School of Computing

Technical Specification

GitMD

Members:

Aaron Crawford	- 20336753
Ciaran Skelly	- 20324213

Supervisor:
Stephen Blott

Date Completed	20/04/2024

Table of Contents

1. Abstract

GitMD is a markdown editor available as a web application and on mobile platforms. The application allows users to create, modify, organise, and access their markdown files effortlessly regardless of the platform they are on. It aims to make the writing process more accessible, shareable and ensure no loss of work through the use of Git. Git repositories allow users to access their files on multiple devices and collaborate with other users on the same markdown files by sharing a repository; these features are ideal for teams working on projects or research groups.

2. Motivation

Many note taking applications are needlessly expensive, cannot be accessed on all platforms or have limited functionality. Our project seeks to provide an easy to use Markdown application for web and mobile platforms that will enable users to create, modify, organise, and access their notes effortlessly. Being able to access our markdown files on a number of platforms and in different contexts was the prime motivator for creating both a web and mobile application. It was also what motivated us to use Git as it would allow users to use git commands to pull files so they can be viewed or edited locally on their own machines.

3. Research

3.1 Django

Django's combination of robust features, security measures and scalability makes it an excellent choice for powering the backend of our project and also allows us to use python which provides a wide range of benefits to us. Python's simplicity and readability made it a great candidate for our shared backend as it meant reviewing and explaining different backend features became easier and less time consuming, increasing productivity. Python also allowed us to use libraries such as requests which simplifies making HTTP requests and handling responses, this made it much easier anytime we had to contact the gitea server for information required by the user. We also researched how we would authenticate users once they logged in or out of their account, we researched into different methods and found pyjwt to be the best option. The jwt library in Python provides functionality for encoding and decoding JSON Web Tokens (JWTs) and we could assign one of these tokens to a user on successful login, check if it exists for authentication and delete the JWT on logout. It also allowed us to encode the JWT using the HS256 algorithm which helped ensure that users accounts and data was kept secure [1]. Django allowed us to use features with clear documentation such as its EmailMultiAlternatives class which made it relatively simple to contact users via email [2]. Django's built-in testing tools provide us with a convenient and powerful way to write and execute tests for our web application. Django's testing framework also provides a wide range of utilities such as simulation HTTP requests, this allows us to create tests that closely mimic real world scenarios, ensuring thorough and accurate testing of our application.

3.2 React

React was used for the frontend as it offered libraries and frameworks that would be key in the development of our project. First it allowed us to use Material UI, providing a comprehensive library of reusable and customizable UI components, covering a wide range of use cases that were used throughout the web application. MUI also adheres closely to Google's Material Design guidelines, offering a visually appealing and consistent design language out of the box. A key component that was used from MUI was the paper component which allowed us to serve user pages to display and edit their markdown files. React also allowed us to use important libraries such as ReactMarkdown [3] and remark-gfm, which were vital to the completion of our project as they allowed us to render user text into markdown. React also allows us to use the jest testing framework [4] , which provides a simple and intuitive interface for writing tests. Jest’s mocking capabilities were important to us as it allowed us to easily mock dependencies and simulate different scenarios in our tests. Jest can also generate code coverage reports easily, helping us to identify areas of code that were not adequately covered by tests.

3.3 Gitea

Gitea is an open-source, self-hosted Git service similar to GitHub and GitLab and is lightweight, easy to install, and highly customizable. We make use of its API to allow us to integrate git features into our application, enabling us to make repositories for users, maintain a history of files created and increase the overall shareability and accessibility of our project.
[5]

3.4 Android Studio

Android studio is the official integrated development environment for android app development. As such it is officially supported and maintained by Google meaning developers get access to the latest features and support from the platform's creators. Built on JetBrains' IntelliJ IDEA software, it offers a feature rich environment. One of the most useful features is the ability to run your apps on an emulated mobile device. Allowing us to efficiently test our app features and changes. [6]

3.5 Robolectric

Robolectric is a framework for writing fast and reliable android tests that don't require an emulator to run. Robolectric tests run in a regular JVM making them much quicker to run on your own machine or they can be run on a continuous integration environment.
We chose to integrate robolectric tests into our continuous integration pipeline so we could ensure that mobile app code was being tested before it could be integrated into the main code base. [7]

3.6 Retrofit

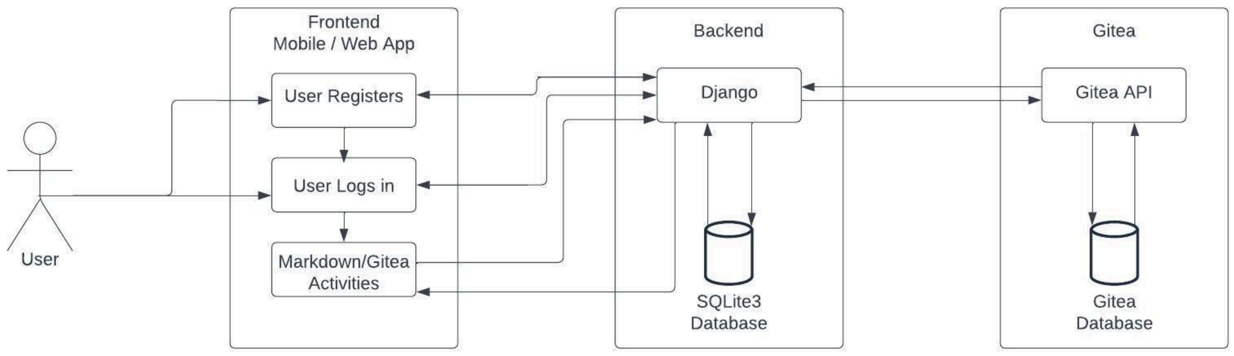
Retrofit is a type-safe HTTP client for Android and Java.
We used retrofit because it gave us a simple but efficient way to perform REST requests to our Django API. Retrofit can be integrated with various JSON parsing libraries (eg. Gson) to easily deserialize API responses into Java objects. [8]

3.7 Markdown rendering (Markwon)

Markwon is a library for android that is able to parse markdown and render it as Android-native spannables. We chose to use Markwon for rendering markdown because it allows us to customise the rendering of markdown text to our needs. It is also highly extensible which allowed us to extend its functionality with a number of plugins. We also liked that it comes with a markdown editor that is capable of highlighting markdown input in edit text components.
[9]

4. Design

4.1 System Architecture

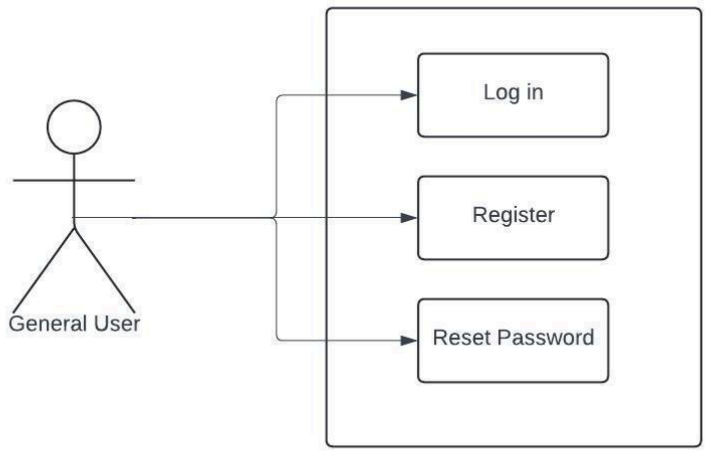


The System architecture shows at a high level the interactions in our system, it consists of the frontend which is our Mobile Application written in java or Web Application written in java script, our django backend written in python and the external Gitea API. A user can register their account on either the web app or mobile app which will create their account on the django backend and Gitea server. After they have done this they can then log in and have access to the rest of our applications functions such as creating and sharing markdown files. The django backend uses sqlite to store user information such as name, email, password and it assigns a JSON Web Token (JWT) token to each user on login which it then uses to decipher who a user is when a request is made to the server. The Django server then makes requests to the Gitea API and manipulates the data returned as it needs before sending it back to the frontend to be displayed to the user.

4.2 High Level Design

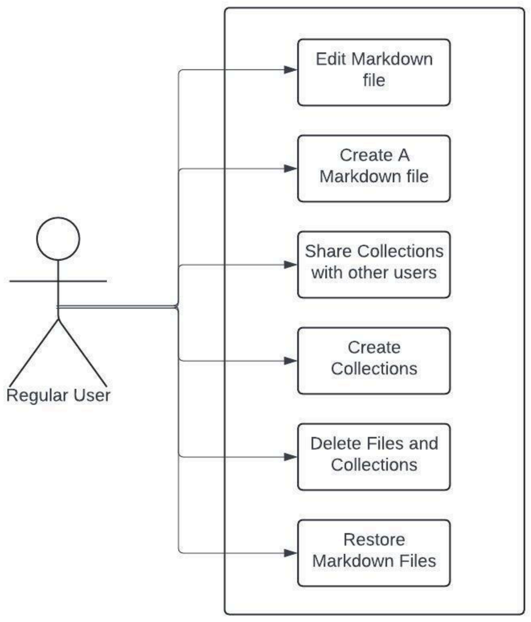
4.2.1 General Capabilities

This shows services a user has access to without being logged into an account. The three features a user can use are registering an account to use our application, logging into an already created account and resetting the password of an already created account however you must have access to the email associated with this account to be able to complete this action.



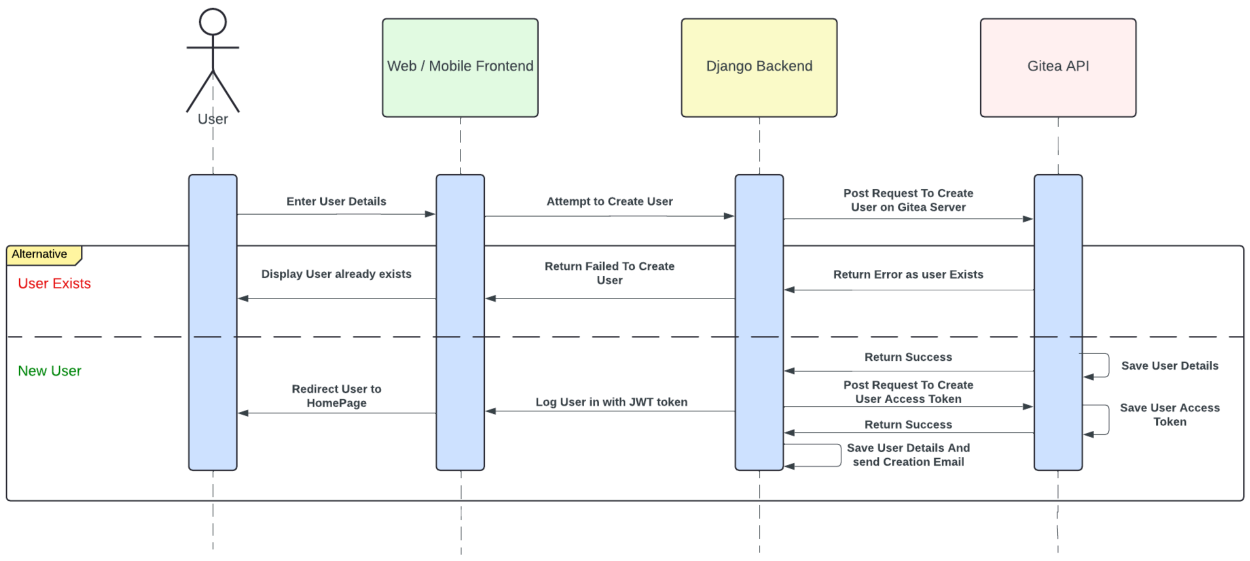
4.2.2 Registered User

This shows the services a user has while being logged into a created account. This is the main bulk of our features and includes creating a collection, creating markdown files inside a collection, editing those created files, deleting markdown files and collections, restoring deleted markdown files and sharing collections with other users.



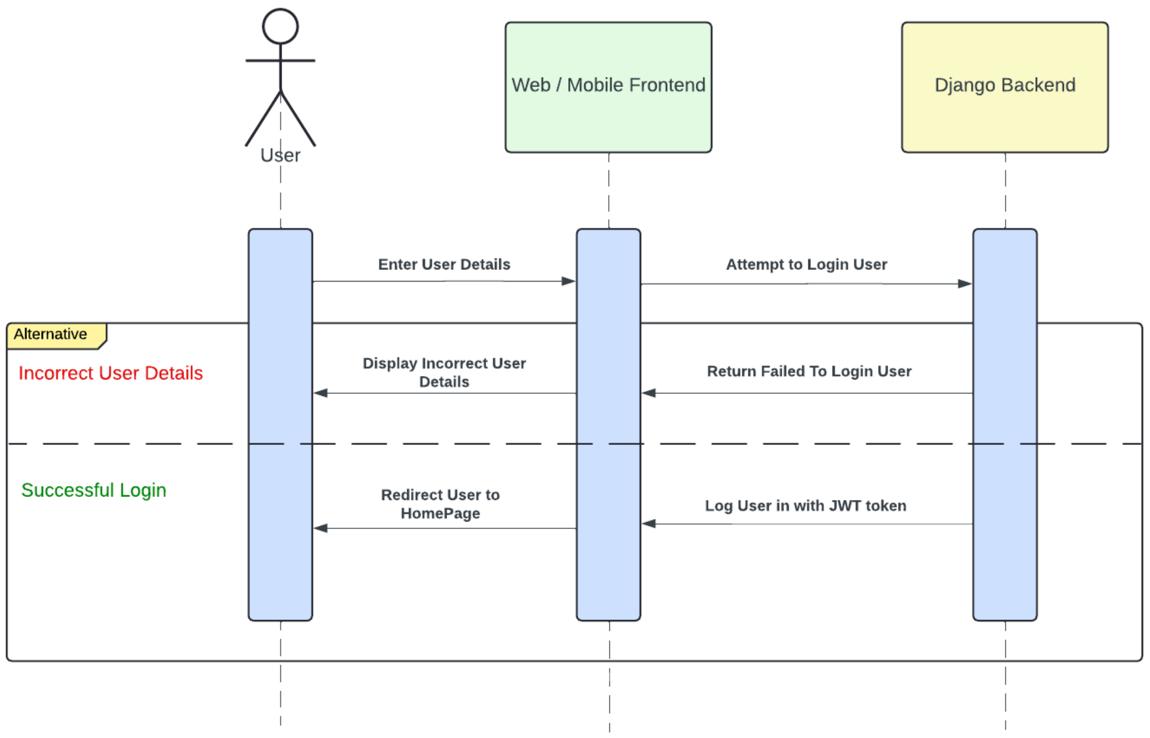
4.3 Sequence Diagrams

4.3.1 Register


User
Web / Mobile Frontend
Django Backend
Gitea API
Enter User Details
Attempt to Create User
Post Request To Create User on Gitea Server
Alternative
User Exists
Display User already exists
Return Failed To Create User
Return Error as user Exists
New User
Return Success
Save User Details
Redirect User to HomePage
Log User in with JWT token
Post Request To Create User Access Token
Save User Access Token
Return Success
Save User Details And send Creation Email

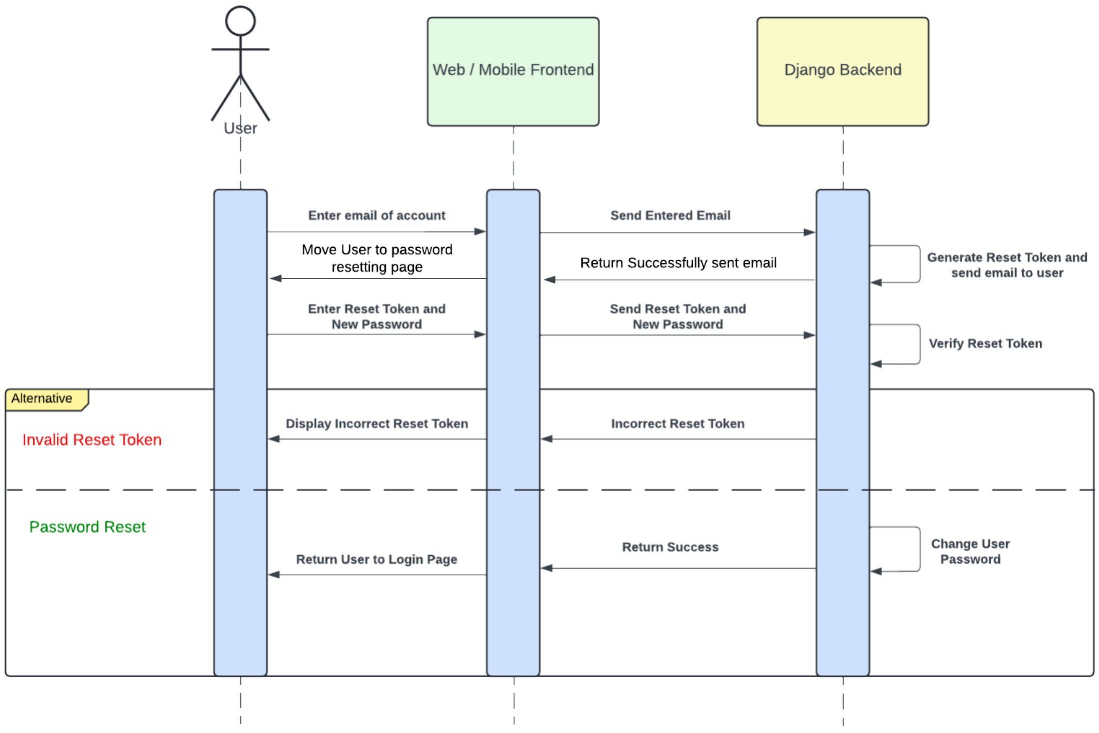
In the register sequence diagram a user enters their account details (username, email and password) which is then sent to the django backend. The django backend then passes this information as a post request to the gitea api which attempts to make the users account. If the user already exists the error message is returned to the django backend so the user will not be saved there and the error is then sent to the frontend where it can be displayed to the user. Otherwise the User is created and saved on the gitea server, following that a user access token is then created and both the user details and the token are sent back to the django backend and saved. The django backend then creates a JSON Web Token to authenticate the user and sets it as a cookie back in the frontend. Finally the user is redirected to the home page.

4.3.2 Login



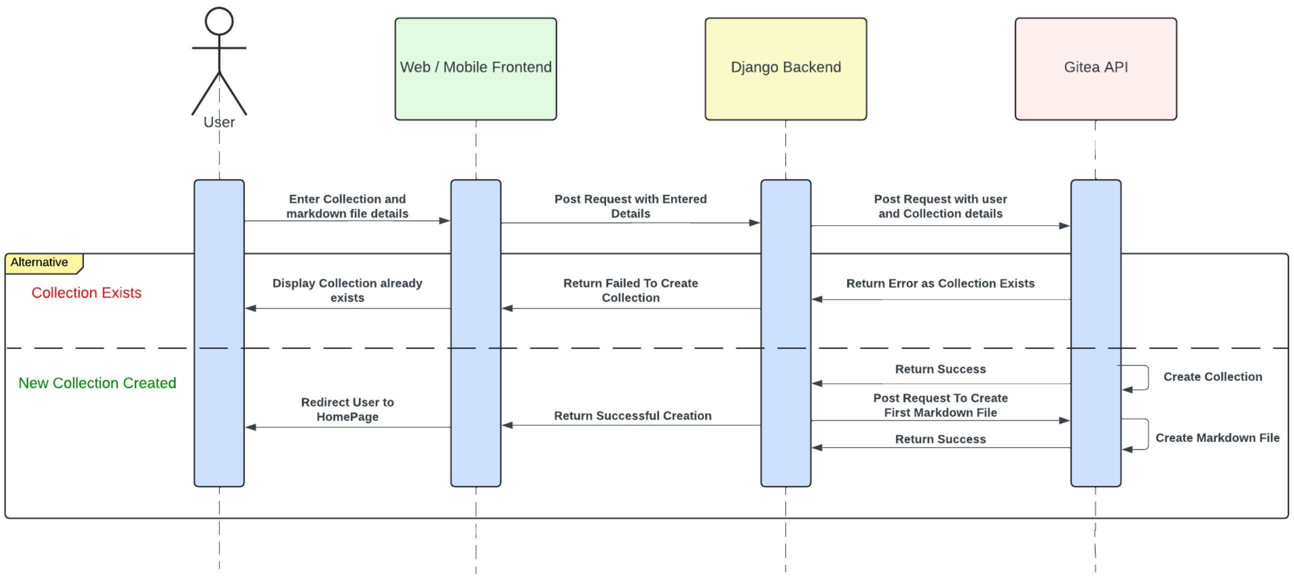
When a user attempts to login their entered details are sent to the django backend where first the entered email is checked to whether the user exists or not, following this the password is then checked to see if the entered password matches the user's password. If either of these fail the error is then sent back to the frontend and displayed to the user otherwise a JWT is created and used for authenticating the user in future api calls.

4.3.3 Reset Password


User
Web / Mobile Frontend
Django Backend
Enter email of account
Send Entered Email
Generate Reset Token and send email to user
Move User to password resetting page
Return Successfully sent email
Enter Reset Token and New Password
Send Reset Token and New Password
Verify Reset Token
Alternative
Invalid Reset Token
Display Incorrect Reset Token
Incorrect Reset Token
Password Reset
Return Success
Change User Password
Return User to Login Page

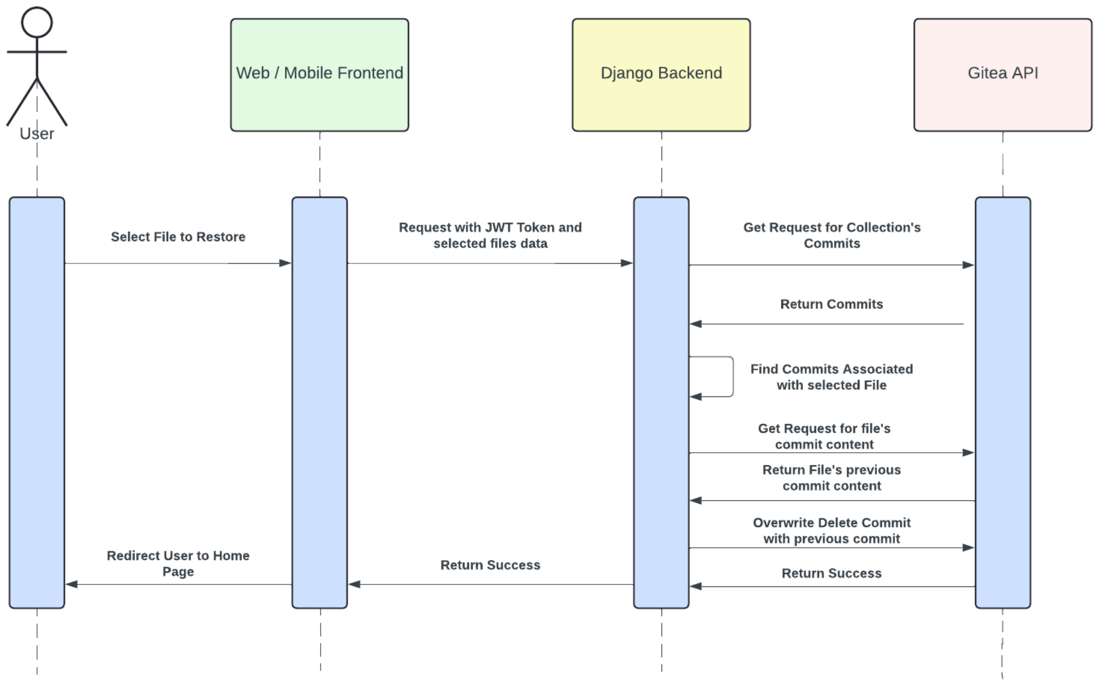
When a user attempts to reset their password, they must enter the email address of the account in which they wish to change the password. After the django backend receives this information it will generate a reset token that the user will need to enter in order to reset their password, and following this an email will be sent to the user containing this token. The user will then enter the reset token they received and their new password, and the django server will verify if this reset token is correct if it is the password will be changed otherwise the user will be displayed the error message.

4.3.4 Create Collection


User
Web / Mobile Frontend
Django Backend
Gitea API
Enter Collection and markdown file details
Post Request with Entered Details
Post Request with user and Collection details
Alternative
Collection Exists
Display Collection already exists
Return Failed To Create Collection
Return Error as Collection Exists
New Collection Created
Return Success
Create Collection
Redirect User to HomePage
Return Successful Creation
Post Request To Create First Markdown File
Return Success
Create Markdown File

On creating a collection a user enters their Collection and markdown file details into the frontend which transform the data to be used in a post request to the gitea API. if the collection is already created the error is sent back to be displayed to the user. Otherwise the collection is made and once the collection is successfully made the first markdown file is created in that collection. Finally the user is returned to the home page.

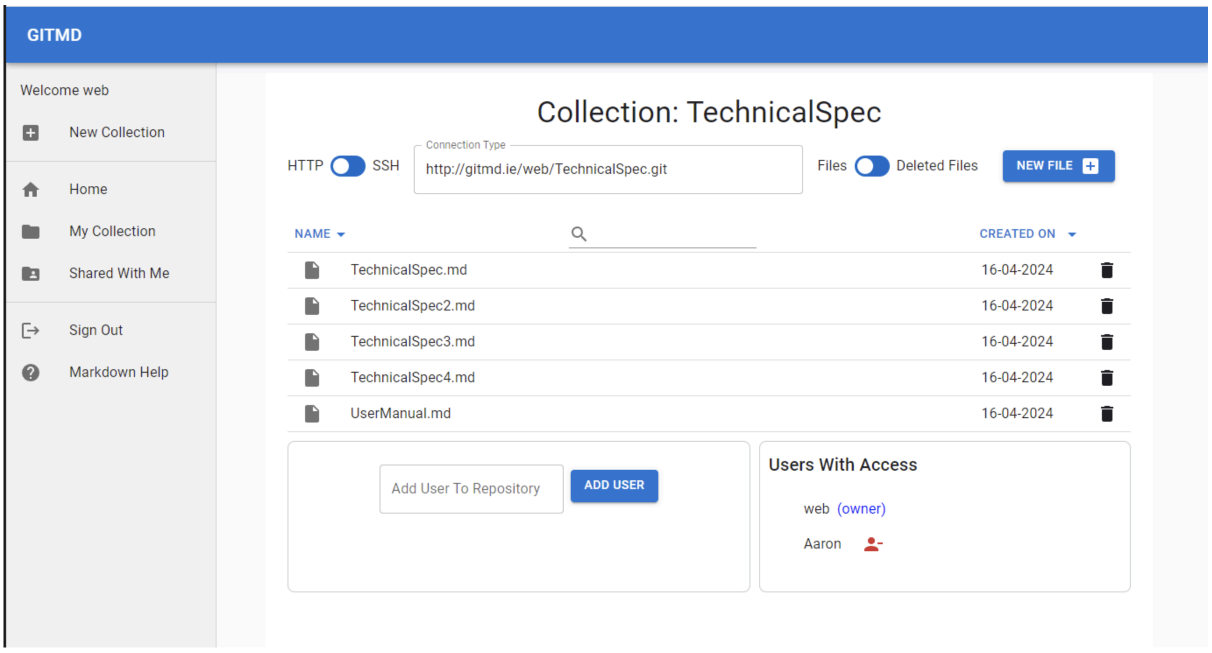
4.3.5 Restore Markdown File To Previous Version


User
Web / Mobile Frontend
Django Backend
Gitea API
Select File to Restore
Request with JWT Token and selected files data
Get Request for Collection's Commits
Return Commits
Find Commits Associated with selected File
Get Request for file's commit content
Return File's previous commit content
Overwrite Delete Commit with previous commit
Redirect User to Home Page
Return Success
Return Success

When restoring a markdown file to a previous version the selected file and users JWT is sent to the django backend. The django backend then makes a get request to get all the commits in the repository that the file belongs to. Once the commits are returned the backend sifts through to find all the commits associated with the selected file to be restored. A get request is then made to get the file to be srestored's content so that we can overwrite the delete commit with a commit that has the old content and a success is returned to the frontend so the user can be redirected back to the homepage.

4.4 UI Design

We followed a common colour scheme throughout web and mobile applications of white and blue. The simplicity of white backgrounds paired with blue creates a polished and professional look that appeals to users, the colours also have good contrast making our project accessible to users with visual impairments as high contrast between both colours improves readability and ensures context is distinguishable. We also used icons such as trash icons for deleting files or a small house for home on the navigation tab to ensure users were able to easily understand different functions without having to spend too much time thinking about each click they have to make. It was important to keep the colour scheme and layout of both the Web application and mobile application the same to ensure users would be able to seamlessly use both without learning a whole new UI. Material UI was used for the web application to ensure a consistent design throughout and also emphasising accessibility as MUI provides accessible components and adheres to accessibility best practices.



Collection: TechnicalSpec

Connection Type: http://gitmd.ie/web/TechnicalSpec.git

NAME	CREATED ON
TechnicalSpec.md	16-04-2024
TechnicalSpec2.md	16-04-2024
TechnicalSpec3.md	16-04-2024
TechnicalSpec4.md	16-04-2024
UserManual.md	16-04-2024

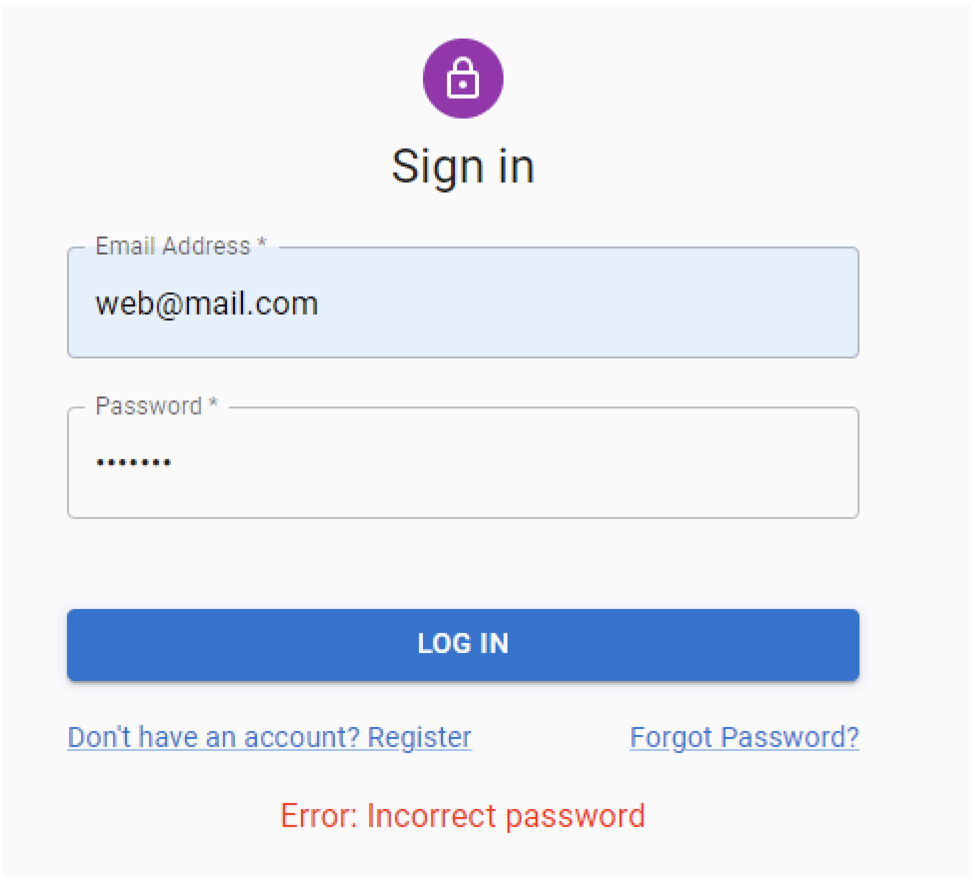
Add User To Repository

Users With Access

web (owner)

Aaron

We also wanted to ensure users got feedback from the applications when they interacted with the interface and did this by quick transitions to new pages, buttons fading darker when hovered and incorrect forms supplied error messages when a problem occurred with a datafield



Sign in

Email Address *
web@mail.com

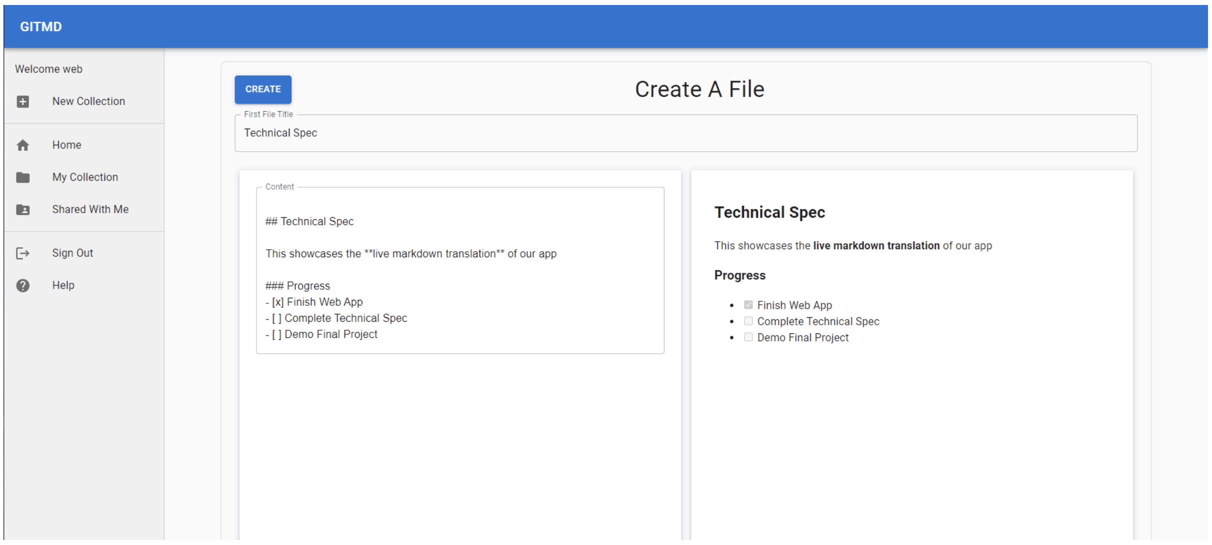
Password *
•••••••

LOG IN

Don't have an account? Register
Forgot Password?

Error: Incorrect password

Below we see a user creating a markdown file within a collection. Any entered content is live translated into markdown on the right page so that users can get a live preview of what they are writing. This is an important feature as we wanted to ensure users are getting constant feedback on their actions and do not have to repeatedly edit files to fix mistakes they might have made in their file creation.



Create A File

First File Title
Technical Spec

Content

```
## Technical Spec

This showcases the **live markdown translation** of our app

### Progress
- [x] Finish Web App
- [ ] Complete Technical Spec
- [ ] Demo Final Project
```

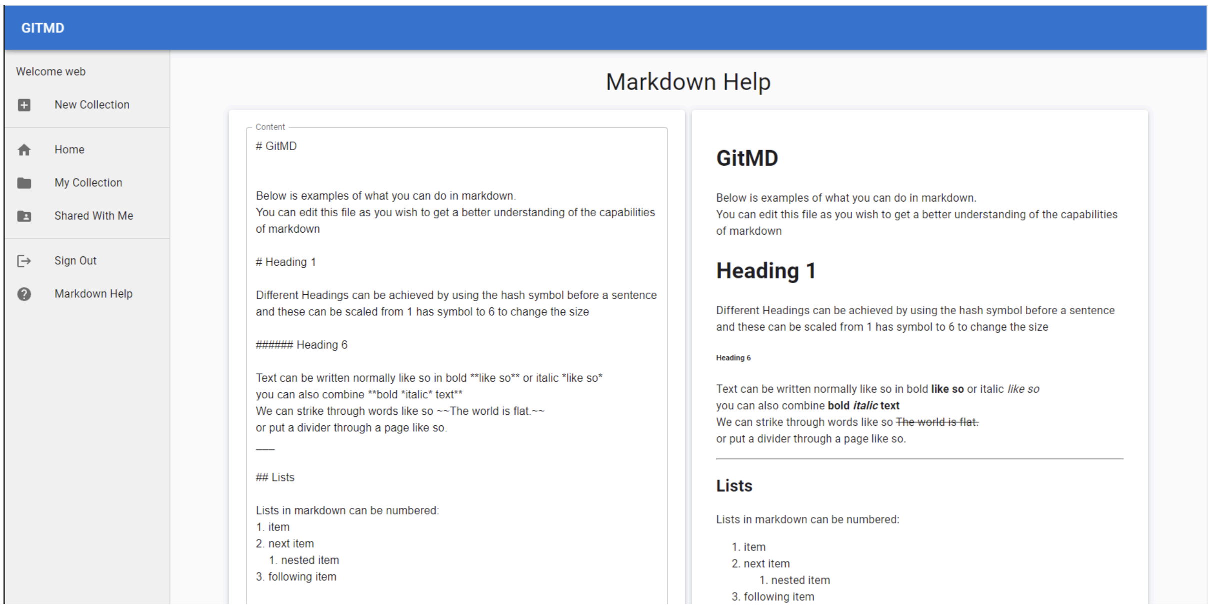
Technical Spec

This showcases the **live markdown translation** of our app

Progress

- ☑ Finish Web App
- ☐ Complete Technical Spec
- ☐ Demo Final Project

We also give users a helping hand with writing in markdown in case any beginners of writing in markdown files wish to use our application. Users are shown all the capabilities of markdown and can edit the file themselves to try and get to grips with how certain features work.


GITMD

Welcome web

New Collection

Home

My Collection

Shared With Me

Sign Out

Markdown Help

Markdown Help

Content
```
# GitMD

Below is examples of what you can do in markdown.
You can edit this file as you wish to get a better understanding of the capabilities
of markdown

# Heading 1

Different Headings can be achieved by using the hash symbol before a sentence
and these can be scaled from 1 has symbol to 6 to change the size

###### Heading 6

Text can be written normally like so in bold **like so** or italic *like so*
you can also combine **bold *italic* text**
We can strike through words like so ~~The world is flat.~~
or put a divider through a page like so.
___

## Lists

Lists in markdown can be numbered:
1. item
2. next item
   1. nested item
3. following item
```

GitMD

Below is examples of what you can do in markdown.
You can edit this file as you wish to get a better understanding of the capabilities of markdown

Heading 1

Different Headings can be achieved by using the hash symbol before a sentence and these can be scaled from 1 has symbol to 6 to change the size

Heading 6

Text can be written normally like so in bold **like so** or italic *like so*
you can also combine **bold *italic* text**
We can strike through words like so ~~The world is flat.~~
or put a divider through a page like so.

Lists

Lists in markdown can be numbered:

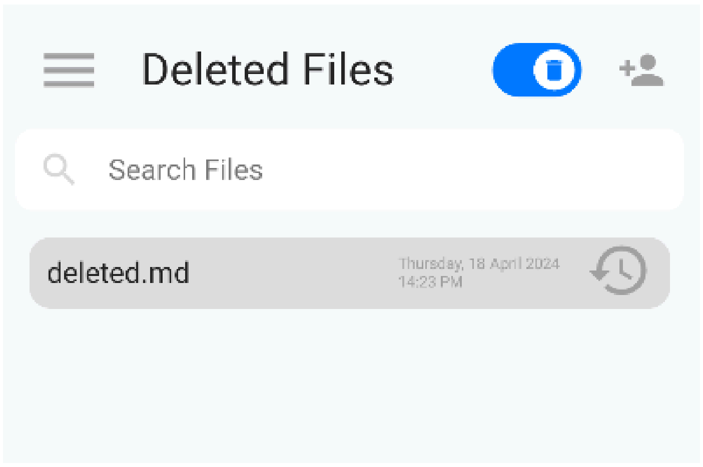
1. item
2. next item
   1. nested item
3. following item

Mobile UI Considerations

Due to mobile devices being touchscreen and generally having a small form factor compared to monitors or laptop screens, we designed some elements of the ui differently. Our overall goal was to make the web and mobile app have a similar look and feel overall but we allowed for some differences that could improve the user experience.

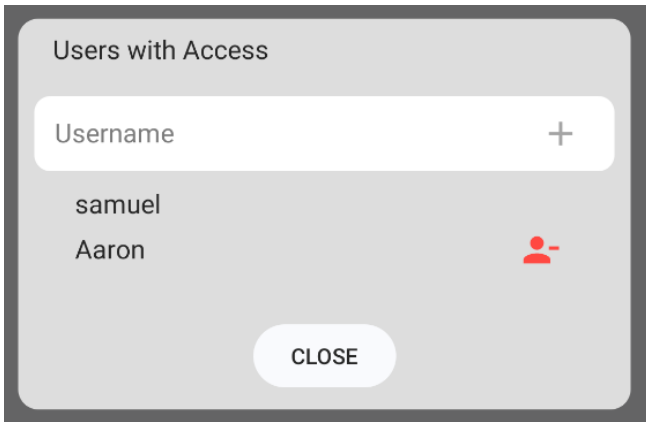
Icons

To conserve space, we used icons to portray the functionality of controls (buttons, switches etc). Using common icons makes the behaviour of the ui more intuitive.



Dialogs

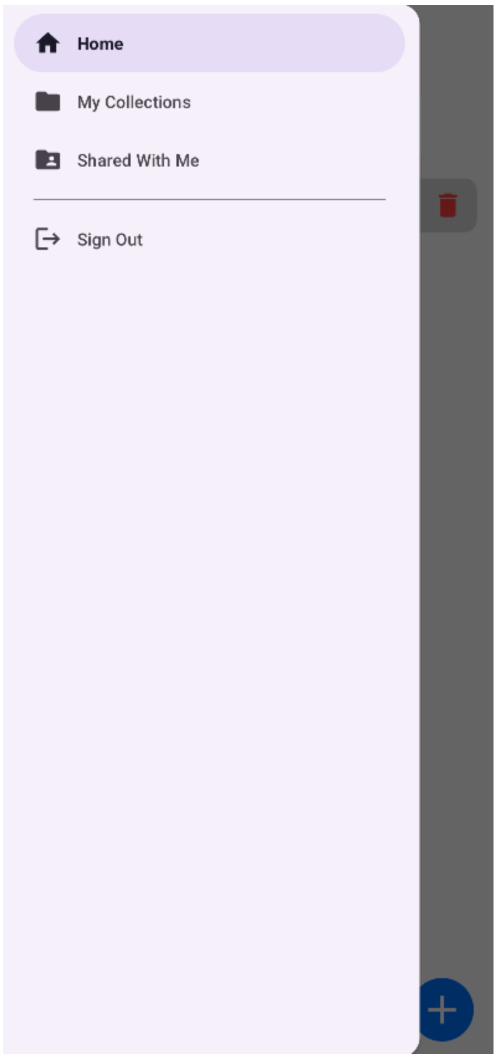
Dialogs are used to provide information to the user or to ask for a decision.They appear in front of app content and remain on screen until dismissed or a required decision is made.



The above dialog example provides the user with information about the Users with access to a specific collection. It also allows users to add or remove access from other users. The dialog can be dismissed by pressing the 'close' button.

Navigation Drawer

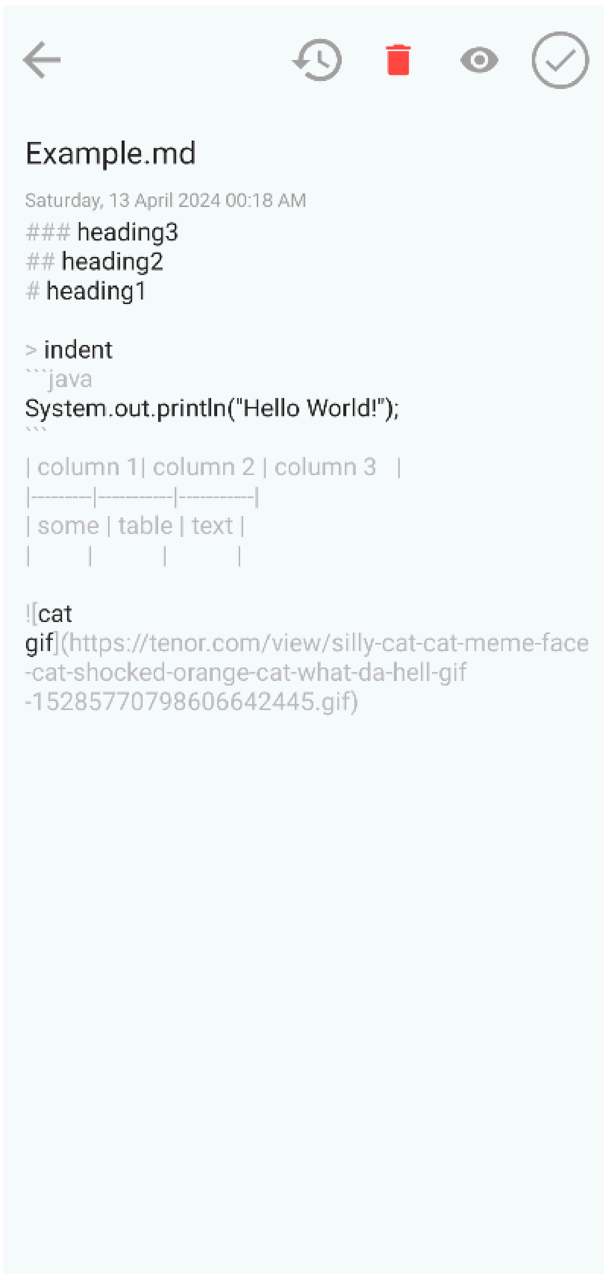
Navigation drawers are used to provide access to navigation and other functionality without having them permanently taking up space on screen, it is accessed by toggling the drawer open or closed.



We use a nav drawer to navigate between all collections, collections that a user owns and collections that have been shared with a user. It also contains the sign out functionality which will navigate the user back to the sign in screen.

Markdown Editor and View button

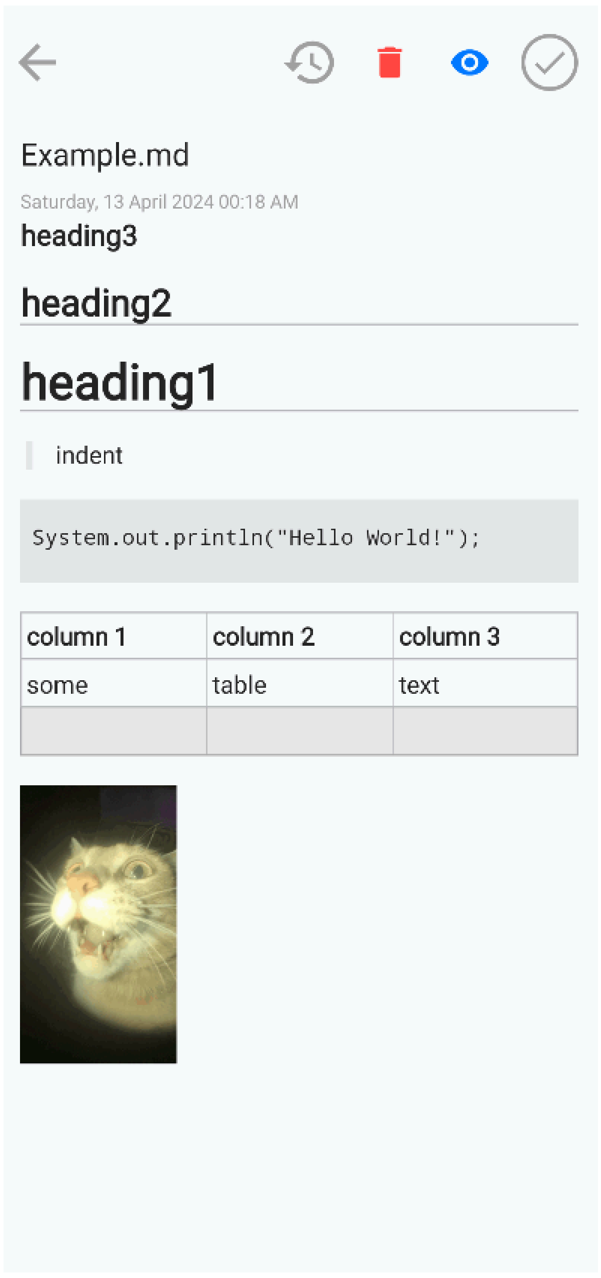
One of the biggest differences in the ui between the mobile and web app is how users can view the rendered markdown. Due to the limited space on mobile devices, we could not fit the two windows for text editing and live translation. Instead the user can edit file content in an editText field which has markdown syntax highlighting and then toggle to view the rendered version.



Example.md

Saturday, 13 April 2024 00:18 AM

~~~
### heading3
## heading2
# heading1

> indent
```java
System.out.println("Hello World!");
```

| column 1| column 2 | column 3   |
|--------|----------|----------|
| some | table | text |
|      |       |       |

![cat
gif](https://tenor.com/view/silly-cat-cat-meme-face
-cat-shocked-orange-cat-what-da-hell-gif
-15285770798606642445.gif)
~~~



Example.md

Saturday, 13 April 2024 00:18 AM

heading3

heading2

heading1

> indent

```
System.out.println("Hello World!");
```

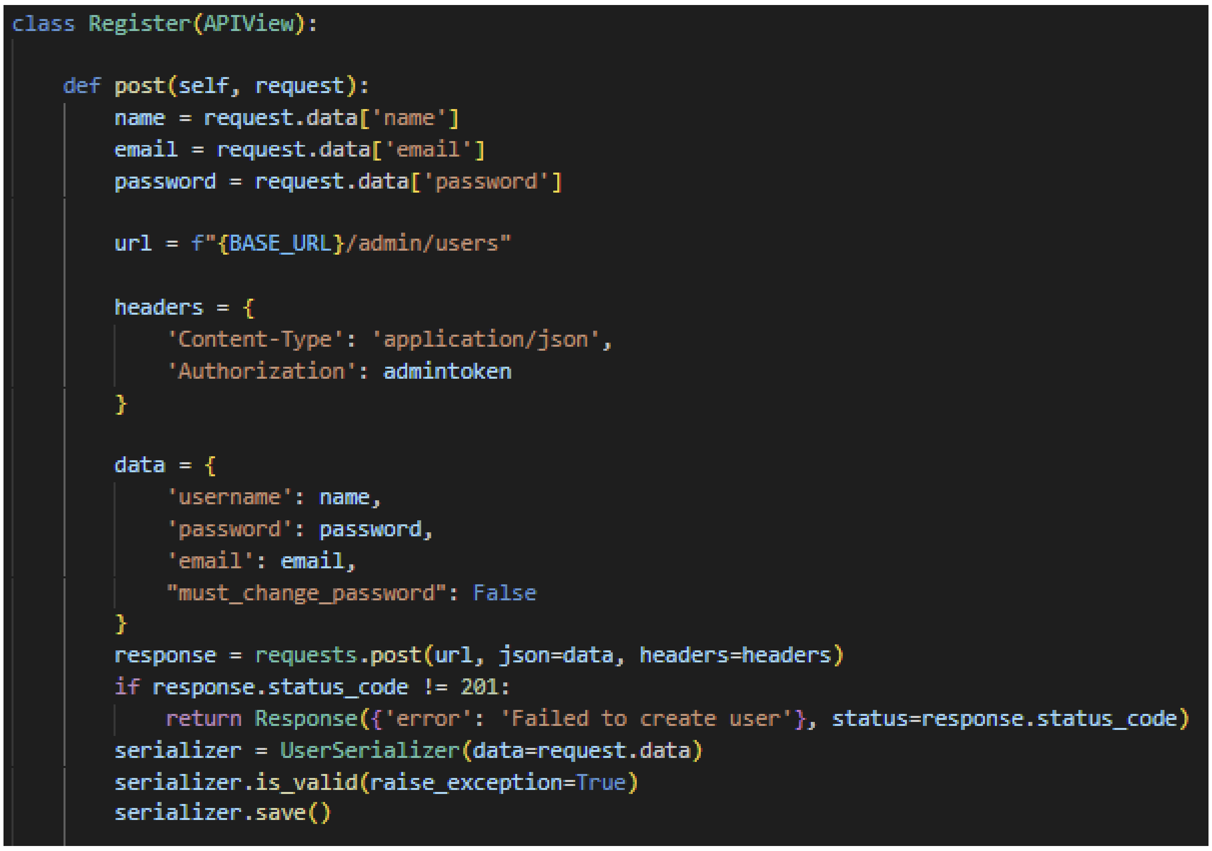
column 1	column 2	column 3
some	table	text

5. Implementation

5.1 Django

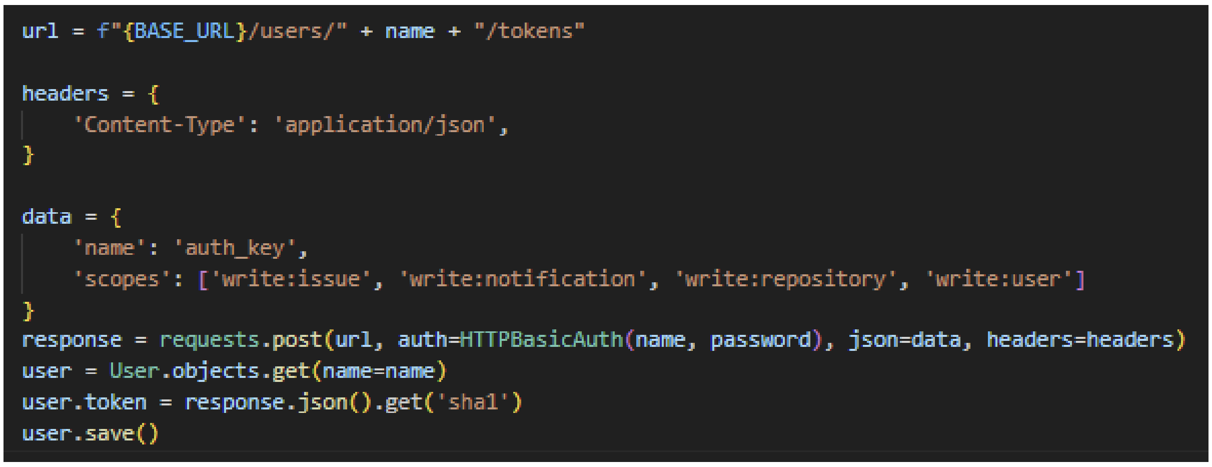
5.1.1 Register

There are three parts to registering a user, first there is creating the user, then we have to create them an access key so that they can interact with the gitea services and finally an email confirmation to the user. To register the user we gather the entered user details (name, email, password) and send them to the gitea api to create the user on the gitea server, and upon successful creation, we then save the user details onto our django server.



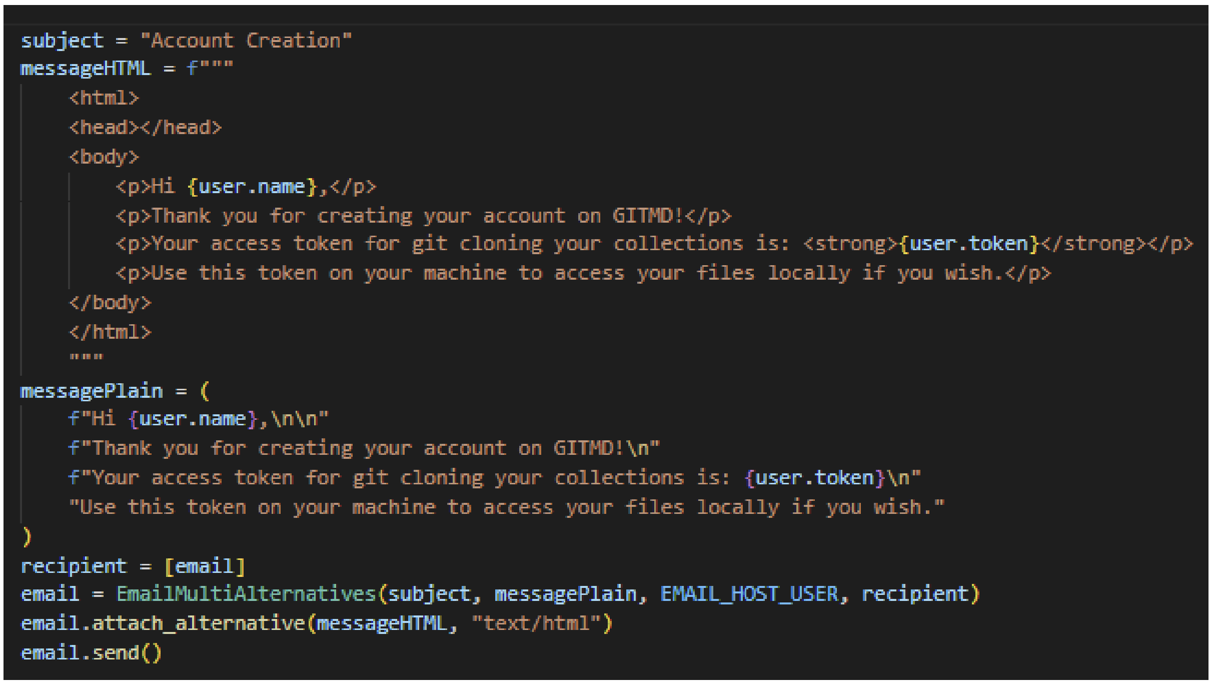
```python
class Register(APIView):

    def post(self, request):
        name = request.data['name']
        email = request.data['email']
        password = request.data['password']

        url = f"{BASE_URL}/admin/users"

        headers = {
            'Content-Type': 'application/json',
            'Authorization': admintoken
        }

        data = {
            'username': name,
            'password': password,
            'email': email,
            "must_change_password": False
        }
        response = requests.post(url, json=data, headers=headers)
        if response.status_code != 201:
            return Response({'error': 'Failed to create user'}, status=response.status_code)
        serializer = UserSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
```

Next we must make the user an access token so that they can make and interact with repositories they make on the gitea server and also clone these repositories to their local machines. Once it has been successfully created we save it to the user model on our django server.



```python
url = f"{BASE_URL}/users/" + name + "/tokens"

headers = {
    'Content-Type': 'application/json',
}

data = {
    'name': 'auth_key',
    'scopes': ['write:issue', 'write:notification', 'write:repository', 'write:user']
}
response = requests.post(url, auth=HTTPBasicAuth(name, password), json=data, headers=headers)
user = User.objects.get(name=name)
user.token = response.json().get('sha1')
user.save()
```

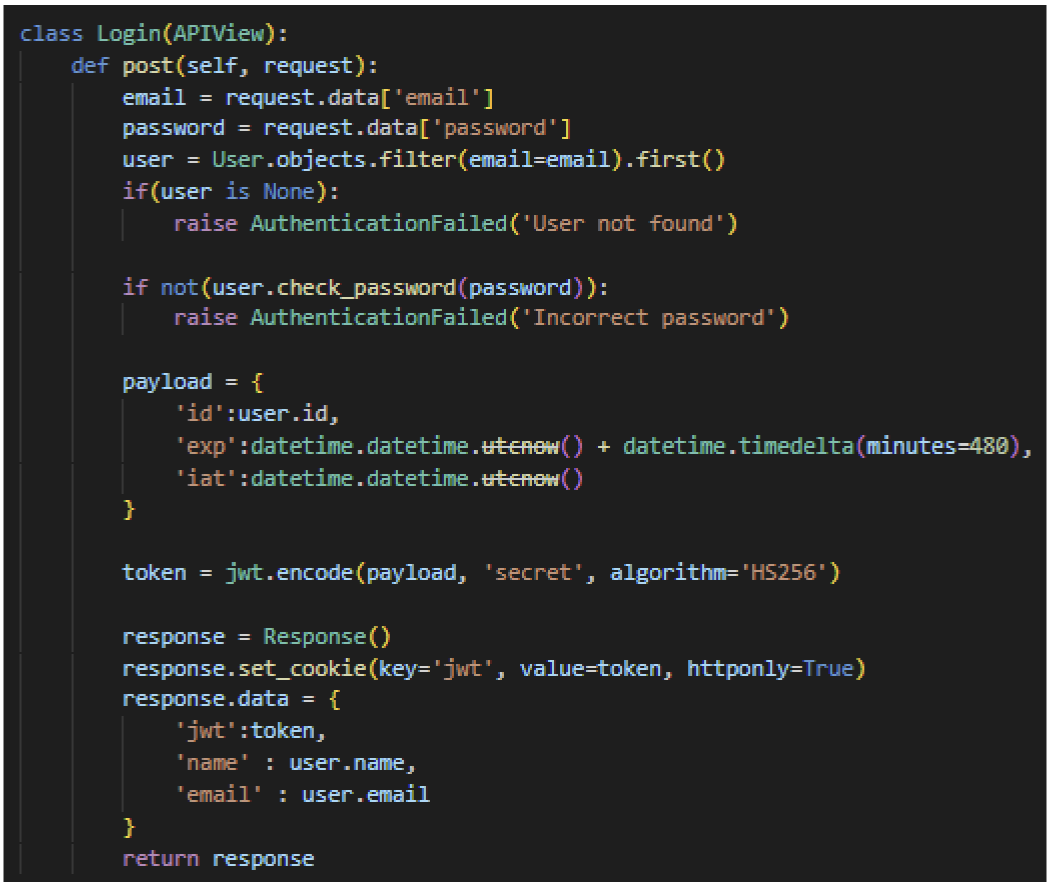
And finally we email the user that their account was successfully created and include their access token within this email so that they can use it to view and edit their collections locally.



```python
subject = "Account Creation"
messageHTML = f"""
    <html>
    <head></head>
    <body>
        <p>Hi {user.name},</p>
        <p>Thank you for creating your account on GITMD!</p>
        <p>Your access token for git cloning your collections is: <strong>{user.token}</strong></p>
        <p>Use this token on your machine to access your files locally if you wish.</p>
    </body>
    </html>
    """
messagePlain = (
    f"Hi {user.name},\n\n"
    f"Thank you for creating your account on GITMD!\n"
    f"Your access token for git cloning your collections is: {user.token}\n"
    "Use this token on your machine to access your files locally if you wish."
)
recipient = [email]
email = EmailMultiAlternatives(subject, messagePlain, EMAIL_HOST_USER, recipient)
email.attach_alternative(messageHTML, "text/html")
email.send()
```

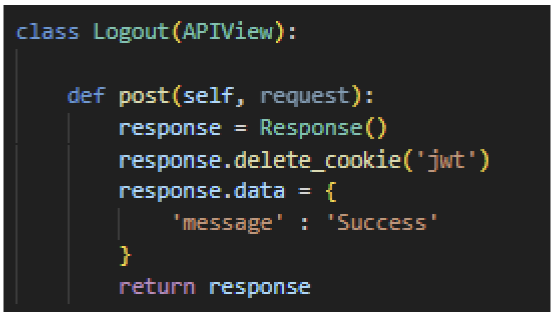
5.1.2 Login / Logout

Logging in and out of an account is relatively simple through the use of JSON Web Tokens. When a user logs in we verify that a user with the entered email exists and if so if the passwords match. If they do we create a JWT that is active for 8 hours before authentication is required again and that JWT is encoded using the HS256
 Algorithm and set as a cookie to be used by the user in future api requests.



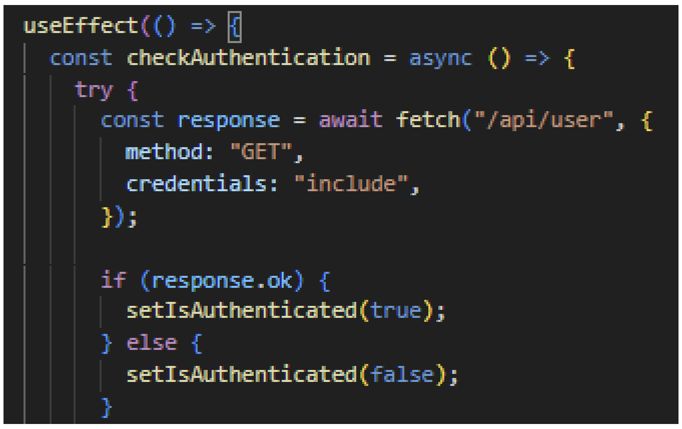
```python
class Login(APIView):
    def post(self, request):
        email = request.data['email']
        password = request.data['password']
        user = User.objects.filter(email=email).first()
        if(user is None):
            raise AuthenticationFailed('User not found')

        if not(user.check_password(password)):
            raise AuthenticationFailed('Incorrect password')

        payload = {
            'id':user.id,
            'exp':datetime.datetime.~~utcnow~~() + datetime.timedelta(minutes=480),
            'iat':datetime.datetime.~~utcnow~~()
        }

        token = jwt.encode(payload, 'secret', algorithm='HS256')

        response = Response()
        response.set_cookie(key='jwt', value=token, httponly=True)
        response.data = {
            'jwt':token,
            'name' : user.name,
            'email' : user.email
        }
        return response
```

Logging out is relatively straightforward as we just delete the JWT cookie



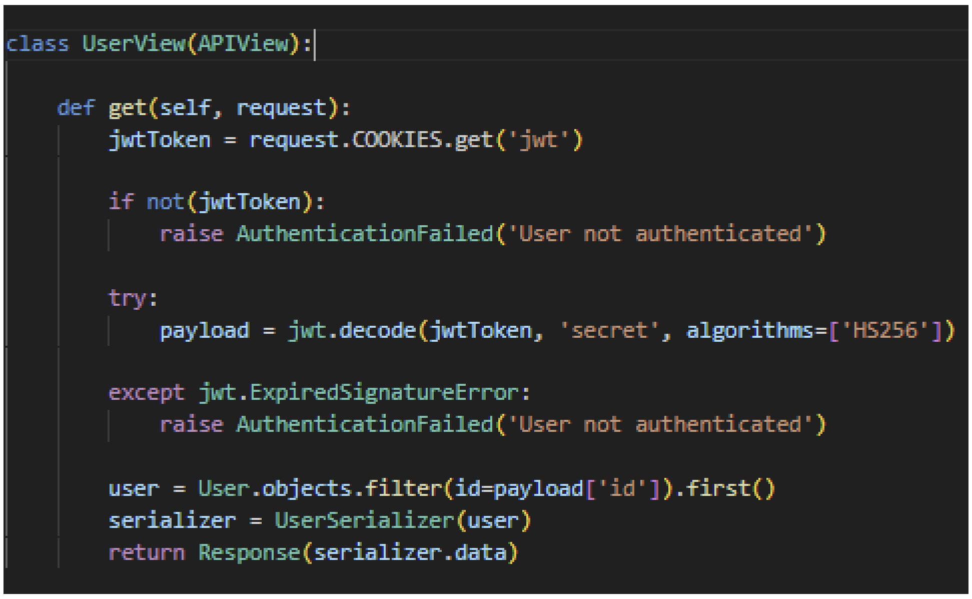
```python
class Logout(APIView):

    def post(self, request):
        response = Response()
        response.delete_cookie('jwt')
        response.data = {
            'message' : 'Success'
        }
        return response
```

Access is checked on the web app when the user attempts to access any page that isn't the login page using a PrivateRoute.js file. This file makes an api request and includes the JWT a user receives on login



```javascript
useEffect(() => {
  const checkAuthentication = async () => {
    try {
      const response = await fetch("/api/user", {
        method: "GET",
        credentials: "include",
      });

      if (response.ok) {
        setIsAuthenticated(true);
      } else {
        setIsAuthenticated(false);
      }
```

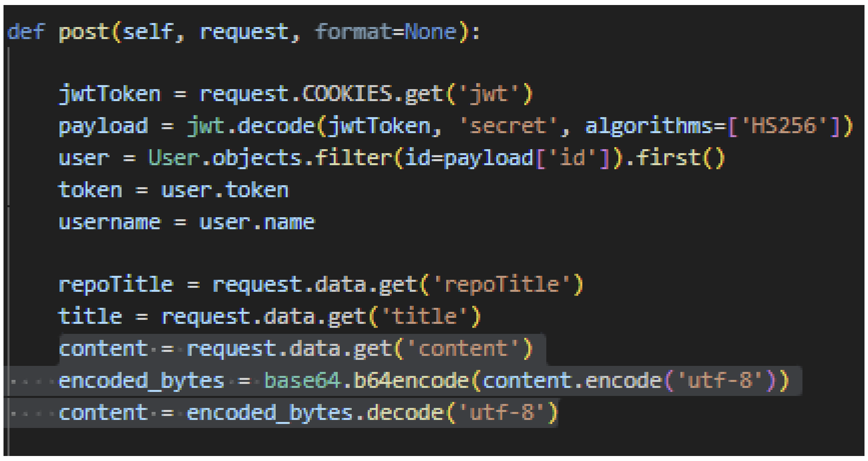
We then take this token and check its authenticity to determine whether a user has access to further web pages



```python
class UserView(APIView):

    def get(self, request):
        jwtToken = request.COOKIES.get('jwt')

        if not(jwtToken):
            raise AuthenticationFailed('User not authenticated')

        try:
            payload = jwt.decode(jwtToken, 'secret', algorithms=['HS256'])

        except jwt.ExpiredSignatureError:
            raise AuthenticationFailed('User not authenticated')

        user = User.objects.filter(id=payload['id']).first()
        serializer = UserSerializer(user)
        return Response(serializer.data)
```

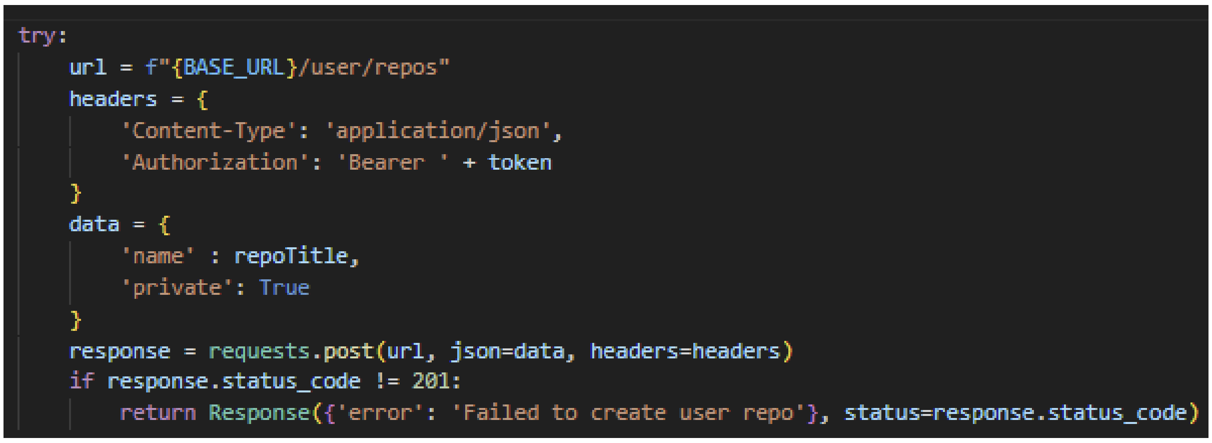
5.1.3 Creating a Markdown file / Collection

For both of these steps we use the same post request. First we have to gather and organise our data. We first use the JWT to determine who the user is and then after gathering the entered data from the user.. We then have to change content to be a base64 encoded string so that we can use it in the api call to the gitea server.



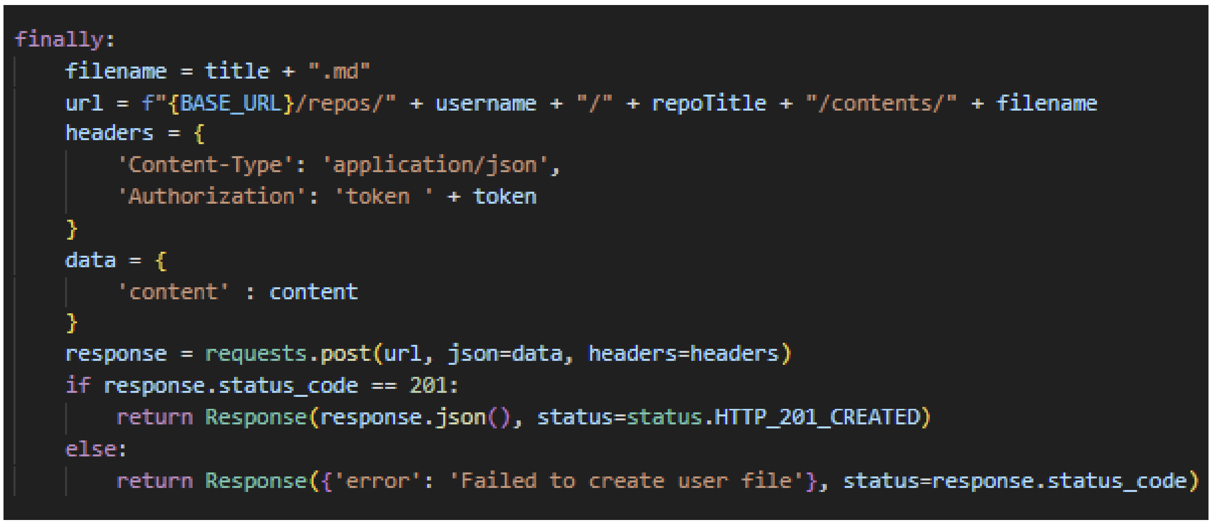
```python
def post(self, request, format=None):

    jwtToken = request.COOKIES.get('jwt')
    payload = jwt.decode(jwtToken, 'secret', algorithms=['HS256'])
    user = User.objects.filter(id=payload['id']).first()
    token = user.token
    username = user.name

    repoTitle = request.data.get('repoTitle')
    title = request.data.get('title')
    content = request.data.get('content')
    encoded_bytes = base64.b64encode(content.encode('utf-8'))
    content = encoded_bytes.decode('utf-8')
```

Next we try to make a repository for the user, if repoTitle is an already created repository this means the user is trying to create a file inside this repository and we skip this step. If not we successfully make the post request setting the repository to private so that only the owner and other shared users can access it.



```
try:
    url = f"{BASE_URL}/user/repos"
    headers = {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + token
    }
    data = {
        'name' : repoTitle,
        'private': True
    }
    response = requests.post(url, json=data, headers=headers)
    if response.status_code != 201:
        return Response({'error': 'Failed to create user repo'}, status=response.status_code)
```

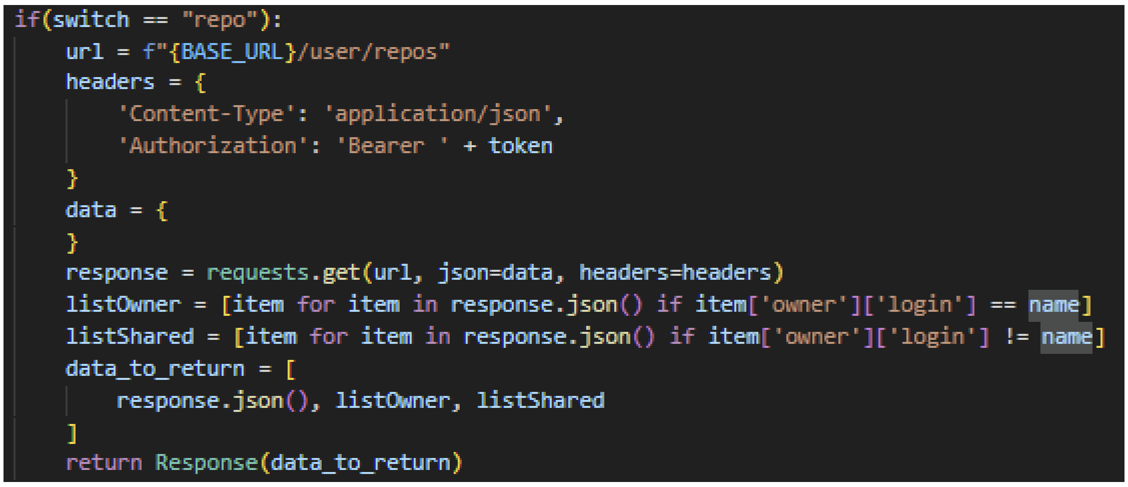
Finally, we add .md to the end of the title to ensure a markdown file is created and make a post request to the users repository to create the desired file.

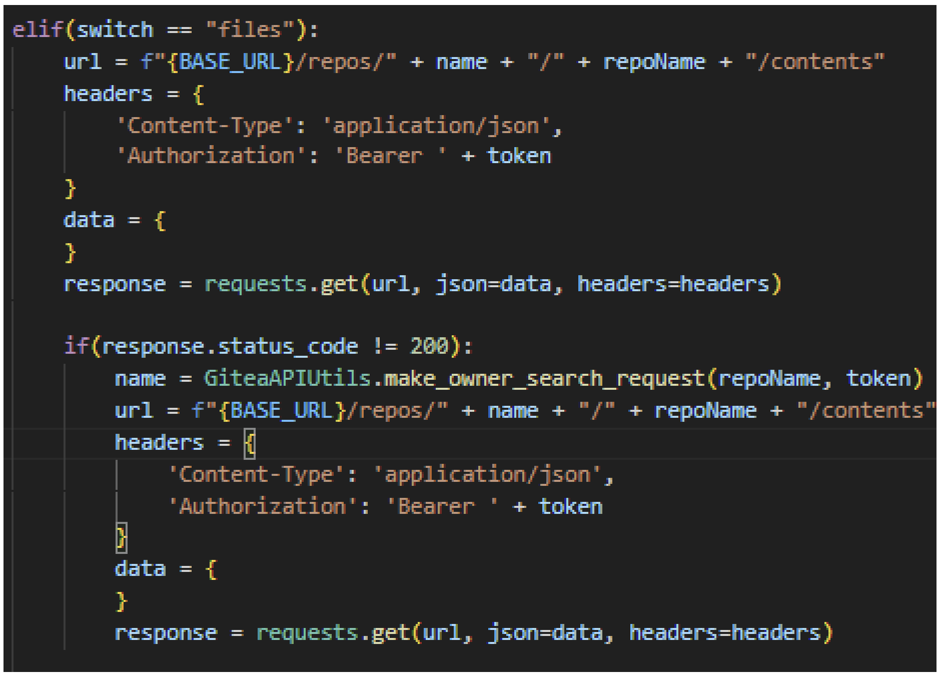


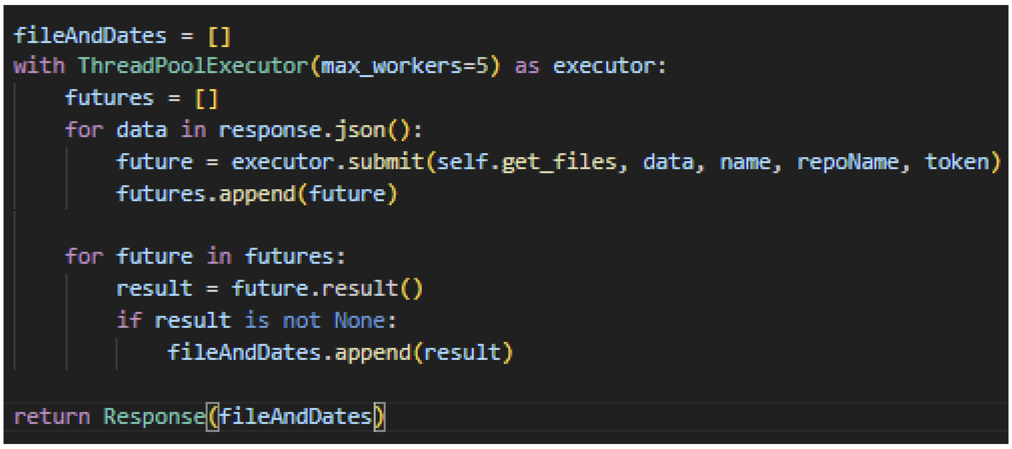
```
finally:
    filename = title + ".md"
    url = f"{BASE_URL}/repos/" + username + "/" + repoTitle + "/contents/" + filename
    headers = {
        'Content-Type': 'application/json',
        'Authorization': 'token ' + token
    }
    data = {
        'content' : content
    }
    response = requests.post(url, json=data, headers=headers)
    if response.status_code == 201:
        return Response(response.json(), status=status.HTTP_201_CREATED)
    else:
        return Response({'error': 'Failed to create user file'}, status=response.status_code)
```

5.1.4 Gathering Markdown files / Collections

To determine whether we are collecting repositories to display to the user or markdown files in a repository we use if statements on the same api call as the required data for both are the same. When gathering repositories we make a get request with the users access token, this will return all repositories they have access to. We then sort these repositories into 3 lists, the first being all repositories, the next being repositories the user is the owner of and last shared repositories.


```python
if(switch == "repo"):
    url = f"{BASE_URL}/user/repos"
    headers = {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + token
    }
    data = {
    }
    response = requests.get(url, json=data, headers=headers)
    listOwner = [item for item in response.json() if item['owner']['login'] == name]
    listShared = [item for item in response.json() if item['owner']['login'] != name]
    data_to_return = [
        response.json(), listOwner, listShared
    ]
    return Response(data_to_return)
```
For gathering markdown files in a repository we make a simple get request and if that doesn't work we change name to the owner of the repository and attempt to make the same call as in rare instances users added to a repository were unable to gain access to files.

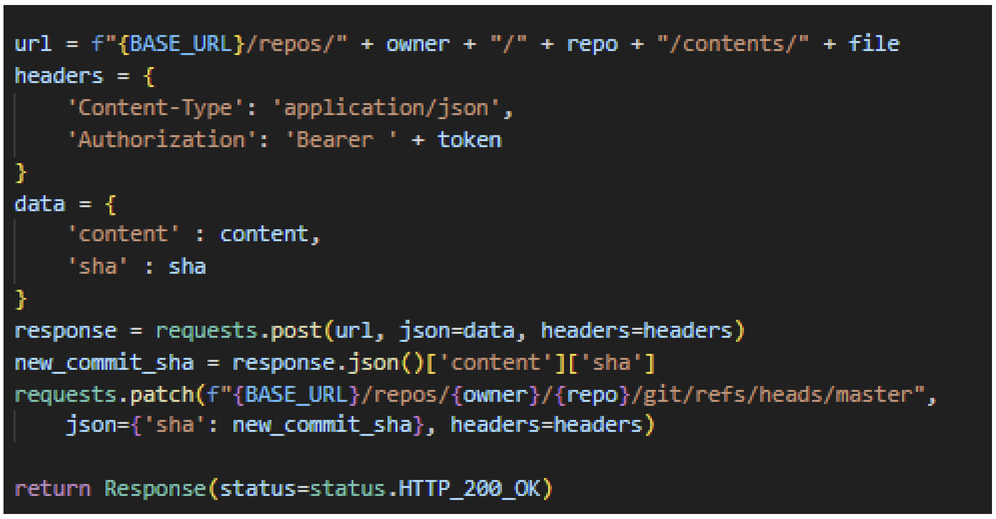

```python
elif(switch == "files"):
    url = f"{BASE_URL}/repos/" + name + "/" + repoName + "/contents"
    headers = {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + token
    }
    data = {
    }
    response = requests.get(url, json=data, headers=headers)

    if(response.status_code != 200):
        name = GiteaAPIUtils.make_owner_search_request(repoName, token)
        url = f"{BASE_URL}/repos/" + name + "/" + repoName + "/contents"
        headers = {
            'Content-Type': 'application/json',
            'Authorization': 'Bearer ' + token
        }
        data = {
        }
        response = requests.get(url, json=data, headers=headers)
```
To get the dates the files were created we had to make a get request to each file in a repository which took a few seconds to complete. To fix this we decided to thread this. We set max_workers to 5 to ensure we can complete 5 tasks concurrently. As we submit tasks, we store the future objects returned by executor.submit in a list called futures. This allows you to keep track of the tasks and retrieve their results later. Each file is sent into the get_files function which will get the name and date created of the file and add it to a list to be displayed to the user.



```
fileAndDates = []
with ThreadPoolExecutor(max_workers=5) as executor:
    futures = []
    for data in response.json():
        future = executor.submit(self.get_files, data, name, repoName, token)
        futures.append(future)

    for future in futures:
        result = future.result()
        if result is not None:
            fileAndDates.append(result)

return Response(fileAndDates)
```

5.1.5 Restoring Deleted File

To restore a deleted file we make a get request on the repository to get all the commits of that repository. We then filter through this response to find commits related to the file we want to restore and get the last commit to this file which isn't the delete commit. We then take the sha (a unique identifier) for this commit and make a get request with this sha to get that content of the last commit before the deleted commit. Once we do that we can make a commit on top of the last non delete commit with the old content of the file and update the pointer of the master branch to point to this new commit with a patch request as seen below.

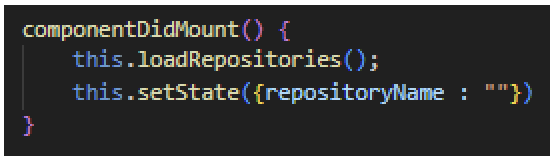


```
url = f"{BASE_URL}/repos/" + owner + "/" + repo + "/contents/" + file
headers = {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer ' + token
}
data = {
    'content' : content,
    'sha' : sha
}
response = requests.post(url, json=data, headers=headers)
new_commit_sha = response.json()['content']['sha']
requests.patch(f"{BASE_URL}/repos/{owner}/{repo}/git/refs/heads/master",
    json={'sha': new_commit_sha}, headers=headers)

return Response(status=status.HTTP_200_OK)
```

5.2 Web app

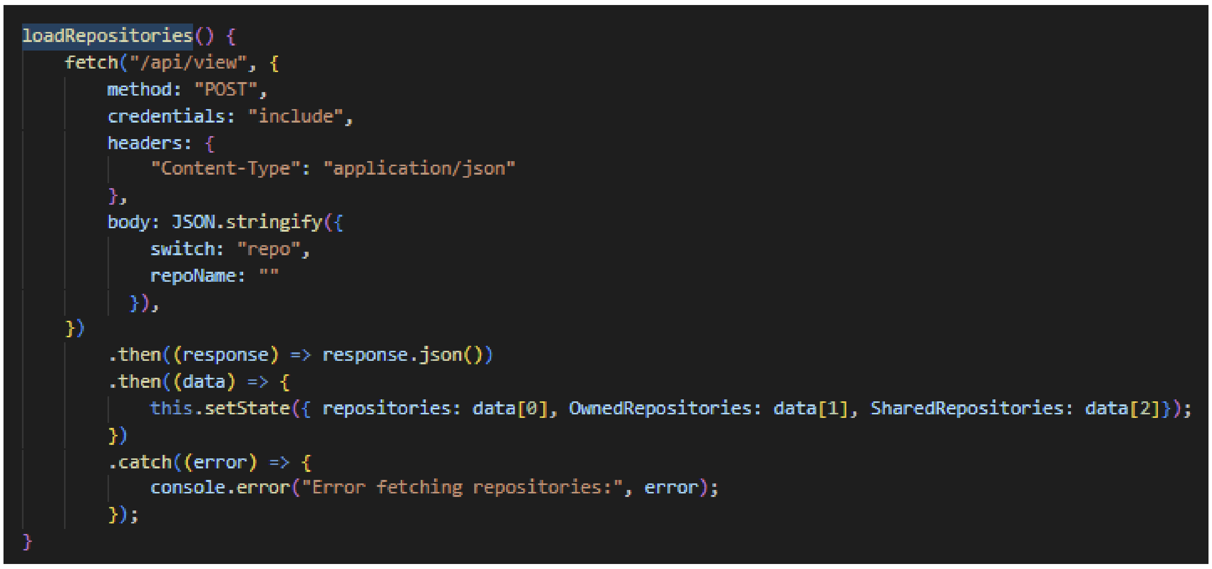
5.2.1 ComponentDidMount

We use the componentDidMount() function of react throughout the web application to make api requests as soon as a user loads the page, this way all information that a user needs will already be displayed as they load the web page. The componentDidMount function below will call the loadRepositories function as soon as the page loads this way we can have the users repositories and all required information displayed to the user straight away.



```
componentDidMount() {
    this.loadRepositories();
    this.setState({repositoryName : ""})
}
```

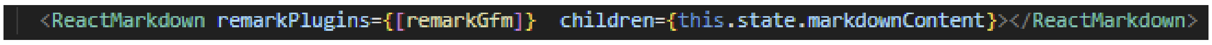
The load repositories function can then make a call to the django backend including the user’s JWT token using the response to set lists equal to all the users repositories, the repositories they own and repositories that were shared with them.



```
loadRepositories() {
    fetch("/api/view", {
        method: "POST",
        credentials: "include",
        headers: {
            "Content-Type": "application/json"
        },
        body: JSON.stringify({
            switch: "repo",
            repoName: ""
          }),
    })
        .then((response) => response.json())
        .then((data) => {
            this.setState({ repositories: data[0], OwnedRepositories: data[1], SharedRepositories: data[2]});
        })
        .catch((error) => {
            console.error("Error fetching repositories:", error);
        });
}
```

5.2.2 Rendering Markdown

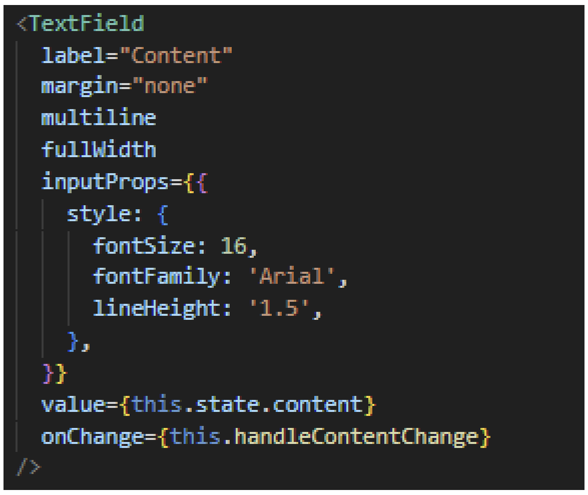
Rendering markdown content on the web application was a relatively straightforward procedure thanks to react libraries React Markdown and remark-gfm. this.state.markdwonContent is simply user entered text and the react libraries take care of the rendering for us.



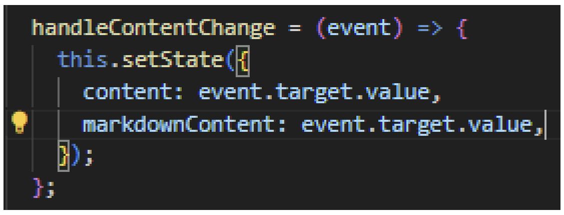
```
  <ReactMarkdown remarkPlugins={[remarkGfm]}  children={this.state.markdownContent}></ReactMarkdown>
```

5.2.3 Updating Fields

We use the onChange react event handler to live update elements such as the content a user is entering, this way we can take a user input as they are typing it and immediately render the markdown version of it beside it.



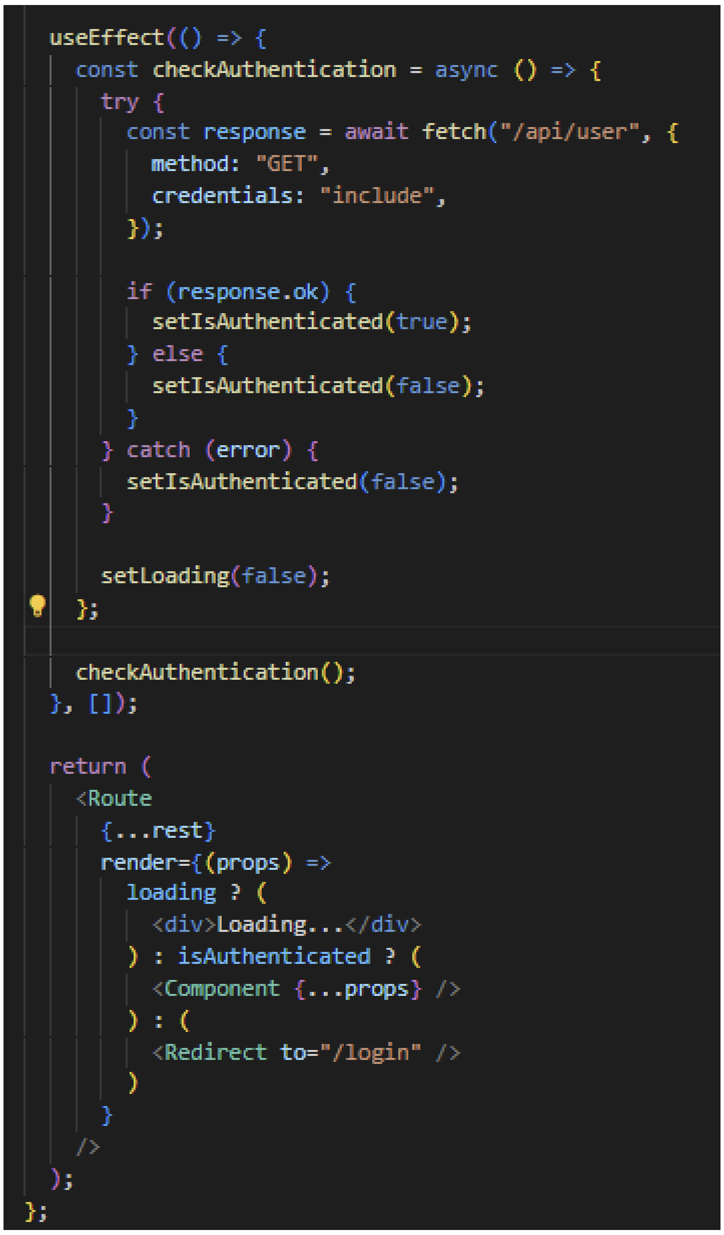
```
<TextField
  label="Content"
  margin="none"
  multiline
  fullWidth
  inputProps={{
    style: {
      fontSize: 16,
      fontFamily: 'Arial',
      lineHeight: '1.5',
    },
  }}
  value={this.state.content}
  onChange={this.handleContentChange}
/>
```



```
handleContentChange = (event) => {
  this.setState({
    content: event.target.value,
    markdownContent: event.target.value,
  });
};
```

5.2.4 Private Route

The private route component is used to ensure users cannot access pages other than the logging in page until they are authenticated. This sets the default route component to now rely on an isAuthenticated value. On load the get request for /api/user is made which will check and verify the users JWT to see if they are a valid user, if they are then they are successfully allowed to go to their desired location otherwise they are redirected back to the login page.



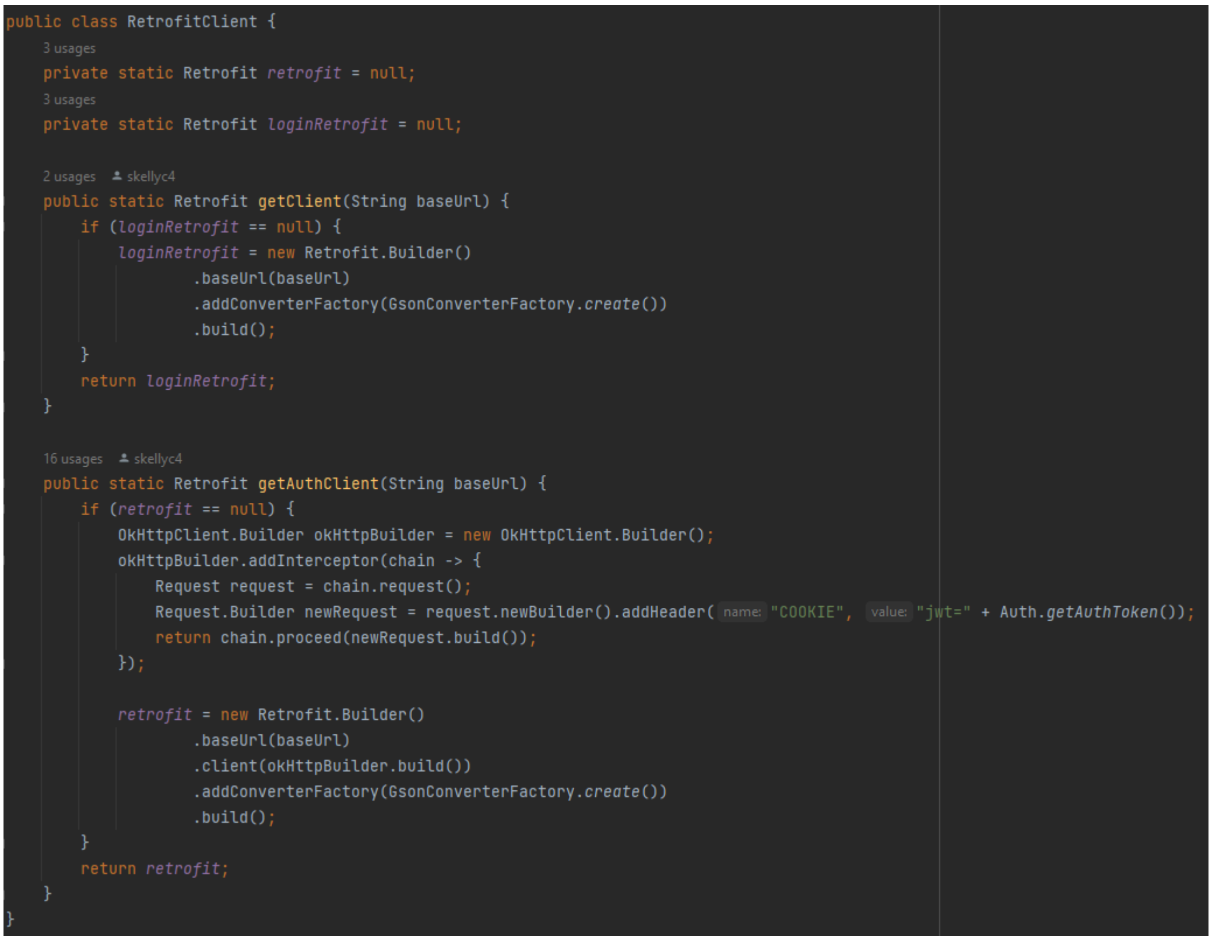
```
  useEffect(() => {
    const checkAuthentication = async () => {
      try {
        const response = await fetch("/api/user", {
          method: "GET",
          credentials: "include",
        });

        if (response.ok) {
          setIsAuthenticated(true);
        } else {
          setIsAuthenticated(false);
        }
      } catch (error) {
        setIsAuthenticated(false);
      }

      setLoading(false);
    };

    checkAuthentication();
  }, []);

  return (
    <Route
      {...rest}
      render={(props) =>
        loading ? (
          <div>Loading...</div>
        ) : isAuthenticated ? (
          <Component {...props} />
        ) : (
          <Redirect to="/login" />
        )
      }
    />
  );
};
```

5.3 Mobile app

5.3.1 Retrofit for api requests

Retrofit was used in the mobile app for communicating with the Restful APIs. Retrofit handles the JSON serialisation, deserialisation and asynchronous execution of network requests. Gson is the Java library we used with retrofit to convert Java objects into JSON representations and JSON strings to an equivalent Java object.

The mobile app has two different retrofit clients that we use to make API requests. The 'loginRetrofit' returned by the 'getClient' method is just a basic version of the retrofit client with a Gson converter.
The 'retrofit' returned by 'getAuthClient' has an additional custom 'okHttpClient' that has an interceptor that is used to add the 'COOKIE' header containing a jwt token.

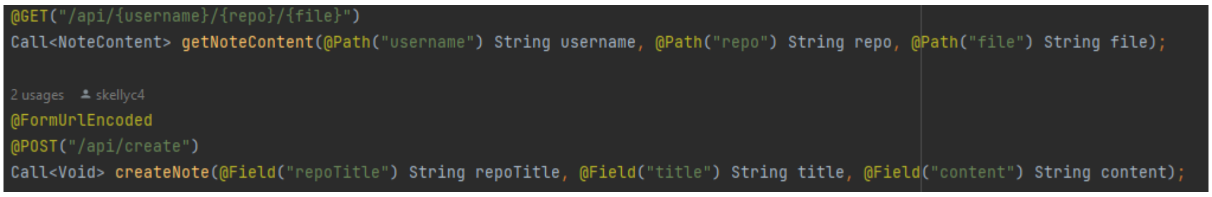


```java
public class RetrofitClient {
    3 usages
    private static Retrofit retrofit = null;
    3 usages
    private static Retrofit loginRetrofit = null;

    2 usages  skellyc4
    public static Retrofit getClient(String baseUrl) {
        if (loginRetrofit == null) {
            loginRetrofit = new Retrofit.Builder()
                    .baseUrl(baseUrl)
                    .addConverterFactory(GsonConverterFactory.create())
                    .build();
        }
        return loginRetrofit;
    }

    16 usages  skellyc4
    public static Retrofit getAuthClient(String baseUrl) {
        if (retrofit == null) {
            OkHttpClient.Builder okHttpBuilder = new OkHttpClient.Builder();
            okHttpBuilder.addInterceptor(chain -> {
                Request request = chain.request();
                Request.Builder newRequest = request.newBuilder().addHeader( name: "COOKIE",  value: "jwt=" + Auth.getAuthToken());
                return chain.proceed(newRequest.build());
            });

            retrofit = new Retrofit.Builder()
                    .baseUrl(baseUrl)
                    .client(okHttpBuilder.build())
                    .addConverterFactory(GsonConverterFactory.create())
                    .build();
        }
        return retrofit;
    }
}
```

Java interfaces are used to define API endpoints with annotations being used to specify request methods, parameters, response type and headers. The code sample below shows examples of two different types of requests.

The first request 'getNoteContent' is annotated with '@Get' because it is a get request. It uses URL manipulation to dynamically update the URL using replacement blocks and method parameters. A replacement block indicated by surrounding a string with'{}' is replaced by the corresponding parameter annotated with '@Path'. The Json response data of 'getNoteContent' requests will be converted to an object of type 'NoteContent'.

The second request 'createNote' is a Post request. It sends Form-encoded data with the annotation 'FormUrlEncoded'. Each key-value pair is annotated with '@Field' and contains the key name and value object.

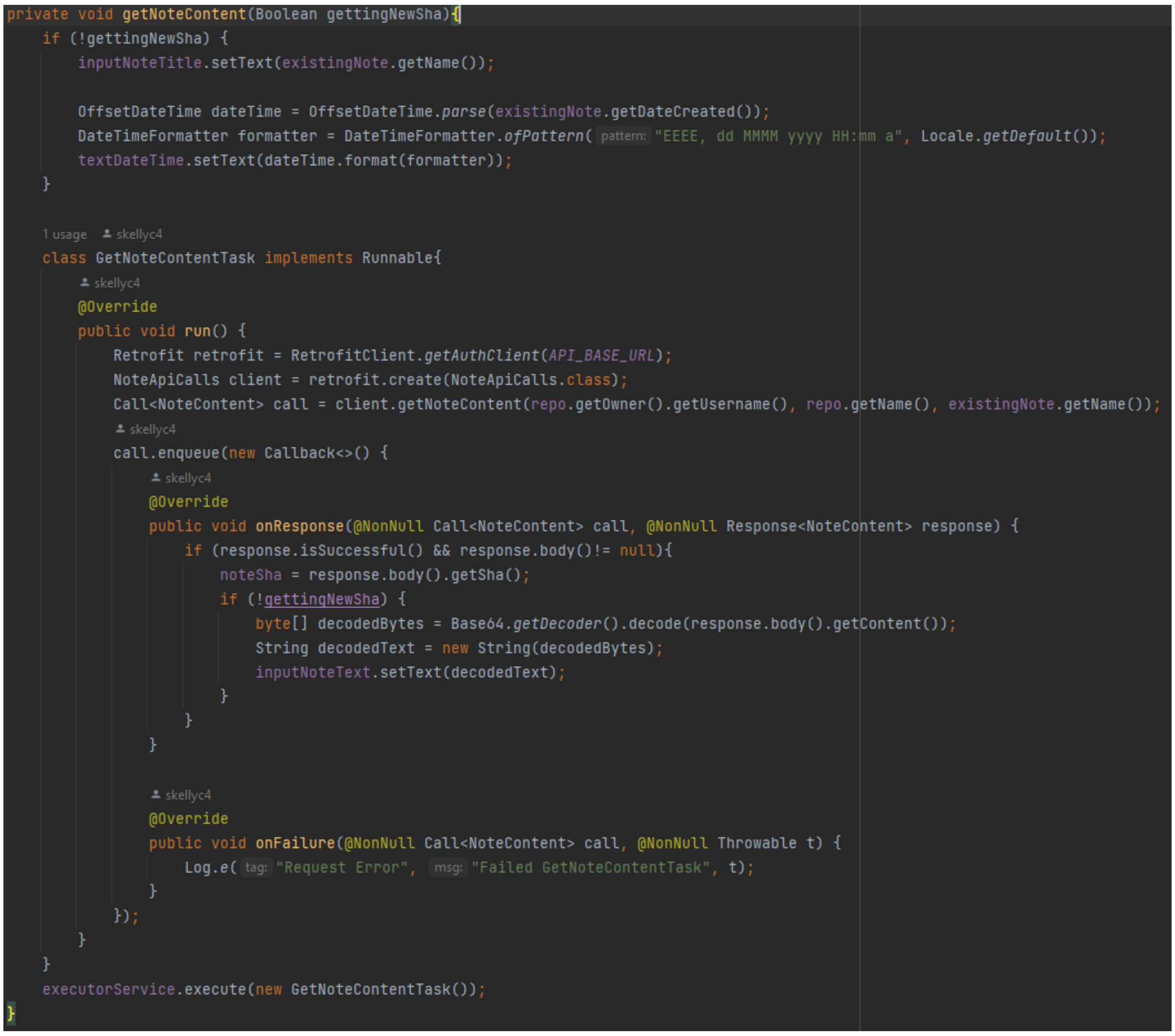


```
@GET("/api/{username}/{repo}/{file}")
Call<NoteContent> getNoteContent(@Path("username") String username, @Path("repo") String repo, @Path("file") String file);

2 usages  skellyc4
@FormUrlEncoded
@POST("/api/create")
Call<Void> createNote(@Field("repoTitle") String repoTitle, @Field("title") String title, @Field("content") String content);
```

5.3.2 Multi-threading tasks

Multi-threading is used to run long tasks like API requests on separate threads than the UI thread so that it doesn't interfere with the smooth rendering and responsiveness of the UI.

Classes that implement Runnable like the one below can be executed by an ExecutorService on a separate thread from the UI thread. The example below makes a request to get the contents of a file and on a successful response sets the file content into the ‘inputNoteText’ UI element. Making this type of request on the UI thread would have caused the UI to freeze until the response is received.

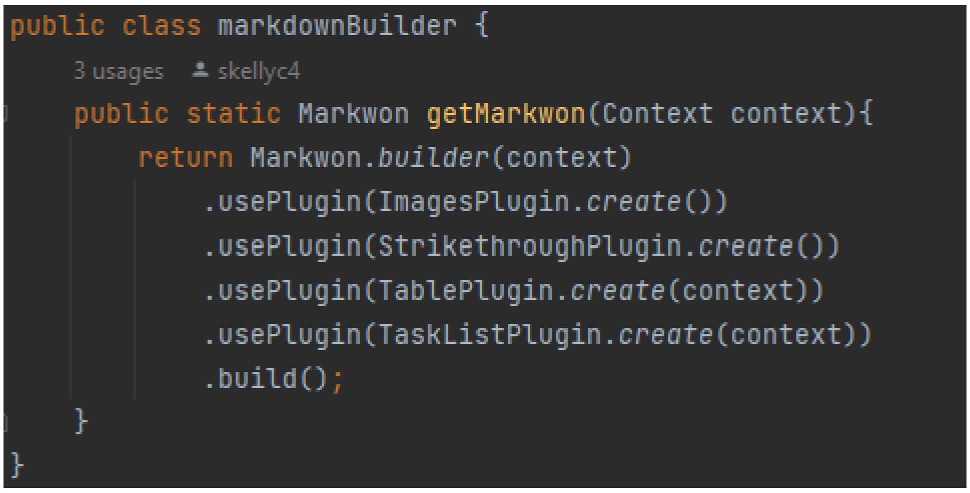


```
private void getNoteContent(Boolean gettingNewSha){
    if (!gettingNewSha) {
        inputNoteTitle.setText(existingNote.getName());

        OffsetDateTime dateTime = OffsetDateTime.parse(existingNote.getDateCreated());
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern( pattern: "EEEE, dd MMMM yyyy HH:mm a", Locale.getDefault());
        textDateTime.setText(dateTime.format(formatter));
    }

    1 usage  skellyc4
    class GetNoteContentTask implements Runnable{
        skellyc4
        @Override
        public void run() {
            Retrofit retrofit = RetrofitClient.getAuthClient(API_BASE_URL);
            NoteApiCalls client = retrofit.create(NoteApiCalls.class);
            Call<NoteContent> call = client.getNoteContent(repo.getOwner().getUsername(), repo.getName(), existingNote.getName());
            skellyc4
            call.enqueue(new Callback<>() {
                skellyc4
                @Override
                public void onResponse(@NonNull Call<NoteContent> call, @NonNull Response<NoteContent> response) {
                    if (response.isSuccessful() && response.body()!= null){
                        noteSha = response.body().getSha();
                        if (!gettingNewSha) {
                            byte[] decodedBytes = Base64.getDecoder().decode(response.body().getContent());
                            String decodedText = new String(decodedBytes);
                            inputNoteText.setText(decodedText);
                        }
                    }
                }

                skellyc4
                @Override
                public void onFailure(@NonNull Call<NoteContent> call, @NonNull Throwable t) {
                    Log.e( tag: "Request Error",  msg: "Failed GetNoteContentTask", t);
                }
            });
        }
    }
    executorService.execute(new GetNoteContentTask());
}
```

5.3.3 Markdown rendering

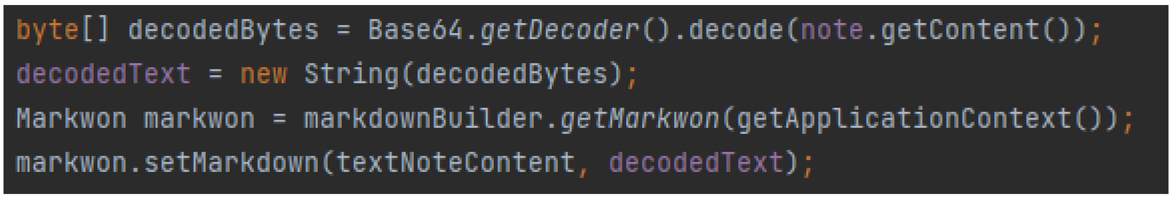
For efficient rendering of Markdown text we used Markwon. Markwon gives the ability to display markdown in all TextView widgets, Toasts and all other places that accept spanned content. It also gives the ability to configure the display styling.

The ‘getMarkwon’ method below returns an instance of Markwon which has been configured with a number of plugins for supporting additional Markdown features (Images, Strikethrough, Tables and Task lists).



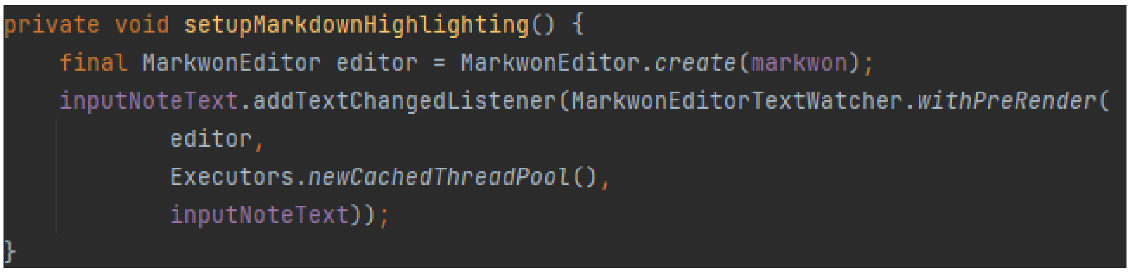
```
public class markdownBuilder {
    3 usages   skellyc4
    public static Markwon getMarkwon(Context context){
        return Markwon.builder(context)
                .usePlugin(ImagesPlugin.create())
                .usePlugin(StrikethroughPlugin.create())
                .usePlugin(TablePlugin.create(context))
                .usePlugin(TaskListPlugin.create(context))
                .build();
    }
}
```

The Markwon instance is used to set the ‘textNoteContent’ UI element to display some text as Markdown.



```
byte[] decodedBytes = Base64.getDecoder().decode(note.getContent());
decodedText = new String(decodedBytes);
Markwon markwon = markdownBuilder.getMarkwon(getApplicationContext());
markwon.setMarkdown(textNoteContent, decodedText);
```

The Markwon library also has a MarkwonEditor for highlighting markdown input in EditText elements. The MarkwonEditor we use runs on a separate thread from the UI thread so that the rendering doesn't affect the responsiveness of the overall UI.



```
private void setupMarkdownHighlighting() {
    final MarkwonEditor editor = MarkwonEditor.create(markwon);
    inputNoteText.addTextChangedListener(MarkwonEditorTextWatcher.withPreRender(
            editor,
            Executors.newCachedThreadPool(),
            inputNoteText));
}
```

6. Problems solved

6.1 Navigation Bar Refresh

A second problem that came up during development was that the navigation bar on the web app refused to refresh after a user signed in which resulted in the user seeing sign in remain in the nav bar until they refreshed the page. This was a frustrating bug to fix as we could see through logging that the backend was returning that the user was logged in and the frontend authentication values were also successfully saying that the user was authenticated. However the issue was discovered to be how we were redirecting the user to the home page after a successful sign in. At first we were using react <Redirect to="/" /> which allows for more control over routing within react applications, however as we were using this the page nav bar was not being reloaded from its last use. To fix this we switched to using the default javascript method window.location.href = '/' which performs a hard reload on the web page allowing our frontend to update its nav bar.

6.2 Slow Gathering Files

The gathering of files from the gitea api was fast however, if we wanted to display the date created of each file in a repository it would slow down the speed of displaying files to users once they click on a repository as we would have to make an api call on each file in the repository. To solve this we decided we could thread the api calls on each file which would greatly reduce the time it takes for the backend to gather all the files along with the dates they were created on.

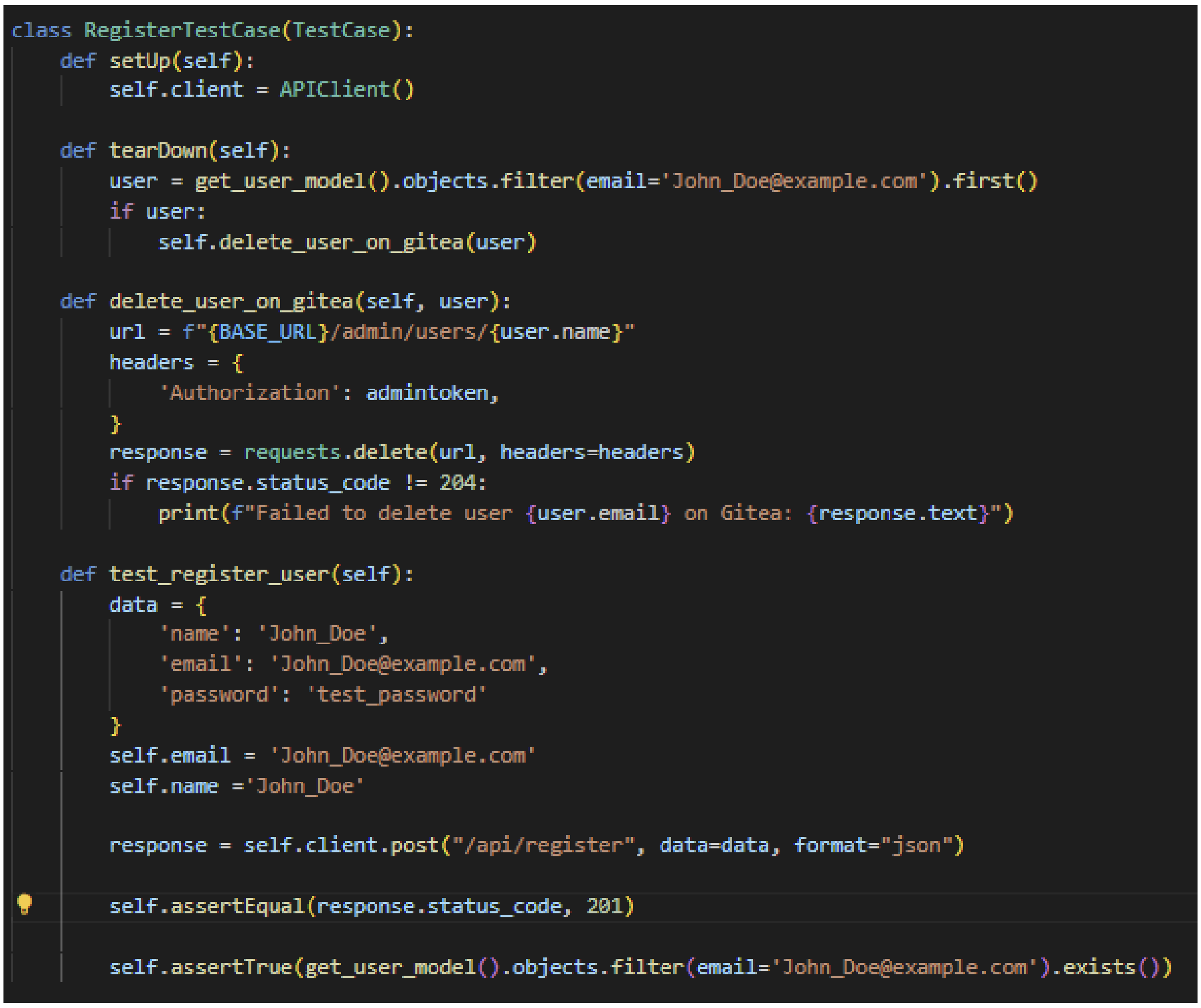
6.3 Consistent UI on different platforms

We came across the issue of consistency early on in our development of this project, as while we required the application to be similar on both platforms, the size difference of screens between a mobile and a computer/laptop was wildly different. One problem that occurred from this was implementing the navigation bar on both platforms as the web app's nav bar was stuck to the screen however this was not possible on the mobile app due to size restrictions. At first we tried not having a navigation bar on the mobile app but this reduced the consistency between both platforms and was pointed out to us in our user testing. Our solution was to have a nav bar on the mobile app but not to have it stuck to the page so that it only appears on the press of a button, this way we could ensure consistency but adhere to the size limitations of the mobile app. Another issue that came from the size limitations of the mobile UI was live translation of user content into markdown, as on the web app we had the space to make these appear side by side so a user could know exactly how their file would appear in markdown. However on the mobile app we did not have the room for this so the solution we came to was having a preview button on the mobile app that when pressed would change the users text to markdown. Doing this allowed us to keep the same features across platforms while also remaining within the limitations of our screen sizes.

7. Testing and Results

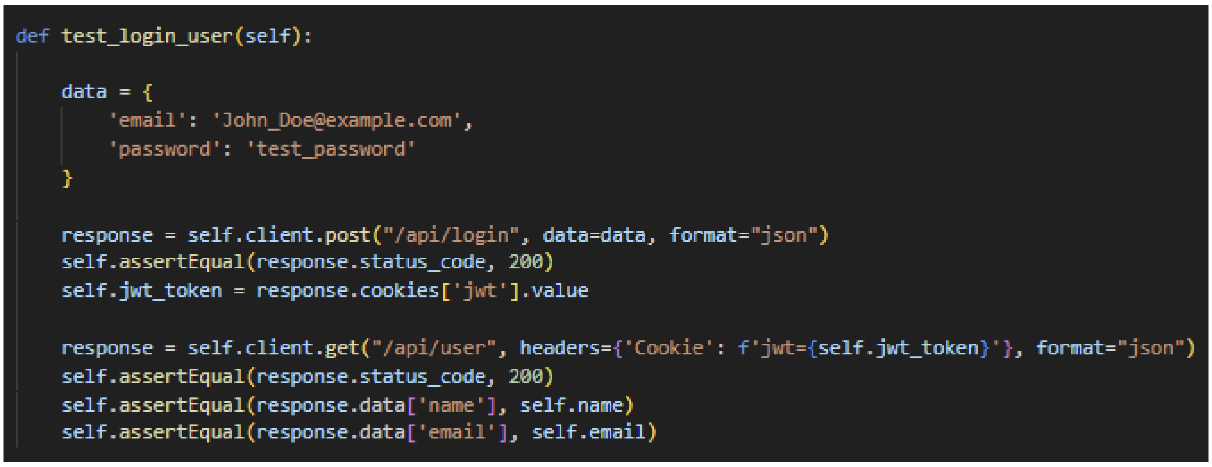
7.1 Unit testing

The unit tests of our application use django's built in testing framework to test all the api endpoints of our backend. Django provides a powerful set of tools for writing and executing unit tests, these tests are designed to simulate different scenarios, ensuring that the application behaves as expected under various conditions.
Below is the test case for registering a user, in the set up we initialise an instance of Django's apiclient class which allows us to make requests to our api endpoints in our tests. After this the test_register_user test takes place where we attempt to create a user with the username John_Doe. We first ensure that the status code is correctly returning 201 and then we check if the user was successfully saved on the django backend. Upon completion we go into the tearDown function which gets the user that was just created and deletes them.



```python
class RegisterTestCase(TestCase):
    def setUp(self):
        self.client = APIClient()

    def tearDown(self):
        user = get_user_model().objects.filter(email='John_Doe@example.com').first()
        if user:
            self.delete_user_on_gitea(user)

    def delete_user_on_gitea(self, user):
        url = f"{BASE_URL}/admin/users/{user.name}"
        headers = {
            'Authorization': admintoken,
        }
        response = requests.delete(url, headers=headers)
        if response.status_code != 204:
            print(f"Failed to delete user {user.email} on Gitea: {response.text}")

    def test_register_user(self):
        data = {
            'name': 'John_Doe',
            'email': 'John_Doe@example.com',
            'password': 'test_password'
        }
        self.email = 'John_Doe@example.com'
        self.name ='John_Doe'

        response = self.client.post("/api/register", data=data, format="json")

        self.assertEqual(response.status_code, 201)

        self.assertTrue(get_user_model().objects.filter(email='John_Doe@example.com').exists())
```

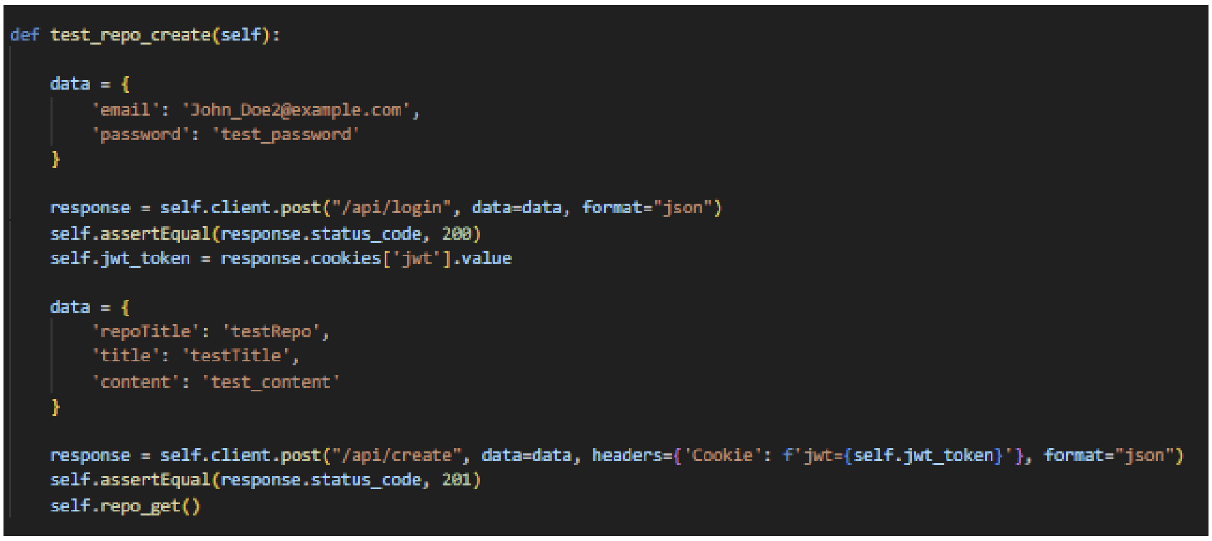
For the login test we check whether the response status code was correctly 200 and then we gather the JSON Web Token that is created upon a successful login. We then check /api/user which is an authentication endpoint including the JWT to check if the user has been successfully logged in. We check the returned status code form this and also if the JWT we supplied matches the JWT of the user we logged into.



```python
def test_login_user(self):

    data = {
        'email': 'John_Doe@example.com',
        'password': 'test_password'
    }

    response = self.client.post("/api/login", data=data, format="json")
    self.assertEqual(response.status_code, 200)
    self.jwt_token = response.cookies['jwt'].value

    response = self.client.get("/api/user", headers={'Cookie': f'jwt={self.jwt_token}'}, format="json")
    self.assertEqual(response.status_code, 200)
    self.assertEqual(response.data['name'], self.name)
    self.assertEqual(response.data['email'], self.email)
```

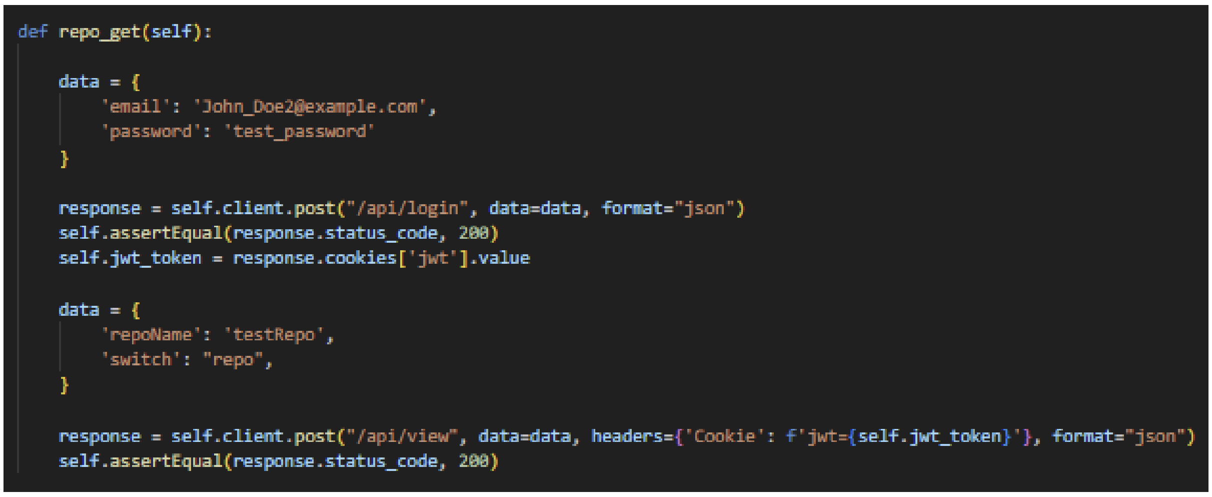
7.2 Integration testing

The set up for our integration test starts with creating a user similar to the unit test seen above instead this time we create two users so we can test interactions between users. Next we create a repository for a user we just created by first logging in to get a jwt token and next using the /create endpoint with sample data



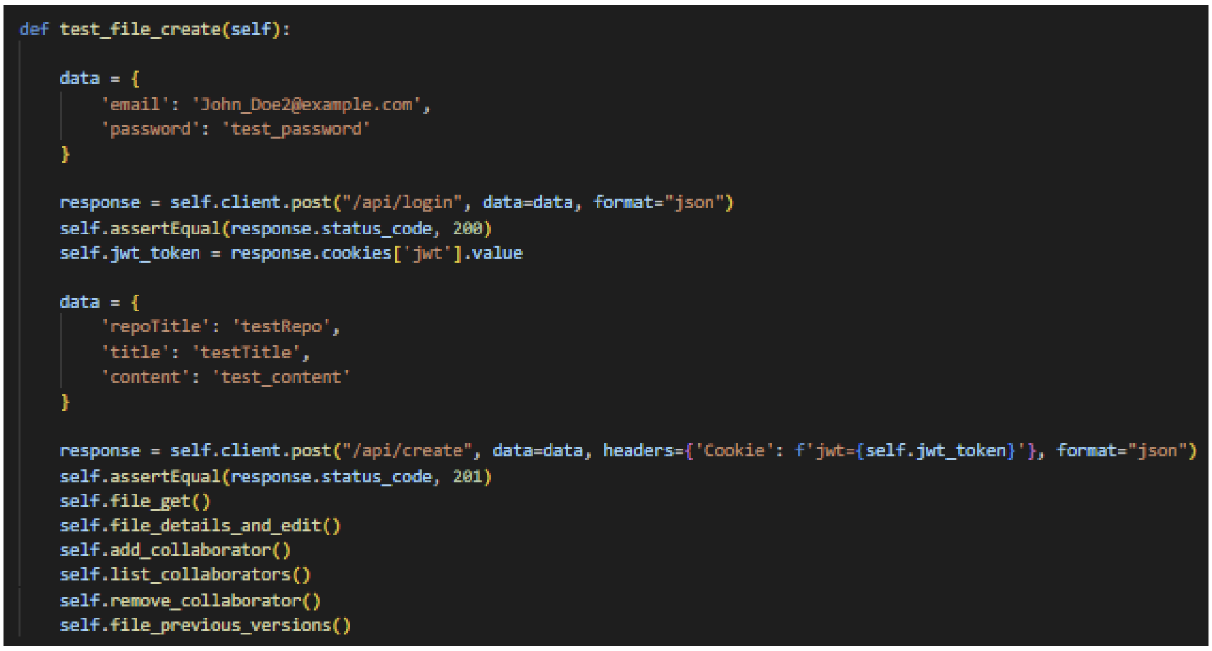
```python
def test_repo_create(self):

    data = {
        'email': 'John_Doe2@example.com',
        'password': 'test_password'
    }

    response = self.client.post("/api/login", data=data, format="json")
    self.assertEqual(response.status_code, 200)
    self.jwt_token = response.cookies['jwt'].value

    data = {
        'repoTitle': 'testRepo',
        'title': 'testTitle',
        'content': 'test_content'
    }

    response = self.client.post("/api/create", data=data, headers={'Cookie': f'jwt={self.jwt_token}'}, format="json")
    self.assertEqual(response.status_code, 201)
    self.repo_get()
```

To ensure that the test worked we check if the repository exists on the gitea server using self.repo_get. This will again log the user in and then check if the repository exists using the /view endpoint, ensuring that the correct status code is returned.



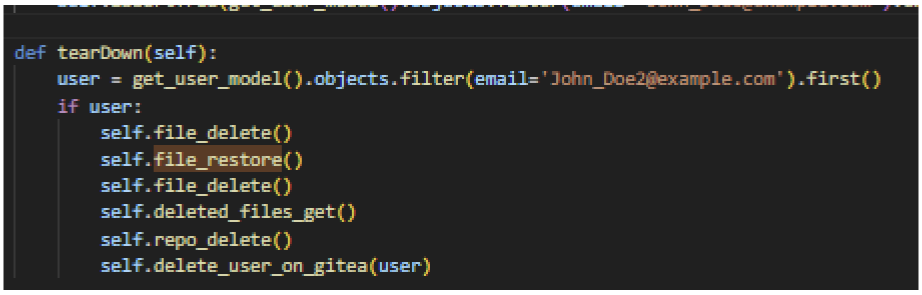
```python
def repo_get(self):

    data = {
        'email': 'John_Doe2@example.com',
        'password': 'test_password'
    }

    response = self.client.post("/api/login", data=data, format="json")
    self.assertEqual(response.status_code, 200)
    self.jwt_token = response.cookies['jwt'].value

    data = {
        'repoName': 'testRepo',
        'switch': "repo",
    }

    response = self.client.post("/api/view", data=data, headers={'Cookie': f'jwt={self.jwt_token}'}, format="json")
    self.assertEqual(response.status_code, 200)
```

We do the same thing when we run tests on creating a file however we test further features of our application such as editing the file, adding, removing and viewing collaborators and viewing a files previous version before our edit



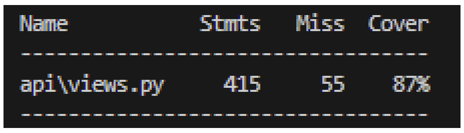
```python
def test_file_create(self):

    data = {
        'email': 'John_Doe2@example.com',
        'password': 'test_password'
    }

    response = self.client.post("/api/login", data=data, format="json")
    self.assertEqual(response.status_code, 200)
    self.jwt_token = response.cookies['jwt'].value

    data = {
        'repoTitle': 'testRepo',
        'title': 'testTitle',
        'content': 'test_content'
    }

    response = self.client.post("/api/create", data=data, headers={'Cookie': f'jwt={self.jwt_token}'}, format="json")
    self.assertEqual(response.status_code, 201)
    self.file_get()
    self.file_details_and_edit()
    self.add_collaborator()
    self.list_collaborators()
    self.remove_collaborator()
    self.file_previous_versions()
```

In our tear down of this test we further test deleting an individual file and then restoring it, gathering deleted files and deleting a repository before finally deleting the test users.



```python
def tearDown(self):
    user = get_user_model().objects.filter(email='John_Doe2@example.com').first()
    if user:
        self.file_delete()
        self.file_restore()
        self.file_delete()
        self.deleted_files_get()
        self.repo_delete()
        self.delete_user_on_gitea(user)
```

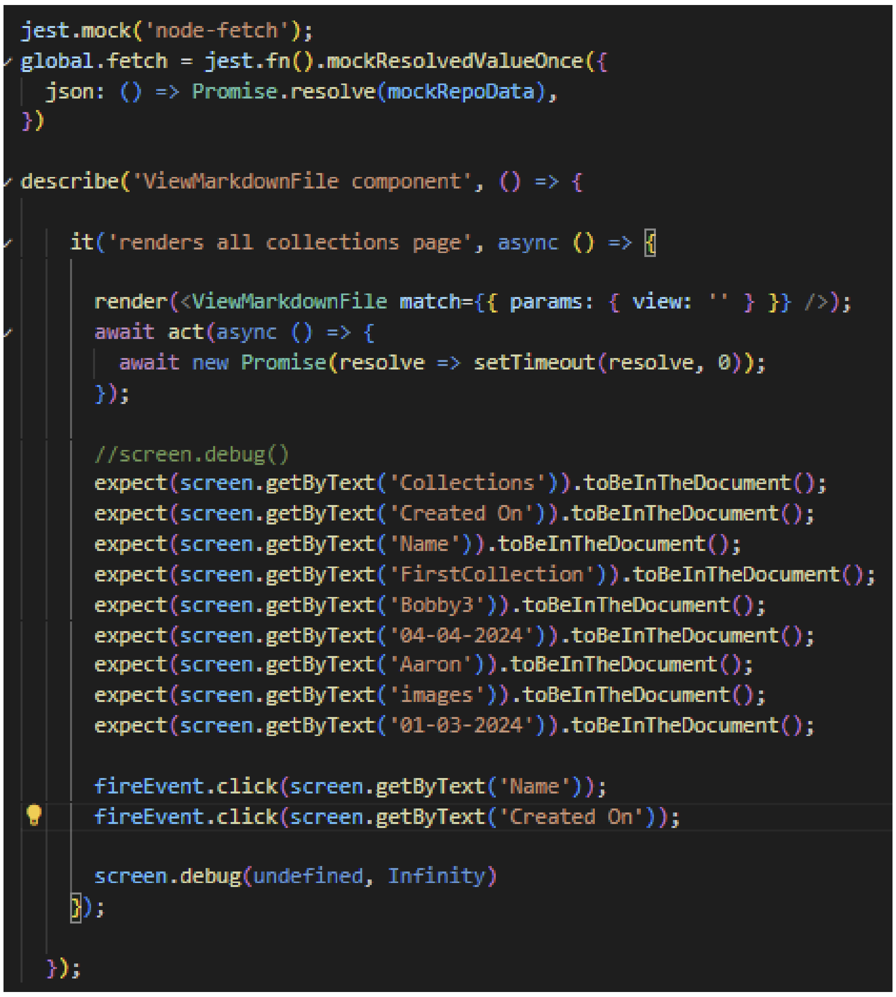
Our code coverage for our backend tests was 91% overall using the python package coverage and was 87% on our views.py file which was where the main bulk of the tests were built for as this was a crucial part of our django backend.



```
Name            Stmts   Miss  Cover
-----------------------------------
api\views.py      415     55    87%
-----------------------------------
```

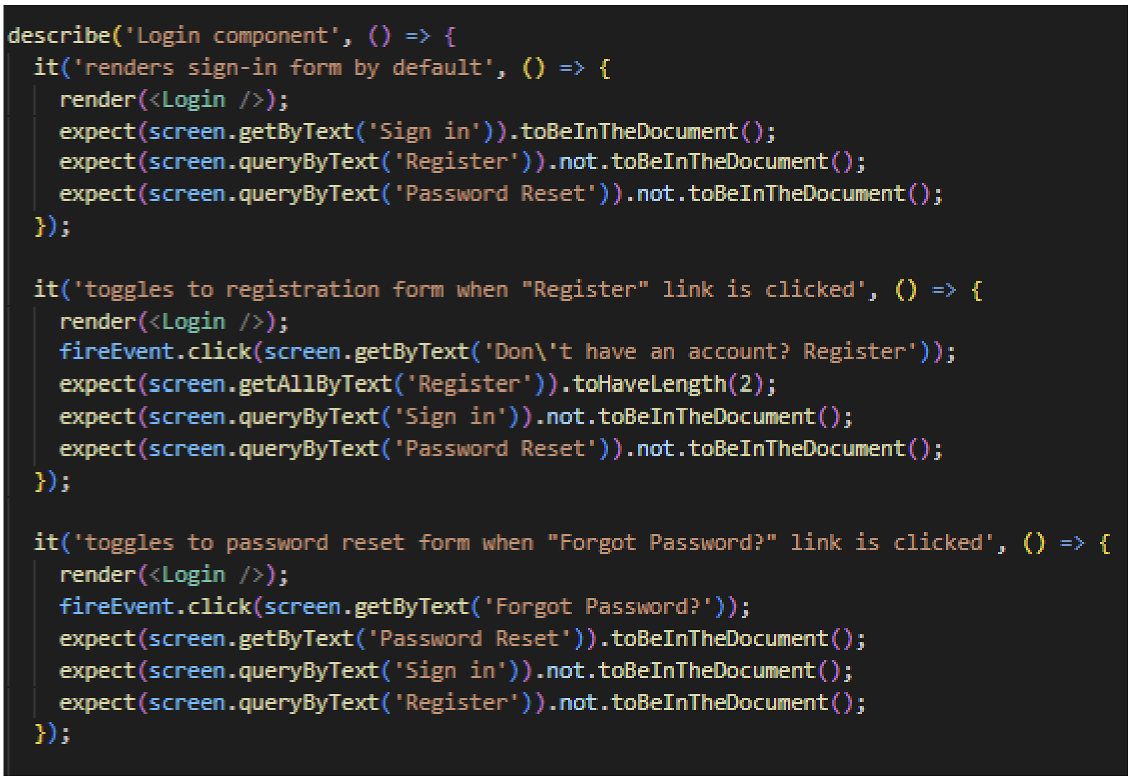
7.3 Frontend testing

For Testing the frontend of the web application I used Jest, it is a JavaScript testing framework that's used for testing JavaScript code, particularly in React applications. It comes with built-in functionalities such as assertion and mocking capabilities that made it easier to set up and execute tests. The below test is ensuring that everything is rendering on the home page correctly, it does this by using
global.fetch = jest.fn(). This line replaces the global fetch function with a Jest mock function created using jest.fn(). By doing this, any calls to fetch within the tested code will be intercepted by Jest and handled by the mock function. This allows us to set the return from the first function which will be gathering a user's repositories to mock data we set called mockRepoData. We then render the home page and use await to ensure that the page is fully loaded before any of our tests take place. After we have done this we can run tests using expect to check if the mock data has been correctly generated onto the page. We then check if we can click on the elements Name and Created on which will sort the list of repositories.



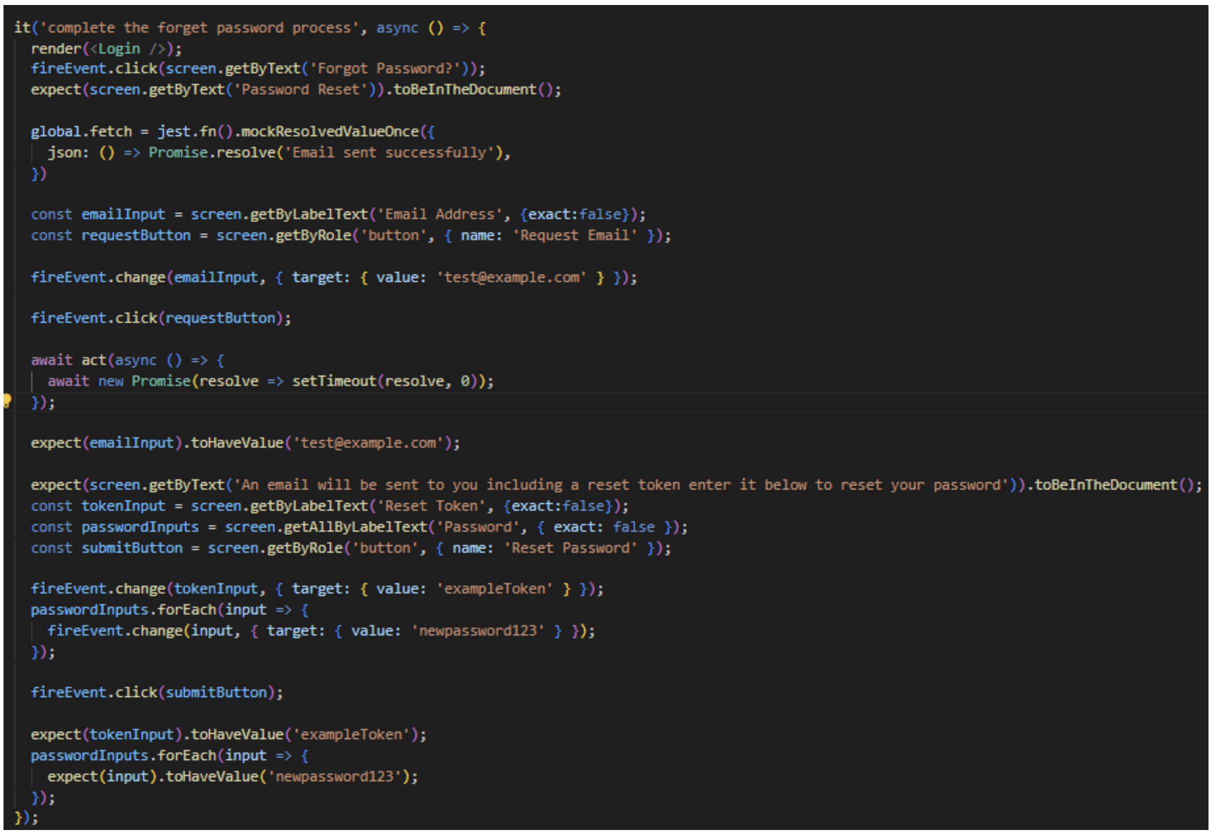
```
jest.mock('node-fetch');
global.fetch = jest.fn().mockResolvedValueOnce({
  json: () => Promise.resolve(mockRepoData),
})

describe('ViewMarkdownFile component', () => {

    it('renders all collections page', async () => {

      render(<ViewMarkdownFile match={{ params: { view: '' } }} />);
      await act(async () => {
        await new Promise(resolve => setTimeout(resolve, 0));
      });

      //screen.debug()
      expect(screen.getByText('Collections')).toBeInTheDocument();
      expect(screen.getByText('Created On')).toBeInTheDocument();
      expect(screen.getByText('Name')).toBeInTheDocument();
      expect(screen.getByText('FirstCollection')).toBeInTheDocument();
      expect(screen.getByText('Bobby3')).toBeInTheDocument();
      expect(screen.getByText('04-04-2024')).toBeInTheDocument();
      expect(screen.getByText('Aaron')).toBeInTheDocument();
      expect(screen.getByText('images')).toBeInTheDocument();
      expect(screen.getByText('01-03-2024')).toBeInTheDocument();

      fireEvent.click(screen.getByText('Name'));
      fireEvent.click(screen.getByText('Created On'));

      screen.debug(undefined, Infinity)
    });

  });
```

For testing the login page we first test if we can successfully navigate between logging in, registering and resetting your password. We click the navigation tools required to get to these pages then run expect tests to see if the title of the page has changed successfully.



```
describe('Login component', () => {
  it('renders sign-in form by default', () => {
    render(<Login />);
    expect(screen.getByText('Sign in')).toBeInTheDocument();
    expect(screen.queryByText('Register')).not.toBeInTheDocument();
    expect(screen.queryByText('Password Reset')).not.toBeInTheDocument();
  });

  it('toggles to registration form when "Register" link is clicked', () => {
    render(<Login />);
    fireEvent.click(screen.getByText('Don\'t have an account? Register'));
    expect(screen.getAllByText('Register')).toHaveLength(2);
    expect(screen.queryByText('Sign in')).not.toBeInTheDocument();
    expect(screen.queryByText('Password Reset')).not.toBeInTheDocument();
  });

  it('toggles to password reset form when "Forgot Password?" link is clicked', () => {
    render(<Login />);
    fireEvent.click(screen.getByText('Forgot Password?'));
    expect(screen.getByText('Password Reset')).toBeInTheDocument();
    expect(screen.queryByText('Sign in')).not.toBeInTheDocument();
    expect(screen.queryByText('Register')).not.toBeInTheDocument();
  });
```

The process for checking if a user can successfully reset their password takes a few more steps. We first render the login page and click on “Forgot Password?” and run an expect test to ensure we have moved to the right page. We then set the mock return of the next request to be “email sent successfully”. After this we can get the email entry field and enter a mock email address before submitting a request to reset password. After this has been done we check if we have successfully moved to a new page that allows the user to enter their reset token and their new password. We finally check if we can enter strings into these fields and that they appear on screen.



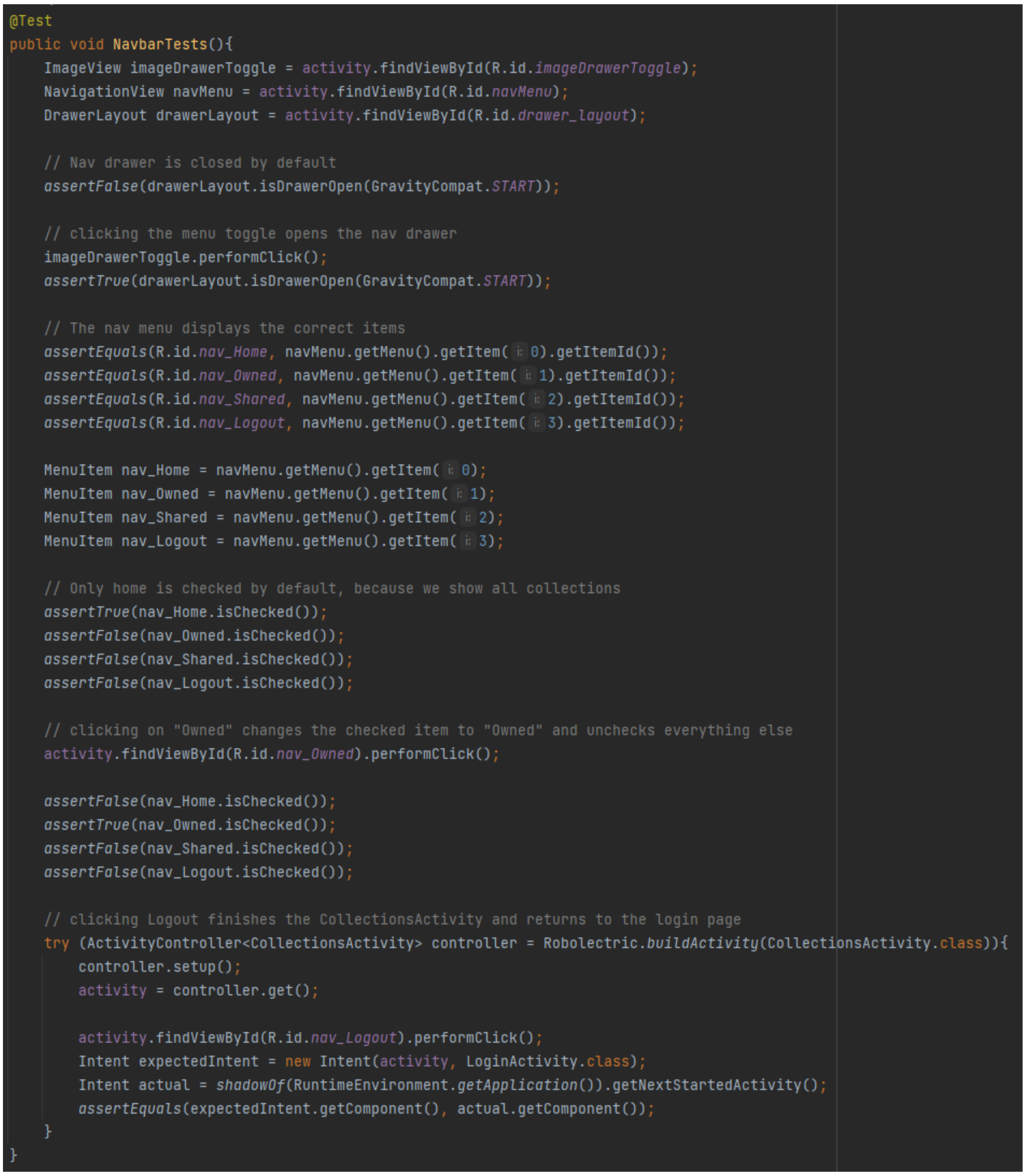
```
it('complete the forget password process', async () => {
  render(<Login />);
  fireEvent.click(screen.getByText('Forgot Password?'));
  expect(screen.getByText('Password Reset')).toBeInTheDocument();

  global.fetch = jest.fn().mockResolvedValueOnce({
    json: () => Promise.resolve('Email sent successfully'),
  })

  const emailInput = screen.getByLabelText('Email Address', {exact:false});
  const requestButton = screen.getByRole('button', { name: 'Request Email' });

  fireEvent.change(emailInput, { target: { value: 'test@example.com' } });

  fireEvent.click(requestButton);

  await act(async () => {
    await new Promise(resolve => setTimeout(resolve, 0));
  });

  expect(emailInput).toHaveValue('test@example.com');

  expect(screen.getByText('An email will be sent to you including a reset token enter it below to reset your password')).toBeInTheDocument();
  const tokenInput = screen.getByLabelText('Reset Token', {exact:false});
  const passwordInputs = screen.getAllByLabelText('Password', { exact: false });
  const submitButton = screen.getByRole('button', { name: 'Reset Password' });

  fireEvent.change(tokenInput, { target: { value: 'exampleToken' } });
  passwordInputs.forEach(input => {
    fireEvent.change(input, { target: { value: 'newpassword123' } });
  });

  fireEvent.click(submitButton);

  expect(tokenInput).toHaveValue('exampleToken');
  passwordInputs.forEach(input => {
    expect(input).toHaveValue('newpassword123');
  });
});
```

The rest of the web applications front-end tests use the same functions and ideas to test the content of different pages, clicking through different options and ensuring users can enter text into text fields.The test statement coverage for the web app frontend is around 61%.

7.4 Mobile App UI Testing

For the mobile app we conducted UI testing using Robolectric. Robolectric is a framework that allows you to run tests on your own machine or on continuous integration environments in a regular JVM, without having to use an emulator.

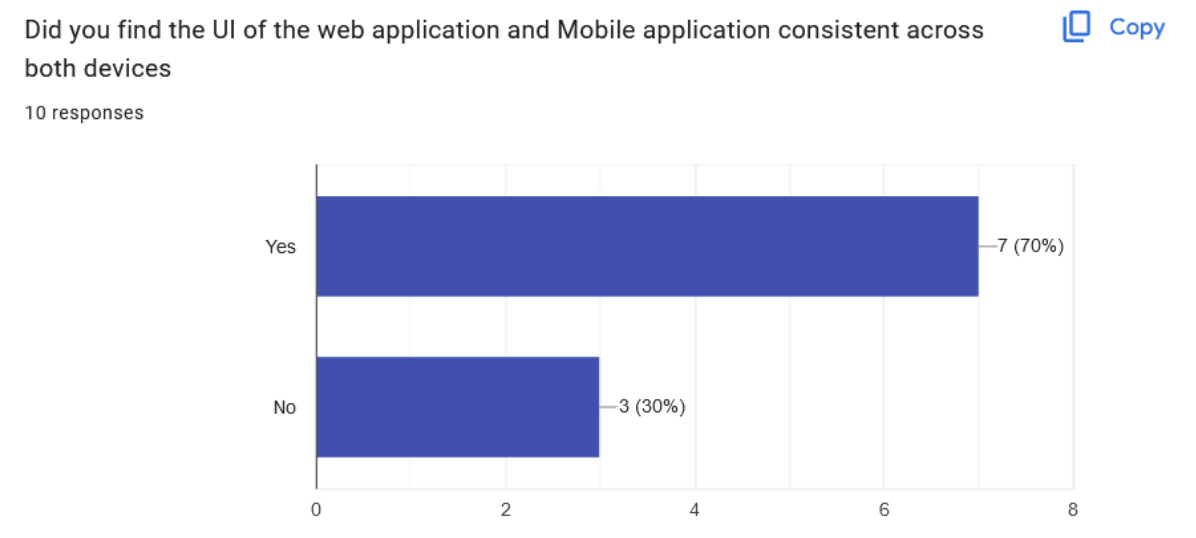
The test below tests the functionality of the navigation bar in the Collections activity. Parts of the test are commented to explain what is being tested.



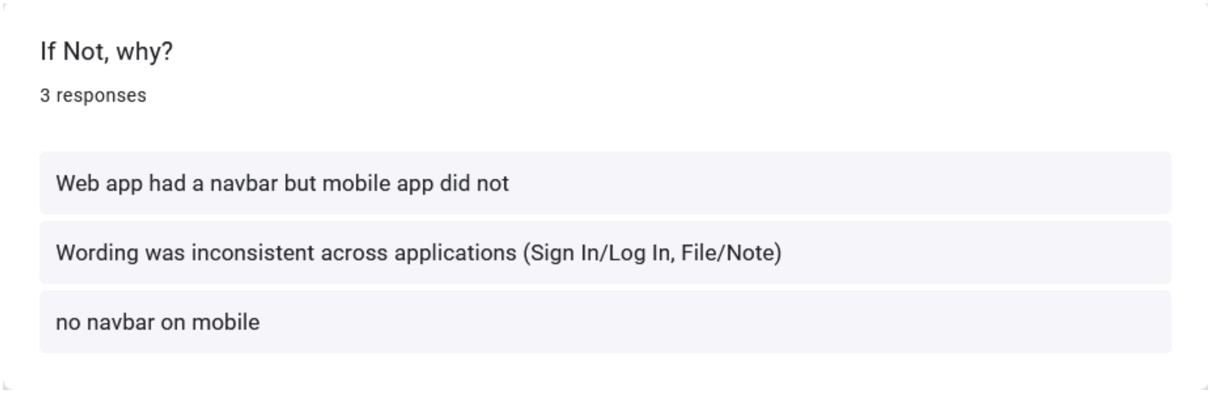
```java
@Test
public void NavbarTests(){
    ImageView imageDrawerToggle = activity.findViewById(R.id.imageDrawerToggle);
    NavigationView navMenu = activity.findViewById(R.id.navMenu);
    DrawerLayout drawerLayout = activity.findViewById(R.id.drawer_layout);

    // Nav drawer is closed by default
    assertFalse(drawerLayout.isDrawerOpen(GravityCompat.START));

    // clicking the menu toggle opens the nav drawer
    imageDrawerToggle.performClick();
    assertTrue(drawerLayout.isDrawerOpen(GravityCompat.START));

    // The nav menu displays the correct items
    assertEquals(R.id.nav_Home, navMenu.getMenu().getItem( i: 0).getItemId());
    assertEquals(R.id.nav_Owned, navMenu.getMenu().getItem( i: 1).getItemId());
    assertEquals(R.id.nav_Shared, navMenu.getMenu().getItem( i: 2).getItemId());
    assertEquals(R.id.nav_Logout, navMenu.getMenu().getItem( i: 3).getItemId());

    MenuItem nav_Home = navMenu.getMenu().getItem( i: 0);
    MenuItem nav_Owned = navMenu.getMenu().getItem( i: 1);
    MenuItem nav_Shared = navMenu.getMenu().getItem( i: 2);
    MenuItem nav_Logout = navMenu.getMenu().getItem( i: 3);

    // Only home is checked by default, because we show all collections
    assertTrue(nav_Home.isChecked());
    assertFalse(nav_Owned.isChecked());
    assertFalse(nav_Shared.isChecked());
    assertFalse(nav_Logout.isChecked());

    // clicking on "Owned" changes the checked item to "Owned" and unchecks everything else
    activity.findViewById(R.id.nav_Owned).performClick();

    assertFalse(nav_Home.isChecked());
    assertTrue(nav_Owned.isChecked());
    assertFalse(nav_Shared.isChecked());
    assertFalse(nav_Logout.isChecked());

    // clicking Logout finishes the CollectionsActivity and returns to the login page
    try (ActivityController<CollectionsActivity> controller = Robolectric.buildActivity(CollectionsActivity.class)){
        controller.setup();
        activity = controller.get();

        activity.findViewById(R.id.nav_Logout).performClick();
        Intent expectedIntent = new Intent(activity, LoginActivity.class);
        Intent actual = shadowOf(RuntimeEnvironment.getApplication()).getNextStartedActivity();
        assertEquals(expectedIntent.getComponent(), actual.getComponent());
    }
}
```

7.5 User testing

User testing was a key component of our development process as we wanted to gather insights and feedback from our intended audience to ensure the overall design and navigation was clear and to find any bugs with our application. We allowed users to use both the web application and the mobile application to ensure both platforms were tested and the main goal wasn't just ensuring the applications worked as intended but ensuring that it worked well for our users. The first stand out response from our user testing was that some users had issues with the differences in the UI of both the web application and mobile application.


Did you find the UI of the web application and Mobile application consistent across both devices

10 responses


If Not, why?

3 responses

Web app had a navbar but mobile app did not

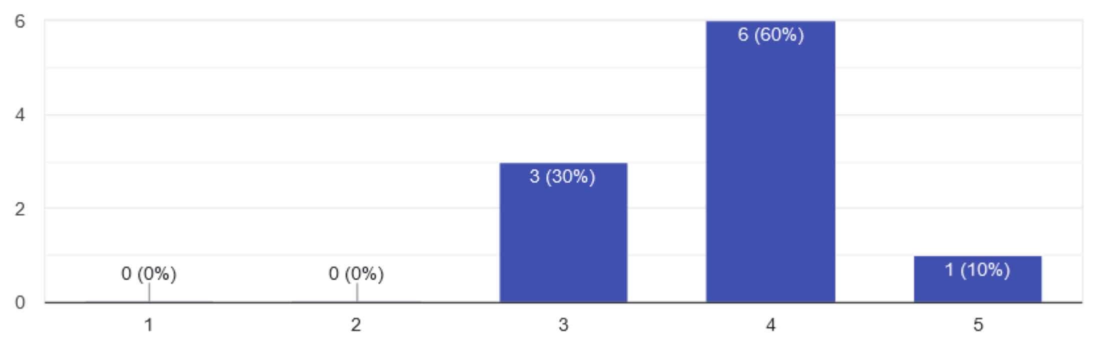
Wording was inconsistent across applications (Sign In/Log In, File/Note)

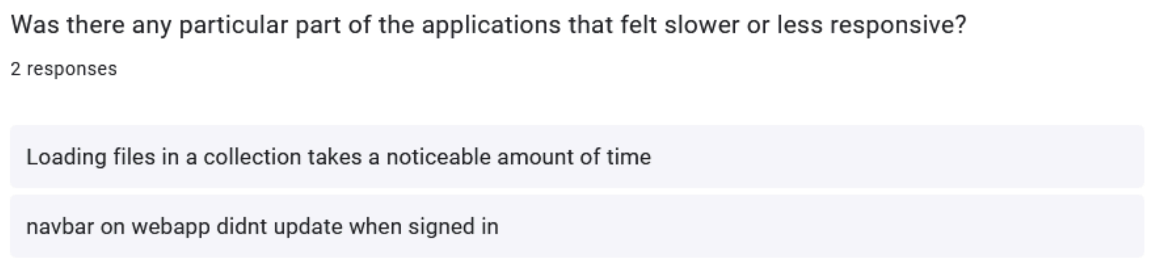
no navbar on mobile

This was extremely valuable data to us as we wanted users to feel as if they were using the same application whether on web or mobile. One issue was that users felt it was confusing that the web app had a navigation bar on the left side of the screen however the mobile app did not, and using this result we acknowledged this problem and developed a similar navigation bar on the mobile app therefore the process to perform functionalities would be the same across platforms. Another user also identified that the wording across platforms was slightly different pointing out that one said log in while the other said sign in, this was a clear and obvious error that we were also able to rectify. The next important response was when asking users on the performance of the application, as we wanted to ensure our app was responsive to users and minimise loading times.

How would you rate the overall performance of the apps?

10 responses




Was there any particular part of the applications that felt slower or less responsive?

2 responses

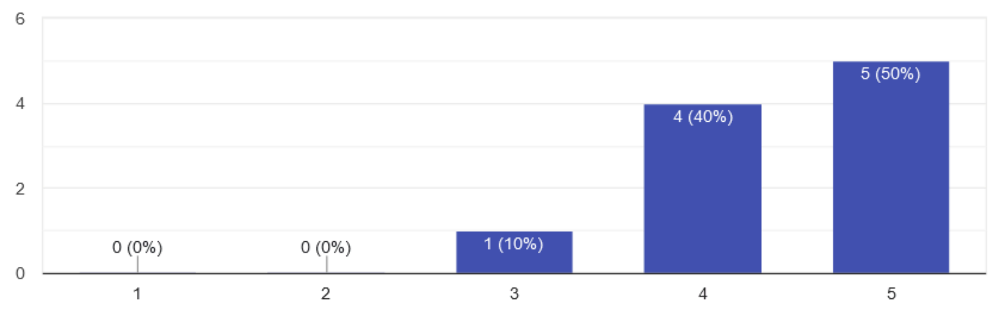
Loading files in a collection takes a noticeable amount of time

navbar on webapp didnt update when signed in

This was more valuable feedback to us as it gave us particular features that we could improve on. The first user complained that the loading time on files was noticeable and we set out to reduce this. We came to the solution that we could thread the api calls for files to significantly reduce the time it takes to display the files in a repository to a user. We also had a user identify that upon log in the nav bar on the web app did not immediately update and still said sign in until the page was manually refreshed or the user navigated to a new page. This was a clear bug that we missed in development and were able to fix from the user's feedback. We finally asked users how likely they would be to use this application again and we received strong feedback which was great and we believe with the further improvements made upon user testing that we could receive an even higher score a second time around.

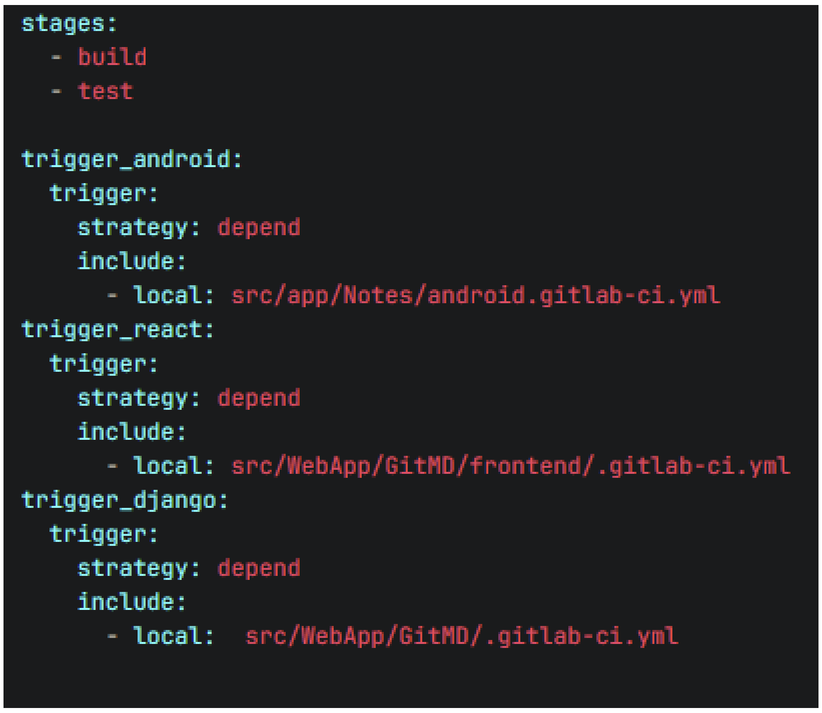
How likely would you be to use this app in a team or project environment?

10 responses



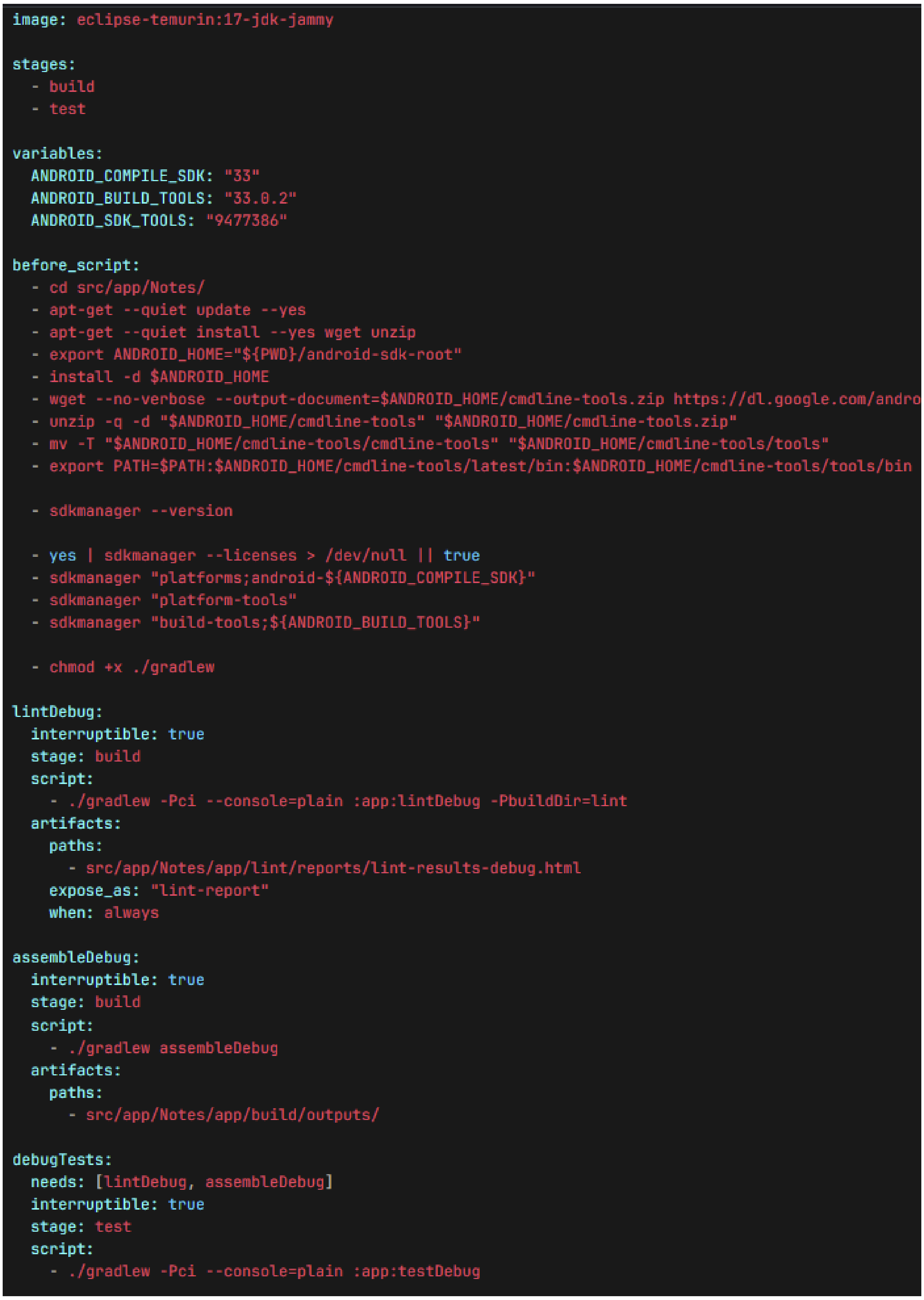
7.6 Results

To ensure we maintained the high quality standard of our code and that any changes made didn't cause regressions, we protected our main branch using pipelines which ran automated tests on our code. These automated tests covered all our unit, integration and frontend tests and would block a push or merge if any of the tests failed. On gitlab we run a parent pipeline on commits that triggers child pipelines to run our tests. We have three child pipelines for running tests on different areas of our code.



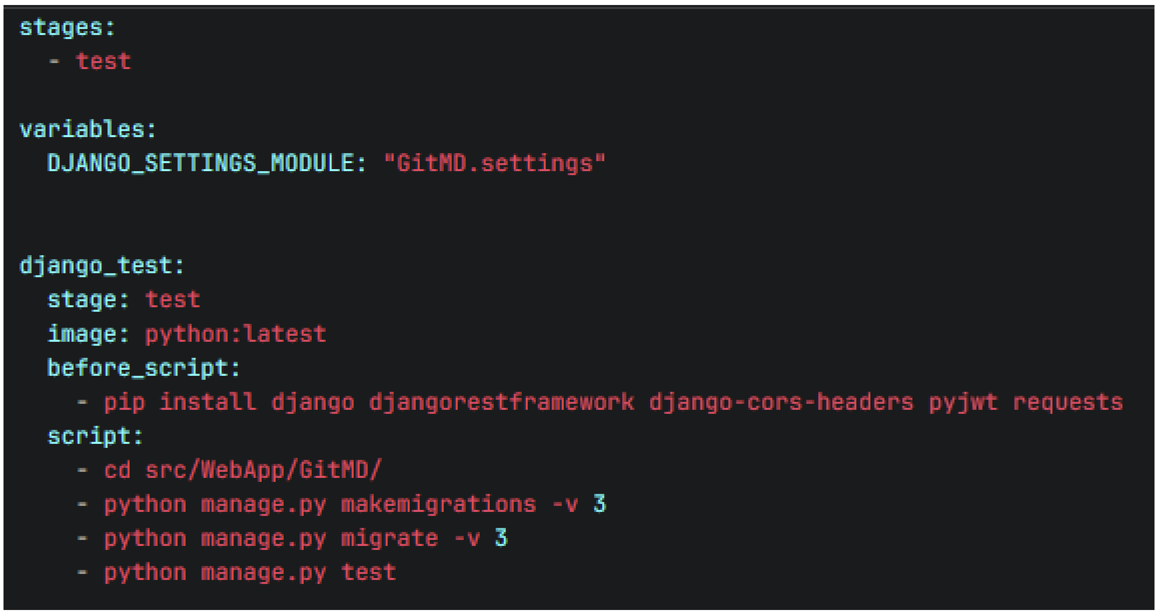
```yaml
stages:
  - build
  - test

trigger_android:
  trigger:
    strategy: depend
    include:
      - local: src/app/Notes/android.gitlab-ci.yml
trigger_react:
  trigger:
    strategy: depend
    include:
      - local: src/WebApp/GitMD/frontend/.gitlab-ci.yml
trigger_django:
  trigger:
    strategy: depend
    include:
      - local:  src/WebApp/GitMD/.gitlab-ci.yml
```

The first pipeline is for the mobile app. The before script installs the necessary tools and packages to build the app. Then linting checks are run and the project is built. When assembleDebug and lintDebug are finished, debugTests runs all the tests. If any of the tests fail the pipeline is interrupted, failing it.



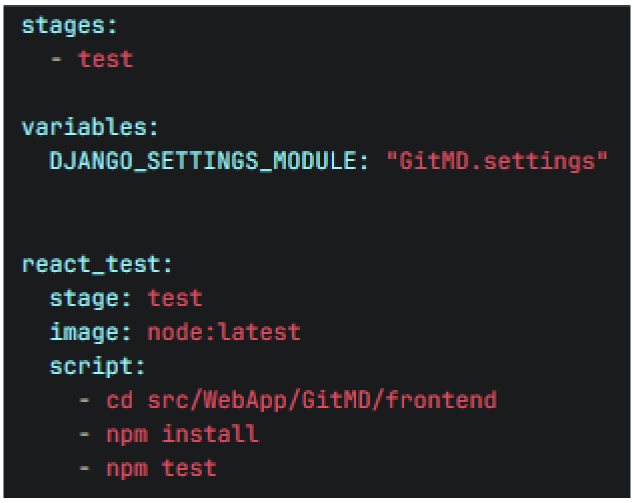
```yaml
image: eclipse-temurin:17-jdk-jammy

stages:
  - build
  - test

variables:
  ANDROID_COMPILE_SDK: "33"
  ANDROID_BUILD_TOOLS: "33.0.2"
  ANDROID_SDK_TOOLS: "9477386"

before_script:
  - cd src/app/Notes/
  - apt-get --quiet update --yes
  - apt-get --quiet install --yes wget unzip
  - export ANDROID_HOME="${PWD}/android-sdk-root"
  - install -d $ANDROID_HOME
  - wget --no-verbose --output-document=$ANDROID_HOME/cmdline-tools.zip https://dl.google.com/andro
  - unzip -q -d "$ANDROID_HOME/cmdline-tools" "$ANDROID_HOME/cmdline-tools.zip"
  - mv -T "$ANDROID_HOME/cmdline-tools/cmdline-tools" "$ANDROID_HOME/cmdline-tools/tools"
  - export PATH=$PATH:$ANDROID_HOME/cmdline-tools/latest/bin:$ANDROID_HOME/cmdline-tools/tools/bin

  - sdkmanager --version

  - yes | sdkmanager --licenses > /dev/null || true
  - sdkmanager "platforms;android-${ANDROID_COMPILE_SDK}"
  - sdkmanager "platform-tools"
  - sdkmanager "build-tools;${ANDROID_BUILD_TOOLS}"

  - chmod +x ./gradlew

lintDebug:
  interruptible: true
  stage: build
  script:
    - ./gradlew -Pci --console=plain :app:lintDebug -PbuildDir=lint
  artifacts:
    paths:
      - src/app/Notes/app/lint/reports/lint-results-debug.html
    expose_as: "lint-report"
    when: always

assembleDebug:
  interruptible: true
  stage: build
  script:
    - ./gradlew assembleDebug
  artifacts:
    paths:
      - src/app/Notes/app/build/outputs/

debugTests:
  needs: [lintDebug, assembleDebug]
  interruptible: true
  stage: test
  script:
    - ./gradlew -Pci --console=plain :app:testDebug
```

The django backend tests are run automatically by installing all dependencies in the before script, moving to the correct directory, making migrations before finally running the tests.



```
stages:
  - test

variables:
  DJANGO_SETTINGS_MODULE: "GitMD.settings"


django_test:
  stage: test
  image: python:latest
  before_script:
    - pip install django djangorestframework django-cors-headers pyjwt requests
  script:
    - cd src/WebApp/GitMD/
    - python manage.py makemigrations -v 3
    - python manage.py migrate -v 3
    - python manage.py test
```

The react frontend tests use the latest version of node, moves to the correct directory then uses npm install to install all the dependencies in the package.json file before finally running tests.



```
stages:
  - test

variables:
  DJANGO_SETTINGS_MODULE: "GitMD.settings"


react_test:
  stage: test
  image: node:latest
  script:
    - cd src/WebApp/GitMD/frontend
    - npm install
    - npm test
```

8. Future work

8.1 Hosting

Currently, our project is not hosted on a web domain or available on the android play store, and has to be run locally or installed as a jdk. Having the service available to anyone who wishes to use it would be an important step in releasing our application and research would be required into different hosting providers that could be able to achieve this for us. It would also be important for us to have a dedicated server to run our gitea server 24/7 as right now it is hosted on a personal machine.

8.2 Code Highlighting

We made attempts to have code highlighting within our services throughout the project but could not have it effectively working in time. We would ideally want to allow users to type ```java “some java code here” ``` and have our application identify the chosen language and highlight specific code accordingly, just as users would see in a code editor such as vs code. Doing so would improve the usability and readability of our service while also strengthening our use case for developers writing documentation for work or college students writing up assignments or documentations for projects.

8.3 Images

A way for users to upload local files to our service so that they can reference and use them in their markdown files would help users use specific images that are not on the web as markdown needs to use the image url to display the image but users local files won't have this. A repository could be set for each user called images where they could upload local files to and we can then take the image url and supply it to users when they are creating or editing files of their own so that they can incorporate these images.

8.4 Live Editing

An interesting concept that would be great to incorporate into the application would be live editing of markdown files so that users can see exactly when changes are being made and who is making them. The challenge that comes with this implementation is working it into git as a live update cant constitute a commit or else every file will have hundreds if not thousands of commits making version control extremely difficult. Research would have to be made to get past this issue and also into ensuring minimum delay between one user editing a file and another user seeing it.

9. References

1.*Welcome to PyJWT — PyJWT 2.8.0 documentation*, https://pyjwt.readthedocs.io/en/stable/.
2. “Sending email.” *Django documentation*, https://docs.djangoproject.com/en/5.0/topics/email/.
3. “remarkjs/react-markdown: Markdown component for React.” *GitHub*, https://github.com/remarkjs/react-markdown.
4. *Jest · Delightful JavaScript Testing*, https://jestjs.io/.
5. “Gitea API.” *Gitea Documentation*, https://docs.gitea.com/api/1.20/.
6. “Download Android Studio & App Tools.” *Android Developers*, https://developer.android.com/studio.
7. *Robolectric*, https://robolectric.org/.
8. “Retrofit.” *Square Open Source*, https://square.github.io/retrofit/.
9. “Introduction | Markwon.” *noties.io*, 11 January 2021, https://noties.io/Markwon/.