COMP3006 Full-Stack Development CW2 Report

Project Choice	Social Media System
GitHub Link	https://github.com/Plymouth-University/coursework-corey-richardson
YouTube Link	https://youtu.be/6XcH9gGSHEw

Requirements

Technical Requirements	**Achieved?**
Frontend client constructed using dynamic web technologies: HTML, CSS, JavaScript, **React** or Angular	Yes
At least 1 backend Node.js server	Yes
At least 1 database (**MongoDB** or PouchDB)	Yes
WebSockets to give the appearance of communication between multiple clients.	Yes
The system must be interactive	Yes
The system must be distributed, capable of running on multiple computers (containers)	Yes

For my Functional and Non-Functional Requirements, I used the MoSCoW prioritisation technique.

Functional Requirements
Must-Have
Authentication and Authorisation to protect
CRUD operations via RESTful API: user must have the ability to create, read, update and delete posts (with sufficient permission levels).
A WebSocket connection which broadcasts real-time updates when new posts are created, or when posts are updated (post edited, post deleted, like/comment count changes)
Should-Have
Different types of feed: global and following. This allows the user to personalise their experience.
Social interactions such as the ability to like or comment on a post; engagement metrics.
Could-Have
Profile customisations such as being able to edit a profile bio.
Won't-Have
To avoid scope creep, I am not implementing images/media to posts. Posts should be text-based only.

Non-Functional Requirements
Must-Be
Distributed across multiple containers
A MongoDB based database for data persistence; Mongoose as ORM?
Interactive, allowing the user to be able to affect the behaviour of the system through input.
Should-Be
A RESTful API design. Resources and subresources should be notated properly (e.g. **/api/posts/:postId/comments/:commentId** as comments are a subresource of posts, despite being their own entity.
The authentication system should use locally stored JWT tokens to manage authentication states.

Some of the initial project requirements were visualised using **low-fidelity wireframe** sketches to act as a blueprint for the eventual frontend design. They also allowed me to visualise the user journey of the application before any code had been written.

Wireframes were produced for the Navbar and Authentication Form components, and the Main Feed and Profile pages. The Feed page low-fidelity design allowed for a fast implementation of the “Feed Toggle” providing a clean UI for users to switch between the Global and Following views from a single-page interface, not requiring a large re-render task and therefore providing a more seamless User Experience.

Discussing the low-fidelity wireframe for the Main Feed page with friends led to the placement of the “New Post” button alongside the feed toggles.

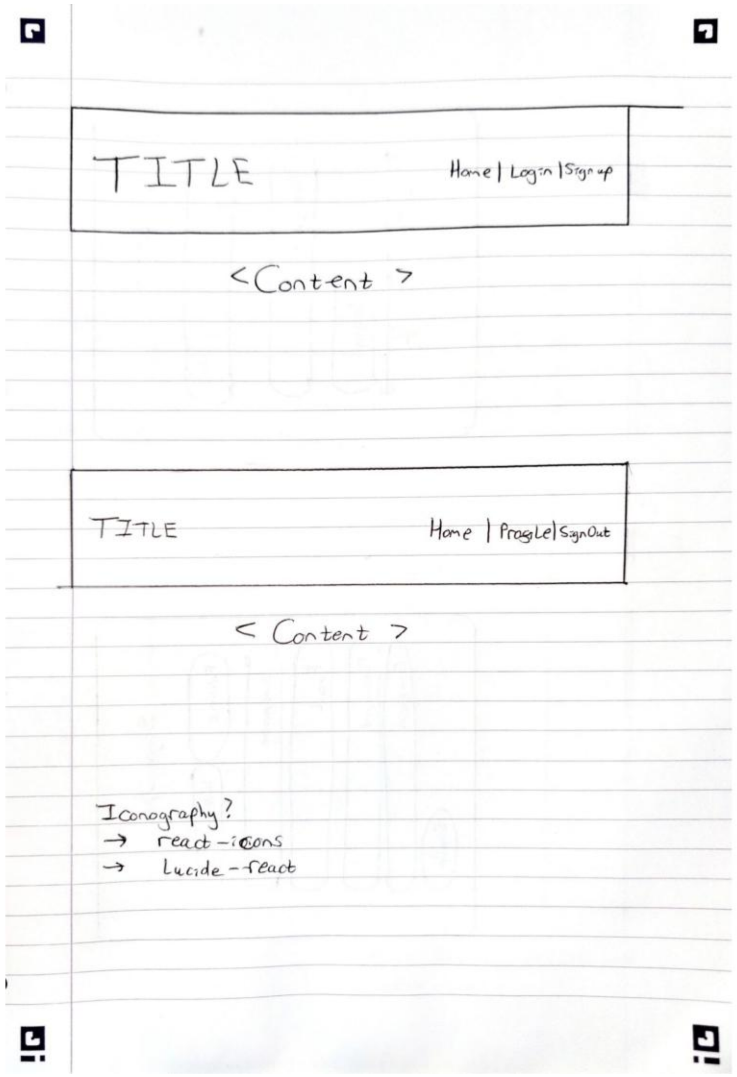


Figure 1: Low Fidelity Wireframe: Navbar Component

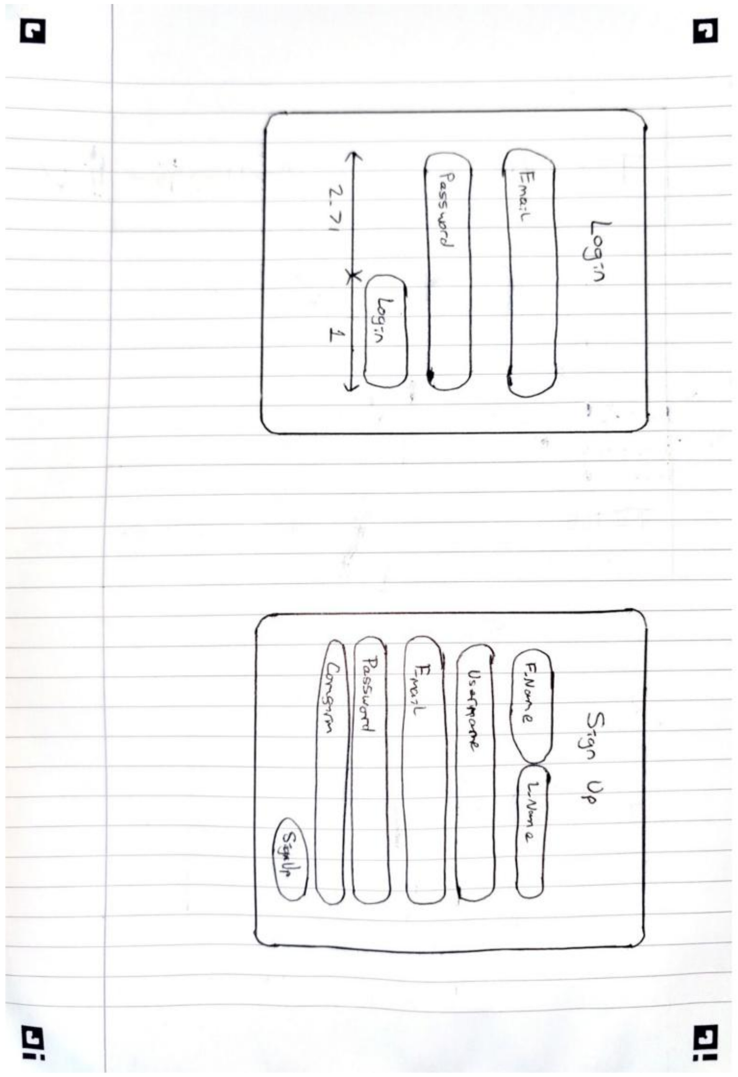


Figure 2: Low Fidelity Wireframe: Authentication Form Components

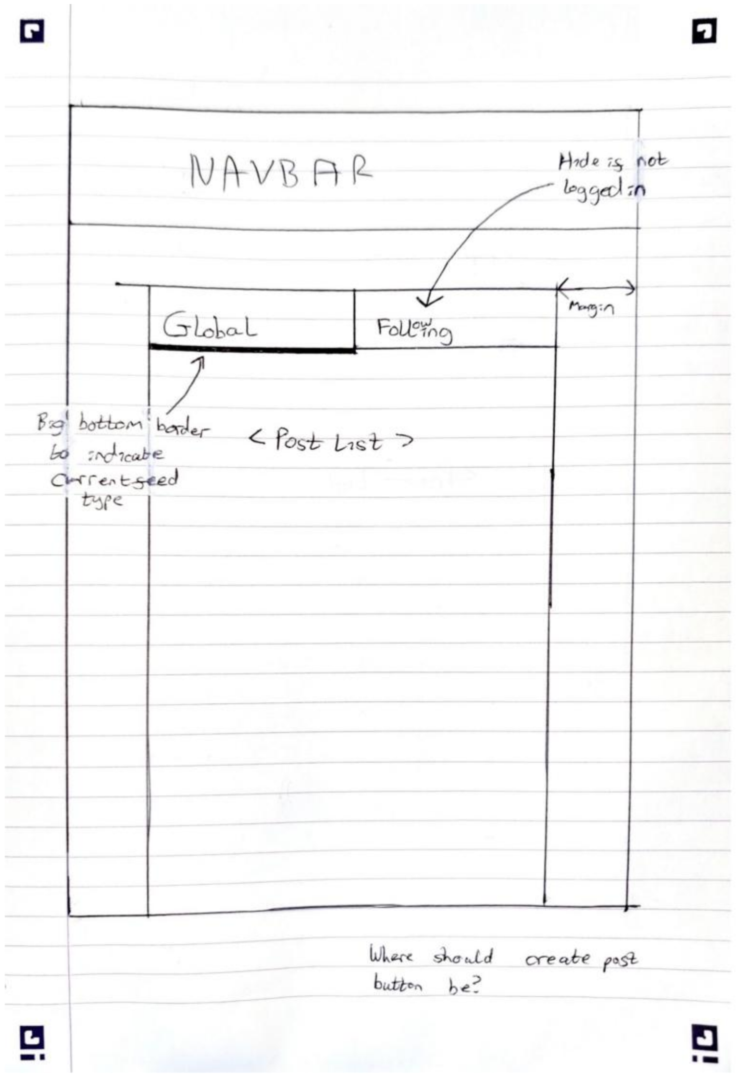

NAVBAR
Hide is not logged in
Global
Following
Margin
Big bottom border to indicate current feed type
< Post List >
Where should create post button be?

Figure 3: Low Fidelity Wireframe: Main Feed Page

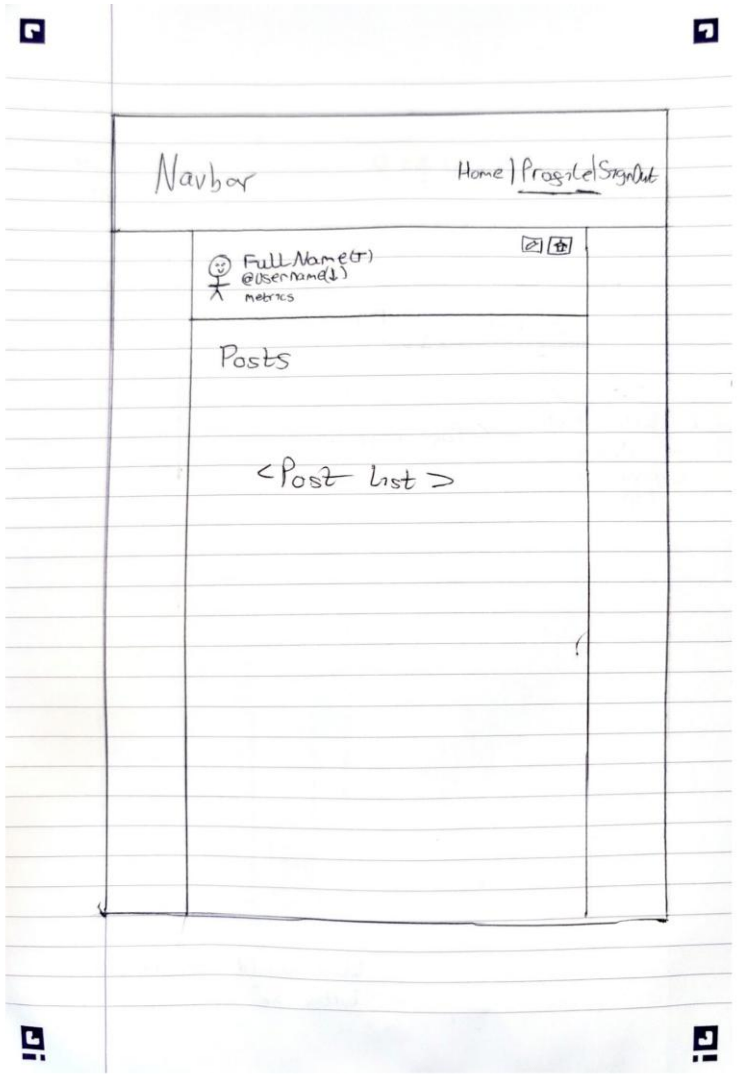


Figure 4: Low Fidelity Wireframe: Profile Page

System Design

The system using containerised microservice architecture, separating concerns between the Client, Server and Database. Containers are orchestrated as independent services using Docker.

The application follows a Model-View-Controller (MVC) design pattern:

- Model: A MongoDB database for persistent data storage
- View: A React frontend user interface, using the Context API for state management
- Controller: A Node.js and Express backend to handle logic, authentication and database operations.

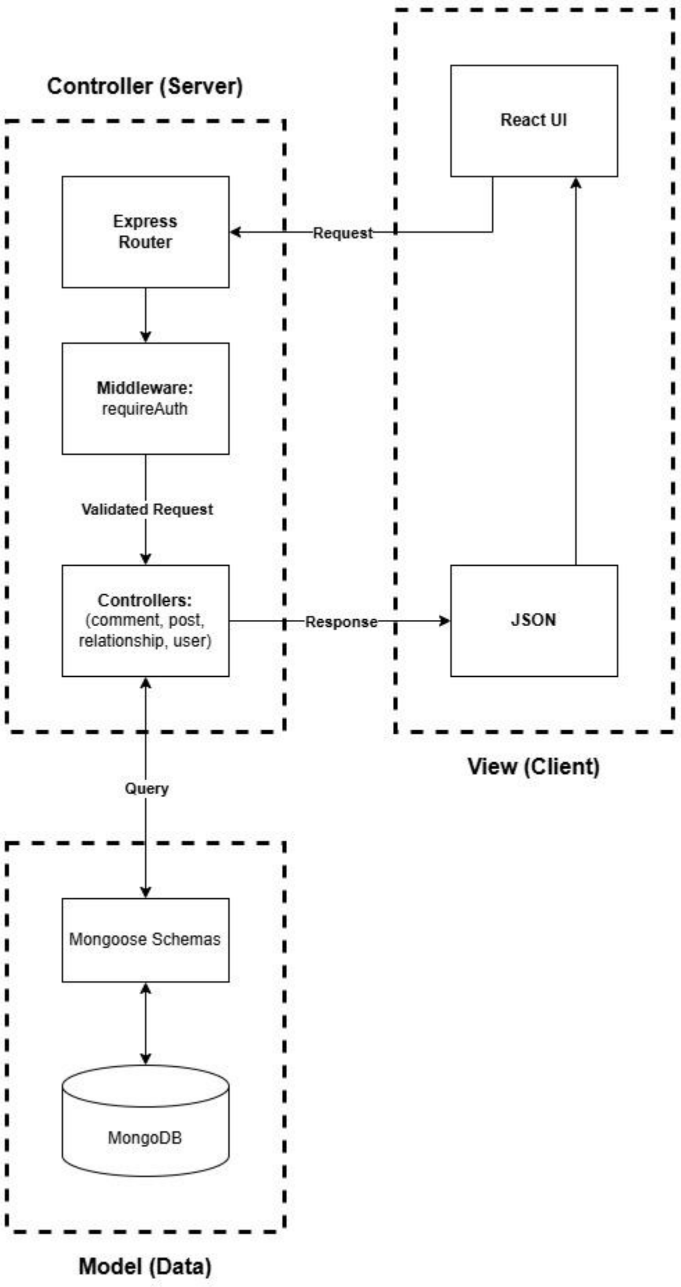


Figure 5: Model-View-Controller

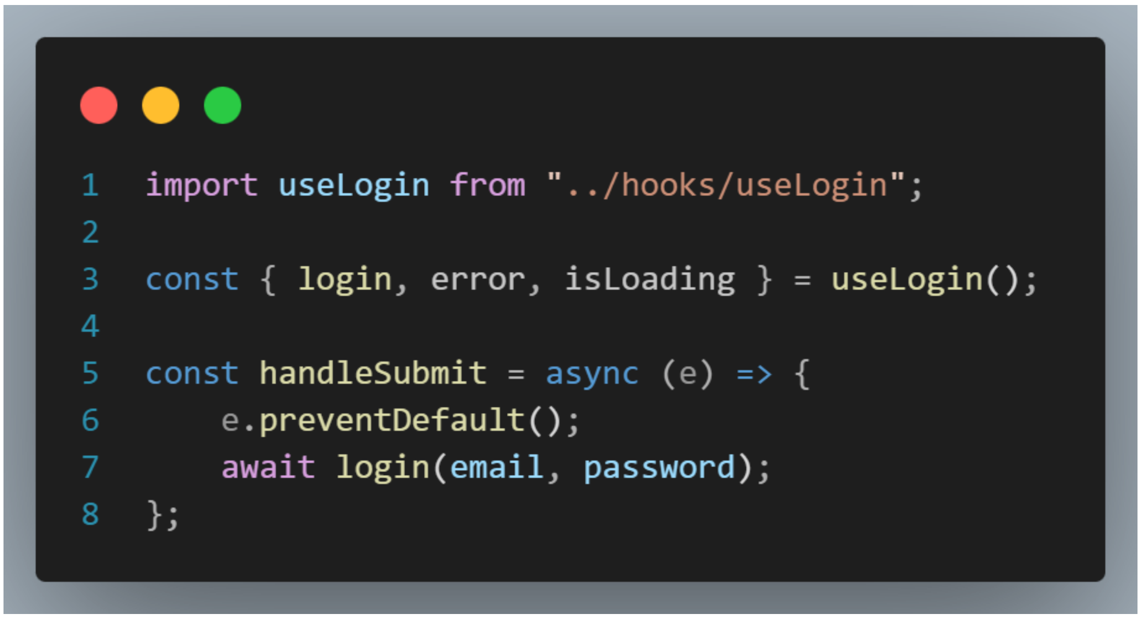
Communications between containers is done through a RESTful API to handle CRUD operations, and a WebSocket channel (via Socket.io) is used with the React Context API to handle real-time UI updates.

Contexts and Reducers to Manage State

To manage the applications global state, I implemented the React Context API with the **useReducer** hook for managing the state of **Authentication**, **Posts** and **Relationships**.

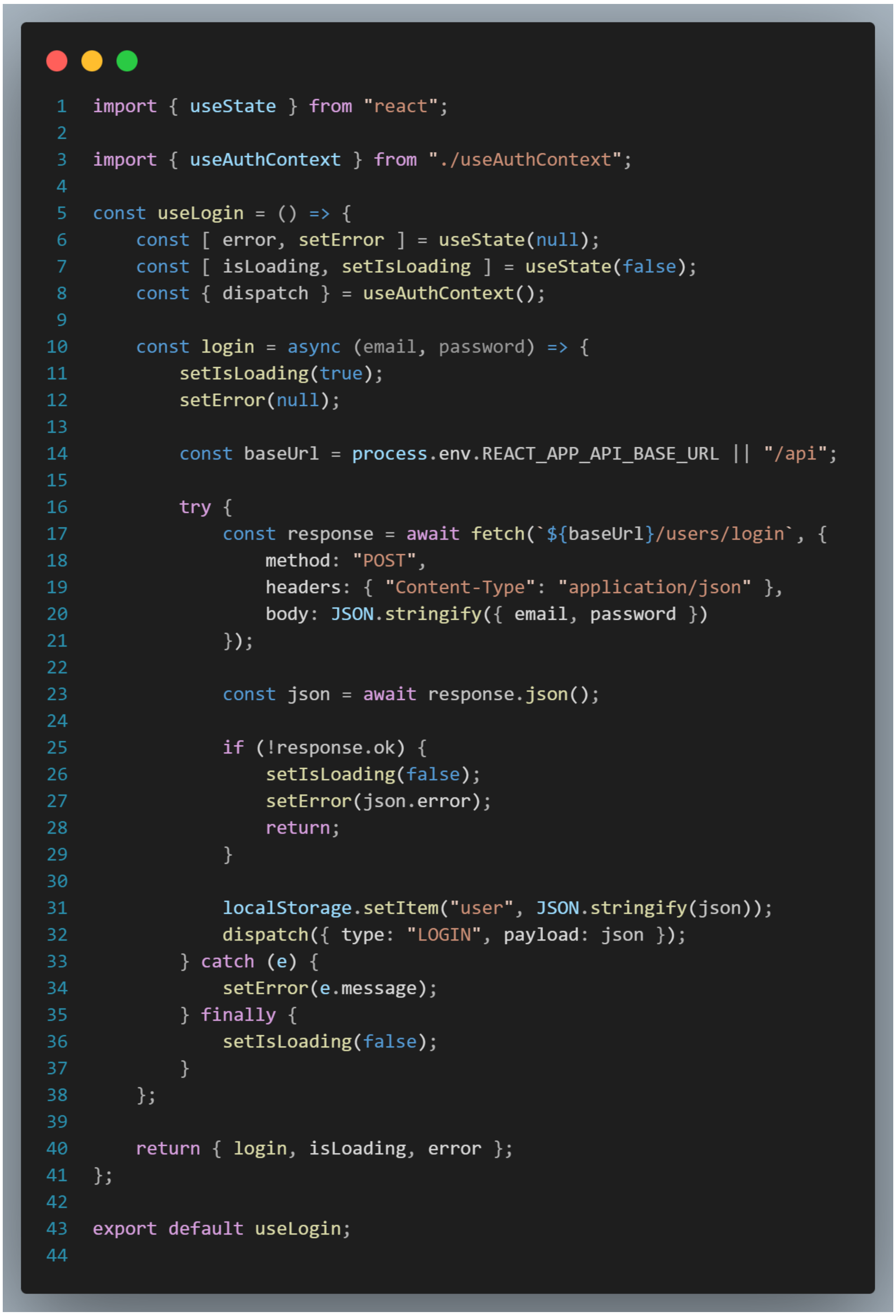
By using a reducer, state transitions are centralised: a single source of truth. A state transition is triggered by a **dispatch** action, containing a **type** and a **payload**.

This design pattern can be used to encapsulate logic away from the components that consume them. For example, the **pages/Login.jsx::handleSubmit** method uses the **login** method provided to it by the **useLogin** hook. This means that the component itself never needs to know the specific implementation details of the database call.



```
import useLogin from "../hooks/useLogin";

const { login, error, isLoading } = useLogin();

const handleSubmit = async (e) => {
    e.preventDefault();
    await login(email, password);
};
```

Figure 6: pages/Login.jsx::handleSubmit



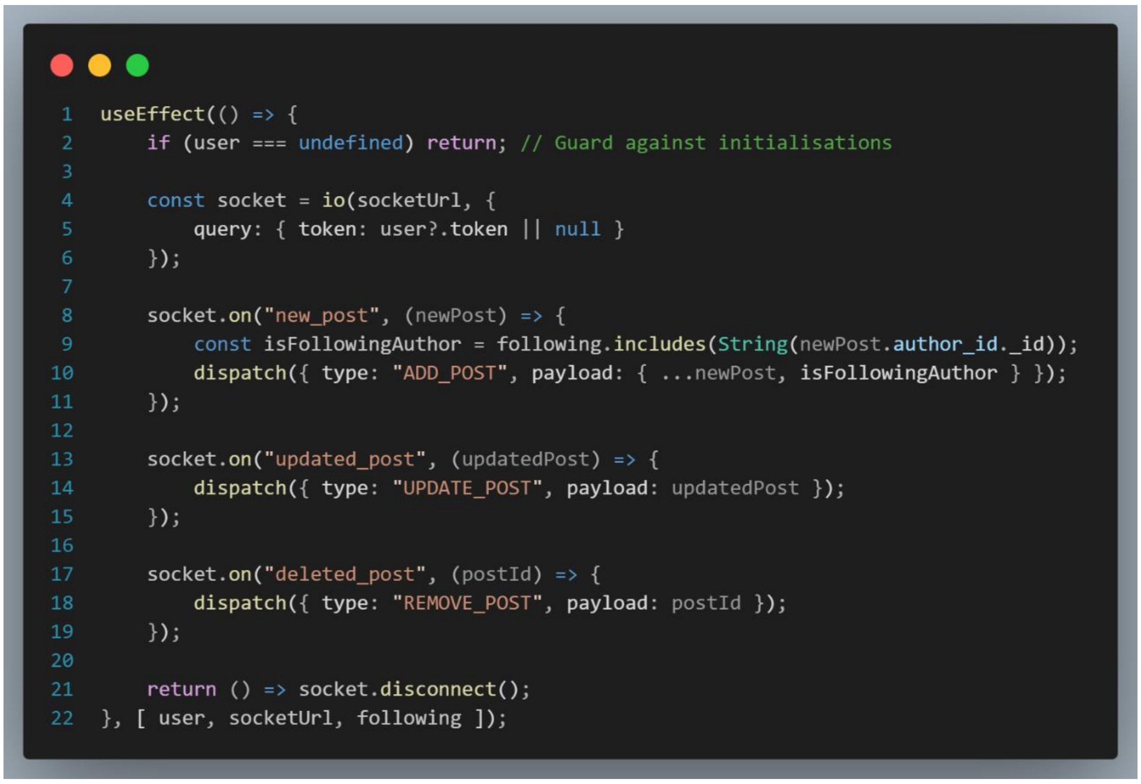
```javascript
import { useState } from "react";

import { useAuthContext } from "./useAuthContext";

const useLogin = () => {
    const [ error, setError ] = useState(null);
    const [ isLoading, setIsLoading ] = useState(false);
    const { dispatch } = useAuthContext();

    const login = async (email, password) => {
        setIsLoading(true);
        setError(null);

        const baseUrl = process.env.REACT_APP_API_BASE_URL || "/api";

        try {
            const response = await fetch(`${baseUrl}/users/login`, {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify({ email, password })
            });

            const json = await response.json();

            if (!response.ok) {
                setIsLoading(false);
                setError(json.error);
                return;
            }

            localStorage.setItem("user", JSON.stringify(json));
            dispatch({ type: "LOGIN", payload: json });
        } catch (e) {
            setError(e.message);
        } finally {
            setIsLoading(false);
        }
    };

    return { login, isLoading, error };
};

export default useLogin;

```

Figure 7: hooks/useLogin.js

The **PostContext** is another, more complex, example of where this pattern has been implemented. It handles conditional fetching of posts to populate the feed, performing filtering for the “Global” feed, only posts from users the current user is following, or posts from a specific user to show on their profile page.

This pattern also allows for pagination by having actions for **SET_POSTS** which sets the global **post** state, and **LOAD_MORE_POSTS** which instead appends the newly loaded posts to the existing **post** state.

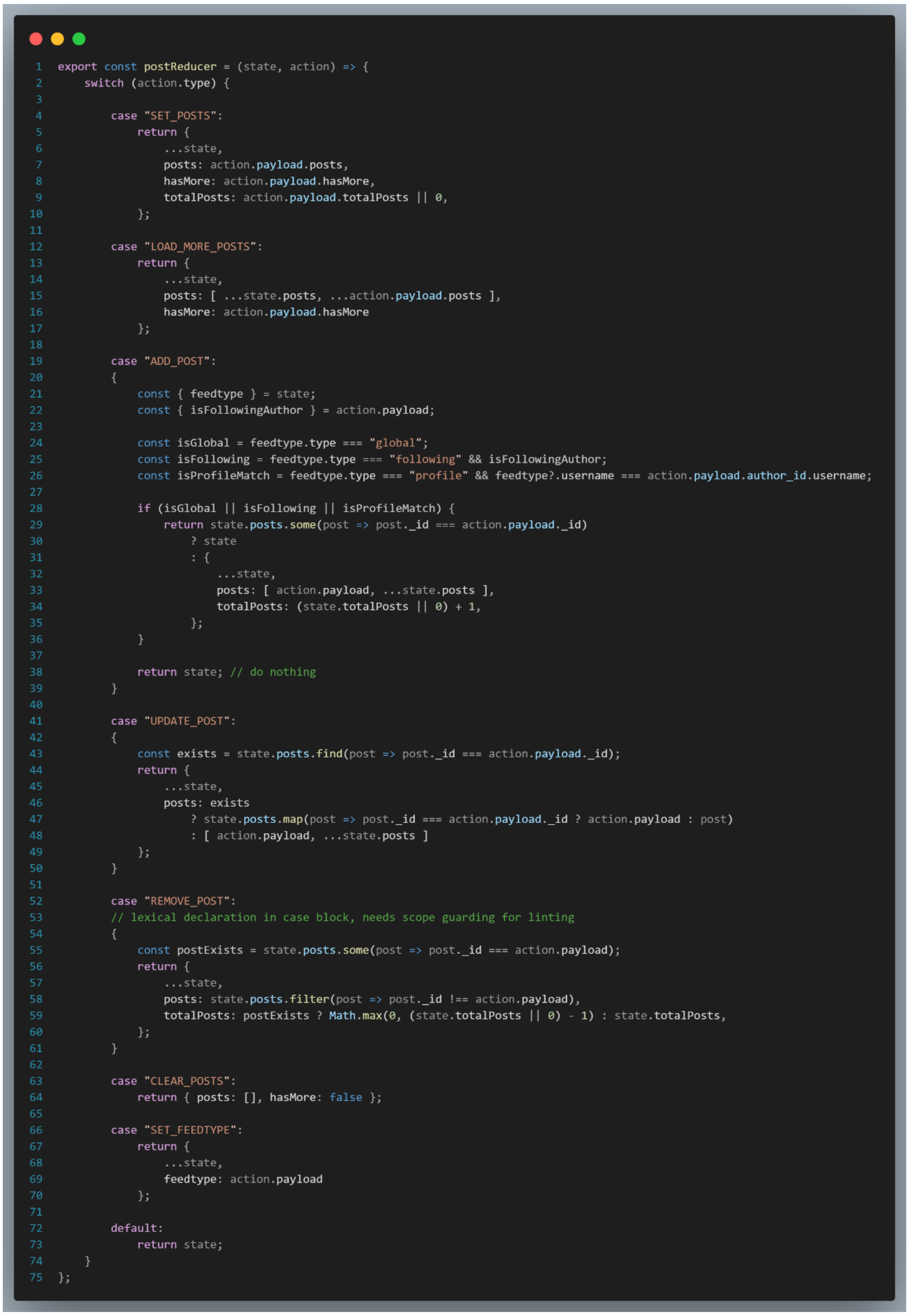
The WebSocket Lifecycle is managed within the **PostContextProvider**, by initialising a socket connection from within a **useEffect** hook to listen to broadcasts emitted by the **PostController**. These events are dispatched the reducer shown on the next page, ensuring that UI updates occur in real time across connected-clients without a page refresh.



```
useEffect(() => {
    if (user === undefined) return; // Guard against initialisations

    const socket = io(socketUrl, {
        query: { token: user?.token || null }
    });

    socket.on("new_post", (newPost) => {
        const isFollowingAuthor = following.includes(String(newPost.author_id._id));
        dispatch({ type: "ADD_POST", payload: { ...newPost, isFollowingAuthor } });
    });

    socket.on("updated_post", (updatedPost) => {
        dispatch({ type: "UPDATE_POST", payload: updatedPost });
    });

    socket.on("deleted_post", (postId) => {
        dispatch({ type: "REMOVE_POST", payload: postId });
    });

    return () => socket.disconnect();
}, [ user, socketUrl, following ]);
```

Figure 8: PostContextProvider Socket Management

During a walkthrough test, I found a bug where the newly created post was being added to the context state without checking what feed the user was on - global, following or on a specific user’s profile - causing an error where somebody else’s post could be added to your profile until there was a page refresh. As such, I moved the **feedtype** state attribute into the **PostContext** and added a guard to the reducers **ADD_POST** case to first check if the post was authored by a user that the current user follows, or if you are on their profile page (see Figure 10). This was done using the **RelationshipContext** and prevents state corruption.



```javascript
export const postReducer = (state, action) => {
    switch (action.type) {

        case "SET_POSTS":
            return {
                ...state,
                posts: action.payload.posts,
                hasMore: action.payload.hasMore,
                totalPosts: action.payload.totalPosts || 0,
            };

        case "LOAD_MORE_POSTS":
            return {
                ...state,
                posts: [ ...state.posts, ...action.payload.posts ],
                hasMore: action.payload.hasMore
            };

        case "ADD_POST":
        {
            const { feedtype } = state;
            const { isFollowingAuthor } = action.payload;

            const isGlobal = feedtype.type === "global";
            const isFollowing = feedtype.type === "following" && isFollowingAuthor;
            const isProfileMatch = feedtype.type === "profile" && feedtype?.username === action.payload.author_id.username;

            if (isGlobal || isFollowing || isProfileMatch) {
                return state.posts.some(post => post._id === action.payload._id)
                    ? state
                    : {
                        ...state,
                        posts: [ action.payload, ...state.posts ],
                        totalPosts: (state.totalPosts || 0) + 1,
                    };
            }

            return state; // do nothing
        }

        case "UPDATE_POST":
        {
            const exists = state.posts.find(post => post._id === action.payload._id);
            return {
                ...state,
                posts: exists
                    ? state.posts.map(post => post._id === action.payload._id ? action.payload : post)
                    : [ action.payload, ...state.posts ]
            };
        }

        case "REMOVE_POST":
        // lexical declaration in case block, needs scope guarding for linting
        {
            const postExists = state.posts.some(post => post._id === action.payload);
            return {
                ...state,
                posts: state.posts.filter(post => post._id !== action.payload),
                totalPosts: postExists ? Math.max(0, (state.totalPosts || 0) - 1) : state.totalPosts,
            };
        }

        case "CLEAR_POSTS":
            return { posts: [], hasMore: false };

        case "SET_FEEDTYPE":
            return {
                ...state,
                feedtype: action.payload
            };

        default:
            return state;
    }
};
```

Figure 9: context/PostContext.jsx::postReducer

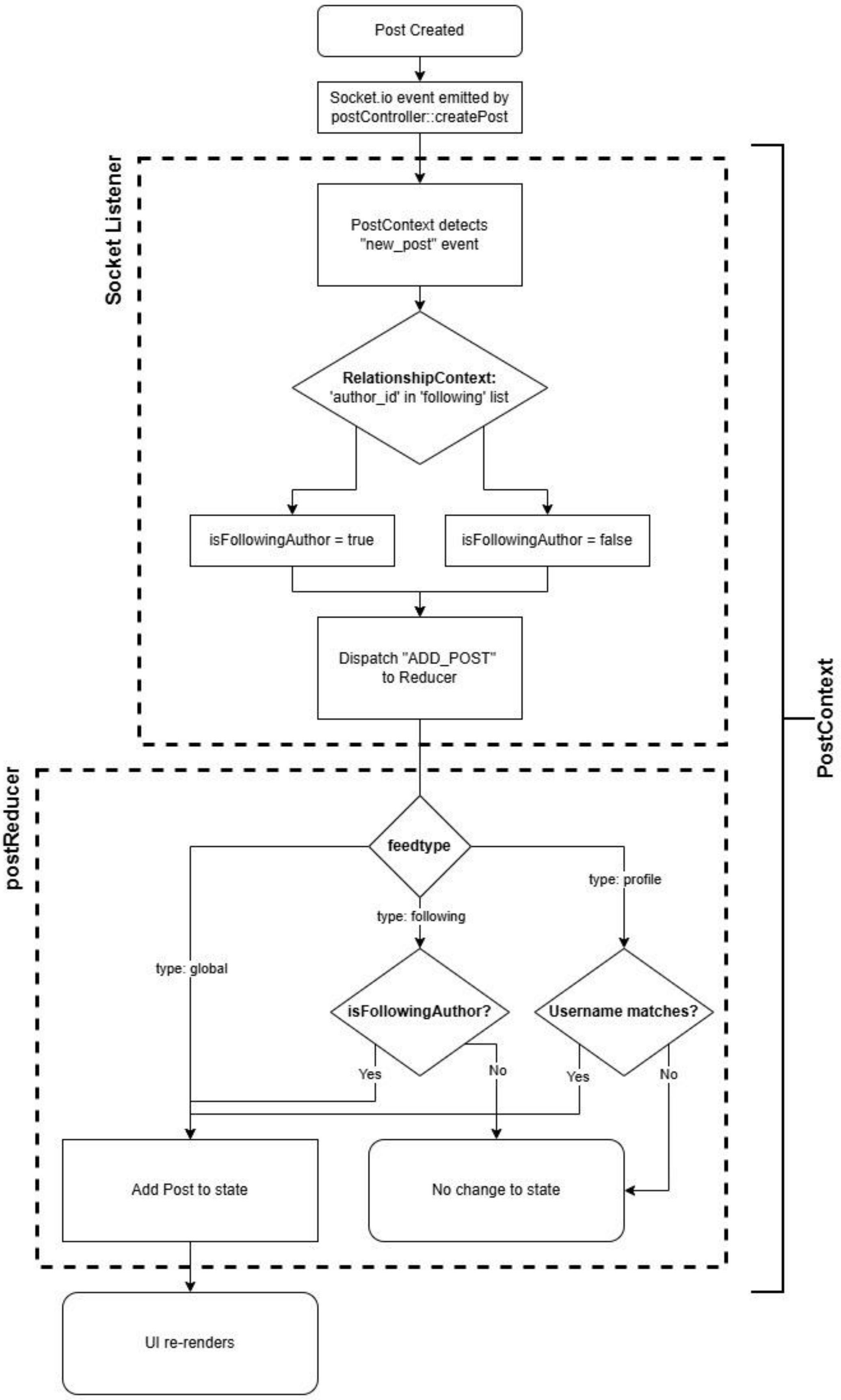
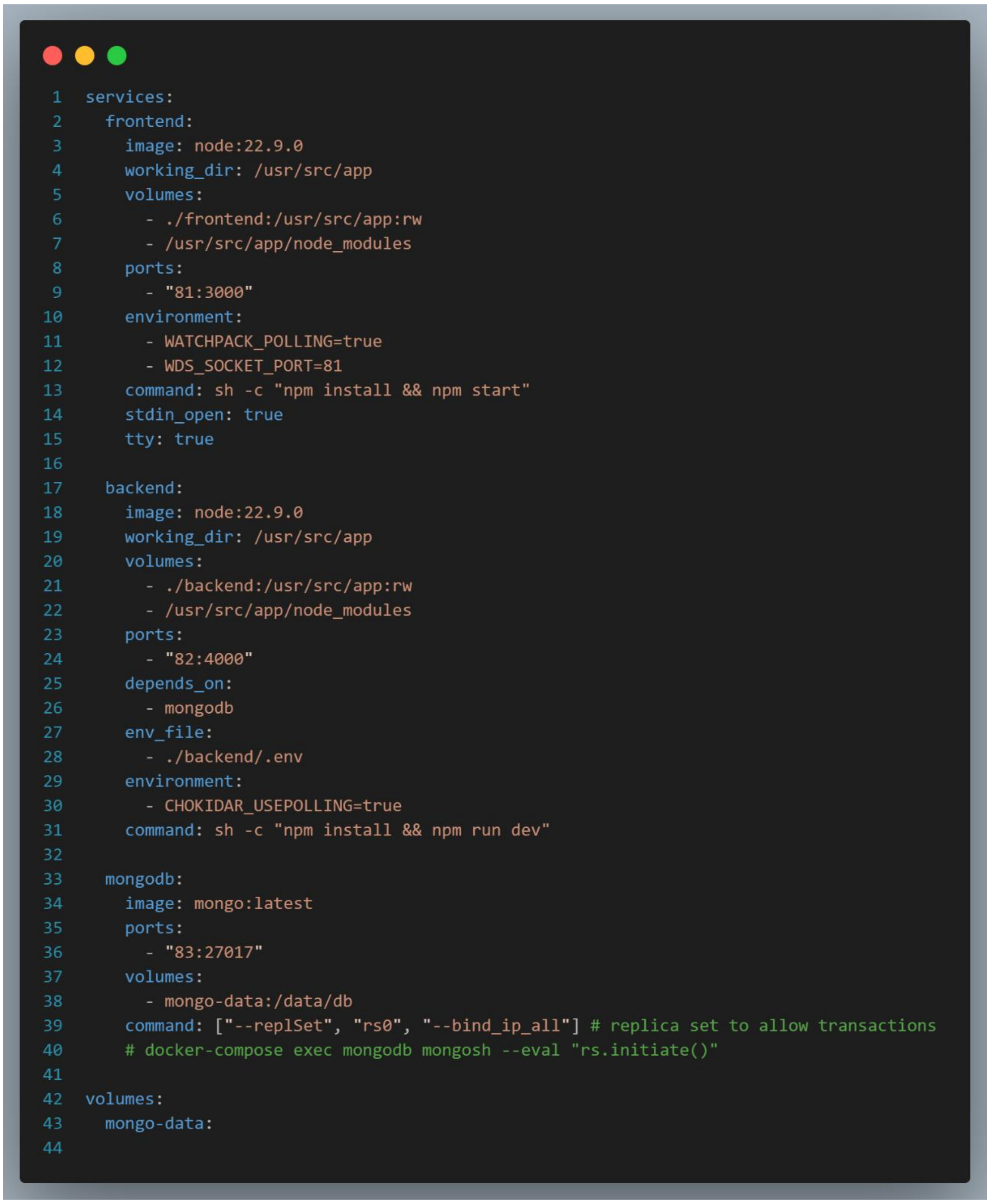


Figure 10: New Post Socket Emit and Context Dispatch

Docker and Containerisation

The system is broken down into three services, orchestrated by a **docker-compose** file. This file ensures that the **database** service is always initialised as a dependency before the **backend** service.

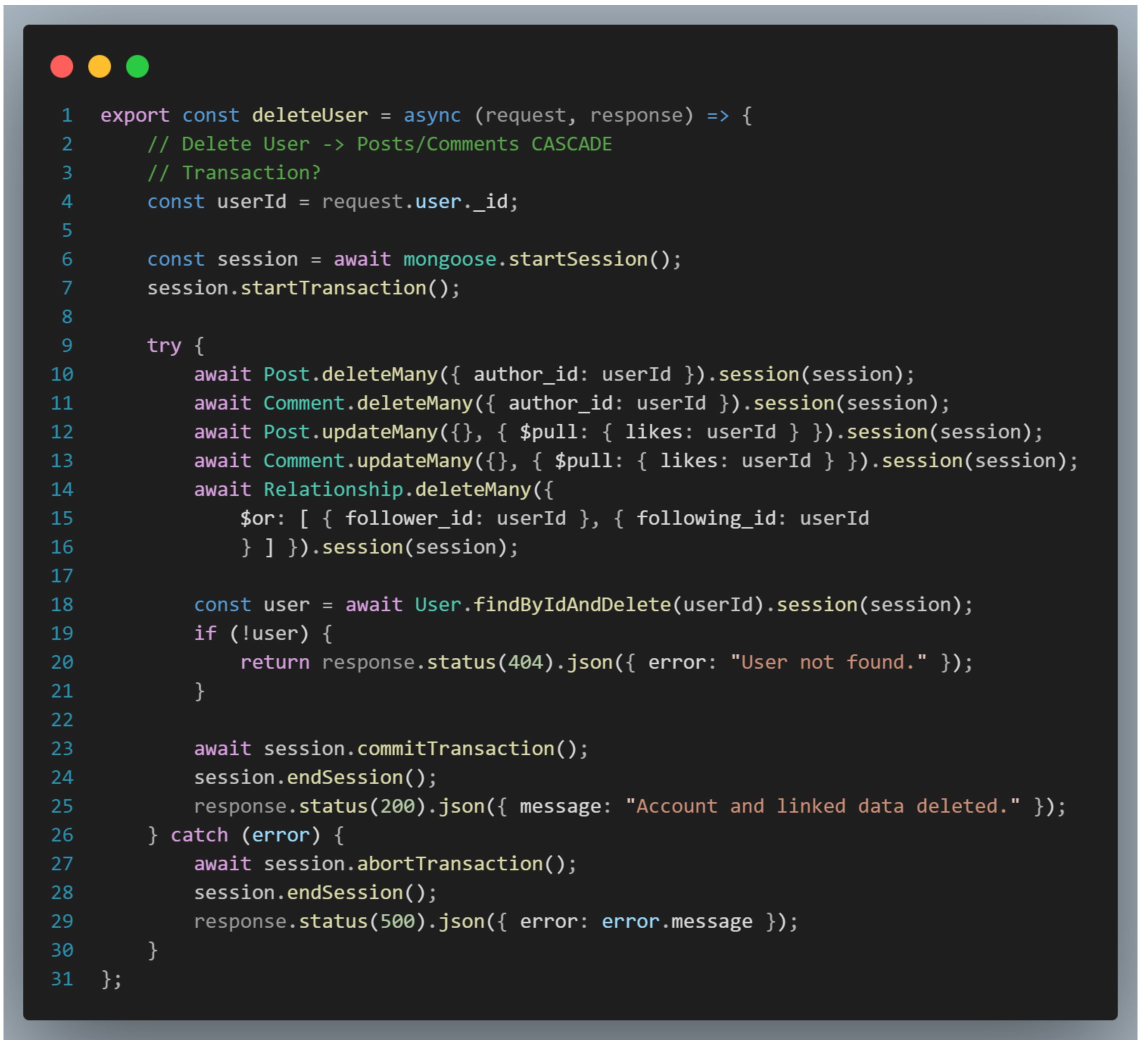


```yaml
services:
  frontend:
    image: node:22.9.0
    working_dir: /usr/src/app
    volumes:
      - ./frontend:/usr/src/app:rw
      - /usr/src/app/node_modules
    ports:
      - "81:3000"
    environment:
      - WATCHPACK_POLLING=true
      - WDS_SOCKET_PORT=81
    command: sh -c "npm install && npm start"
    stdin_open: true
    tty: true

  backend:
    image: node:22.9.0
    working_dir: /usr/src/app
    volumes:
      - ./backend:/usr/src/app:rw
      - /usr/src/app/node_modules
    ports:
      - "82:4000"
    depends_on:
      - mongodb
    env_file:
      - ./backend/.env
    environment:
      - CHOKIDAR_USEPOLLING=true
    command: sh -c "npm install && npm run dev"

  mongodb:
    image: mongo:latest
    ports:
      - "83:27017"
    volumes:
      - mongo-data:/data/db
    command: ["--replSet", "rs0", "--bind_ip_all"] # replica set to allow transactions
    # docker-compose exec mongodb mongosh --eval "rs.initiate()"

volumes:
  mongo-data:

```

Figure 11: Docker Compose File

The **--replSet rs0 --bind_ip_all** command turns the MongoDB instance into a replica set, allowing the use of database transactions in my controllers.

This allows for atomic operations, such as deleting a user from the database. Before deleting the user, their posts, comments, likes and relationships must first also be deleted to prevent orphaned items being present in the database. However, if any of these operations fail, all operations as part of that transaction should fail too to prevent inconsistent data.



```
export const deleteUser = async (request, response) => {
    // Delete User -> Posts/Comments CASCADE
    // Transaction?
    const userId = request.user._id;

    const session = await mongoose.startSession();
    session.startTransaction();

    try {
        await Post.deleteMany({ author_id: userId }).session(session);
        await Comment.deleteMany({ author_id: userId }).session(session);
        await Post.updateMany({}, { $pull: { likes: userId } }).session(session);
        await Comment.updateMany({}, { $pull: { likes: userId } }).session(session);
        await Relationship.deleteMany({
            $or: [ { follower_id: userId }, { following_id: userId
            } ] }).session(session);

        const user = await User.findByIdAndDelete(userId).session(session);
        if (!user) {
            return response.status(404).json({ error: "User not found." });
        }

        await session.commitTransaction();
        session.endSession();
        response.status(200).json({ message: "Account and linked data deleted." });
    } catch (error) {
        await session.abortTransaction();
        session.endSession();
        response.status(500).json({ error: error.message });
    }
};
```

Figure 12: controllers/userController.js::deleteUser

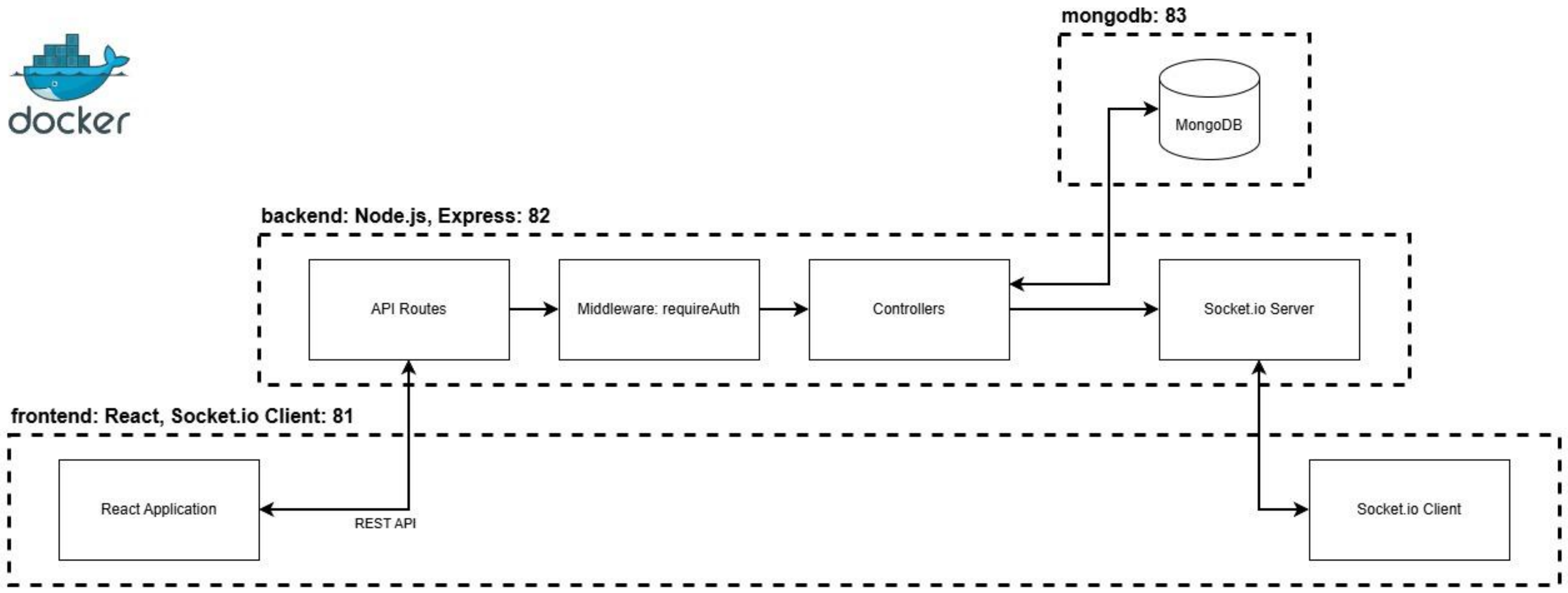


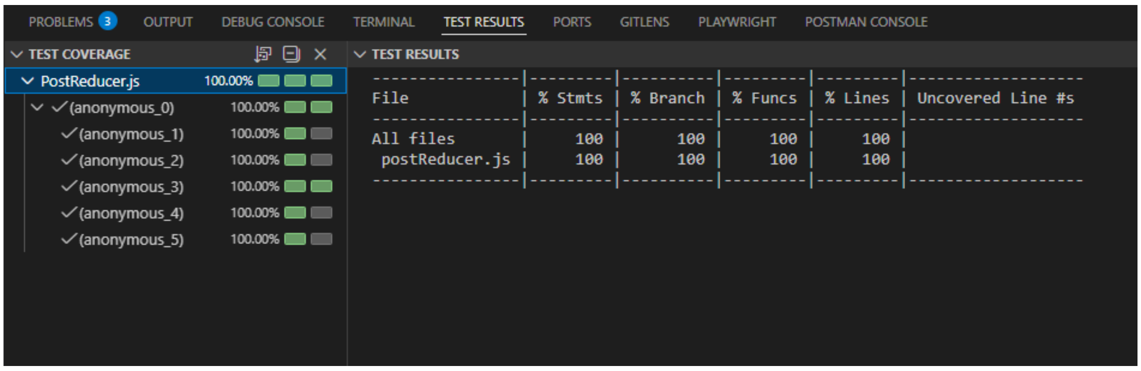
Figure 13: Docker Containerisation Architecture

Service	Host Port	Container Port
frontend	81	3000
backend	82	4000
mongodb (Database)	83	27017

Testing

I used the **Vitest** and **Supertest** libraries to implement Unit and Integration testing to the project on both the front and back end of the system. Whilst test coverage isn't 100% across the codebase, it does cover a breadth of different functionality to demonstrate an understanding of the test process and methodology. I prioritised tests for high-complexity areas of functionality, such as the real-time state management context and reducer functions, authentication middleware and data controllers.

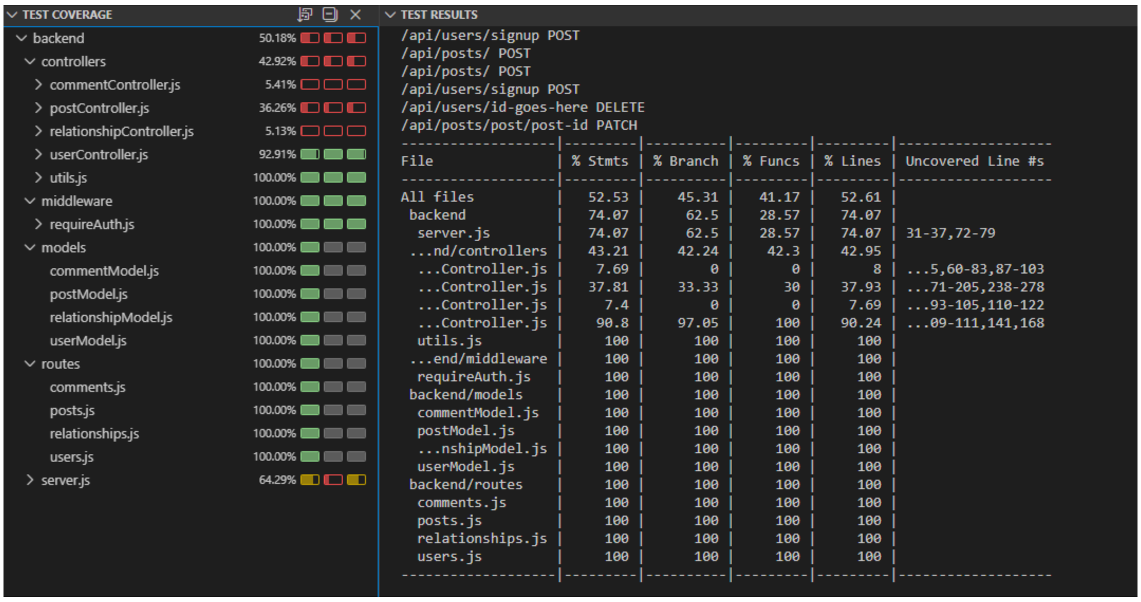
As well as these implemented tests, Postman was also used for manual testing of the API routes early in the development process.



```
PostReducer.js      100.00%
  (anonymous_0)     100.00%
   (anonymous_1)    100.00%
   (anonymous_2)    100.00%
   (anonymous_3)    100.00%
   (anonymous_4)    100.00%
   (anonymous_5)    100.00%
```

```
----------------|---------|----------|---------|---------|-------------------
File            | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------|---------|----------|---------|---------|-------------------
All files       |     100 |      100 |     100 |     100 |
 postReducer.js |     100 |      100 |     100 |     100 |
----------------|---------|----------|---------|---------|-------------------
```

Figure 14: Frontend Test Coverage

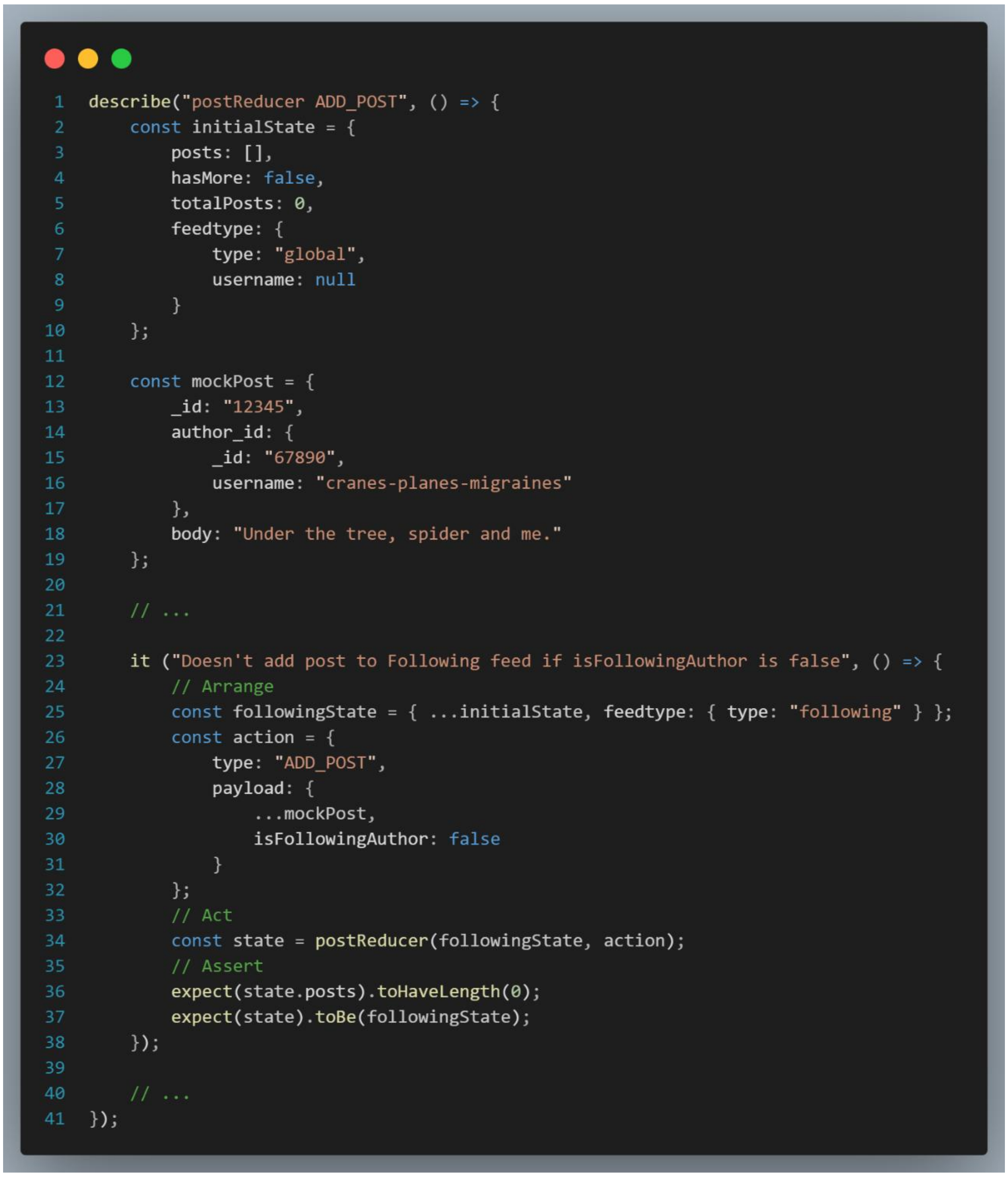


```
backend                       50.18%
  controllers                 42.92%
    commentController.js       5.41%
    postController.js         36.26%
    relationshipController.js  5.13%
    userController.js         92.91%
    utils.js                 100.00%
  middleware                 100.00%
    requireAuth.js           100.00%
  models                     100.00%
    commentModel.js          100.00%
    postModel.js             100.00%
    relationshipModel.js     100.00%
    userModel.js             100.00%
  routes                     100.00%
    comments.js              100.00%
    posts.js                 100.00%
    relationships.js         100.00%
    users.js                 100.00%
  server.js                   64.29%
```

```
/api/users/signup POST
/api/posts/ POST
/api/posts/ POST
/api/users/signup POST
/api/users/id-goes-here DELETE
/api/posts/post/post-id PATCH
-------------------|---------|----------|---------|---------|-------------------
File               | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------|---------|----------|---------|---------|-------------------
All files          |   52.53 |    45.31 |   41.17 |   52.61 |
 backend           |   74.07 |     62.5 |   28.57 |   74.07 |
  server.js        |   74.07 |     62.5 |   28.57 |   74.07 | 31-37,72-79
 ...nd/controllers |   43.21 |    42.24 |    42.3 |   42.95 |
  ...Controller.js |    7.69 |        0 |       0 |       8 | ...5,60-83,87-103
  ...Controller.js |   37.81 |    33.33 |      30 |   37.93 | ...71-205,238-278
  ...Controller.js |     7.4 |        0 |       0 |    7.69 | ...93-105,110-122
  ...Controller.js |    90.8 |    97.05 |     100 |   90.24 | ...09-111,141,168
  utils.js         |     100 |      100 |     100 |     100 |
 ...end/middleware |     100 |      100 |     100 |     100 |
  requireAuth.js   |     100 |      100 |     100 |     100 |
 backend/models    |     100 |      100 |     100 |     100 |
  commentModel.js  |     100 |      100 |     100 |     100 |
  postModel.js     |     100 |      100 |     100 |     100 |
  ...nshipModel.js |     100 |      100 |     100 |     100 |
  userModel.js     |     100 |      100 |     100 |     100 |
 backend/routes    |     100 |      100 |     100 |     100 |
  comments.js      |     100 |      100 |     100 |     100 |
  posts.js         |     100 |      100 |     100 |     100 |
  relationships.js |     100 |      100 |     100 |     100 |
  users.js         |     100 |      100 |     100 |     100 |
-------------------|---------|----------|---------|---------|-------------------
```

Figure 15: Backend Test Coverage

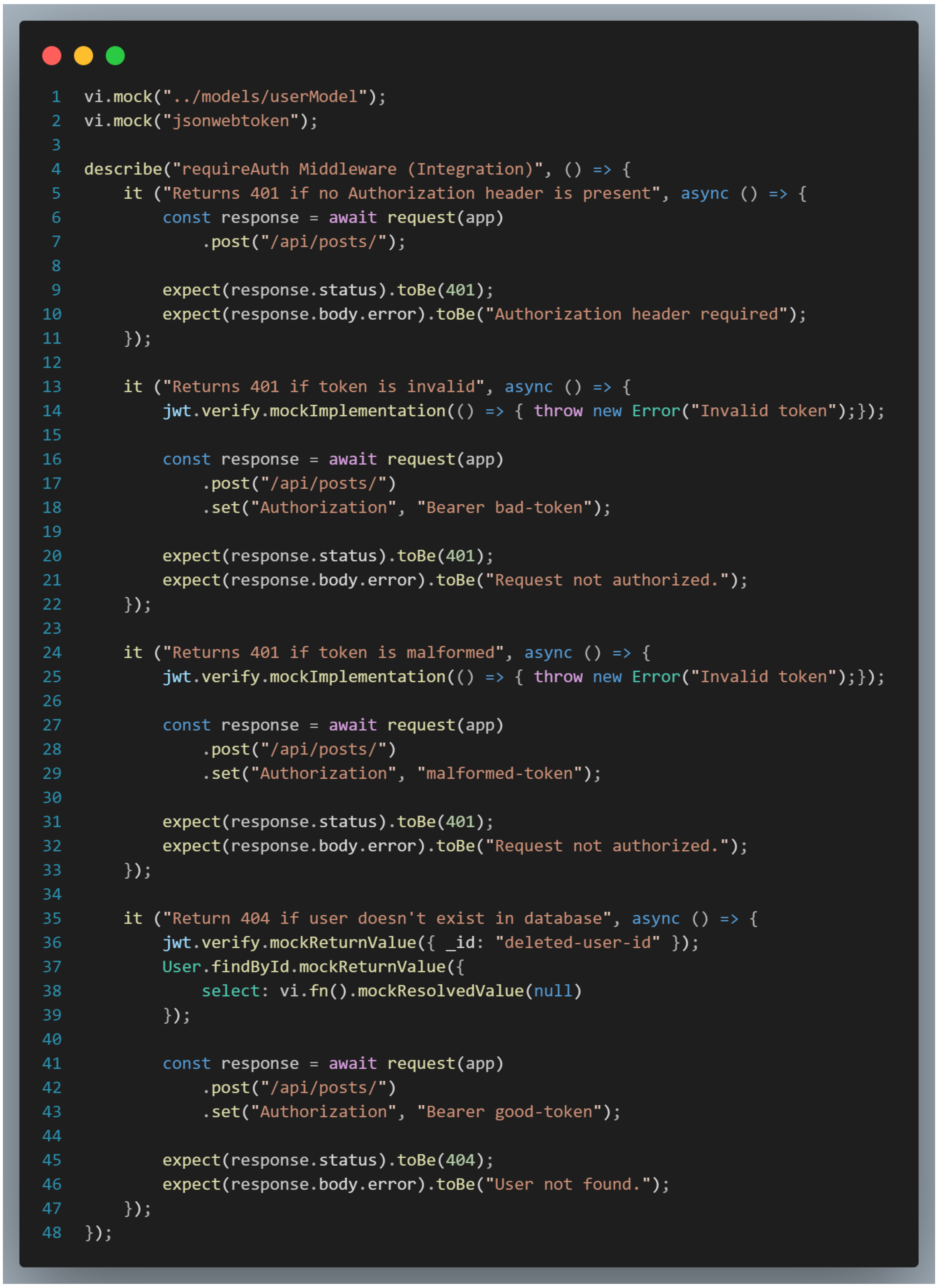
I used the **Arrange-Act-Assert** pattern to structure my **65 different tests**. This pattern is used to ensure that tests have a single responsibility on what they are testing and are atomic. The **Arrange** stage sets up the test case and prepares any required data objects to be used in the **Act** stage when the code to be tested is ran. The **Assert** stage is where the outcome is compared to the expected state.



```
describe("postReducer ADD_POST", () => {
    const initialState = {
        posts: [],
        hasMore: false,
        totalPosts: 0,
        feedtype: {
            type: "global",
            username: null
        }
    };

    const mockPost = {
        _id: "12345",
        author_id: {
            _id: "67890",
            username: "cranes-planes-migraines"
        },
        body: "Under the tree, spider and me."
    };

    // ...

    it ("Doesn't add post to Following feed if isFollowingAuthor is false", () => {
        // Arrange
        const followingState = { ...initialState, feedtype: { type: "following" } };
        const action = {
            type: "ADD_POST",
            payload: {
                ...mockPost,
                isFollowingAuthor: false
            }
        };
        // Act
        const state = postReducer(followingState, action);
        // Assert
        expect(state.posts).toHaveLength(0);
        expect(state).toBe(followingState);
    });

    // ...
});
```

Figure 16: postReducer ADD_POST Test Case Example

The above test case checks that the reducer correctly applies conditional state updates based on the currently set **feedtype.type** property. All new posts are broadcast to connected clients by the WebSocket, however, not every post should be reflected in the users local state. If the feedtype is set to **following**, only posts authored by users the current user follows should be added to the context state. In the **Arrange** stage, the state is configured to use the **following** feedtype and sets the action-to-be-dispatched's payload to include **isFollowingAuthor: false**; this property is passed to the reducer from the context provider. The **Assert** stage confirms that the posts array remains unchanged and that the reducer has returned the original unaltered state object. This preserves immutability.



```
vi.mock("../models/userModel");
vi.mock("jsonwebtoken");

describe("requireAuth Middleware (Integration)", () => {
    it ("Returns 401 if no Authorization header is present", async () => {
        const response = await request(app)
            .post("/api/posts/");

        expect(response.status).toBe(401);
        expect(response.body.error).toBe("Authorization header required");
    });

    it ("Returns 401 if token is invalid", async () => {
        jwt.verify.mockImplementation(() => { throw new Error("Invalid token");});

        const response = await request(app)
            .post("/api/posts/")
            .set("Authorization", "Bearer bad-token");

        expect(response.status).toBe(401);
        expect(response.body.error).toBe("Request not authorized.");
    });

    it ("Returns 401 if token is malformed", async () => {
        jwt.verify.mockImplementation(() => { throw new Error("Invalid token");});

        const response = await request(app)
            .post("/api/posts/")
            .set("Authorization", "malformed-token");

        expect(response.status).toBe(401);
        expect(response.body.error).toBe("Request not authorized.");
    });

    it ("Return 404 if user doesn't exist in database", async () => {
        jwt.verify.mockReturnValue({ _id: "deleted-user-id" });
        User.findById.mockReturnValue({
            select: vi.fn().mockResolvedValue(null)
        });

        const response = await request(app)
            .post("/api/posts/")
            .set("Authorization", "Bearer good-token");

        expect(response.status).toBe(404);
        expect(response.body.error).toBe("User not found.");
    });
});
```

Figure 17: requireAuth() Middleware Integration Tests

These tests validate the **require_auth()** authentication middleware, using **Supertest** to pass HTTP requests into the middleware to ensure that request handling, execution and response all work together as expected. External dependencies are mocked to protect the live dataset and to ensure deterministic behaviour.

In the “Return 404 if user doesn’t exist in database” test case, the **jsonwebtoken** module is mocked to return a “valid” id:key response containing a user identifier which allows the test to proceed past the initial authentication checks. The **User** model method, **findByID**, is also mocked to return **null**, simulating a case where whilst a user may have a valid JSON Web Token, the User has been deleted from the database, validating the middleware’s ability to handle failure conditions across different layers of the application. The **404** result confirms that the middleware enforces this access control based on authentication states and user existence.

All of this ensures that the systems authentication works as intended and the security of the application is correct.

DevOps Pipeline

Code quality checks are done frequently at the earliest point (Shift-Left) in the development process.

To prevent low quality of unformatted code from being committed to the git repository, I implemented a **Husky pre-commit hook** which runs **lint-staged** whenever someone makes a commit in the repository. **lint-staged** runs static analysis with **ESLint** on the files which have been changed and staged since the last commit, enforcing style rules defined in **SourceCode/eslint.config.mjs**. These rules include preventing unused variables from being declared, ensuring consistent indentation is used and imports are grouped and sorted alphabetically. Whilst these rules don't change the effects of the code, they do improve the readability and maintainability of the project by ensuring consistency across files and a high code quality.

The next quality gate is a **Continuous Integration pipeline**, ran on each push to the main branch of the repository with **GitHub Actions**. The **lint** job ensures that code is styled consistently and is a dependency of the following jobs: if this job fails, they don't run (fail early). After linting passes, **backend-tests** and **frontend-tests** run in parallel with coverage metrics. There is a check to see if the workflow is being run from **GitHub Local Actions** but if not, the **upload-artifact** step uploads the test coverage reports to the run so that tests can be reviewed.

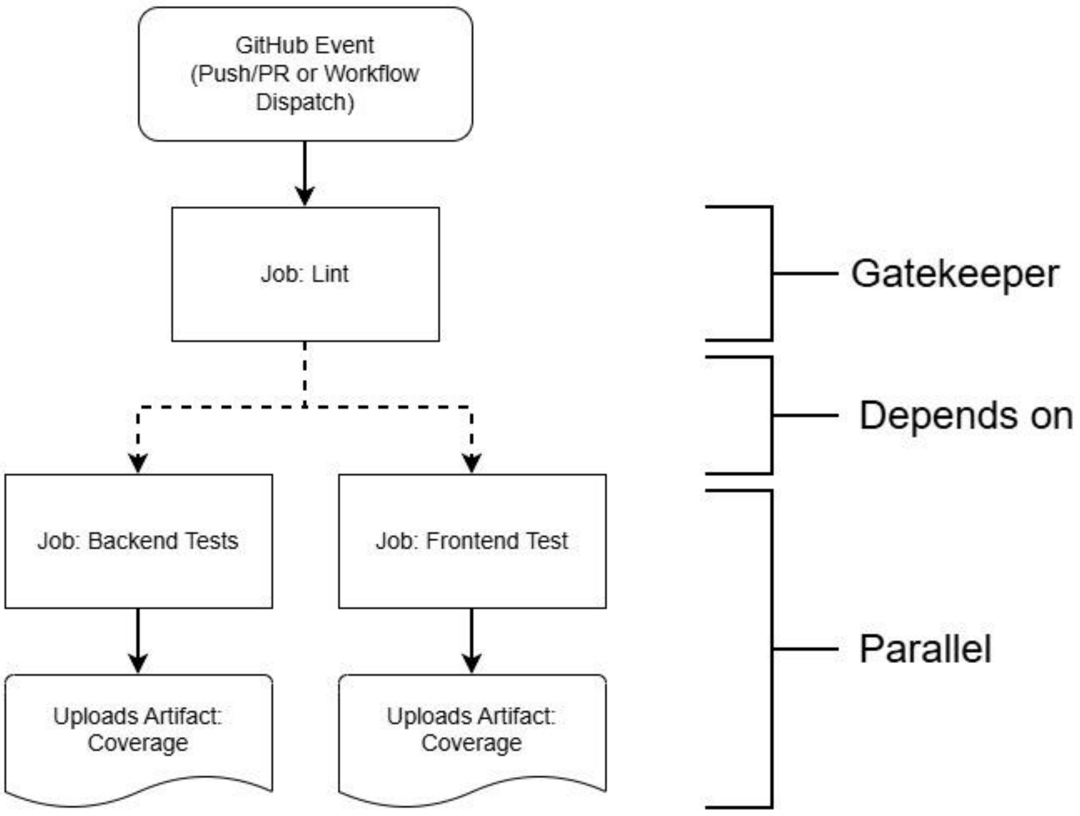
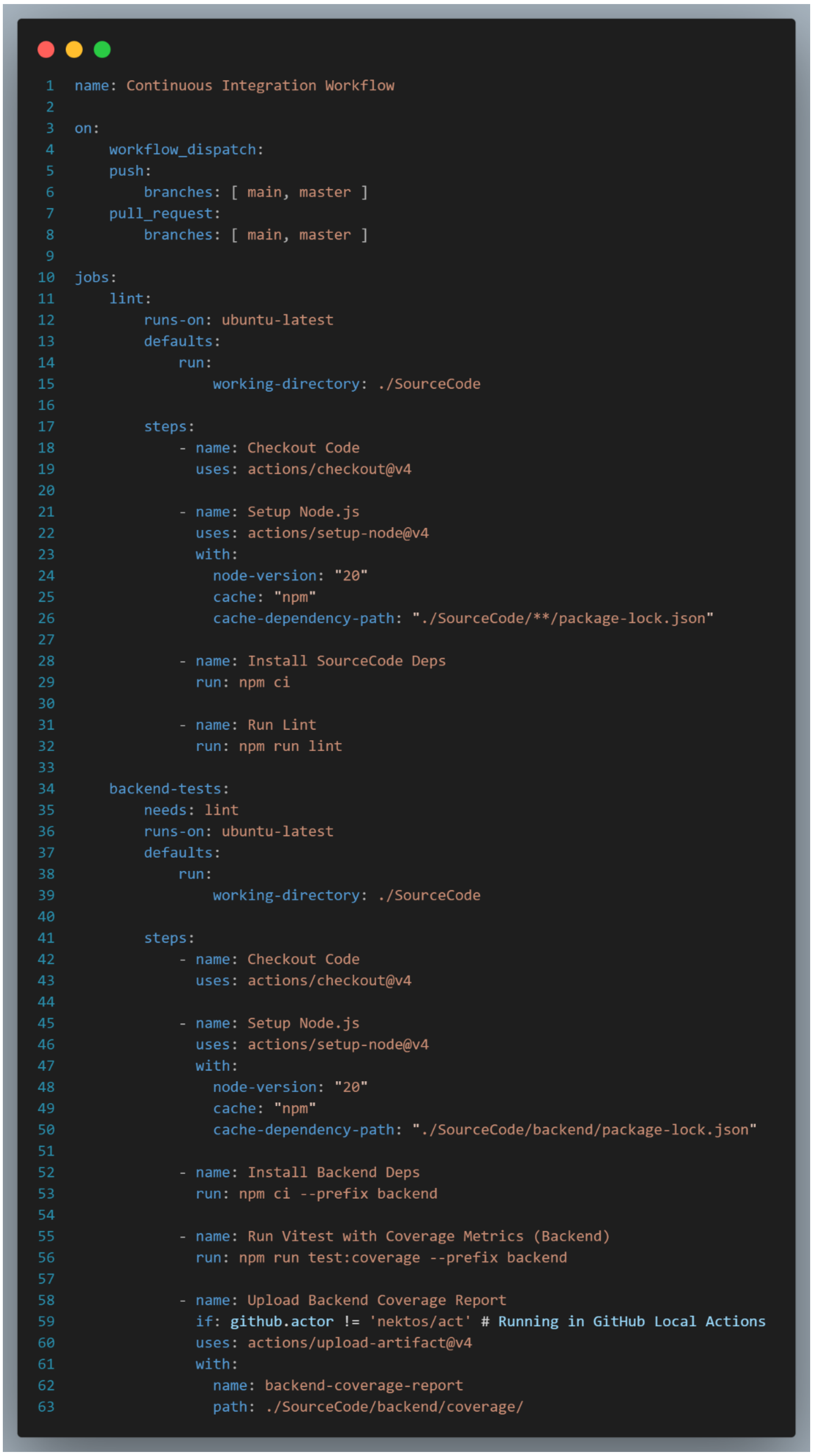
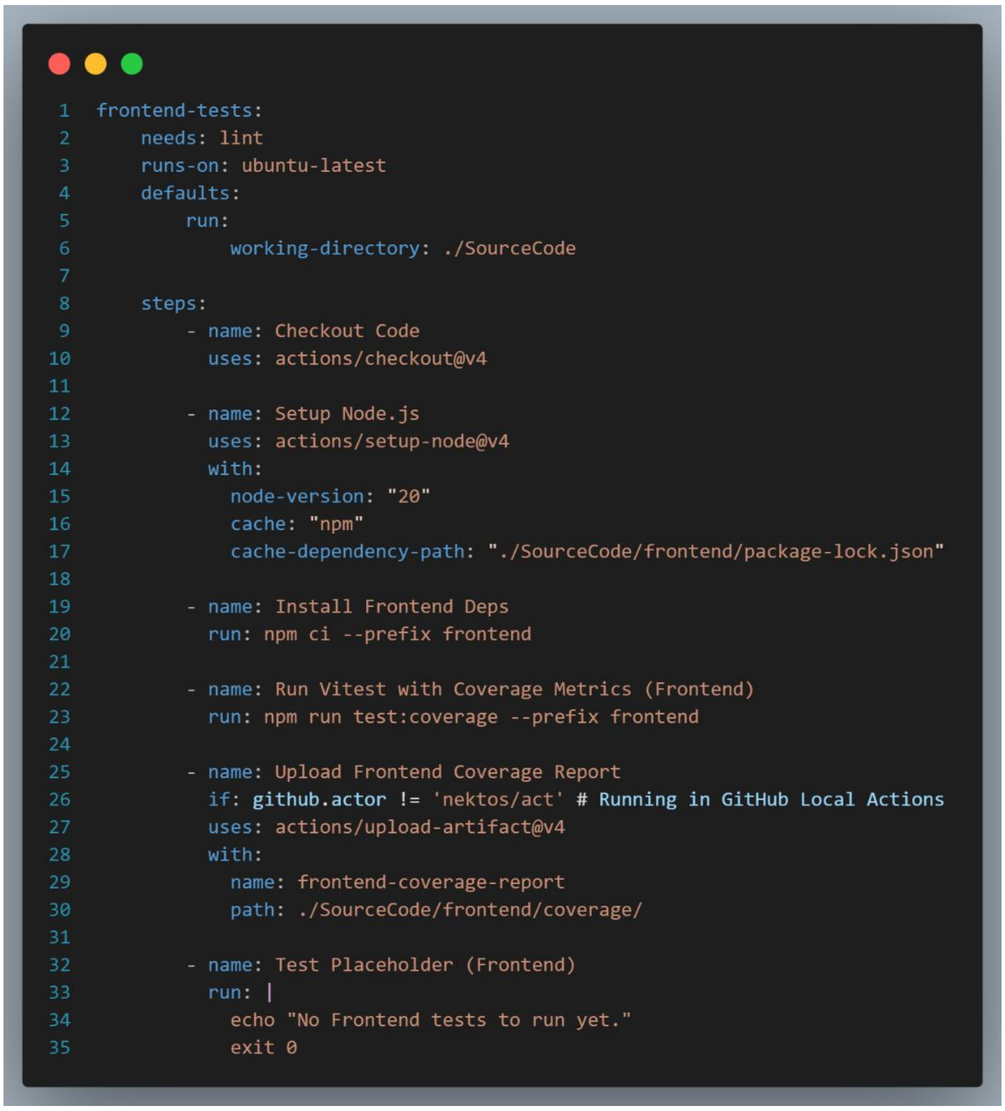


Figure 18: GitHub Actions Workflow



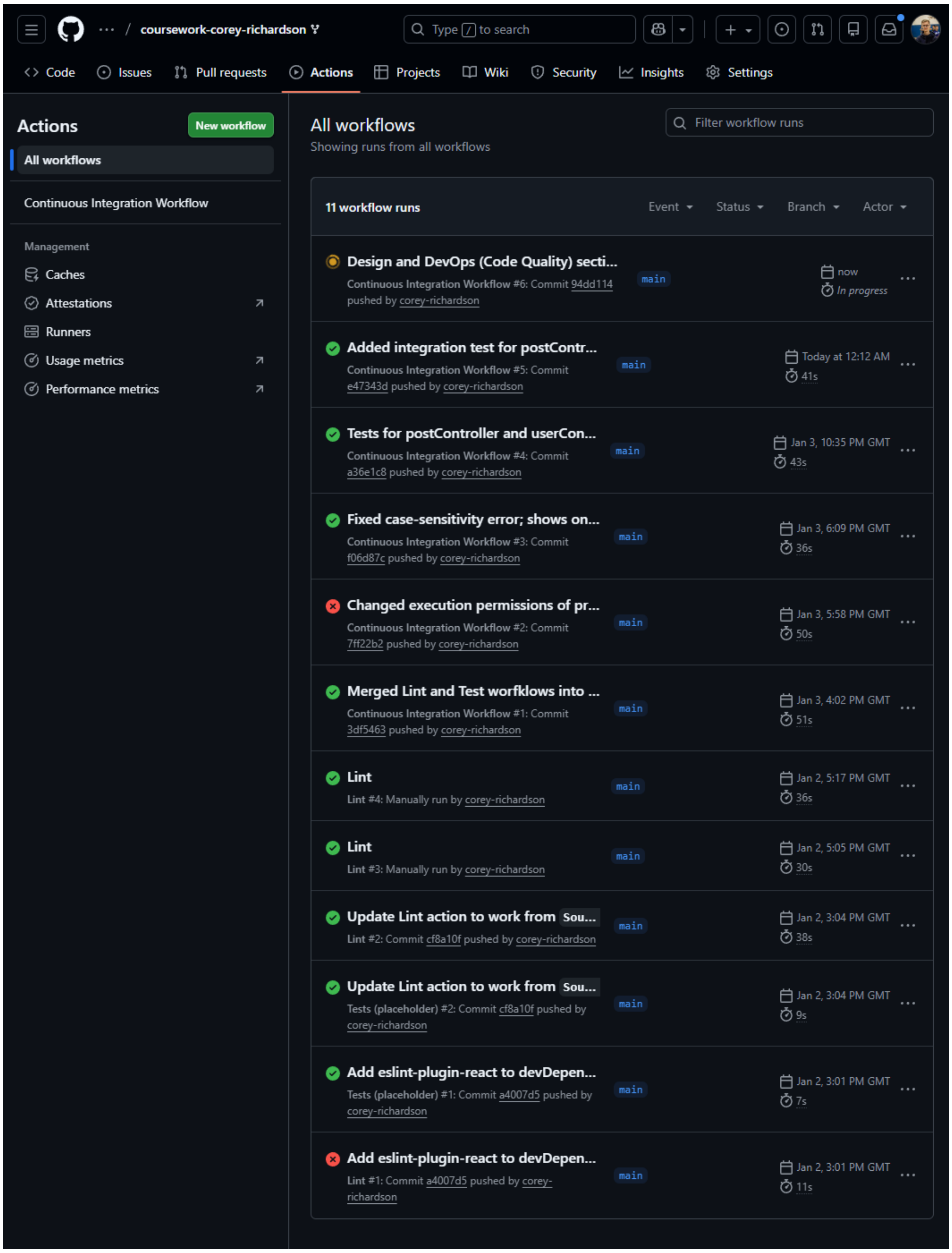
```yaml
name: Continuous Integration Workflow

on:
    workflow_dispatch:
    push:
        branches: [ main, master ]
    pull_request:
        branches: [ main, master ]

jobs:
    lint:
        runs-on: ubuntu-latest
        defaults:
            run:
                working-directory: ./SourceCode

        steps:
            - name: Checkout Code
              uses: actions/checkout@v4

            - name: Setup Node.js
              uses: actions/setup-node@v4
              with:
                node-version: "20"
                cache: "npm"
                cache-dependency-path: "./SourceCode/**/package-lock.json"

            - name: Install SourceCode Deps
              run: npm ci

            - name: Run Lint
              run: npm run lint

    backend-tests:
        needs: lint
        runs-on: ubuntu-latest
        defaults:
            run:
                working-directory: ./SourceCode

        steps:
            - name: Checkout Code
              uses: actions/checkout@v4

            - name: Setup Node.js
              uses: actions/setup-node@v4
              with:
                node-version: "20"
                cache: "npm"
                cache-dependency-path: "./SourceCode/backend/package-lock.json"

            - name: Install Backend Deps
              run: npm ci --prefix backend

            - name: Run Vitest with Coverage Metrics (Backend)
              run: npm run test:coverage --prefix backend

            - name: Upload Backend Coverage Report
              if: github.actor != 'nektos/act' # Running in GitHub Local Actions
              uses: actions/upload-artifact@v4
              with:
                name: backend-coverage-report
                path: ./SourceCode/backend/coverage/
```

Figure 19: CI Workflow 1



```yaml
frontend-tests:
    needs: lint
    runs-on: ubuntu-latest
    defaults:
        run:
            working-directory: ./SourceCode

    steps:
        - name: Checkout Code
          uses: actions/checkout@v4

        - name: Setup Node.js
          uses: actions/setup-node@v4
          with:
            node-version: "20"
            cache: "npm"
            cache-dependency-path: "./SourceCode/frontend/package-lock.json"

        - name: Install Frontend Deps
          run: npm ci --prefix frontend

        - name: Run Vitest with Coverage Metrics (Frontend)
          run: npm run test:coverage --prefix frontend

        - name: Upload Frontend Coverage Report
          if: github.actor != 'nektos/act' # Running in GitHub Local Actions
          uses: actions/upload-artifact@v4
          with:
            name: frontend-coverage-report
            path: ./SourceCode/frontend/coverage/

        - name: Test Placeholder (Frontend)
          run: |
            echo "No Frontend tests to run yet."
            exit 0
```

Figure 20: CI Workflow 2



coursework-corey-richardson

Type / to search

Code · Issues · Pull requests · Actions · Projects · Wiki · Security · Insights · Settings

Actions · New workflow

All workflows

Continuous Integration Workflow

Management

Caches

Attestations

Runners

Usage metrics

Performance metrics

All workflows

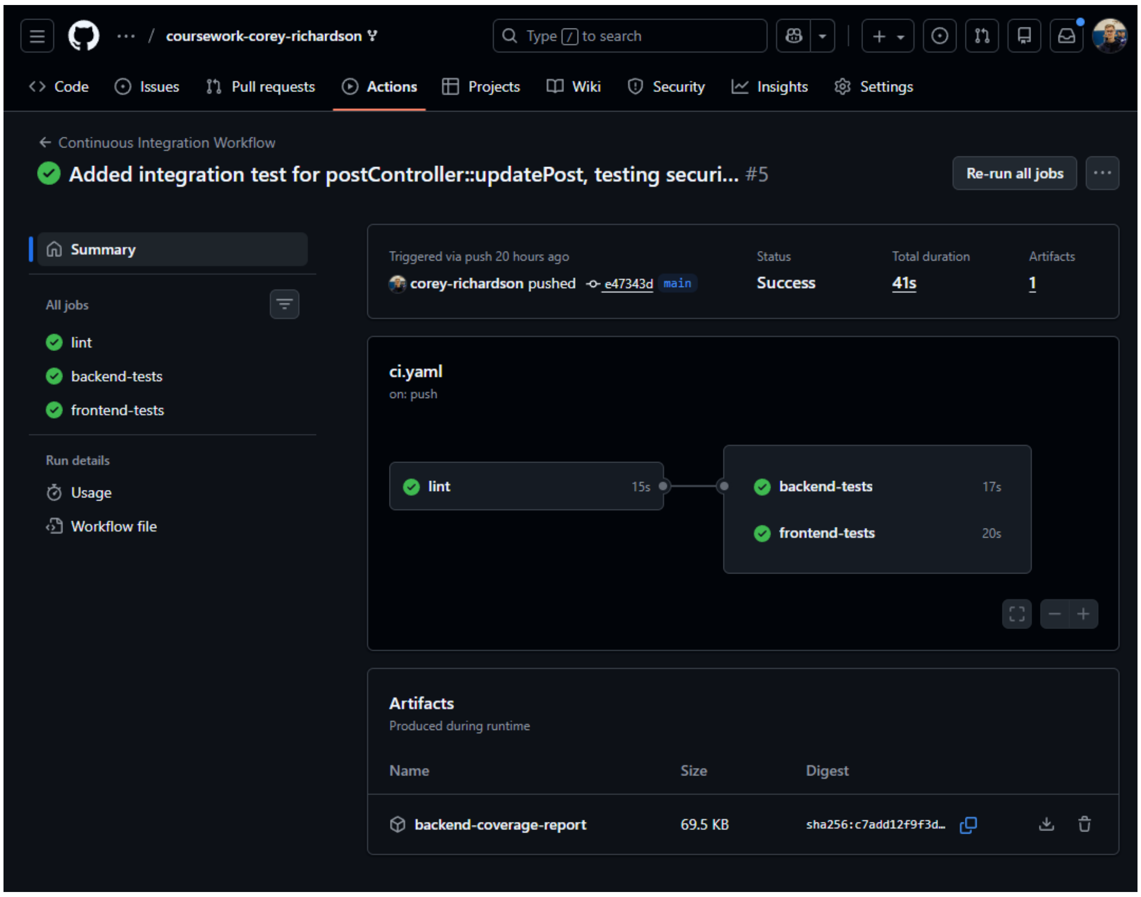
Showing runs from all workflows

Filter workflow runs

11 workflow runs · Event · Status · Branch · Actor

Design and DevOps (Code Quality) secti... Continuous Integration Workflow #6: Commit 94dd114 pushed by corey-richardson	main	now	*In progress*
Added integration test for postContr... Continuous Integration Workflow #5: Commit e47343d pushed by corey-richardson	main	Today at 12:12 AM	41s
Tests for postController and userCon... Continuous Integration Workflow #4: Commit a36e1c8 pushed by corey-richardson	main	Jan 3, 10:35 PM GMT	43s
Fixed case-sensitivity error; shows on... Continuous Integration Workflow #3: Commit f06d87c pushed by corey-richardson	main	Jan 3, 6:09 PM GMT	36s
Changed execution permissions of pr... Continuous Integration Workflow #2: Commit 7ff22b2 pushed by corey-richardson	main	Jan 3, 5:58 PM GMT	50s
Merged Lint and Test worfklows into ... Continuous Integration Workflow #1: Commit 3df5463 pushed by corey-richardson	main	Jan 3, 4:02 PM GMT	51s
Lint Lint #4: Manually run by corey-richardson	main	Jan 2, 5:17 PM GMT	36s
Lint Lint #3: Manually run by corey-richardson	main	Jan 2, 5:05 PM GMT	30s
Update Lint action to work from Sou... Lint #2: Commit cf8a10f pushed by corey-richardson	main	Jan 2, 3:04 PM GMT	38s
Update Lint action to work from Sou... Tests (placeholder) #2: Commit cf8a10f pushed by corey-richardson	main	Jan 2, 3:04 PM GMT	9s
Add eslint-plugin-react to devDepen... Tests (placeholder) #1: Commit a4007d5 pushed by corey-richardson	main	Jan 2, 3:01 PM GMT	7s
Add eslint-plugin-react to devDepen... Lint #1: Commit a4007d5 pushed by corey-richardson	main	Jan 2, 3:01 PM GMT	11s

Figure 21: GitHub Actions Page



Added integration test for postController::updatePost, testing securi... #5

Triggered via push 20 hours ago
corey-richardson pushed e47343d main

Status	Total duration	Artifacts
Success	41s	1

ci.yaml
on: push

Artifacts
Produced during runtime

Name	Size	Digest
backend-coverage-report	69.5 KB	sha256:c7add12f9f3d…

Figure 22: GitHub Actions Page - Coverage Report Artifact

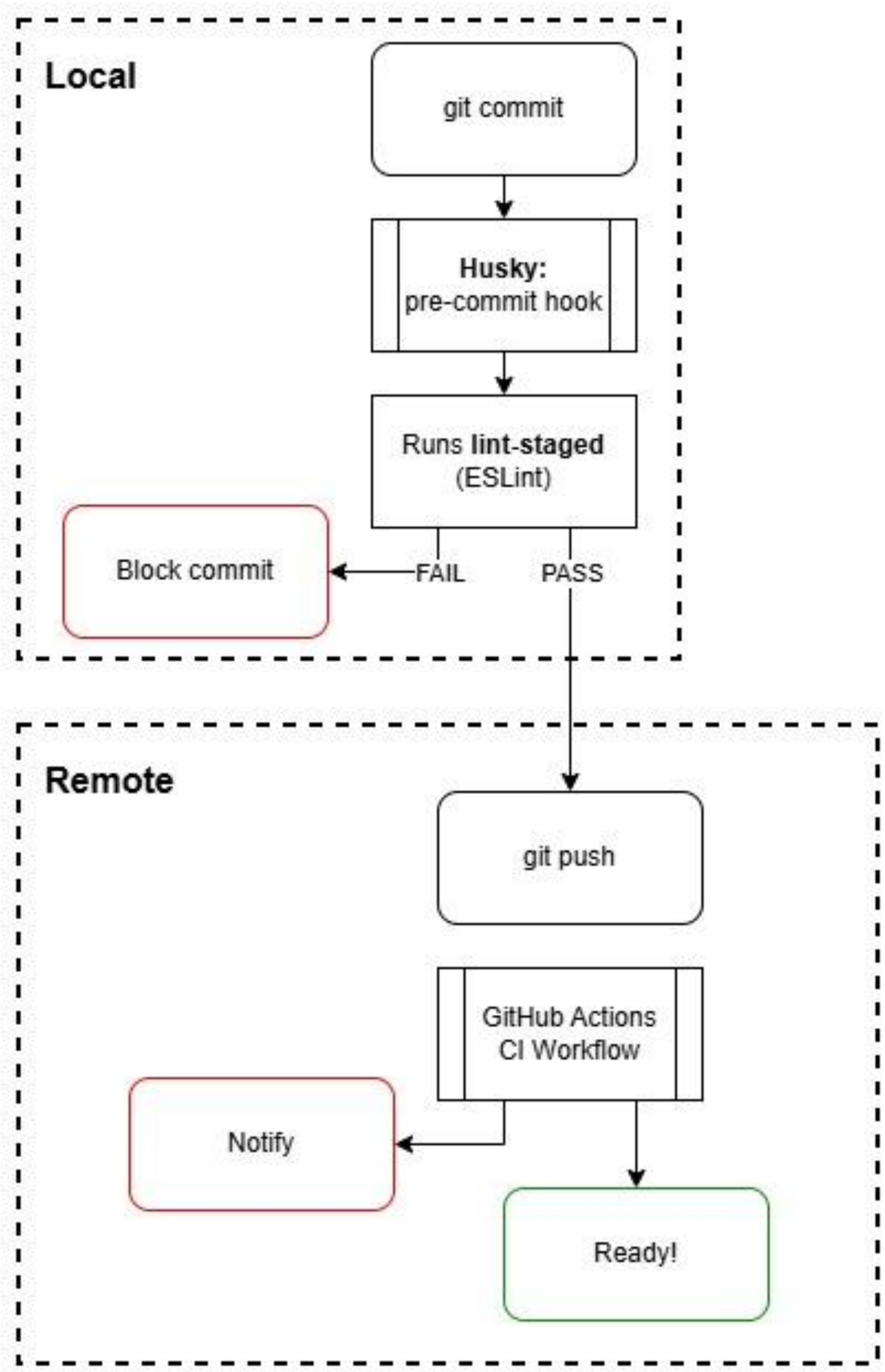
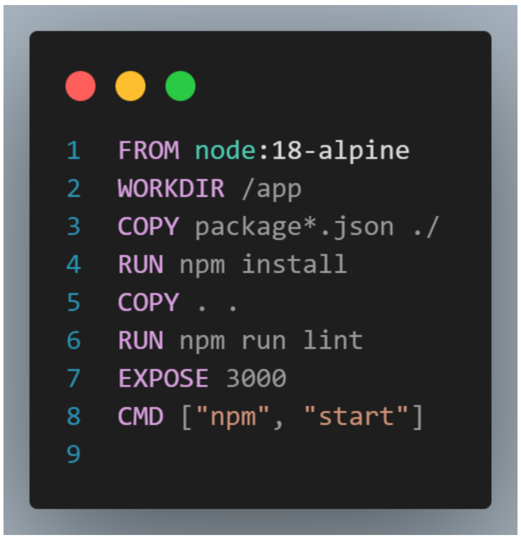

Local
git commit
Husky: pre-commit hook
Runs **lint-staged** (ESLint)
FAIL
PASS
Block commit
Remote
git push
GitHub Actions CI Workflow
Notify
Ready!

Figure 23: Code Quality Pipeline

The **Dockerfile** configurations (**frontend/Dockerfile** and **backend/Dockerfile**) provide consistent environment parity between development, testing, and deployments.

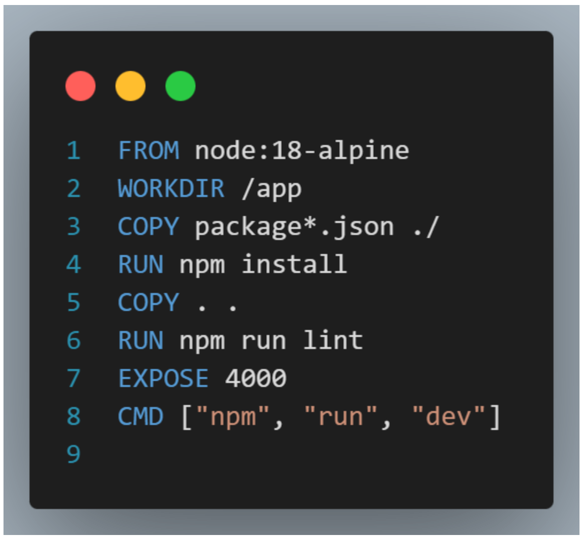
GitHub Actions will use the same **node:18-alpine** image used during development and deployment, meaning that if the CI pipeline tests pass there, they will pass in a production container too.

npm run lint is also found in the Dockerfiles before exposing the port, providing a final check that below-quality code isn't being provided to the deployment.



```dockerfile
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run lint
EXPOSE 3000
CMD ["npm", "start"]
```

Figure 24: frontend/Dockerfile



```dockerfile
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run lint
EXPOSE 4000
CMD ["npm", "run", "dev"]
```

Figure 25: backend/Dockerfile

Code Quality Gates			
Stage	**Tool**	**Trigger**	**Action**
Commit	Husky / lint-staged	git commit	Block commit if linting fails.
Push	GitHub Actions	git push	Notifies on lint or test failure.
Build	Docker	docker-compose	Runs **npm run lint** before EXPOSE.

Evaluation

The system met all of my “Must-Have” requirements such as having a containerised MERN architecture and the implementation of real-time UI updates via Socket.io.

My use of the React Context/Reducer pattern provided an easily scalable foundation with high robustness, as evidenced by the use of Unit and Integration tests.

Whilst early peer-feedback on the low fidelity wireframes helped finalise the blueprint of the design, a more formal Usability Testing process would likely have allowed me to improve the User Interface and Experience even further.