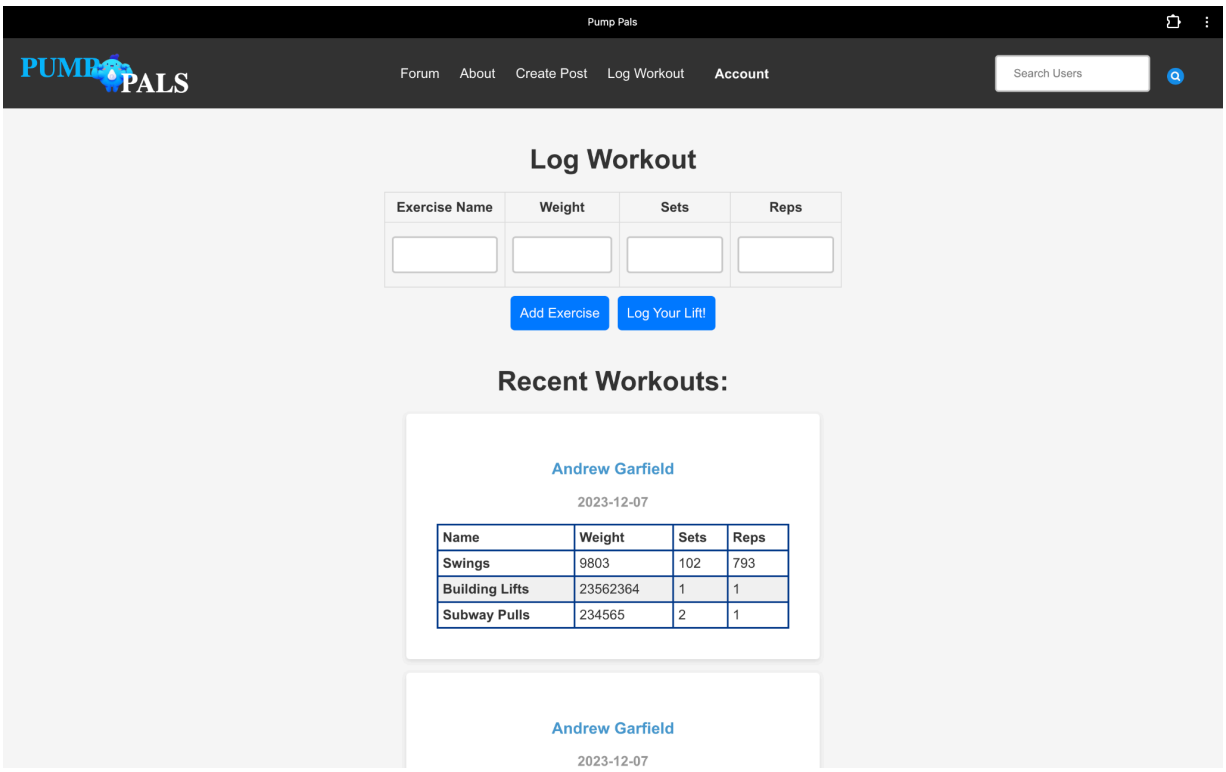
User Page	Displays the profile of the selected user as denoted by “username” in /user/{username}. This page may be navigated to via the search bar or by clicking a direct link to the user’s page from a card. This page allows the user to follow them, interact with their posts, and view their profile.



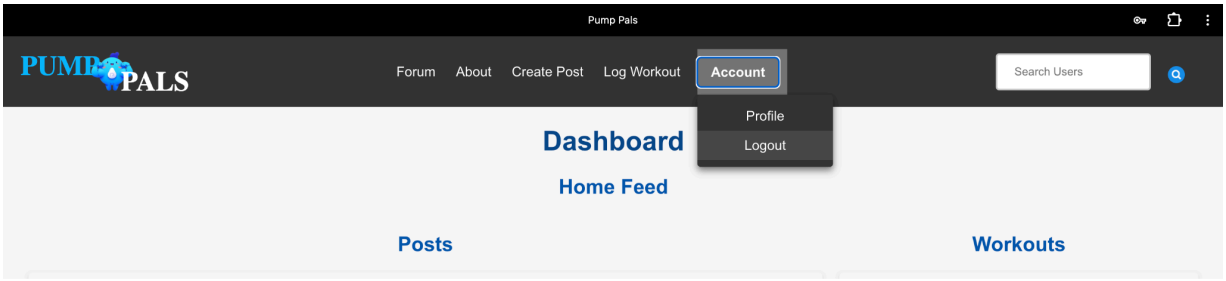
Edit Profile	Allows the user to modify their information as well as upload a profile picture. All new accounts are given a default profile picture, and this is the only way to upload a new one. All profile pictures must be in png, jpg, or jpeg format.



Create Post	Create a post that will be shown on the global forum, the user's profile, the user's dashboard, and the dashboard of users that follow the user. The user may upload media in the form of a png, jpg, jpeg, or gif. The user is also given the option to input a title and content for their post.



Log Workout	Log a workout that will be shown in the log workout page, global forum, the user's profile, the user's dashboard, and the dashboard of users that follow the user.



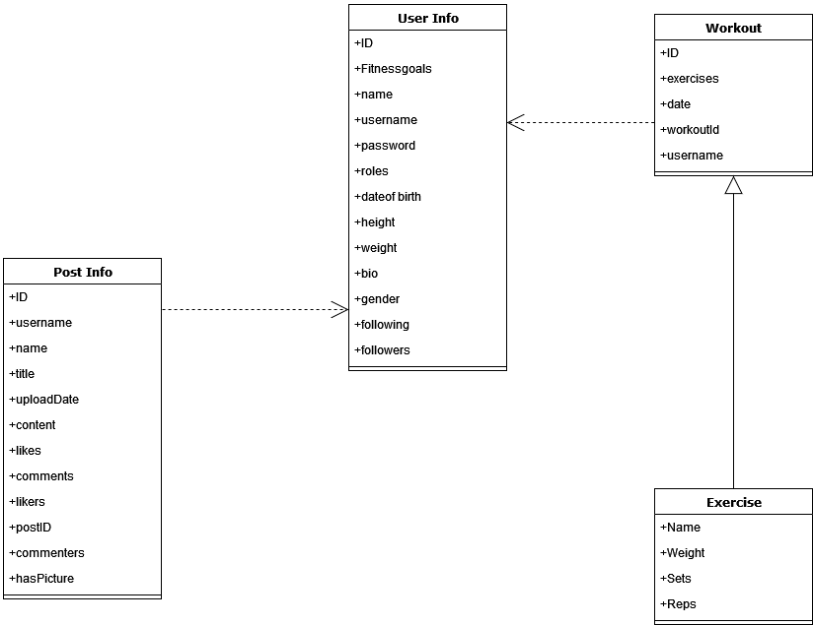
Logout	Not a page with a graphical interface. Simply deletes the user's token and routes to the login form.

With that, the general architecture of the software is laid out. Further details and specifications will be covered in the programming analysis section of this document. In conclusion, this software has three major components: The cloud database and S3 bucket, Spring Boot backend, and React frontend. Each component interacts only with its nearest neighbor

(Database/Cloud Storage < – > Backend < – > Frontend) and the application as a whole requires authentication at every step other than logging in and account creation.

Data Management

In the system modeling section the flow of data for the backend of our application is covered, and here will be more comprehensive insight into how data is managed throughout the application.



Pictured above is a diagram covering the structure of the application's database. Modeled after the Java model classes, there are four collections: UserInfo, PostInfo, WorkoutInfo, and ExerciseInfo. One user can have many posts and workouts, and one workout can have many exercises. Additionally, each user is uniquely identified by their username, given that a username may only be used once. Each post and workout has a unique string ID that is generated using the Object ID that is generated whenever a document is inserted into a collection in MongoDB. After

the document is inserted into the database, it is then re-obtained so that its Object ID may be converted to string and re-saved. There is no exercise collection in the database, instead a workout contains a list of exercise objects in its exercise array list variable. This database is only accessible via authenticated API requests or by an administrator with the correct connection string (which would contain the database's password).

POST


```
{
  "_id": {
    "$oid": "6571603911a63c467094147b"
  },
  "username": "i_heart_trees",
  "name": "Lorax",
  "title": "me convincing bro not to hit legs",
  "content": "no, it's not leg day. it's never been leg day. it's
earth day. earth day every day.",
  "uploadDate": {
    "$date": "2023-12-07T06:03:36.812Z"
  },
  "likes": 4,
  "comments": [

"{\"comment\":\"real\",\"commenter\":\"PumpPalsLuvr29\"
}"
  ],
  "commenters": [
    "PumpPalsLuvr29"
  ],
  "likers": [
    "i_heart_trees",
    "cleanairguy1",
    "PumpPalsLuvr29",
    "PumpPalsLuvr29"
  ],
  "postId": "6571603911a63c467094147b",
  "hasPicture": true,
  "pictureName":
"4e316fb57c09c059e7735fddcb0bde0d.gif",
  "_class": "com.pumppals.pumppalsapi.model.PostInfo"
}
```


USER


```
{
  "_id": {
    "$oid": "65715fbb11a63c4670941479"
  },
  "fitnessGoals": [
    "Sports",
    "General health"
  ],
  "name": "Lorax",
  "username": "i_heart_trees",
  "password":
"$2a$10$lPOoSFkju/GKOhlImdkx6eaNchqyp1duAAy42IQ
/f3r6aYVQgjqDK",
  "roles": "ROLE_USER",
  "dateOfBirth": "1926-02-02",
  "height": 2,
  "weight": 32,
  "bio": "i speak for the trees, they do be talkin",
  "gender": "Other",
  "following": [
    "Carlos",
    "etims334",
    "PumpPalsLuvr29",
    "NotBatman45",
    "lasagna_dude",
    "cleanairguy1",
    "my_mom17"
  ],
  "followers": [],
  "_class": "com.pumppals.pumppalsapi.model.UserInfo"
}
```


WORKOUT


```
{
  "_id": {
    "$oid": "6571623511a63c467094147d"
  },
  "exercises": [
    {
      "name": "Bench Press",
      "weight": 536,
      "sets": 10,
      "reps": 3
    },
    {
      "name": "Deadlifts",
      "weight": 732,
      "sets": 3,
      "reps": 10
    },
    {
      "name": "Squats",
      "weight": 1092,
      "sets": 5,
      "reps": 5
    }
  ],
  "date": {
    "$date": "2023-12-07T05:00:00Z"
  },
  "username": "i_heart_trees",
  "workoutId": "6571623511a63c467094147d",
  "_class": "com.pumppals.pumppalsapi.model.Workout"
}
```


Above is an example of what a document looks like in the database. This user's username is i_heart_trees and to the left and right are a post that they've posted as well as a workout that they've logged. Notice that the password is encrypted; all user passwords are encrypted using BCrypt and only able to be unencrypted using the application's authentication manager.

As for user submitted media, as previously mentioned it is only able to be accessed by either an administrator with AWS root privileges or by the API's IAM service account, which requires both an access key and secret key. AWS provides robust security and will automatically notify administrators any time these keys become public, and will automatically quarantine

privileges. These keys are to be routinely rotated for best security practices, and only may be stored in a .env file that is never made public (this is the same case for the MongoDB access information).

Programming Analysis

For the programming analysis, given the extensive amount of code that was written for this project the best approach is to go into detail on how the code works and the source code itself will be found in this zip file along with this document. To begin, below is a list of the API endpoints as well as the function they serve:

/api/posts [GET]	Get a list of all posts, sorted by most recent
/api/posts/{username} [GET]	Get a list of all posts by a user
/api/posts/comment/{id} [PUT]	Comment on a post
/api/posts/create [POST]	Create a post
/api/posts/create/picture [POST]	Create a post that contains a picture
/api/posts/delete/{id} [DELETE]	Delete a post
/api/posts/following [GET]	Get a list of all posts from all users the user follows
/api/posts/like/{id} [PUT]	Like a post
/api/posts/picture/{id} [GET]	Get a post's picture
/api/posts/uncomment/{id} [PUT]	Delete a comment on a post
/api/posts/unlike/{id} [PUT]	Unlike a post
/api/posts/update [PUT]	Update a post
/api/refresh [GET]	Refresh the user's saved token
/api/create [POST]	Create a new user account
/api/authenticate [POST]	Authenticate a user using their username and password, returns a string JSON Web Token to be saved in the user's local storage
/api/pfp [PUT]	Upload a user's profile picture
/api/pfp/{username} [GET]	Get a user's profile picture, cropped to a square aspect ratio
/api/user/{username} [GET]	Get the user's information via their username
/api/user/checkfollow/{username} [GET]	Check if the user is following another user
/api/user/follow/{username} [PUT]	Follow a user
/api/user/recommended [GET]	Get a list of recommended users
/api/user/unfollow/{username} [PUT]	Unfollow a user
/api/user/update [PUT]	Update a user's profile information
/api/user/username [GET]	Get the current user's username from their token
/api/workout/{username} [GET]	Get a list of a user's recent workouts
/api/workout/{username}/{date} [GET]	Get a list of a user's workouts from a specific day
/api/workout/all [GET]	Get a list of all user's workouts, sorted by most recent

/api/workout/create [POST]	Create a post
/api/workout/date/{date} [GET]	Get a list of all user's workouts for a certain date
/api/workout/following [GET]	Get a list of workouts from users that the user follows
/api/workout/id/{workoutld} [GET]	Get a specific workout by its workoutID
/api/workout/user [GET]	Get a list of the user's recent workouts

All of the above endpoints may be found in the UserController.java, PostController.java, and WorkoutController.java classes. Requests to these endpoints are made to the frontend using Axios to build HTTP requests. These request functions may be found in ServerConnector.js in the frontend's source code. Below is the function that is used to create authentication headers using the user's token:

```
const createAuthHeader = async (otherHeaderInfo) => ({
  checkLock: await checkLock(),
  headers: {
    ...(otherHeaderInfo || {}),
    Authorization: `Bearer ${localStorage.getItem("token")}`,
  },
});
```

Once a request is received, the backend will first check to see if there is a token after the string "Bearer," and the request may only pass to a protected endpoint if a valid one is present. Below is an example of how a HTTP API request is built, as well as an example of the path a request may take throughout the full application:

```
export async function commentPost(id, comment) {
  try {
    await Axios.put(
      `/api/posts/comment/${id}`,
      comment,
      await createAuthHeader()
    );
  } catch (error) {
    console.error("Failed to comment on post:", error);
```

```
    return error;
  }
}
```

The above function is used to build a request to comment on a post, and accepts the post's ID and the comment string as parameters. This is a put request with the endpoint defined first (having the ID as a request parameter), then the body (in this case being the comment String), and then the authentication header is created (the await keyword tells the program to wait for the function to complete before continuing). The above function is a void method, so it does not need to return any information, and only needs to print to the console and return an error if an exception occurred. This specific function is used in PostCard.js, where the post card object is defined:

```
const handleCommentSubmit = () => {
  if (comment.trim() === "") {
    return; // Do not submit blank comments
  }

  const newComment = {
    comment: comment,
    commenter: user,
  };
  commentPost(postId, newComment);
  setAllComments([...allComments, newComment]);
  setComment("");
};
```

This function is called when the user clicks on the comment button after inputting their comment in the text area, as defined here in the HTML code:

```
<div id="comment">
  <button
```

```
            className="comment-button"
            onClick={handleCommentSubmit}
          >
            Comment
          </button>
        </div>
```

Before the request is executed by the backend, it must first be authenticated using its bearer token by the authentication filter. Every request is authenticated using the below function in JwtAuthFilter.java:

```
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws
ServletException, IOException {
        String authHeader = request.getHeader("Authorization");
        String token = null;
        String username = null;
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            token = authHeader.substring(7);
            username = jwtService.extractUsername(token);
        }

        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails = userDetailsService.loadUserByUsername(username);
            if (jwtService.validateToken(token, userDetails)) {
                UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(userDetails, null,
userDetails.getAuthorities());
                authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }
        filterChain.doFilter(request, response);
    }
```

This method in the authentication filter classes calls upon the JwtService class, which handles the generation and validation of tokens. The authentication filter uses the two following method in JwtService.java:

```
public String extractUsername(String token) {
    return extractClaim(token, Claims::getSubject);
}
public Boolean validateToken(String token, UserDetails userDetails) {
    final String username = extractUsername(token);
    return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
}
```

Once this function is called and properly authenticated by the backend's filter, it is met by the following function in PostController.java:

```
@PutMapping("/posts/comment/{id}")
public ResponseEntity<?> commentPost(@PathVariable String id, @RequestBody String comment, Principal principal) {
    ObjectId objectId = new ObjectId(id);
    postService.commentPost(objectId, principal.getName(), comment);
    return ResponseEntity.ok().build();
}
```

This method then calls the PostService class in PostService.java to execute the commentPost() method:

```
public void commentPost(ObjectId id, String username, String comment) {
    PostInfo post = postRepository.findById(id).get();
    post.getComments().add(comment);
    post.getCommenters().add(username);
    postRepository.save(post);
}
```

This method then calls the PostRepository's save() method, which does not need to be explicitly defined in the code since it is defined by MongoRepository's interface, which

PostRepository's interface extends. From here the comment is successfully saved in the database, and no return value needs to be passed so if no exception occurs then a 200 status may be sent back.

Additionally, the frontend code does also perform computations to further process user information, as well as validate inputs before they are sent to the API. For example all fields must be filled out in any given form and the user's username must not contain any special characters. An example of the frontend's code processing user information is when the application determines whether to display a follow button or edit profile button. To avoid excess code, instead of creating an entirely different card just to display user information, a simple check is made when a user card is loaded to see if the card belongs to the user that is logged in. This is done using the user's token which is sent to the backend using getUsername() from ServerConnecor.js and returns the user's username. If the username belonging to the card is equal to the returned username from the API, then isCurrentUser is set to true and the edit profile button is displayed rather than a follow button. This implementation may be seen in the below code which executes as soon as a card is rendered:

```
useEffect(() => {
  const fetchProfilePicture = async () => {
    let imageUrl = await getProfilePicture(username);
    if (imageUrl === null) {
      imageUrl = process.env.PUBLIC_URL + "/account_icon.svg";
    }
    setProfilePicture(imageUrl);
  };

  const checkFollowStatus = async () => {
    setFollowed(await isFollowing(username));
  };

  const checkFollowStatus = async () => {
    setFollowed(await isFollowing(username));
  };

  const checkCurrentUser = async () => {
    const response = await getUsername();
    const user = response.data;
    if (user === username) {
      setIsCurrentUser(true);
    }
  };

  fetchProfilePicture();
  checkFollowStatus();
  checkCurrentUser();
}, [username]);
```

In addition to JavaScript being used for the page's functions and HTML being used to format the page, CSS has also been utilized to stylize the application and provide additional formatting rules. Provided below is the general CSS that is applied to all buttons, unless overridden by a className:

button {
  padding: 10px 20px;
  border: none;
  background-color: #007BFF;
  color: white;
  cursor: pointer;
  border-radius: 5px;
  margin-top:10px;
}
	Defines the margins, padding, color, etc. for every button unless overridden

button:not(:last-child) {
  margin-right: 10px;
}
 	
button:hover {
  background-color: #0056b3;
}
	If the button is not the last child, add a 10px margin to the right, and darken the color when highlighted

This programming analysis has covered several examples to demonstrate the flow of data and implementation of the system model and requirements. Further information can be gathered from the source code itself and the comments provided within.

Testing & Use Cases

Use Case	**Input**	**Output**
Make a profile	User information	Profile page showing the users newly created profile
Create a post	Post information, optional media	Profile page showing newest post at the top
Log a workout	Exercises	Newly logged workout at the

		top of the list of recent workouts
Comment on a post	Comment	Comment under post
Like/unlike post	User click on like button	Post is liked/unliked
Login	Username and password	User's dashboard
Navigation to any page in the navigation bar	User click on link to page	The requested page
Logout	User click on logout button	Logs out the user, sends them to login form
Follow/unfollow a user	User click on follow button	Text changes from follow to following, depending on current following status
Click on link to user profile	User click on profile link	Requested user's profile
User search bar	Desired profile's username	Requested user's profile
Delete a post	Click on post's delete button	Post deleted confirmation
Upload profile picture	New profile picture	Profile picture on the left side of the page changes to new profile picture
Edit profile	New profile information	Profile page displaying new profile information
Dashboard	Navigation to page	List of posts/workouts from user and users the user follows
Forum	Navigation to page	List of posts/recommended users/workouts from all users
Profile	Navigation to page	Users information, statistics, posts, and workouts
Log Workout	Navigation to page	Form to log a new workout and recent workouts
About	Navigation to page	About summary for Pump Pals

Appendices

Appendix 1: Hours Spent

	Ariya Seng		Carlos Negretes		Kelvin Tims		Michael Liam Sinclair		S M Raisul Alam Bhuiyan		Tom Tran	
Weekly Updates	Hours	%	Hours	%	Hours	%	Hours	%	Hours	%	Hours	%
Week of 8/28	10	16	10	16	10	16	10	18	10	16	10	18
Week of 9/4	10	16	10	16	10	16	10	18	10	16	10	18
Week of 9/11	10	16	10	16	10	16	10	18	10	16	10	18
Week of 9/18	10	16	10	16	10	16	10	18	10	18	10	16
Week of 9/25	10	16	10	16	10	16	10	18	10	16	10	16
Week of 10/2	10	16	10	16	10	16	10	18	10	16	10	16
Week of 10/9	10	18	10	16	10	16	10	16	10	16	10	16
Week of 10/16	10	16	10	18	10	16	10	24	10	8	10	16
Week of 10/23	10	16	10	16	10	16	10	24	10	8	10	16
Week of 10/30	10	16	10	16	10	16	10	24	10	8	10	16
Week of 11/6	10	16	10	16	10	18	10	18	10	16	10	16
Week of 11/13	10	16	10	16	10	16	10	24	10	8	10	18
Week of 11/20	10	16	10	16	10	16	10	24	10	8	10	18
Week of 11/27	10	16	10	16	10	16	10	24	10	8	10	18
Week of 12/4	Presentation + Final Report											

Appendix 2: Requirement Checklists

Submitted with this document as Requirements Checklist Fall 2023.xlsx

Appendix 3: Program Code and Executable

Included in the root directory of this submission, separated between backend (with compiled frontend) and frontend source code.

PumpPals folder: backend source code with compiled frontend
PumpPals_React folder: frontend source code

An executable of the software can be found as PumpPals.jar and must meet the following requirements to run:

- Java 17
- No VPN currently running
- Preferably run on MacOS or Linux

The executable will start a local server hosting the application
The application itself can be viewed at http://localhost:8080 after the application loads
Use the command "java -jar PumpPals.jar" to run the application and wait for the confirmation command

Appendix 4: Project Status Reports

For our status reports, we have chosen to put them into separate files that are part of the final report themselves.

Appendix 5: Test Cases

Test	**Successful/ Unsuccessful**	**Comments**
User can create an account	*Successful*	Must fill out all fields
User can upload a profile	*Successful*	Must be png, jpg, or jpeg

picture		
User can create a post	*Successful*	
User can upload media to a post	*Successful*	Must be png, jpg, or jpeg
User can follow/unfollow other users	*Successful*	
Users cannot edit information of other users or delete their posts/workouts	*Successful*	
User's dashboard is a feed of users they follow and their own posts	*Successful*	If the user doesn't follow anyone a message displays encouraging them to check out the forum
Global feed shows posts, users, and workouts	*Successful*	From all users, including the current users and users they follow
User can log workouts	*Successful*	User can add as many exercises as they want, as long as all fields other than name are numbers only
User can't access without being logged in	*Successful*	User will be routed to login page if not logged in
Logout logs out the user	*Successful*	Deletes token and routes to /login
User can click user links to open their profile	*Successful*	Fixed bug where /user/{username} did not re render the user page
User can search other users by their username	*Successful*	Fixed same bug as above
Users can like/unlike posts and number of likes displays	*Successful*	Liked posts have darkened like button
Users can comment on other users' posts	*Successful*	Comment field cannot be blank
Usernames can only be taken once	*Successful*	Error message will show if user tries to take taken username
All media loads when page is loaded	*Successful*	Large media may take some time to load

References

Sources

Source 1: All of our sources can be found at our GitHub, which you can see here at:

Frontend: https://github.com/tommytran9/PumpPals_React

Backend: https://github.com/mliamsinclair/PumpPals