

# **How to Build Web Apps:**

## **Finding the right tools to build web apps**

By: Matthieu Dubray

CSD480 Capstone Project

Lake Washington Institute of Technology

April 29, 2022

# 1. Table of Contents

## 1. TABLE OF CONTENTS

## 2. ABSTRACT

## 3. INTRODUCTION

- a. What is the purpose of this paper?
- b. What is a web app?
- c. What is a tech stack?

## 4. PROJECT SPECS FOR THE WEB APPS

- a. Defining the app
- b. Defining the models
  - i. User model
  - ii. Post model
  - iii. Comment model
- c. Defining the back-ends
  - i. Python stacks
    - 1. Django/PostgreSQL stack
    - 2. Flask/SQLite
  - ii. JavaScript stack
    - 1. NodeJS/Express/MongoDB
  - iii. Cloud Provider
    - 1. Amazon web services (AWS)
- d. Defining the API endpoints

- e. Defining the front-end**
  - i. React
  - ii. Tailwind CSS/Post CSS
- f. External tools to help development**

## **5. BACK-END SETUPS**

### **a. Code Walkthrough (Django Stack)**

- i. Installing Django and Creating base app
- ii. Setting up postgresSQL database
- iii. Creating post model
- iv. Adding routes & validating user data for post model
- v. Creating user model
- vi. Adding routes, validating user data, & user authentication for user model
- vii. Creating comment model
- viii. Adding routes & validating user data for comment model

### **b. Code Walkthrough (Flask Stack)**

- i. Creating base app
- ii. Setting up the structure of the project
- iii. Setting up SQLite database
- iv. Creating post model
- v. Adding routes & validating user data for post model
- vi. Creating user model
- vii. Adding routes & validating data for user model
- viii. Adding user authentication with JSON web token
- ix. Adding user functionality to post model
- x. Creating comment model
- xi. Adding routes & validating user data for comment model
- xii. Adding comment functionality to post model

### **c. Code Walkthrough (NodeJS/Express Stack)**

- i. Creating base app

- ii. Setting up the structure of the project
- iii. Setting up MongoDB database
- iv. Creating post model
- v. Adding routes & validating user data for post model
- vi. Creating user model
- vii. Adding routes & validating user data for user model
- viii. Adding user authentication with JSON web tokens
- ix. Adding user functionality to post model
- x. Creating comment model
- xi. Adding routes & validating user data for comment model
- xii. Adding comment functionality to post model

## **6. FRONT-END SETUP**

### **a. Code Walkthrough (React)**

- i. Creating base app
- ii. Removing template code
- iii. Auth Context
- iv. Post Context
- v. Creating the header & nav
- vi. Login and signup forms
- vii. Main post page & post form
- viii. Individual post, update, delete, retrieve & comments, commentForm section

## **7. CLOUD SETUP**

### **a. AWS Beanstalk Deploy with Github Actions & Docker**

- i. Download docker
- ii. Staging app for deployment
- iii. Creating docker development files
- iv. Creating docker production files
- v. Creating AWS account

- vi. Setting up IAM user
- vii. Creating beanstalk app/environment
- viii. Setting up AWS environment variables
- ix. Creating Github account
- x. Creating repo of project
- xi. Setting up github action secrets
- xii. Creating github deploy file
- xiii. Testing CI/CD deployment to AWS

## **8. COMPARING TECH STACKS**

### **a. Back-end Comparison**

- i. Advantages & Disadvantages (Django)
- ii. Advantages & Disadvantages (Flask)
- iii. Advantages & Disadvantages (NodeJS/Express)
- iv. Brief look at other popular back-end frameworks
  - 1. Fastify
  - 2. NestJS
  - 3. ASP.NET

### **b. Front-end comparison**

- i. Advantages & Disadvantages (React)
- ii. Brief look at other popular front-end frameworks
  - 1. VueJS
  - 2. Angular

### **c. Database comparison**

- i. Advantages & Disadvantages (SQL based)
  - 1. PostgreSQL
  - 2. SQLite
- ii. Advantages & Disadvantages (NoSQL based)
  - 1. MongoDB
- iii. Brief look at other popular SQL & NoSQL databases
  - 1. DynamoDB

## 2. MySQL

### **d. Cloud provider comparison**

- i. Advantages & Disadvantages (AWS)
- ii. Brief look at other popular cloud providers
  - 1. Azure (Microsoft)
  - 2. Google cloud (Google)

### **e. Brief look at other popular tech stacks**

- i. MEAN/MEVN
- ii. LAMP

## 9. CONCLUSION

## 2. Abstract

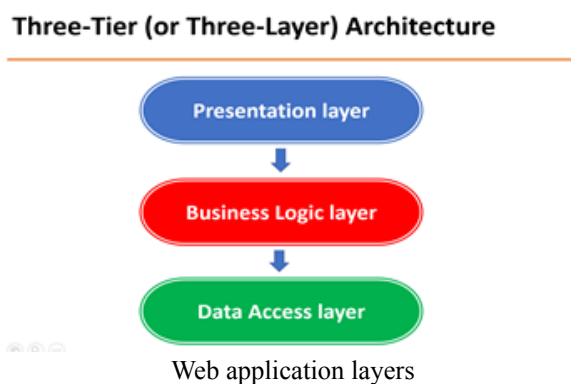
In programming there is an ever growing number of new tools and technologies a developer can use to create something. There are dozens of languages that are being used today, and among some of the most popular languages individuals or companies have created tools to help make it easier for other developers to create apps. There is unity combined with C# or Unreal with C++ to give game developers an engine to create stunning games. There is python along with a few libraries, or R and jupyter notebooks to provide good data analysis solutions. Similarly, web applications have a plethora of options for their back end component, from C# with ASP.NET, PHP, Golang, Java servlets, and the three featured in this paper which will be Python using Flask and another with Python using Django, as well as JavaScript using NodeJS with Express. With these techstacks, there will be a coding walkthrough on how to start from scratch and build a web app both front-end and back-end and then put it on AWS cloud platform to be reachable by other people. With this ever increasing number of tools it can be hard to understand which tools to use for what types of projects, and why some tools might be better than others. Sometimes the tools don't even need to be performatively better but can be based on popularity as well as ease of use. This paper analyzes three different tech stacks, comparing their advantages and disadvantages and comparing them to each other, in terms of how easy they were to set up, how easy it was to learn, and how well they can scale.

## 3. Introduction

### 3.1 What is the purpose of this paper?

There is an ever growing number of new tools and technologies, and with that a growing number of jobs that seek to hire people that know how to use them. This paper will look at 3 popular back-ends, and database tools along with a popular front-end and cloud provider to create 3 different tech stacks, and try to provide some comparison among them and look at some alternatives not used for the stacks. These tools will demonstrate how to build web apps from the ground up, with a similar approach to real world development. while also showing different tech stacks to show different tools that a dev or team can use for their idea and needs.

### 3.2 What is a web app?



#### 3.2.1 What is it?

A web app (or web application) is a client-server application that runs on a browser. This application can be broken down into layers, these layers consist of the presentation layer or UI, the business layer, and the Data access layer, and are used to reduce dependency between the layers to allow for different teams to work simultaneously, as well as more security as the user won't have access to the data layer directly:



Presentation Layer (client side): A client would visit a website and interact with the front-end

Business Logic Layer (server side): handles the users request and interacts with the data layer

Data layer (server side): takes data that is processed by the business layer and stores it in a database, also used to retrieve data for the business layer to process back to the client if needed

### **3.2.2 How does it work?**

Let's say there is a blog app, when a user visits the blog, they would be seeing the front end which can have sign in/up buttons, blog posts for the user to see, and other things. These blogs are populated by sending a request to the back-end's API associated with the posts which will then grab the post data from the database and return it back to the front-end for it to format the data into components for the user to see. These types of requests are typically done automatically when the page is visited or reloaded.

The user can also invoke requests themselves and pass data to the back-end. For instance, if the user wanted to sign up, they would click the button which would bring up the form and fill it out then clicking a submit button that data would be sent to another API call which would handle validating the user inputted data, and eventually sticking that data into a database table to be looked at again when the user wanted to login

## **3.3 What is a tech stack?**

### **3.3.1 What is it?**

A tech stack is a set of technologies used to create an application. Rather than coding everything from scratch there are a lot of tools out there that provide the basic necessities to creating a web app. Such as front-end libraries to help make web pages more dynamic and interactive for the user. back-ends have frameworks that help to automate the overhead associated with common activities performed in web development. For example, many

web frameworks provide libraries for database access, templating frameworks, session management, and they often promote code reuse, and also they provide interaction with the browser.

The web apps demonstrated in this paper will each have a different tech stack, so that they can be compared against each other, for ease of use, popularity, and scalability. However for this paper there will only be one front-end and one cloud provider as it adds more complexity than needed to this paper to have more than one of each, but there will be brief looks at alternative technologies for each part of the stack.

### **3.3.2 What makes a tech stack or library/framework popular?**

This can be a tough question, sometimes it can be how performative the technologies are, sometimes it can just be what is trendy.

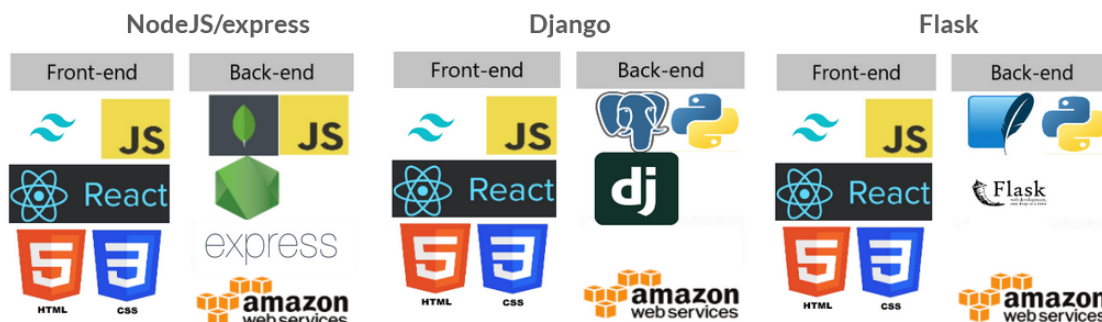
Some popular stacks include the MEAN/MERN/MEVN stacks which all use mongoDB express and nodejs but change their front-end with either react, angular, or vuejs. Other stacks to mention are LAMP/WAMP which are apache, mysql, and php used either with linux or windows servers, and C#'s ASP.NET, with some sql or noSQL based databases and some front-end frameworks mentioned before.

## 4. Project Specs for the Web Apps

## 4.1 Defining the app

For the three different tech stacks, the project that will be built is a simple blog app. This blog app will contain a front-end component where the user will be able to signup and login to authenticate themselves. As well as view all other users posts, and creating, updating, and deleting their own posts which will require the client to be logged in to do. Finally there will be comments on posts that any user can add to any post if they are logged in, these comments will not be able to be edited or deleted (however the functionality can be added if wanted). The front-end will then call the back-end through its exposed APIs to interact with the actual data itself, whenever the client sends a valid request. The architecture of this entire app will be a RESTful API architecture on the back-end hosted on a cloud server as well as an SPA (Single Page Application) architecture on the front-end also hosted on a cloud server.

The three different tech stacks will be the first, MERN, which stands for MongoDB/Express/React/Nodejs and is a fairly popular and easy to learn tech stack. The second, Postgres/Django/React, this one can be a little more challenging for people just starting web development, as django has a standard that can be a bit rigid. Finally the third, SQLite/Flask/React, which is also an easy to learn tech stack.



## 4.2 Defining the models

Models represent individual tables in a sql database that are a representation of the data, for instance a field in a model will represent a column in that model's table. For this project there will be 3 different models:

User Model:

the user model will hold basic information for the user such as a unique identifier, a username, an email, a password (the password will be hashed so there are no plain text passwords in the database), a first and last name, and for foreign keys/references there is a reference to posts to keep track of that individual users posts (this will be one-to-many).

User	
PK	Id
FK1	post: posts
	username: string
	first_name: string
	last_name: string
	email: string
	password: string

Post Model:

The post model will hold a unique identifier, a title, an image url path, a body, a date created and update field, and for foreign keys/references there is a reference to the user to keep track of the author of the post.

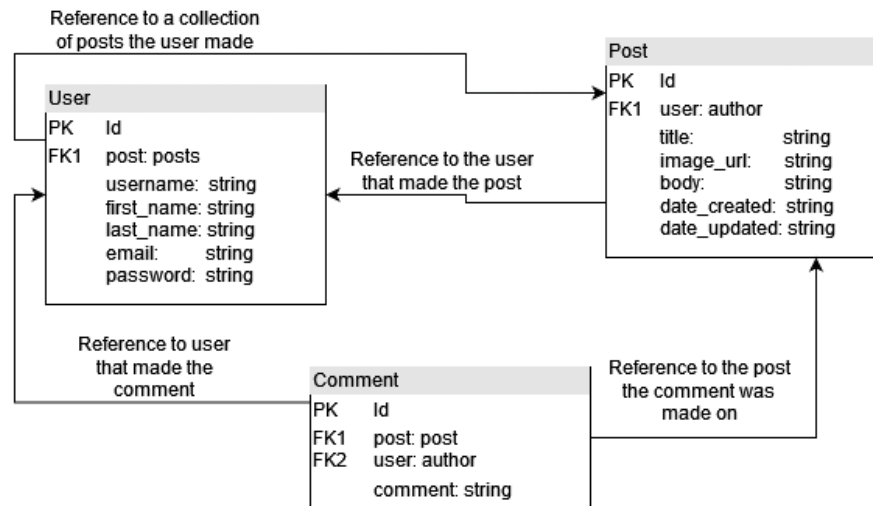
Post	
PK	Id
FK1	user: author
	title: string
	image_url: string
	body: string
	date_created: string
	date_updated: string

Comment Model:

The comment model will hold a unique identifier, a comment, and 2 foreign keys/references, one of which will be a reference to the post the comment was made on and the second being the user who made that comment.

Comment	
PK	Id
FK1	post: post
FK2	user: author
	comment: string

## Relationships between the 3 models:



### 4.3 Defining the back-ends

For the back-ends, there will be 3 different tech stacks.

The first stack will be in python using the Django framework and postgresSQL as the database. The second stack will also be in python and it will be using the Flask framework and SQLite as the database. The third tech stack will be in javascript and it will be using NodeJS along with the Express framework and MongoDB as the database. All of these back-ends will then be hosted on separate AWS instances that can then be called by the front-end.

#### 4.3.1 What is a RESTful back-end?

The back-ends will be using the RESTful architecture. The REST architecture uses a set of constraints that the APIs have to follow, as stated by restfulapi (<https://restfulapi.net/rest-architectural-constraints/>):

1. **Uniform Interface:** standardized communication between the client server, such as using HTTP with URI resources, CRUD and JSON.
  - a. Such as `http://blogapp.com/posts` (send GET, POST to retrieve or create data), `http://blogapp.com/post/<postId>` (to PUT or DELETE, data), `http://blogapp.com/signup`.
2. **Client-Server:** Client and server applications must not depend on each other.
3. **Stateless:** Between different calls to the APIs there is no session or history being stored and that every request is treated as a new,

- a. In between a client calling to create a post and retrieving that post the server should not store any history or any form of memory of the previous call to the next call
- 4. Cacheable: In some form or other the application should cache data as it can improve client experience as well as offload some work for the server.
  - a. cached data can include caching user tokens for easy authentication and to keep users logged in after exiting
- 5. Layered System: A layered system is where the entire app is layered out into different servers,
  - a. Previously shown, the approach for this will be a presentation layer where the client will be able to interact with the app, a business layer that will handle calls from the client, and a data layer that will handle all of the data storage and manipulation of data to and from the business layer.
- 6. Code on demand (optional): Most of the time the response from an API call will be in the form of XML/JSON, however it is allowed to send back executable code

#### 4.3.2 Python stacks

The 2 stacks used with python will be the Django and PostgreSQL and, Flask and SQLite, and both will be hosted on AWS

**Django:** (<https://www.djangoproject.com/>)

Django is one of python's most popular web frameworks that gives its users an easier time creating web apps. Having its own file structure to keep things orderly and including a plug-and-play module system, form validation, data abstraction, etc, it offers a good scalable back-end solution. Django follows the MTV (model-template-view) architecture pattern, using its own templating language it can pass variables, or even execute code in the django template HTML files, where django will then pre process (executing all the template code) the HTML files and return them to the client with the data it injected. This however is not the approach of this project, as previously stated the back-ends will be using a RESTful architecture, where in Django will receive requests from the user, interact with the data, and then send the user back some JSON typically with a status code, a message, and sometimes some actual data.

**Flask:** (<https://flask.palletsprojects.com/en/2.1.x/>)

Flask is another popular python based web framework, however unlike django it is a minimalist framework that includes pretty much the bare bones just the ability to handle requests and responses, to a browser. However, combined with tools such as werkzeug WSGI (Web Server Gateway Interface) and jinja templating language, and flask's various extension libraries, flask can quickly become as powerful as django. However, without all the nice structuring that django offers. In exchange the framework includes exactly what is needed and nothing more.

**PostgreSQL:** (<https://www.postgresql.org/>)

Postgres (or PostgreSQL) is a free and open source database system that extends what SQL can do by adding features that offer better security and scalability. Some of these features include offering support to different data types, and providing better indexing methods to make searches more performant. Postgres can also be used as a NoSQL document based database. Django also offers default ORM tools for PostgreSQL and a few other popular SQL options; an ORM tool generates objects which map to tables in the database virtually, so if a field called username with type of string is in a model, the orm tool will create a new column in the sql table of the model called username with type string.

**SQLite:** (<https://sqlite.org/index.html>)

SQLite is a minimal, fast, reliable SQL engine that fits the theme of Flask's minimal style. As per [SQLite's own words](#) "SQLite works great as the database engine for most low to medium traffic websites. ... The amount of web traffic that SQLite can handle depends on how heavily the website uses its database. Generally speaking, any site that gets fewer than 100K hits/day should work fine with SQLite." So for a small blog this will work ok, but if the app

begins to scale it might be better to look at something like postgresSQL.

### 4.3.3 JavaScript Stack

The final back-end stack will be using NodeJS to provide the ability for javascript to be run outside of a browser, and paired with Express web framework and mongoDB to server as the full back-end

**NodeJS/Express:** (<https://nodejs.org/en/>)

NodeJS is an open source server environment that allows for javascript to be run outside of a browser and onto a server. To be honest, it's pretty much the only option to run javascript on a server.

As for Express, there are plenty of other options such as NestJS which uses typescript (a superset of javascript that uses typing), NextJS which works with reactJS to essentially be a full stack framework kind of similar to django. Express is being used as it's fairly minimalist and easy to learn as well as being popular in the MExN (x => react, vue, or angular) stacks.

**MongoDB:** (<https://www.mongodb.com/>)

MongoDB is a NoSQL document based database and is popular again in the MExN stacks. It integrates well into most cloud platforms and does performant searches.

### 4.3.4 Cloud Provider (AWS) (<https://aws.amazon.com/>)

For the cloud provider that will host the apps, Amazon Web Service (AWS) will be the choice. AWS is currently the most popular cloud provider, coming with an abundance of services that provide everything, serverless APIs, budgeting tools, virtual machines, database tools, logistics, etc. With these tools it can allow any single user to an enterprise level company to build scalable apps for their users.



## 4.4 Defining the API endpoints

The API endpoints for this project are as listed, they can also be found (<https://documenter.getpostman.com/view/17038140/UyxdLUZF>):

**GET** request to ‘<some URL>/api/posts/<page:int>’: this endpoint requires no special headers or body, and will return a list of posts paginated to the page give (pagination will return  $n * p - 1$  to  $n * p$  posts, i.e. set max posts per page to 3 (n) and on page 2 (p) it will return posts 4-6, where on page 1 would return posts 1-3).

**GET** request to ‘<some URL>/api/post/<postId:int>’: this endpoint requires no special headers or a body, and will return a single post with the same post Id provided in the url (<some URL>/post/2 will return only the post with id of 2) if the post with that id doesn’t exist it will return an error.

**POST** request to ‘<some URL>/api/posts’: this endpoint provided with the correct body (title:string, body:string, image:file) and a special authorization header (JSON web token encoded with users credentials), will allow the user to create a new post.

**POST** request to ‘<some URL>/api/post/comments/<postId:int>’: this endpoint provided with the post id from the URL and the authorization header will allow the user to create a comment on a post (note this application does not allow users to change or delete comments).

**PUT** request to ‘<some URL>/api/post/<postId:int>’: this endpoint provided the correct body (title:string, body:string, image:file, any of these fields can be left blank and the field will not be changed) and authorization header (this should be the same token used when creating the post, i.e. the original author of the post is the only one that can send a request to it) will update any given fields.

**DELETE** request to ‘<some URL>/api/post/<postId:int>’: this endpoint provided the correct authorization header (this should be the same token used when creating the post, i.e. the original author of the post is the only one that can send a request to it) will delete the post fields.

**PUT** request to ‘<some URL>/api/signup’: this endpoint provided with the correct body (email:string, first\_name: string, last\_name: string, username:string, password:string, confirmPassword:string) will allow the user to create an account to then be logged in to.

**POST** request to ‘<some URL>/api/login’: this endpoint will allow a user that has signed up an account to login and be provided with credentials to create posts and comments etc.

## 4.5 Defining the front-end

For the front-end portion of the stacks will be kept simple, ReactJS to act as the front-end library to provide an SPA (Single Page Application) for better user experience and Tailwind CSS to provide simple styling to the ui (to avoid writing all of the css), and lastly Post CSS which is just a CSS preprocessor.

**React:** (<https://reactjs.org/>)

React is a state based library that will re-render individual components of a web page as it updates. This is good for a few reasons, it provides a better user experience as they don’t need to reload a page to get new data, it also takes off the load of the back-end from having to preprocess a template HTML file and then send it back to the client. React uses JSX which basically lets us use html elements in javascript files which react will handle rendering all of the components.

**Tailwind CSS:** (<https://tailwindcss.com/>)

Tailwind is a css framework that is also performance focused as it will produce css files for only the css used in the project rather than its entire library. This means that not only is there no need to make the styles but it also won’t bloat what is served to the client.

## 4.6 External tools to help development

There are some useful tools that can help the development process. The first tool is some sort of text editor as expected, visual studio code (<https://code.visualstudio.com/>) offers a really good free solution to this.

Additionally another free tool known as Postman (<https://www.postman.com/>) which is a tool that will store the APIs to be called whenever needed as well as help to document them to make it easier to know its function and is a big want especially when releasing public APIs for other developers to know how to use.

## 5. Back-End Setups

For the python stacks download python:<https://www.python.org/downloads/>, from this link find the download button for python which will download an executable to run. After installing with the default settings, if it did not automatically add python to the PATH.

### 5.1 Code Walkthrough (Django Stack)



#### 5.1.1 Installing Django and Creating the base app

Start by creating an empty directory, whether through a file manager or from a terminal.

```
matt@ubuntu:~/Documents$ mkdir NewProject
matt@ubuntu:~/Documents$ cd NewProject/
matt@ubuntu:~/Documents/NewProject$
```

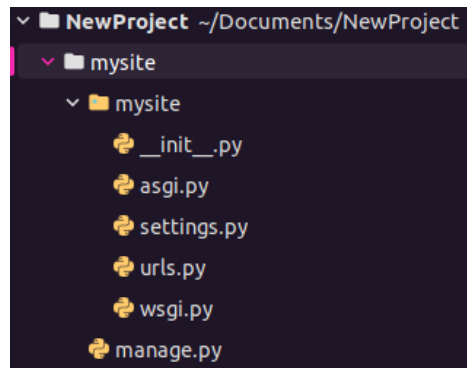
Then after creating a new directory, open an IDE in the directory and open the terminal or stay on the same terminal and run **pip install Django**.

```
matt@ubuntu:~/Documents/NewProject$ pip install Django
```

Afterwards use a Django command in the terminal inside of the directory, to initialize a base Django project: **django-admin startproject <Name of app>**

```
matt@ubuntu:~/Documents/NewProject$ django-admin startproject mysite
```

After there should be a file structure in the directory similar to this:



### 5.1.2 Setting up postgresQL database

For the django stack. Postgres will be the database that holds the data, to set up a local database, go to <https://www.postgresql.org/download/> and download the installer or install through a cli for the OS and follow the installation steps.

Side note: there will be a different setup when putting the django project on AWS using AWS RDS.

**For linux/mac users**, after installing postgres check to make sure it is good by typing **sudo systemctl status postgresql**.

```
matt@ubuntu:~$ sudo systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
   Loaded: loaded (/lib/systemd/system/postgresql.service; enabled; vendor pre
   Active: active (exited) since Mon 2022-05-02 22:28:25 PDT; 5min ago
   Main PID: 5645 (code=exited, status=0/SUCCESS)
     Tasks: 0 (limit: 9439)
    Memory: 0B
         CPU: 0
    CGroup: /system.slice/postgresql.service

May 02 22:28:25 ubuntu systemd[1]: Starting PostgreSQL RDBMS...
May 02 22:28:25 ubuntu systemd[1]: Finished PostgreSQL RDBMS.
```

Afterwards access postgres by using **sudo su postgres** and **psql** in the following order.

```
matt@ubuntu:~$ sudo su postgres
postgres@ubuntu:/home/matt$ psql
could not change directory to "/home/matt": Permission denied
psql (14.2 (Ubuntu 14.2-1.pgdg21.10+1), server 13.6 (Ubuntu 13.6-0ubuntu0.21.10.1))
Type "help" for help.

postgres=#
postgres=#
```

After entering the postgres terminal use **createdb -h localhost -p <port> -U <username> <db name>**, the default port is 5432 and default username is postgres so those will be left as they are.

```
postgres=# createdb -h localhost -p 5432 -U postgres testdb
```

Finally after creating the database to confirm it was made use **\l** To then open a list of databases.

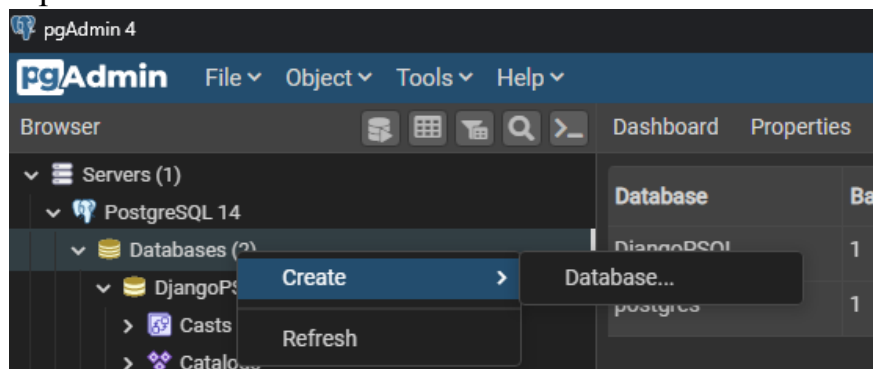
```
postgres-# \l
```

List of databases					
Name	Owner	Encoding	Collate	Ctype	
postgres	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	
template0	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	
template1	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	
testdb	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	
(4 rows)					

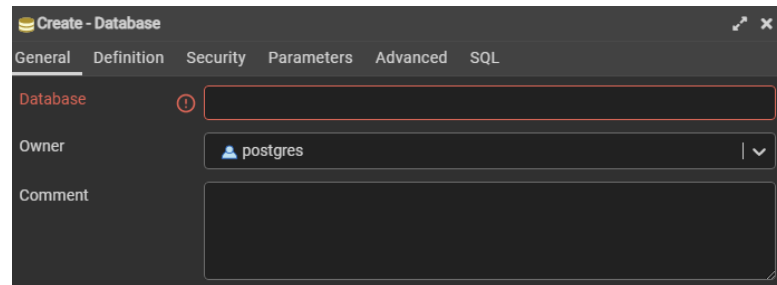
**For Windows/Mac users**, postgres has installers that will automatically set up everything and prompt to create a username and password (remember this as it is needed to login to postgres app and also connect to django).

Side note: downloading pgAdmin (postgres admin) will help setting up the database for django to connect.

From the postgres admin application, right click the databases to pull up a drop down and create a new database.



Specify the database name and owner can be left as the default user postgres.



Now then move back into the django project and visit the settings.py found in the project name directory, scroll down to the database section

```
# Database
# https://docs.djangoproject.com/en/4.0/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

This is where django holds the information of what database to connect to, doing so will let django automatically map the models and make new tables.

Django can also connect to database by replacing the default config to the following, and also replace any lines with <> with what was used when setting up the database:

BlogProject/Settings.py	<pre>DATABASES = {     'default': {         'ENGINE':             'django.db.backends.postgresql_psycopg2'         ,         'NAME': '&lt;DB name&gt;',         'USER': '&lt;username&gt;',         'PASSWORD': '&lt;password&gt;',         'HOST': 'localhost',         'PORT': '&lt;port&gt;',     } }</pre>
-------------------------	--

### 5.1.3 Creating post model

Now that the database connection is set up, start by making the post model. However, before doing that, understand that django apps have a unique file structure. New files or directories can't just be added straight into the app and just code away, django works by having modules which each contain a part of the app. These modules will typically contain an entire part of the project, in the case of this app there will be two modules (user and blog), where the user module will contain all logic for handling the users, and blog which will contain the logic for the posts and comments.

To create a new module in django open a terminal in the app root directory and use **python manage.py startapp <app name>**.

```
matt@ubuntu:~/Documents/NewProject/mysite$ python3 manage.py startapp newApp
```

There should now be a new directory within the project with the same name as the name input in the command. After creating the app, it needs to be registered in the django settings config, similar to the database.

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

This is the place where django will include modules in the build process. To add the new module simply adding its name to the list of apps, (replace **'blog'** with the name of the app):

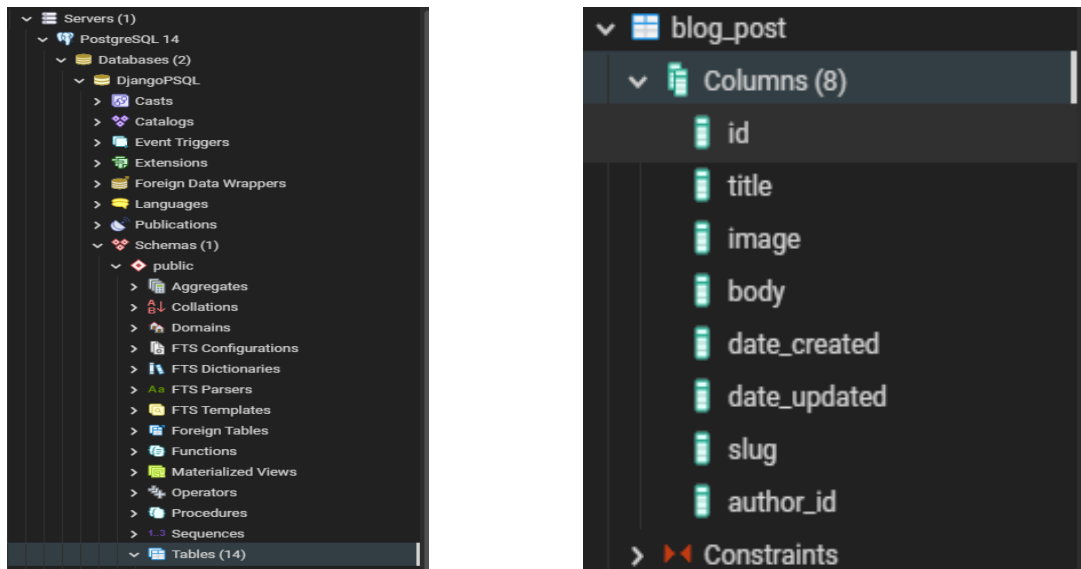
```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog'
]
```



Finally to add the post model. When creating new models, django will look in all models.py files of all connected modules and look for classes that inherit from a Model class. Inside the class fields can be specified which will act as the columns in a SQL table. Go to the new app and visit the models.py file, copy the code below:

blog/models.py	<pre> from django.conf import settings from django.db import models  class Post(models.Model):     title = models.CharField(max_length=100)     image = models.ImageField(upload_to='images', null=True)     body = models.TextField()     date_created = models.DateTimeField(auto_now_add=True)     date_updated = models.DateTimeField(auto_now=True)      # this will use the default django user model as the     model     author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name='author')      # when a detailed post is needed     def as_json(self):         return dict(             id=self.id,             title=self.title,             imageUrl=str(self.image.url),             body=self.body,             createdAt=self.date_created,             author=self.author.as_json(),         )      # when a less detailed list of posts is needed     def list_json(self):         return dict(             id=self.id,             title=self.title,             createdAt=self.date_created,             author=self.author.as_json()         ) </pre>
----------------	--

After adding both posts to the installed\_apps config and adding in this post model, using **python manage.py runserver**, django will start up a live local server. While there are currently no endpoints to visit, looking in the database from the pg admin app there should be a new table under **Server>PostgreSQL>Databases> <db name> > Schemas > Tables**, will be a few django default tables but also a <module name>\_post table with all of the fields from the model.



#### 5.1.4 Adding routes & validating user data for post model

Django offers predefined class structures that can be inherited from to make creating views easier. Views are where the logic for listening for requests and serving back responses to the user, typically with the MTV architecture, views would return a render call pointing to an HTML django template file. For this project, the views will return JSON.

As for the Django view classes, there are different kinds that are used for different cases. In this case, with the help of an external library, called Django REST-framework. This library provides useful view classes, methods, and other things that help to convert Django's MTV architecture into one that more fits with RESTful. To install this, similar to Django, use **pip install djangorestframework**.

```
matt@ubuntu:~/Documents/NewProject$ pip install djangorestframework
Collecting djangorestframework
```

Now with the package installed, visit the views.py file in the blog module and start adding two new views to the class that inherit from the ModelViewSet class from the rest framework.

blog/views.py	<pre>from rest_framework import viewsets  class PostsViewSet(viewsets.ModelViewSet):     pass</pre>
---------------	---

	<pre>class PostViewSet(viewsets.ModelViewSet):     pass</pre>
--	---

The ModelViewSet inherits from the ViewSet class which provides overridable functions for create, list, retrieve, destroy, and update (aka POST, GET, DESTROY, and PUT; HTTP methods). These methods can alter the data as needed and can easily be hooked up with the model as well as the routing for the app to handle on its own. The viewsets are separated into two to make routing more simple and similar to the other tech stacks, it is however possible to make custom routers.

Next add some new settings to the settings.py config file to allow for file uploads, and add rest\_framework to the installed apps list.

BlogProject/settings.py	<pre>INSTALLED_APPS = [     'django.contrib.admin',     'django.contrib.auth',     'django.contrib.contenttypes',     'django.contrib.sessions',     'django.contrib.messages',     'django.contrib.staticfiles',     'rest_framework',     'blog', ]  # where files that can be retrieve/uploaded to MEDIA_ROOT = BASE_DIR / 'uploads' # specify where files can be retrieved MEDIA_URL = '/user-media/'</pre>
-------------------------	---

Before creating the views, the way that data validation will be handled for this project is through serializers that will act as a very simple and easy validation setup. Start by creating a serializers.py file in the module.

blog/serializers.py	<pre>from rest_framework import serializers from .models import Post  class PostSerializer(serializers.ModelSerializer):      class Meta:         model = Post         fields = '__all__'</pre>
---------------------	---

Model Serializers in the rest framework convert objects into data types understandable by javascript and front-end frameworks (JSON). Serializers also parse data to first validate the incoming data, and then will handle saving the data to a new object of the model.

blog/views  
.py  
Get list &  
Create

```
from django.contrib.auth.models import AnonymousUser
from django.shortcuts import get_object_or_404
from django.core.paginator import Paginator
from rest_framework import viewsets, status
from rest_framework.decorators import parser_classes
from rest_framework.response import Response
from rest_framework.parsers import MultiPartParser

from .models import Post
from .serializers import PostSerializer

# Create your views here.
class PostsViewSet(viewsets.ModelViewSet):
    queryset = Post.objects.all()
    # specifying the serializer class to use
    serializer_class = PostSerializer

    def list(self, request, **kwargs):
        pass

    def retrieve(self, request, pk=None, **kwargs):
        # returning all posts (this can be a good place to add
        pagination)
        posts = self.queryset

        p = Paginator([post.list_json() for post in posts], 3)
        return Response({'posts': p.page(pk).object_list, 'message':
        'Posts Retrieved Successfully', 'totalPages': p.num_pages},
        status.HTTP_200_OK)

    @parser_classes(MultiPartParser)
    def create(self, request):
        if type(request.user) is AnonymousUser:
            return Response({'message': 'Not Authorized'},
            status.HTTP_401_UNAUTHORIZED)
        # when sending a post request, django will store files in
        # request.FILES with the name of form items being keys
        if 'image' not in request.FILES:
            return Response({'message': 'No image uploaded'},
            status.HTTP_422_UNPROCESSABLE_ENTITY)
        if request.FILES['image'].name.split('.')[1].lower() not in
        ['png', 'jpeg', 'jpg']:
            return Response({'message': 'incorrect file type submitted
            (accepted: PNG, JPG, JPEG)'},
            status.HTTP_422_UNPROCESSABLE_ENTITY)

        post = Post(
            title=request.data['title'],
            image=request.FILES['image'],
            body=request.data['body'],
            author=request.user
        )
        # .save() is a method that django dynamically makes to save
        model
        post.save()

        return Response({'post': post.as_json(), 'message': 'Post
        Created Successfully'}, status.HTTP_200_OK)

    @parser_classes(MultiPartParser)
    def update(self, request, pk=None):
```

	<pre> pass  def destroy(self, request, pk=None, **kwargs):     pass </pre>
--	--

With the first viewset, the create method and get for list of posts is used. While the others 3 are set to not work so that there wouldn't be extra endpoints when not needed.

The retrieve method receives an input from the URL for the page number as pagination will be used to control how much data is being sent over at a time, paginator will separate out the list given to it into pages containing the amount specified as the second param. Afterwards, it can use a page method to specify what page is wanted and then send the list back with the object\_list field containing all entries on the page specified.

The create method uses a decorator that will parse the incoming request with the form data passed through it to a request.data dictionary and files to request.FILES. The request holds headers where it will first check that a user provided an auth header with their JWT token (this will be implemented later). Afterwards, it will check that a file has been provided and that the file has a mime type of png, jpeg, or jpg. Finally, it will create a new post object and save it to the database, and send back a confirmation response with the post details in the response.

<b>blog/views.py</b> <b>Get single post</b> <b>Edit post &amp;</b> <b>Delete post</b>	<pre> class PostViewSet(viewsets.ModelViewSet):     queryset = Post.objects.all()     # specifying the serializer class to use     serializer_class = PostSerializer      def list(self, request, **kwargs):         pass      def create(self, request):         pass      def retrieve(self, request, pk=None, **kwargs):         # attempting to see if post with the given id exists         try:             post = Post.objects.get(pk=pk)         except Post.DoesNotExist:             return Response({'message': 'That post does not exist'}, status.HTTP_404_NOT_FOUND) </pre>
--	---

```

        return Response({'post': post.as_json(), 'message': 'Post
Retrieved Successfully'}, status.HTTP_200_OK)

    @parser_classes(MultiPartParser)
    def update(self, request, pk=None):
        if type(request.user) is AnonymousUser:
            return Response({'message': 'Not Authorized'},
status.HTTP_401_UNAUTHORIZED)
        # attempting to see if post with the given id exists
        try:
            post = Post.objects.get(pk=pk)
        except Post.DoesNotExist:
            return Response({'message': 'That post does not
exist'}, status.HTTP_404_NOT_FOUND)

        # user authentication to see if post author is the user
        if post.author == request.user:
            storage, path, image = post.image.storage,
post.image.path, str(post.image).split('/')[1]
            # checking if the user passed in a new image
            if 'image' in request.FILES:
                if
request.FILES['image'].name.split('.')[1].lower() not in
['png', 'jpeg', 'jpg']:
                    return Response({'message': 'incorrect file type
submitted (accepted: PNG, JPG, JPEG)'},
status.HTTP_422_UNPROCESSABLE_ENTITY)
                    storage.delete(path)
                    post.image = request.FILES['image']

            post.title = request.data['title'] or post.title
            post.body = request.data['body'] or post.body
            # after changing the wanted variables we call .save()
            to update the database
            post.save()
            return Response({'post': post.as_json(), 'message':
'Post Updated Successfully'}, status.HTTP_200_OK)
        else:
            return Response({'message': 'User is not author of
post'}, status.HTTP_401_UNAUTHORIZED)

    def destroy(self, request, pk=None, **kwargs):
        if type(request.user) is AnonymousUser:
            return Response({'message': 'Not Authorized'},
status.HTTP_401_UNAUTHORIZED)
        # attempting to see if post with the given id exists
        try:
            post = Post.objects.get(pk=pk)
        except Post.DoesNotExist:
            return Response({'message': 'That post does not
exist'}, status.HTTP_404_NOT_FOUND)

        # user authentication to see if post author is the user
        if post.author is request.user:
            # deleting the image before we delete the post
            storage, path = post.image.storage, post.image.path
            storage.delete(path)
            # .delete() is another method django makes to delete
rows
            post.delete()
            return Response({'post': post.as_json(), 'message':

```

	<pre>'Post Deleted Successfully'}, status.HTTP_200_OK)     else:         return Response({'message': 'User is not author of post'}, status.HTTP_401_UNAUTHORIZED)</pre>
--	---

Much like the other viewset, this viewset contains logic for retrieve, update, and destroy, while some methods will be passed to avoid unwanted calls.

The retrieve method will accept a param through the URL, which will act as the id of the post, and return a more detailed single post back as the response (for when individual posts are looked up on the front-end).

The update method will first check that an auth header with a valid JWT is provided and that the user is the author of the post. Afterwards it will check for any new file submissions with the same process as the create method, delete the current image if a new one is sent and replace it with the new image. Finally it will check for if the body or title has changed and will update them accordingly and save to the database.

The delete method will first check that an auth header with a valid JWT is provided and that the user is the author of the post. Afterwards it will delete the image attached to the post and then delete the post from the database.

Now that the views have been set up, the routing will need to be set up to actually be able to reach these endpoints. Using another useful tool from the rest framework called routers, the viewset can conveniently be used to set the url endpoint, default routers will have specific syntax that the URL must follow depending on what endpoint it is trying to access (i.e. for PUT requests it will look for /<endpoint>/<id>. As mentioned before a custom router can be made but this will work fine. Using the django convention of variable named urlpatterns as a list, that django will go and look to register these paths.

blog/urls.py	<pre> from django.urls import path, include from rest_framework import routers  from . import views  router = routers.DefaultRouter() # GET, POST http://localhost:8000/posts/ router.register('posts', views.PostViewSet) # GET, PUT, DELETE http://localhost:8000/post/&lt;id&gt; router.register('post', views.PostViewSet)  urlpatterns = [     path('', include(router.urls)) ]</pre>
--------------	--

Now to attach it to the project, visit the urls.py file located in the main directory with the rest of the config files (this will be named after the project when django project was first created).

BlogProject/urls.py	<pre> from django.conf import settings from django.conf.urls.static import static from django.contrib import admin from django.urls import path, include  urlpatterns = [     path('admin/', admin.site.urls),     path('', include('blog.urls')) ] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT) # this bit is part of the image file to give access to django to use those locations for storing/getting files</pre>
---------------------	--

Now boot up the postman app (if postman was not installed go to <https://www.postman.com/downloads/> and follow the download instructions). Start the django server by opening a terminal in the root directory of the project and use **python manage.py runserver**.

```

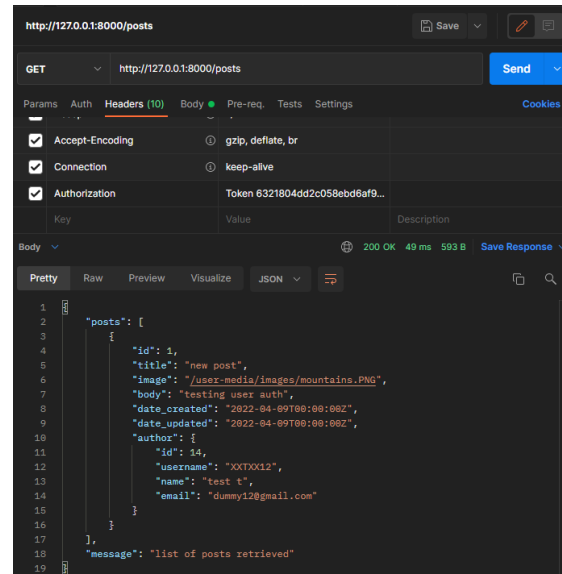
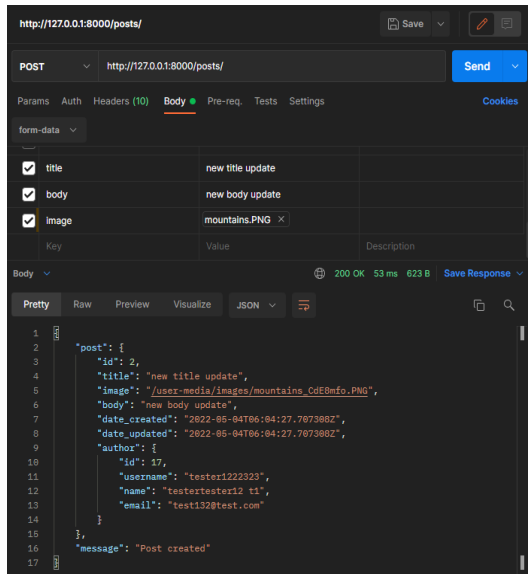
natt@ubuntu:~/Documents/NewProject/mysite$ python3 manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly
Run 'python manage.py migrate' to apply them.
May 04, 2022 - 05:31:14
Django version 4.0.4, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```



Once the server is up and running, test with postman using the endpoints at **http://localhost:8000/post** or **http://localhost:8000/posts**, refer to the [API docs](#) included in **section 4** if needed.



### 5.1.5 Creating user model

First create a new module that will hold all the logic for the users. Use **python manage.py startapp <app name>**.

```
matt@ubuntu:~/Documents/NewProject/mysite$ python3 manage.py startapp user
```

Next go to the models.py file inside the new module. Django offers a default user model without even needing to make one or a module for it. However, the user model for this project will be different so instead it will inherit `AbstractBaseUser` which will give a lot of the functionality of the default user model but the model can have its own fields and methods.

user/models.py	<pre> from django.contrib.auth.models import AbstractBaseUser from django.db import models  # Create your models here. class User(AbstractBaseUser):     email = models.EmailField(verbose_name='email', max_length=50, unique=True)     username = models.CharField(max_length=25, unique=True)     date_joined = models.DateTimeField(verbose_name='date joined', auto_now_add=True)     last_login = models.DateTimeField(verbose_name='last login', auto_now=True)     is_admin = models.BooleanField(default=False)     is_active = models.BooleanField(default=True) </pre>
----------------	---

```

is_staff = models.BooleanField(default=False)
is_superuser = models.BooleanField(default=False)
first_name = models.CharField(max_length=20)
last_name = models.CharField(max_length=20)

# setting login to use email as the login field
USERNAME_FIELD = 'email'
REQUIRED_FIELDS = ['username', 'first_name', 'last_name',
'password']

def __str__(self):
    return self.username

def has_perm(self, perm, obj=None):
    return self.is_admin

def has_module_perms(self, app_label):
    return True

def as_json(self):
    return dict(
        id=self.id,
        username=self.username,
        name=(self.first_name + ' ' + self.last_name),
    )

```

Finally hook up the new module to the settings config as well as adding in a new setting to set the default user model for django.

BlogProject/settings.py

```

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'blog',
    'user'
]

# used to set default user model for django
AUTH_USER_MODEL = 'user.User'

```

### 5.1.6 Adding routes, validating user data, & user authentication for user model

Before creating routes, some new settings will be used to make sure that django uses JSON web tokens, as well as cors. Start by adding the rest framework auth token and corsheaders to the app list list. Along with a few other needed settings, so that it matches this.

<b>BlogProject</b> <b>/settings.py</b>	<pre> INSTALLED_APPS = [     'django.contrib.admin',     'django.contrib.auth',     'django.contrib.contenttypes',     'django.contrib.sessions',     'django.contrib.messages',     'django.contrib.staticfiles',     'django.contrib.sites',     'corsheaders',     'rest_framework',     'rest_framework.authtoken',     'blog',     'user' ]  # somewhere in the file SITE_ID=1 MIDDLEWARE = [     'django.middleware.security.SecurityMiddleware',     'django.contrib.sessions.middleware.SessionMiddleware',     'corsheaders.middleware.CorsMiddleware',     'django.middleware.common.CommonMiddleware',     'django.middleware.csrf.CsrfViewMiddleware',     'corsheaders.middleware.CorsPostCsrfMiddleware',     'django.contrib.auth.middleware.AuthenticationMiddleware',     'django.contrib.messages.middleware.MessageMiddleware',     'django.middleware.clickjacking.XFrameOptionsMiddleware', ]  # REST framework settings REST_AUTH_REGISTER_SERIALIZERS = {     "REGISTER_SERIALIZER":     "user.serializers.RegisterUserSerializer", } REST_FRAMEWORK = {     'DEFAULT_PERMISSION_CLASSES': [         'rest_framework.permissions.AllowAny',     ],     'DEFAULT_AUTHENTICATION_CLASSES': (         'rest_framework_simplejwt.authentication.JWTAuthentication',     ), } REST_AUTH_SERIALIZERS = {     "USER_DETAILS_SERIALIZER":     "user.serializers.UserSerializer", }  REST_USE_JWT = True </pre>
---	---

```
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=60),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=10),
    'ROTATE_REFRESH_TOKENS': True,
    'BLACKLIST_AFTER_ROTATION': True,
    'ALGORITHM': 'HS256',
    'SIGNING_KEY': SECRET_KEY,
    'VERIFYING_KEY': None,
    'AUTH_HEADER_TYPES': ('JWT',),
    'USER_ID_FIELD': 'id',
    'USER_ID_CLAIM': 'user_id',
    'AUTH_TOKEN_CLASSES':
    ('rest_framework_simplejwt.tokens.AccessToken',),
    'TOKEN_TYPE_CLAIM': 'token_type',
}

# CORS Settings
CORS_ORIGIN_ALLOW_ALL = True
```

user/managers.py	<pre> from django.contrib.auth.base_user import BaseUserManager  # custom manager for user model class CustomUserManager(BaseUserManager):     # create_user will take all the inputs and will make     # a new user entry in the table     def create_user(self, username, email, first_name, last_name, password, **extra_fields):         if not email:             raise ValueError('the email must be set')         if not username:             raise ValueError('the username must be set')         norm_email = self.normalize_email(email)         user = self.model(             username=username,             email=norm_email,             first_name=first_name,             last_name=last_name,         )         # django will automatically hash this password         # before setting it in the table to ensure         # better security standards         user.set_password(password)         user.save()         return user  # this isn't necessary but if you wish to make an admin # account you can include this def create_superuser(self, email, password):     user = self.create_user(         email=self.normalize_email(email),         username='admin',         password=password,     )     user.is_admin = True     user.is_active = True     user.is_staff = True     user.is_superuser = True     user.save() </pre>
------------------	---

Then set the manager of the model inside the model class.

user/models.py	<pre> class User(AbstractBaseUser):     # . . . other code     objects = CustomUserManager()     # . . . other code </pre>
----------------	--

Setting the objects field in the model, which is what will be able to interact with the data layer, and what was used previously for the post views. This is what the manager does, acts as a layer of abstraction between django models and the database.

First create the serializers, by creating a new serializers.py file in the user module. There are going to be four different serializers one is for the user model, one is the register serializer that will handle validating and creating a new user object. The last two are for logging in, rest framework provides two classes TokenObtain and TokenRefresh that will take in a user login, and attempt to authenticate the user and then return a jwt with that users data, and token expiration date.

user/serializers.py	<pre> from django.contrib.auth.password_validation import validate_password from .models import User from rest_framework import serializers  class UserSerializer(serializers.ModelSerializer):     class Meta:         model = User         fields = '__all__'  class RegisterUserSerializer(serializers.ModelSerializer):     email = serializers.EmailField(required=True)     username = serializers.CharField(required=True)     first_name = serializers.CharField(required=True)     last_name = serializers.CharField(required=True)     password = serializers.CharField(         write_only=True, required=True, validators=[validate_password])     confirmPassword = serializers.CharField(write_only=True, required=True)      class Meta:         model = User         fields = ('email', 'username', 'password', 'confirmPassword', 'first_name', 'last_name')      # custom validation     def validate(self, attrs):         if attrs['password'] != attrs['confirmPassword']:             raise serializers.ValidationError(                 {'password': 'passwords must match'})          if User.objects.filter(username=attrs['username']).exists():             raise serializers.ValidationError(                 {'username': 'User with this username already exists'})          if User.objects.filter(email=attrs['email']).exists():             raise serializers.ValidationError(                 {'email': 'This email is already registered with us, please login'})          return attrs </pre>
---------------------	--

	<pre> def create(self, validated_data):     password = validated_data.pop('password', None)     confirmPassword = validated_data.pop('confirmPassword', None)     # as long as the fields are the same, we can just use this     post = self.Meta.model(**validated_data)     if password is not None:         post.set_password(password)     post.save()     return post  class TokenObtainLifetimeSerializer(TokenObtainPairSerializer):      def validate(self, attrs):         data = super().validate(attrs)         refresh = self.get_token(self.user)         # renaming access key to token to avoid changing frontend for django specific         data['token'] = data.pop('access')         # setting additional keys to provide necessary information for frontend         data['lifetime'] = int(refresh.access_token.lifetime.total_seconds())         data['userId'] = self.user.id         data['username'] = self.user.username         return data  class TokenRefreshLifetimeSerializer(TokenRefreshSerializer):      def validate(self, attrs):         data = super().validate(attrs)         refresh = RefreshToken(attrs['refresh'])         data['lifetime'] = int(refresh.access_token.lifetime.total_seconds())         return data </pre>
--	---

Now that the user model and data validation is handled, the only thing left to add is views that will let the user use the signup and login feature of the app.

user/views.py	<pre> from rest_framework import status from rest_framework.decorators import api_view from rest_framework.response import Response from rest_framework_simplejwt.views import TokenViewBase  from .serializers import RegisterUserSerializer, TokenObtainLifetimeSerializer, TokenRefreshLifetimeSerializer  @api_view(['PUT']) def register(request): </pre>
---------------	--

	<pre> serializer = RegisterUserSerializer(data=request.data)  if serializer.is_valid():     user = serializer.save()     return Response({'message': 'user signed up successfully'}, status=status.HTTP_201_CREATED)     return Response(serializer.errors, status=status.HTTP_422_UNPROCESSABLE_ENTITY)  class TokenObtainPairView(TokenViewBase):     """     Return JWT tokens (access and refresh) for specific users based on username and password.     """     serializer_class = TokenObtainLifetimeSerializer  class TokenRefreshView(TokenViewBase):     """     Renew tokens (access and refresh) with new expire time based on specific user's access token.     """     serializer_class = TokenRefreshLifetimeSerializer </pre>
--	---

The TokenObtain and TokenReview, are classes that provide logic to check user login and return JSON web tokens that will contain relevant data for authenticating the user (the JWT itself encrypted with user credentials, a user id and username to provide to the front end to make user experience better).

Now that all of the views have been setup, create a urls.py file in the user module and register them in the urlpatterns list, and then register the new user urls to the main project urls.py.

user/urls.py	<pre> from django.urls import path from .views import register, TokenObtainPairView  urlpatterns = [     path('signup', register),     path('login', TokenObtainPairView.as_view(), name='login') ] </pre>
BlogProject/ urls.py	<pre> from django.conf import settings from django.conf.urls.static import static from django.contrib import admin from django.urls import path, include  urlpatterns = [     path('admin/', admin.site.urls),     path('', include('blog.urls')), </pre>



	<pre>path('', include('user.urls')) ] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)</pre>
--	---

### 5.1.7 Creating comment model

As before, create a new comment model in the models.py file from the blog module. Since the comments are related to the posts they are fine to be in the same module as the posts.

blog/models.py	<pre>class Comment(models.Model):     comment = models.TextField()     author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name='user_comments')     post = models.ForeignKey(Post, on_delete=models.CASCADE, related_name='comments', null=True, blank=True)      def as_json(self):         return dict(             id=self.id,             comment=self.comment,             author=self.author.as_json(),         )</pre>
----------------	---

The comment model will hold to foreign keys, one that points to the author who made the comment and one that points to the post the comment was made on.

### 5.1.8 Adding routes & validating user data for comment model

Similar to the post a serializer will handle validating the data input.

blog/serializers.py	<pre>class CommentSerializer(serializers.ModelSerializer):      class Meta:         model = Comment         fields = '__all__'</pre>
---------------------	--

Create a viewset for the comments, which will only include a create as the app won't let the user update or delete comments, and retrieving will be done with the post retrieve method (it can be implemented if wanted).

blog/views.py	<pre> from .serializers import PostSerializer, CommentSerializer  class CommentsViewSet(viewsets.ModelViewSet):     queryset = Comment.objects.all()     serializer_class = CommentSerializer      def list(self, request, **kwargs):         pass      @parser_classes(MultiPartParser)     def create(self, request):         if type(request.user) is AnonymousUser:             return Response({'message': 'Not Authorized'},                 status.HTTP_401_UNAUTHORIZED)         post = get_object_or_404(Post.objects.all(),             pk=request.data['postId'])          comment = Comment(             comment=request.data['comment'],             post=post,             author=request.user         )         comment.save()         return Response({'message': 'Comment Created             Successfully', 'comment': comment.as_json()},             status.HTTP_200_OK)      def retrieve(self, request, pk=None, **kwargs):         pass      # preventing user from trying to update comment     def update(self, request, pk=None, **kwargs):         pass      # preventing user from trying to delete comment     def destroy(self, request, pk=None, **kwargs):         pass </pre>
---------------	--

Finally, move onto the urls.py file in the blog module, and register the viewset like the posts viewset.

blog/urls.py	<pre> # . . .other code router.register('posts/comments', views.CommentsViewSet) # . . .other code </pre>
--------------	---

The django back-end is now complete and now either move onto the front end or build out the other two back-ends using different tech stacks.

## 5.2 Code Walkthrough (Flask Stack)



### 5.2.1 Creating base app

Start by creating an empty directory.

```
matt@ubuntu:~/Documents$ mkdir NewProject  
matt@ubuntu:~/Documents$ cd NewProject/  
matt@ubuntu:~/Documents/NewProject$
```

After creating a new directory, open it in an IDE and open the terminal or open it on the same terminal and type in **pip install Flask**.

```
matt@ubuntu:~/Documents/NewProject$ pip install Flask
```

In the new directory open up an ide inside of it and create a new `app.py` file in the root directory and include the following.

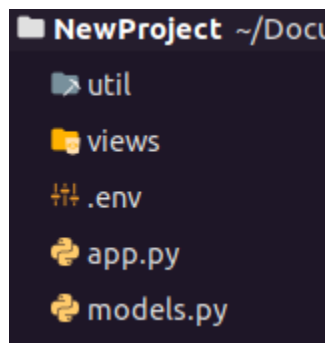
<code>app.py</code>	<pre>from flask import Flask  app = Flask(__name__)  @app.route("/") def hello_world():     return "&lt;p&gt;Hello, World!&lt;/p&gt;"</pre>
---------------------	---

Now using the **flask run** command in the terminal in the root directory of the project, flask will start up a local server that can be accessed.

```
matt@ubuntu:~/Documents/NewProject$ flask run
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
```

### 5.2.2 Setting up the structure of the project

Unlike django, flask is a minimalistic almost bare bones web framework, which basically means it doesn't come with anything that django provides like a pre-made file structure. There are different file structures that can be found online (<https://newbedev.com/common-folder-file-structure-in-flask-app>, more than just this too). In this project, this is the file structure that will be used, the util directory will hold any functionality needed in different parts of the app, views will hold the different endpoints for different parts of the app, .env is a file that will hold any secrets that do not want to be made public for instance on a github repo, app.py will be the main file that starts the server, models.py will hold all of the models.



It would be ideal for the models.py file to actually be a directory where each model is separated into its own file. However, for the project complexity and scope of the back-end it is not necessary.

### 5.2.3 Setting up SQLite database

To set up the sqlite database for flask, open the app.py file and add some config options.

app.py	<pre> from flask import Flask import os  # basedir holds the path to the root of the project directory basedir = os.path.abspath(os.path.dirname(__file__))  app = Flask(__name__)  # config options # tell flask what the path of the db will be app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' + os.path.join(basedir, 'blog.db')  # . . .other code </pre>
--------	--

Basedir will give a path string to the root directory of the project, and the sqlalchemy uri config will tell flask that this is where the database can be located.

### 5.2.4 Creating post model

As flask doesn't have a built in ORM tool like django a new package is needed, Flask-SQLAlchemy is an extension for flask to use SQLAlchemy to have access to an ORM tool. **pip install Flask-SQLAlchemy.**

```
matt@ubuntu:~/Documents/NewProject$ pip install Flask-SQLAlchemy
```

Now open the models.py file and make a new post model as well as initializing a database object.

models.py	<pre> from flask_sqlalchemy import SQLAlchemy  db = SQLAlchemy()  class PostModel(db.Model): </pre>
-----------	---

	<pre> __tablename__ = 'post'  id = db.Column(db.Integer, primary_key=True, autoincrement=True) title = db.Column(db.String(255), nullable=False) imageUrl = db.Column(db.String(255), unique=True) body = db.Column(db.Text(), nullable=False)  createdAt = db.Column(db.DateTime, nullable=False) updatedAt = db.Column(db.DateTime, nullable=False)  def json(self):     return {         'id': self.id,         'title': self.title,         'body': self.body,         'imageUrl': self.imageUrl,         'createdAt': self.createdAt,     }  def list_json(self):     return {         'id': self.id,         'title': self.title,         'createdAt': self.createdAt,     } </pre>
--	---

Now that the post model and db object are set up, can go back to the app.py file and hook up this db object to the flask app. SQLAlchemy doesn't provide migrations so the database file would need to be recreated every time any models change. However, Flask has another nice extension Flask-Migrate that handles migrations using alembic (a migration tool). **pip install Flask-Migrate.**

app.py	<pre> from flask import Flask from models import db import os  env = environ.Env(DEBUG=(bool, False)) # basedir holds the path to the root of the project directory basedir = os.path.abspath(os.path.dirname(__file__)) # reads variables from the .env file environ.Env.read_env(os.path.join(basedir, '.env'))  app = Flask(__name__)  # config options app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' + os.path.join(basedir, 'blog.db') app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False app.config['SECRET_KEY'] = env('SECRET_KEY') app.config['DEBUG'] = True  db.init_app(app) </pre>
--------	---

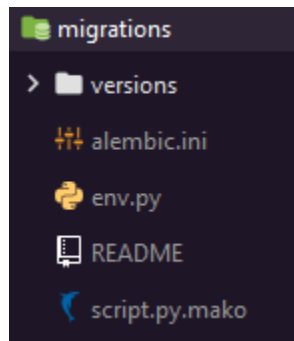
```

app.app_context().push()
migrate = Migrate(app, db)

# this should only be run once
db.create_all(app=app)

```

Now with the database object made, a post model created, and the migration tool set up, run **flask db init**, which will set up a migrations directory in the project root directory that looks like this.



With the migrations directory setup, use two new commands **flask db migrate** and **flask db upgrade** anytime there are changes to any models.

```

matt@ubuntu:~/Documents/NewProject$ flask db migrate
* Tip: There are .env or .flaskenv files present. Do "pip install python-alembic"
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Will assume non-transactional DDL.
INFO [alembic.env] No changes in schema detected.
matt@ubuntu:~/Documents/NewProject$ flask db upgrade
* Tip: There are .env or .flaskenv files present. Do "pip install python-alembic"
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Will assume non-transactional DDL.

```

The first command will stage any changes that have been made to any models, and the second will apply those to the database without needing to delete the schema or database and the data along with it.

Now there should be a `blog.db` file found in the root directory of the project that has a post table with the same fields as the post model in the `model.py` file.

### 5.2.5 Adding routes & validating user data for post model

As flask does not provide a default views.py file, create a new view in the views directory titled **post\_views.py**. This view file will contain all of the logic for the posts as well as the comments. To avoid needing to create many different route functions for flask, there will be separate methods that are then called through one main method for each of the different routes. This wouldn't be completely ideal as it can cause confusion or ambiguous code so feel free to change that. However, for the scope and complexity of this project it helps to keep the main app.py file cleaner.

<p>views/post_views.py Create:</p>	<pre> import os  from flask import request, jsonify, make_response from datetime import datetime as dt  from werkzeug.utils import secure_filename  from models import PostModel as Post from models import db from util.error_handler import respond_error  # basedir gives the path to the root directory of the # project basedir = os.path.abspath(os.path.dirname(os.path.dirname(__file__)))  # file_upload will point to a dir where images will be # stored file_upload = os.path.join(basedir, 'uploads/images')  # these are the file types that are allowed to be # uploaded ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg'}  def create_post(userid):     title = request.form['title']     body = request.form['body']      # on form submissions flask will store file upload     # in request.files     if 'image' not in request.files:         return respond_error('No image uploaded', 422)      # key from the form where the file is located     image = request.files['image']     if image.filename == '':         return respond_error('Invalid filename', 422)     if image and allowed_file(image.filename):         filename = upload_image(image)     else:         return respond_error('incorrect file type submitted (accepted: PNG, JPG, JPEG)')     post = Post(         title=title,         imageUrl=filename or None, </pre>
--	--



	<pre>         body=body,         author_id=userid,         createdAt=dt.now(),         updatedAt=dt.now()     )     db.session.add(post)     db.session.commit()     return response_post('Post created Successfully', post)  <i># will handle mutating filename to make sure it is</i> <i># unique and saving it to the specified upload folder</i> def upload_image(image):     filename = str(dt.now().timestamp()) + '-' + secure_filename(image.filename)     image.save(os.path.join(file_upload, filename))     return filename  <i># checks the extension type of the file uploaded and</i> <i># check if it is in ALLOWED_EXTENSIONS</i> def allowed_file(filename):     return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS  def response_post(message, requested_post):     return make_response(jsonify({         'message': message,         'post': requested_post.json()     }), 200) </pre>
--	---

The create method will handle all of the logic to get data from a POST request from the API endpoint, and create a new post object from the data sent to it.

views/post_views.py Retrieve:	<pre> def retrieve_all_posts(page):     posts = Post.query.paginate(page, 3)      return make_response(jsonify({         'message': 'Posts Retrieved Successfully',         'posts': [post.list_json() for post in posts.items],         'totalPages': posts.pages     }), 200) </pre>
----------------------------------	--

This retrieve method will retrieve all posts. Similar to django this return will also be paginated.

views/post_views.py Update:	<pre> def update_post(requested_post):     title = request.form['title'] or requested_post.title </pre>
--------------------------------	---

	<pre> body = request.form['body'] or requested_post.body requested_post.title = title requested_post.body = body requested_post.updatedAt = dt.now()  if 'image' in request.files:     image = request.files['image']     if image:         if allowed_file(image.filename):             delete_image(requested_post.imageUrl)             filename = upload_image(image)             requested_post.imageUrl = filename         else:             return respond_422('incorrect file type submitted (accepted: PNG, JPG, JPEG)')     db.session.commit()     return response_post('Post updated Successfully', requested_post)  # handles deleting images that are no longer needed def delete_image(filename):     os.remove(os.path.join(file_upload, filename)) </pre>
--	--

To update a post a user must send a PUT request to the server which will later (when the user model is set up) check for user authentication. But for now it won't check for user authentication.

views/post_views.py Delete:	<pre> def delete_post(requested_post):     delete_image(requested_post.imageUrl)     db.session.delete(requested_post)     db.session.commit()     return response_post('Post deleted Successfully', requested_post) </pre>
--------------------------------	---

The delete method will delete the image tied to the post and then the post. The user authentication will be implemented later. Now that all four of the methods, the main methods can be created that will be attached to the main app file to attach to endpoints.

views/post_views.py	<pre> def RUD_post(index):     requested_post = Post.query.get(index)     if requested_post is None:         return respond_error('That post does not exist', 404)     if request.method == 'GET':         return response_post('Post retrieved Successfully', requested_post)     if request.method == 'PUT':         return update_post(requested_post)     elif request.method == 'DELETE':         return delete_post(requested_post) </pre>
---------------------	--

	<pre> return respond_error('Not Authorized', 401)  def C_posts():     userId = # some userId     return create_post(userId) </pre>
--	--

Some of the functionality of the main methods are not currently implemented, such as the user authentication portion, but a token will need to be provided in the authorization header for some endpoints. Before creating the user model, add a new file to the utils directory which will hold methods that call error responses back to the user (not necessary).

utils/error_handler.py	<pre> from flask import make_response, jsonify def respond_error(message, status):     return make_response(jsonify({         'message': message     }), status) </pre>
------------------------	---

## 5.2.6 Creating user model

First install another package **Flask-Bcrypt** which is a flask extension that provides bcrypt's password hashing/checking. **pip install Flask-Bcrypt**.

```
matt@ubuntu:~/Documents/NewProject$ pip install Flask-Bcrypt
```

In the models.py file create a new user model, a special decorator can be used to create properties that can then use the setter decorator to tell how the value of an object should be set as. Using these decorators, the password will be automatically hashed using bcrypt's hash method, which will take the password input, hash it and add salting rounds to make it more secure. The check password method will use a function that will check the hashed password with the input password when authenticating a user to check if the input password is the correct password. Also add in two new fields to the post model that will point to the user model as a foreign key.

models.py	<pre> from flask_bcrypt import Bcrypt flask_bcrypt = Bcrypt()  class PostModel(db.Model):     # what sqlalchemy will name the table </pre>
-----------	--

```

__tablename__ = 'post'

id = db.Column(db.Integer, primary_key=True,
autoincrement=True)
title = db.Column(db.String(255), nullable=False)
imageUrl = db.Column(db.String(255), unique=True)
body = db.Column(db.Text(), nullable=False)

author_id = db.Column(db.Integer, db.ForeignKey('user.id'))
author = db.relationship('UserModel', back_populates='posts')

createdAt = db.Column(db.DateTime, nullable=False)
updatedAt = db.Column(db.DateTime, nullable=False)

def json(self):
    return {
        'id': self.id,
        'title': self.title,
        'body': self.body,
        'imageUrl': self.imageUrl,
        'createdAt': self.createdAt,
        'author': self.author.json(),
    }

def list_json(self):
    return {
        'id': self.id,
        'title': self.title,
        'createdAt': self.createdAt,
        'author': self.author.json(),
    }

class UserModel(db.Model):
    __tablename__ = 'user'

    id = db.Column(db.Integer, primary_key=True,
autoincrement=True)
    email = db.Column(db.String(255), unique=True, nullable=False)
    username = db.Column(db.String(50))
    first_name = db.Column(db.String(50))
    last_name = db.Column(db.String(50))
    password_hash = db.Column(db.String(100))
    createdAt = db.Column(db.DateTime, nullable=False)
    posts = db.relationship('PostModel', back_populates='author')

    @property
    def password(self):
        raise AttributeError('password: write-only field')

    @password.setter
    def password(self, password):
        self.password_hash =
flask_bcrypt.generate_password_hash(password).decode('utf-8')

    def check_password(self, password):
        return flask_bcrypt.check_password_hash(self.password_hash,
password)

    def json(self):
        return {
            'name': self.first_name + ' ' + self.last_name,
            'id': self.id,

```

	<pre>         'username': self.username     } </pre>
--	--

Finally run the two migrate commands in the terminal in the root directory of the project, **flask db migrate** and **flask db upgrade** to apply the changes and create the user model.

### 5.2.7 Adding routes & validating user data for user model

The user model will only need two routes, a signup and login route that will create a new user in the database and the second will authenticate the user, providing an auth token.

views/user_view.py	<pre> from flask import request, jsonify, make_response from datetime import datetime as dt  from models import UserModel as User from models import db from util.error_handler import respond_error from util.user_jwt import create_jwt  def user_signup():     if request.method == 'PUT':         email = request.form['email']         userExists = User.query.filter_by(email=email).first()         if userExists:             return respond_error('This email is already registered with us, please login', 422)         password = request.form['password']         confirm_password = request.form['confirmPassword']         if password != confirm_password:             return respond_error('passwords must match', 422)          first_name = request.form['first_name']         last_name = request.form['last_name']         username = request.form['username']          user = User(             email=email,             username=username,             # password hashing is done through the password.setter method in the user model             password=password,             first_name=first_name,             last_name=last_name,             createdAt=datetime.now()         )          db.session.add(user)         db.session.commit() </pre>
--------------------	---

	<pre> return make_response(jsonify(     {         'message': 'user has been registered',         'user': user.json()     }), 200)  def user_login():     if request.method == 'POST':         email = request.form['email']         userExists =         User.query.filter_by(email=email).first()         if userExists:             password = request.form['password']             if userExists.check_password(password):                 token = create_jwt(userExists)                 return make_response(jsonify(                     {                         'message': 'user has logged in',                         'token': token                     }), 200)             else:                 return respond_error('incorrect password', 422)         else:             return respond_error('This email does not exist, please signup', 422) </pre>
--	--

### 5.2.8 Adding user authentication with JSON web token

Create a new file in utils that will provide the application with JSON web tokens, which will be implemented with the JWT package. **pip install jwt**.

```
matt@ubuntu:~/Documents/NewProject$ pip install jwt
```

utils/user_jwt.py	<pre> import jwt  secret = 'enter secret'  def create_jwt(user):     return jwt.encode(         {             'userId': user.id,             'username': user.username,         },         secret, algorithm="HS256")  def decode_jwt(token):     return jwt.decode(token, secret, algorithms=["HS256"])  def is_user_authenticated(userId, token_userId): </pre>
-------------------	---

	<code>return userId == token_userId</code>
--	--

### 5.2.9 Adding user functionality to post model

Now that the user model and routes have been created, go back and include any user related data back into the post routes. JWTs can be implemented by simply checking for a header that will be set in the front-end, and can access it and send it to methods that need that token data or to use the **is\_user\_authentication** method to see if the user is the author of the post.

views/post_views.py	<pre> from util.user_jwt import decode_jwt, is_user_authenticated  def RUD_post(index):     requested_post = Post.query.get(index)     if requested_post is None:         return respond_error('That post does not exist', 404)     if request.method == 'GET':         return response_post('Post retrieved Successfully', requested_post.json())     token = request.headers['Authorization']     if token:         decoded = decode_jwt(token)         if is_user_authenticated(requested_post.author_id, decoded['userId']):             if request.method == 'PUT':                 return update_post(requested_post)             elif request.method == 'DELETE':                 return delete_post(requested_post)             else:                 return respond_error('User is not author of post', 401)             return respond_error('Not Authorized', 401)  def C_posts():     if request.method == 'POST':         token = request.headers['Authorization']         if token:             decoded = decode_jwt(token)             return create_post(decoded['userId'])         else:             return respond_error('Not Authorized', 401) </pre>
---------------------	--

### 5.2.10 Creating comment model

In the `models.py` create a new comment model. After adding in the model run the two commands **flask db migrate** and **flask db upgrade** to apply the changes to the database.

models.py	<pre> class PostModel(db.Model):     # what sqlalchemy will name the table     __tablename__ = 'post'      id = db.Column(db.Integer, primary_key=True, autoincrement=True)     title = db.Column(db.String(255), nullable=False)     imageUrl = db.Column(db.String(255), unique=True)     body = db.Column(db.Text(), nullable=False)      author_id = db.Column(db.Integer, db.ForeignKey('user.id'))     author = db.relationship('UserModel', back_populates='posts')      comments = db.relationship('CommentModel', back_populates='post')      createdAt = db.Column(db.DateTime, nullable=False)     updatedAt = db.Column(db.DateTime, nullable=False)      def json(self):         return {             'id': self.id,             'title': self.title,             'body': self.body,             'imageUrl': self.imageUrl,             'createdAt': self.createdAt,             'author': self.author.json(),             'comments': [comment.json() for comment in self.comments]         }  class CommentModel(db.Model):     __tablename__ = 'comment'      id = db.Column(db.Integer, primary_key=True, autoincrement=True)     comment = db.Column(db.String(255), nullable=False)      author_id = db.Column(db.Integer, db.ForeignKey('user.id'))      post_id = db.Column(db.Integer, db.ForeignKey('post.id'))     post = db.relationship('PostModel', back_populates='comments')      createdAt = db.Column(db.DateTime, nullable=False)      def json(self):         return {             'id': self.id,             'comment': self.comment,             'author': self.author.json()         } </pre>
-----------	---

### 5.2.11 Adding routes and validating data for comment model



For the routes there will only be one as the user will only be allowed to create a comment and not update or delete, and retrieve the comments when the post is retrieved.

views/post_views.py	<pre>def create_comment_on_post(postId, userId):     user_comment = request.form['comment']     comment = Comment(         comment=user_comment,         author_id=userId,         post_id=postId,         createdAt=dt.now()     )     db.session.add(comment)     db.session.commit()     return make_response(jsonify({         'message': 'Comment created Successfully',         'comment': comment.json()     }), 200)</pre>
---------------------	--

With the view a new main method will be created that will attach to the app.

views/post_views.py	<pre>def C_comments(index):     token = request.headers['Authorization']     if token and request.method == 'POST':         decoded = decode_jwt(token)         if request.method == 'POST':             return create_comment_on_post(index,                 decoded['userId'])</pre>
---------------------	--

### 5.2.12 Adding all views to routes on the app

Before attaching all of the endpoints to the app, first another new flask extension is needed **flask-cors** which will allow cors to be enabled for any domain, for this all domains will be enabled but for real apps that likely isn't the desired wait to set up cors. As it can have security implications but since the app uses JWT for authentication that should not be an issue.

app.py	<pre>from flask import Flask, send_file from flask_cors import CORS  from models import db, flask_bcrypt import environ import os</pre>
--------	---

```

from views.user_views import user_signup, user_login
from views.post_views import C_posts, RUD_post, C_comments,
retrieve_all_posts
from flask_migrate import Migrate

application = Flask(__name__, static_url_path='',
                    static_folder='uploads')
CORS(application, resources=r'/*')
# config options
application.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' +
os.path.join(basedir, 'blog.db')
application.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
application.config['SECRET_KEY'] = os.environ['SECRET_KEY']
application.config['DEBUG'] = True
application.config['ENV'] = 'development'
application.config['UPLOAD_FOLDER'] = os.path.join(basedir,
'uploads\\images')

db.init_app(application)
flask_bcrypt.init_app(application)

application.app_context().push()
migrate = Migrate(application, db)
db.create_all()

@app.route('/static/<path:path>')
def send_report(path):
    img_path = os.path.join(application.config['UPLOAD_FOLDER'],
str(path))
    return send_file(img_path)

@app.route('/signup', methods=['PUT'])
def signup():
    return user_signup()

@app.route('/login', methods=['POST'])
def login():
    return user_login()

@app.route('/post/<int:index>', methods=['GET', 'PUT',
'DELETE'])
def post(index):
    return RUD_post(index)

@app.route('/post/comments/<int:index>', methods=['POST'])
def comment(index):
    return C_comments(index)

@app.route('/posts', methods=['POST'])
def CreatePost():
    return C_posts()

@app.route('/posts/<int:page>', defaults={'page': 1})
def posts(page):
    return retrieve_all_posts(page)

if __name__ == '__main__':
    application.run(port=8080)

```

### 5.3 Code Walkthrough (NodeJS/Express)



Before creating a new project, download & install NodeJS (<https://nodejs.org/en/>).

#### 5.3.1 Creating base app

Start by creating an empty directory.

```
matt@ubuntu:~/Documents$ mkdir NewProject
matt@ubuntu:~/Documents$ cd NewProject/
matt@ubuntu:~/Documents/NewProject$
```

After creating a new directory, open it in an IDE and open the terminal or open it on the same terminal and type in **npm init**.

```
matt@ubuntu:~/Documents/New-Project$ npm init
```

When `npm init` is used a new `package.json` file will be created containing info about the app including the dependency list for npm packages, as well as the commands used to build, run, test, etc. for the app itself.

```
{
  "name": "new-project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Afterwards, the other big component of this back-end app will be the web framework, there are many to choose from but this project will be using expressJS. **npm install express.**

```
matt@ubuntu:~/Documents/New-Project$ npm install express
```

Finally, setup a basic app.js file in the root directory of the project and include the following snippet.

app.js	<pre>const express = require('express')  const app = express()  app.get('/', function (req, res) {   res.send('Hello World') })  app.listen(8080)</pre>
--------	---

A useful package for development workflow will be nodemon, which will refresh the server anytime a file is changed making it easy to view changes without having to stop and restart the server every time. **npm install --save-dev nodemon.** --save-dev is used since it is only used for the development phase and should not be pushed to production.

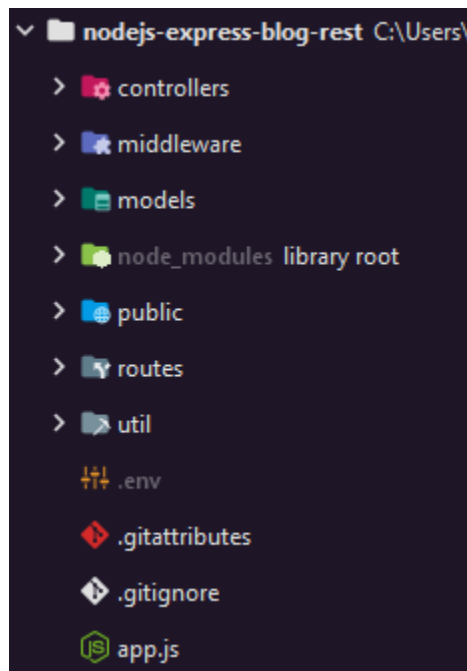
```
matt@ubuntu:~/Documents/New-Project$ npm install --save-dev nodemon
```

Adding a new start script using the nodemon package to the package.json, will allow a local server to be started.

package.json	<pre>{   "name": "new-project",   "version": "1.0.0",   "main": "index.js",   "scripts": {     "test": "echo \"Error: no test specified\" &amp;&amp; exit 1",     "start": "nodemon app.js"   },   "author": "",   "license": "ISC",   "dependencies": {     "express": "^4.18.1"   },   "devDependencies": {     "nodemon": "^2.0.16"   } }</pre>
--------------	--

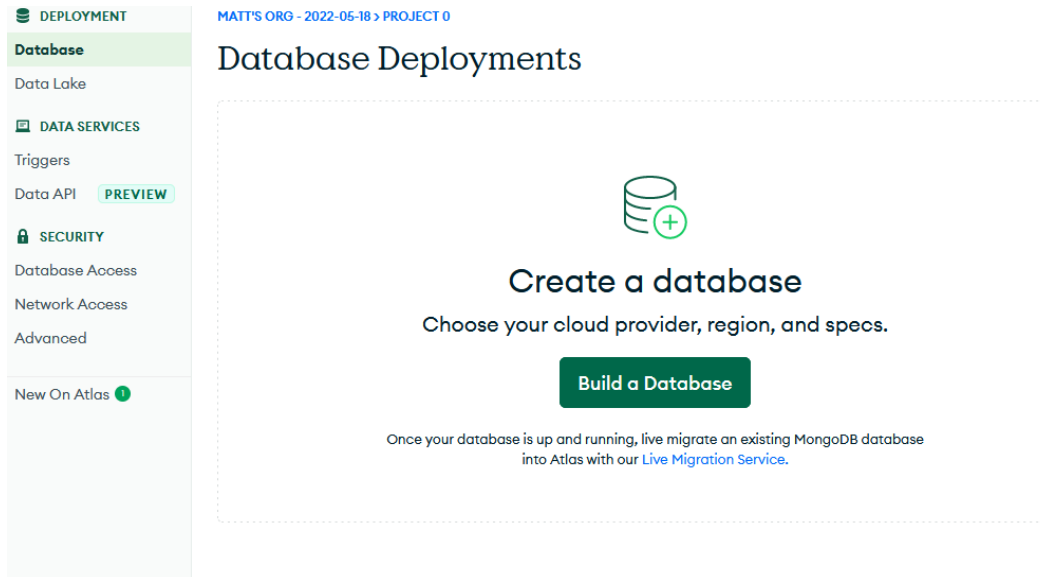
### 5.3.2 Setting up the structure of the project

For this project, the file structure will be simple, there will be a controller directory that holds the functions for each of the endpoints, the middleware directory will hold the logic to decode a JWT from a user to get that user's id, the models directory will hold all three models, node\_modules is just a folder where node will store all of the packages installed in the project (i.e. express, nodemon), public will store the image uploads for the blog posts, routes will use those methods from controllers to set up endpoints for the user to access them, util will provide some error handling. The other files are just some included files from setting up a git repo, and .env to hold secrets that are not sent to the repo.

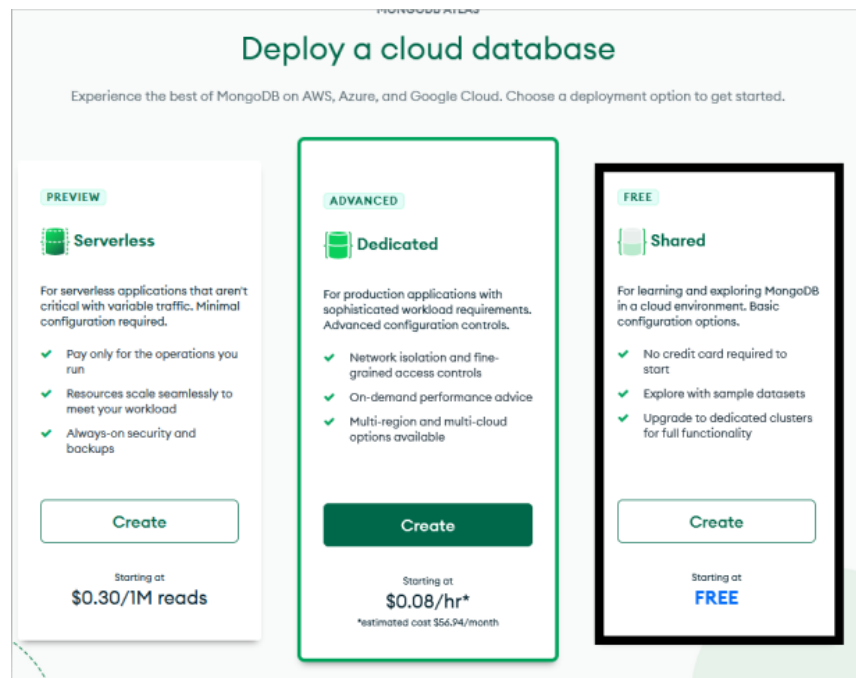


### 5.3.3 Setting up MongoDB database

The first thing to do is to visit the mongodb website and create a cloud atlas account (<https://www.mongodb.com/cloud/atlas/register>). Afterwards it should open up a portal that is similar to



Click on Build a database, and pick the Shared free tier (unless a more powerful database is needed).



Leaving all of the default settings will be fine, so long as there is no money being spent unless wanted.

Cloud Provider & Region	AWS, Oregon (us-west-2) ^
Cluster Tier	M0 Sandbox (Shared RAM, 512 MB Storage) ^ Encrypted
Additional Settings	MongoDB 5.0, No Backup ^
Cluster Name	Cluster0 ^

Finally a username and password as well as the ip of where the database will be accessed from. Remember the username and password as it will be needed to connect to the database from the app.

1 How would you like to authenticate your connection?

Your first user will have permission to read and write any data in your project.

Username and Password

Certificate

Create a database user using a username and password. Users will be given the *read and write to any database* privilege by default. You can update these permissions and/or create additional users later. Ensure these credentials are different to your MongoDB Cloud username and password.

Username

Password






2 Where would you like to connect from?

Enable access for any network(s) that need to read and write data to your cluster.

**My Local Environment**  
Use this to add network IP addresses to the IP Access List. This can be modified at any time.

ADVANCED

**Cloud Environment**  
Use this to configure network access between Atlas and your cloud or on-premise environment. Specifically, set up IP Access Lists, Network Peering, and Private Endpoints.

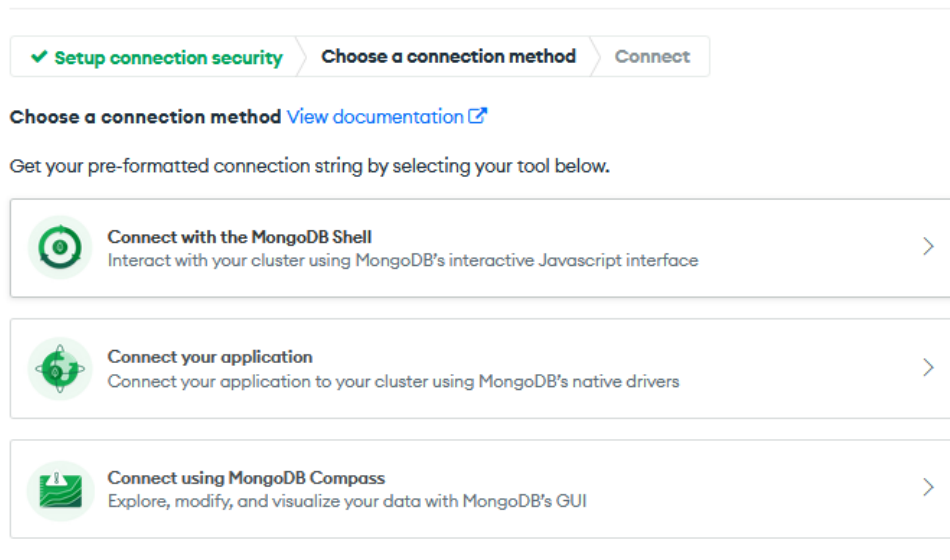
Add entries to your IP Access List

Only an IP address you add to your Access List will be able to connect to your project's clusters.

IP Address

Description

After a little bit the cluster should be available, another useful tool to use for mongodb would be the mongodb compass (<https://www.mongodb.com/products/compass>). Which does a similar thing to the pgAdmin that postgres uses, it shows the different databases and tables and different documents (rows, but mongodb is a document based database). In the mongodb cloud atlas portal, there will be an option to connect to a cluster, this will have options to connect to a shell, an application (code snippets to show how to connect), and a connect string for connecting in the compass app.



Now to setup the connection in the app itself. A new package will be used, mongoose which uses the mongodb package and also provides an ODM (ORM for documents) tool that will turn the models into documents and insert them into the database. **npm install mongoose**. Afterwards add the following changes to the app.py file.

app.py	<pre> const express = require('express') const mongoose = require('mongoose')  const MONGODB_URI = // Your mongoDB connect string provided in cluster  const app = express()  app.use(express.json()) </pre>
--------	--



```
// connects to the mongoDB db and if successful will start up server
mongoose.connect(MONGODB_URI)
  .then(result => {
    console.log('Server Active')
    app.listen(8080)
  })
  .catch(err => console.log(err))
```

If the app starts up then the connection to the mongodb was successful. If not, check the connection string for any typos and make sure the cluster is active.

### 5.3.4 Creating post model

Unlike flask and django, making a model in mongoose will use the schema class that takes an object as an argument where the fields will be defined. The fields use the key as the name of the column and the value will be the type, or a nested object can be used as the value where more parameters can be defined.

Create a new post.js file in the models directory. Currently the only fields will be the title, image, and body, but a comment and user field will be added later.

models/post.js

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema

const postSchema = new Schema({
  title: {
    type: String,
    required: true
  },
  imageUrl: {
    type: String,
    required: true
  },
  body: {
    type: String,
    required: true
  }
}, {timestamps: true})

postSchema.options.toJSON = {
  transform: function (doc, ret, options) {
    ret.id = ret._id
    delete ret._id
    delete ret.__v
```

	<pre>         delete ret.updatedAt         return ret       }     }      module.exports = mongoose.model('Post', postSchema) </pre>
--	---

Unlike the SQL databases used in the other two back-ends, mongodb has no need to create tables or migrate, as it will dynamically change the data within a document. The `toJSON` method will tell mongoose how to parse the post when it is being sent back. The last line **`module.exports`** allows this schema object to be used outside of this file using an import statement **`require()`**.

### 5.3.5 Adding routes & validating user data for post model

Express will handle the routing for this app, and it will use a router. To setup this router create a new `post-routes.js` file in the routes directory. Currently this will hold all of the endpoints for the post model, however no functionality is implemented. Express router offers the different HTTP methods as methods to use so that it will tell express the method needed to use that endpoint.

routes/post-routes.js	<pre> const express = require('express')  const router = express.Router()  router.get('/posts/:page')  router.get('/post/:postId')  router.post('/posts')  router.put('/post/:postId')  router.delete('/post/:postId')  module.exports = router </pre>
-----------------------	--

Another useful util file will be needed, this will take any errors and throw an error that express will handle. Create a new `errorHandler.js` file in the util folder.

util/errorHandler.js	<pre> exports.throwError = (err, errorCode, next) =&gt; {   const error = new Error(err)   error.statusCode = errorCode   if (next)     next(error)   else     throw error } </pre>
----------------------	---

The next step is to create the logic for the endpoints. Create a new post-controller.js file in the controllers directory.

controllers/post-controller.js  Retrieve & Retrieve Paginated list	<pre> const path = require("path") const {unlink} = require("fs")  const Post = require('../models/post') const {throwError} = require('../util/errorHandler')  const POSTS_PER_PAGE = 3  exports.getPosts = async (req, res, next) =&gt; {   try {     const page = +req.params.page    1     // . find() is a method that mongoose creates for     // its schemas (returns all documents by default)     const totalItems = await     Post.find().countDocuments()     const posts = await Post.find()     // populate() grabs the related data to the     // reference collection     // here we are grabbing the user document that     // belongs to the post     .skip((page - 1) * POSTS_PER_PAGE)     // limit 4 posts per page     .limit(POSTS_PER_PAGE)      posts.forEach((post) =&gt; {       // mapping out data I don't want front-end to       have       post.body = undefined       post.imageUrl = undefined       return post     })     res.status(200).json({       message: 'Posts Retrieved Successfully',       posts: posts,       totalPages: Math.ceil(totalItems /       POSTS_PER_PAGE)     })   } catch (err) {     return throwError(err, 500, next)   } }  exports.getPost = async (req, res, next) =&gt; {   try {     const postId = req.params.postId </pre>
--	--

	<pre> const post = await Post.findById(postId)  if (!post)   return throwError('post not found', 404, next)  res.status(200).json({   message: 'Post retrieved Successfully',   post: post }) } catch (err) {   return throwError(err, 500, next) } </pre>
--	--

The `getPosts` method will receive an additional arg through the url that will tell what page of the paginated list of posts, to avoid having to send all of the posts at once. The second method `getPost` will send a more detailed single post that is used when an individual post is wanted.

Before creating the create and update methods a new middleware will be required. Multer, which will parse a form for the app into an object it can access and grab the data from. **npm install multer**

In the middleware directory create a new `multerSetup.js` file.

middleware/ multerSetup. js	<pre> const multer = require('multer')  const storage = multer.diskStorage({   // setting file upload destination to public/images from   // project root dir   destination: function (req, file, cb) {     cb(null, 'public/images')   },   // setting up custom file names to not overwrite existing   // files with exact same names   filename: function (req, file, cb) {     const uniqueSuffix = Date.now() + '-' +       Math.round(Math.random() * 1E9)     const filename = file.originalname.replace('public',       '')     cb(null, uniqueSuffix + '-' + filename)   } })  // check if file is of type png, jpg, or jpeg const fileFilter = (req, file, cb) =&gt; {   if (file.mimetype === 'image/png'    file.mimetype ===     'image/jpg'    file.mimetype === 'image/jpeg')     cb(null, true)   else     cb(null, false) } </pre>
-----------------------------------	---

	<pre> module.exports = [   multer({storage: storage, fileFilter: fileFilter}).single('image') ] </pre>
--	--

As stated, multer will handle multipart form data which helps when a file upload is used, which is needed for the posts for this project, multer will parse text based data into a dictionary with the key words being the input names from the form, and it will create a separate dictionary for file uploads unless the single method is used which will tell multer that only one file upload will be present.

<b>controllers/post-controller.js</b>  <b>Create</b>	<pre> exports.postPost = async (req, res, next) =&gt; {   try {      // multer or json parse will put related data onto     req.body     const title = req.body.title     const body = req.body.body     // multer grabs the image and puts it onto the     req.file object     const image = req.file.path.replace("\\", "/").replace('public', '')      const post = await new Post({       title: title,       body: body,       imageUrl: image,     }).save()      res.status(201).json({       message: 'Post Created Successfully',       post: post,     })   } catch (err) {     throwError(err, 500, next)   } } </pre>
--	---

The create method will take a form as the body, will then take the data from the 3 inputs and create a post object which mongoose will then create a new document in the table. A user token will be required eventually to create a post but for now it is not.

<b>controllers/post-controller.js</b>  <b>Update</b>	<pre> exports.updatePost = async (req, res, next) =&gt; {   try {     const postId = req.params.postId      const post = await Post.findById(postId) </pre>
--	---

	<pre>         if (!post)             return throwError('post not found', 404, next)          // if multer parsed info then grab that if not then         grab post elements         const title = req.body.title    post.title         const body = req.body.body    post.body          let imageUrl = req.body.image    post.imageUrl         if (req.file) {             imageUrl = req.file.path.replace("\\", "/")         }         // remove old image if it was updated to a new image         if (imageUrl !== post.imageUrl)             clearImage(post.imageUrl)          post.title = title         post.body = body         post.imageUrl = imageUrl          await post.save()          res.status(200).json({             message: 'Post Updated Successfully',             post: post,         })     } catch (err) {         throwError(err, 500, next)     } }  const clearImage = async imagePath =&gt; {     let filePath = path.join(__dirname, '..', 'public' +     imagePath)     await unlink(filePath, err =&gt; {         if (err)             throwError(err, 500)     }) } </pre>
--	---

The update method is similar to the create in that it will accept a form as the body, with 3 possible inputs. If any of the inputs are missing then mongoose will just take the values from the current post and just set them to those. If an image is changed then the clearImage method will delete the previous image from the directory and multer will upload the new one. Eventually it will check for the author of the post to ensure only they can update it but for now anyone can.

<b>controllers/post-controller.js</b>	<pre> exports.deletePost = async (req, res, next) =&gt; {     try {         const postId = req.params.postId         const post = await Post.findById(postId)     } } </pre>
---------------------------------------	--

Delete	<pre>         if (!post)             return throwError('post not found', 404, next)          await clearImage(post.imageUrl)          await Post.deleteOne({_id: postId})          res.status(200).json({             message: 'Post Deleted Successfully',             post: post         })     } catch (err) {         return throwError(err, 500, next)     } } </pre>
--------	--

The delete method will delete a post, delete the image from the directory, and finish by returning that post's data. Eventually it will check for the author of the post to ensure only they can delete it but for now anyone can.

Now that all of the methods are created for each of the endpoints, they can be added to routes file.

routes/post-routes.js	<pre> const express = require('express')  const postsController = require('../controllers/post-controller')  const router = express.Router()  router.get('/posts/:page', postsController.getPosts)  router.get('/post/:postId', postsController.getPost)  router.post('/posts', postsController.postPost)  router.put('/post/:postId', postsController.updatePost)  router.delete('/post/:postId', postsController.deletePost)  module.exports = router </pre>
-----------------------	--

The only thing left is to add validation for the create and update methods. This can be done directly in the methods in the controller or a third party package called express-validator can be used and it can validate the data directly in the routes as a middleware that will pass the data through it before reaching the controller methods. **npm install express-validator**

routes/post-routes.js	<pre>// other code  router.post('/posts',   body('title').isLength({min: 15}).trim()     .withMessage('Title is too short, min: 15 characters'),   body('body').isLength({min: 25}).trim()     .withMessage('Body is too short, min: 25 characters'),   check('image')     .custom((value, {req}) =&gt; {       return req.file.mimetype === 'image/png'    req.file.mimetype === 'image/jpg'    req.file.mimetype === 'image/jpeg'     })     .withMessage('incorrect file type submitted (accepted: PNG, JPG, JPEG)'),   postsController.postPost)  router.put('/post/:postId',   body('title').isLength({min: 15}).trim()     .withMessage('Title is too short, min: 15 characters'),   body('body').isLength({min: 25}).trim()     .withMessage('Body is too short, min: 25 characters'),   check('image').optional({ nullable: true })     .custom((value, {req}) =&gt; {       return req.file.mimetype === 'image/png'    req.file.mimetype === 'image/jpg'    req.file.mimetype === 'image/jpeg'     })     .withMessage('incorrect file type submitted (accepted: PNG, JPG, JPEG)'),   postsController.updatePost)  // other code</pre>
-----------------------	--

Express validator has many default sanitizers and validators that will check and mutate data as need be, and then finally pass the data to the controller methods, custom ones can be created if need be but for now the default ones will work. If any of the validators return false it will return an error message or a custom one using the withMessage method, however the express validator will still pass through to the controller methods even if the data input is incorrect. So to make sure that error handling is done properly an error check needs to be added to both the create and update methods for post.

controllers/post	<pre>const {validationResult} = require('express-validator')</pre>
------------------	--



<b>-controller.js</b>	<pre> exports.postPost = async (req, res, next) =&gt; {   try {     const errors = validationResult(req);     if (!errors.isEmpty()) {       return res.status(422).json({ errors: errors.array() });     }     // . . . the rest of the update method   }    exports.updatePost = async (req, res, next) =&gt; {     try {       const errors = validationResult(req);       if (!errors.isEmpty()) {         return res.status(422).json({ errors: errors.array() });       }       // . . . the rest of the update method     }   } </pre>
-----------------------	---

With this simple little addition of code, simple validation is set up for when a user tries to create or update a post.

Now that the post routes and logic has been set up the final step is to attach the router, as well as the multer setup to the main app.

<b>app.py</b>	<pre> // . . . other imports // . . . other imports  const postRoutes = require('./routes/post-routes') const multer = require('./middleware/multerSetup')  // . . . other code // . . . other code  const app = express()  // allows express to parse json data input if needed app.use(express.json())  // able to get images from localhost:8080/images/&lt;image file&gt; without publicly exposing the directory public/images app.use('/images', express.static(path.join(__dirname, 'public/images')))  // configuring express app to use multer, and only accept a single file upload per request app.use(multer)  app.use(postRoutes)  // default error handler for express app.use((error, req, res, next) =&gt; {   console.log(error)   const status = error.statusCode    500   const message = error.message </pre>
---------------	---

```

const data = error.data
res.status(status).json({message: message, data: data})
})

// . . . mongoose connect/app start

```

### 5.3.6 Creating user model

Similar to the post model, create a new user.js file in the models directory and create a new schema object.

models/user.py

```

const mongoose = require('mongoose')
const Schema = mongoose.Schema

const userSchema = new Schema({
  username: {
    type: String,
    required: true
  },
  first_name: {
    type: String,
    required: true
  },
  last_name: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true
  },
  password: {
    type: String,
    required: true
  },
  posts: [{
    type: Schema.Types.ObjectId,
    ref: 'Post'
  }]
}, {timestamps: true})

userSchema.options.toJSON = {
  transform: function (doc, ret, options) {
    ret.id = ret._id
    ret.name = ret.first_name + ' ' + ret.last_name
    delete ret._id
    delete ret.__v
    delete ret.password
    delete ret.createdAt
    delete ret.updatedAt
    delete ret.posts
    delete ret.email
    delete ret.first_name
    delete ret.last_name
    return ret
  }
}

```

	<code>module.exports = mongoose.model('User', userSchema)</code>
--	--

As well as adding in the user relation into the post model. When creating relations it is important to specify the ref field as the same value as the specified model name in the **model()** method.

models/post.py	<pre> const mongoose = require('mongoose') const Schema = mongoose.Schema  const postSchema = new Schema({   title: {     type: String,     required: true   },   imageUrl: {     type: String,     // required: true   },   body: {     type: String,     required: true   },   author: {     type: Schema.Types.ObjectId,     ref: 'User',     required: true   } }, {timestamps: true}) </pre>
----------------	---

### 5.3.7 Adding routes & validating user data for user model

First setup a new routes file, user-routes.js in the routes directory. Also add some simple validation for the signup endpoint to check that a user with the email provided does not already exist, that the passwords match, and that the username is provided. For a few of these validations, custom validators will need to be created as they are too specific to the purpose of this project.

routes/user-routes.js	<pre> const express = require('express') const {body} = require('express-validator')  const User = require('../models/user')  const router = express.Router()  router.put('/signup',   body('email').isEmail().custom((value, { req }) =&gt; {     return User.findOne({email: value}).then(user =&gt; {       if (user) </pre>
-----------------------	---

	<pre>         return Promise.reject(`user with email: \${value} already exists`)       })     },     body('username').isLength({min: 8}).trim()       .withMessage('Username too short, min: 8 characters'),     body('password').isLength({min: 8, max: 40}).trim()       .withMessage('password length must be between 8 and 40 characters'),     body('confirmPassword').custom((value, { req }) =&gt; {       if (value !== req.body.password) {         throw new Error('passwords must match');       }       // Indicates the success of this synchronous custom       validator       return true;     })   ))    router.post('/login')    module.exports = router </pre>
--	---

Some of these validators are custom validators, what happens when a custom validator is created, express-validator will pass 2 args, one is value which will hold the value of what is being validated, and the req arg which holds the request object to the callback method. In the custom validators the first is an email that is checked if a user with the same email already exists, and the second checks if both passwords match.

Now that the endpoints are created, create a new user-controller.js file in the controllers directory that will contain the logic for actually using these endpoints. Two new packages will be needed, jsonwebtoken, which will provide all the necessary logic to create and validate JWTs for user authentication, and bcryptjs, which will handle all the logic for hashing and checking hashed passwords to log the user in. **npm install bcryptjs** and **npm install jsonwebtoken**.

controllers/user-controller.py	<pre> const bcrypt = require('bcryptjs') const jwt = require('jsonwebtoken') const {validationResult} = require("express-validator") require('dotenv').config()  const User = require("../models/user") const {throwError} = require("../util/errorHandler")  exports.signup = async (req, res, next) =&gt; {   try {     const errors = validationResult(req); </pre>
--------------------------------	--

```

    if (!errors.isEmpty()) {
      return res.status(422).json({ errors:
errors.array() });
    }

    const username = req.body.username
    const email = req.body.email
    const first_name = req.body.first_name
    const last_name = req.body.last_name
    const password = req.body.password

    // hashing password with 12 rounds of salting
    const hashedPass = await bcrypt.hash(password, 12)
    const user = await new User({
      username: username,
      first_name: first_name,
      last_name: last_name,
      email: email,
      password: hashedPass
    }).save()
    res.status(201).json({
      message: 'user has been signed up',
      user: user.toJSON(),
    })
  } catch (err) {
    return throwError(err, 500, next)
  }
}

exports.login = async (req, res, next) => {
  try {
    // since username is not unique we don't want to
    check for username as we will get more than one
    const email = req.body.email
    const password = req.body.password

    const user = await User.findOne({email: email})
    if (!user)
      return throwError(`user with email: ${email} not
found`, 404, next)

    const passwordMatch = await bcrypt.compare(password,
user.password)
    if (!passwordMatch)
      return throwError('incorrect password', 401,
next)

    // https://jwt.io/ can visually see decoded tokens
    here by copy-pasting the response token you get
    const token = jwt.sign({
      user: {id: user._id, username: user.username}
    }, 'your unique string', { expiresIn: '1h' })

    res.status(201).json({
      message: 'user has been logged in',
      token: token,
      userId: user._id,
      username: user.username
    })
  } catch (err) {
    return throwError(err, 500, next)
  }
}

```

Now add those two new methods to the respective endpoints within the routes file and connect that router to the main app file.

<b>routes/user-routes.js</b>	<pre> const express = require('express') const {body} = require('express-validator')  const userController = require('../controllers/user-controller') const User = require('../models/user')  const router = express.Router()  router.put('/signup',   body('email').isEmail().custom((value, { req }) =&gt; {     return User.findOne({email: value}).then(user =&gt; {       if (user)         return Promise.reject(`user with email: \${value} already exists`)     })   }),   body('username').isLength({min: 8}).trim()     .withMessage('Username too short, min: 8 characters'),   body('password').isLength({min: 8, max: 40}).trim()     .withMessage('password length must be between 8 and 40 characters'),   body('confirmPassword').custom((value, { req }) =&gt; {     if (value !== req.body.password) {       throw new Error('passwords must match');     }   })   // Indicates the success of this synchronous custom validator   return true; }), userController.signup)  router.post('/login', userController.login)  module.exports = router </pre>
<b>app.py</b>	<pre> // . . . other imports const userRoutes = require('./routes/user-routes')  // . . . other code  app.use(userRoutes)  // . . . other code </pre>

### 5.3.8 Adding user authentication with JSON web tokens

Since the login endpoint provides the jwt to the user when they login, the only thing to do is to now accept that jwt when a request is sent and decode it to get the user credentials for specific endpoints.

To accomplish this, an authenticate.js middleware will be created that will be passed through the routes similar to how the validation was. This method will look at the authorization header and try to decode the token from the header and then set a new request key value pair that holds the user id.

middleware/ authenticate. js	<pre>const jwt = require('jsonwebtoken') require('dotenv').config()  const {throwError} = require('../util/errorHandler')  module.exports = (req, res, next) =&gt; {   const authHeader = req.get('Authorization')   if (!authHeader) {     throwError('not authenticated', 401)   }   try {     const decodedToken = jwt.verify(authHeader, 'your unique string')     req.userId = decodedToken.user.id   } catch (err) {     throwError(err, 500, next)   }   next() }</pre>
------------------------------------	--

Now that the authenticate logic is set up, it just needs to be attached to the necessary routes in the post-routes.js file where user authentication is wanted.

<b>routes/post-routes.js</b>	<pre> const express = require('express')  const postsController =   require('../controllers/post-controller') const authenticate = require('../middleware/authenticate') const {body, check} = require('express-validator')  const router = express.Router()  router.post('/posts',   body('title').isLength({min: 15}).trim()     .withMessage('Title is too short, min: 15 characters'),   body('body').isLength({min: 25}).trim()     .withMessage('Body is too short, min: 25 characters'),   check('image')     .custom((value, {req}) =&gt; {       return req.file.mimetype === 'image/png'    req.file.mimetype === 'image/jpg'    req.file.mimetype === 'image/jpeg'     })     .withMessage('incorrect file type submitted (accepted: PNG, JPG, JPEG)'),   authenticate, postsController.postPost)  router.put('/post/:postId',   body('title').isLength({min: 15}).trim()     .withMessage('Title is too short, min: 15 characters'),   body('body').isLength({min: 25}).trim()     .withMessage('Body is too short, min: 25 characters'),   check('image').optional({ nullable: true })     .custom((value, {req}) =&gt; {       return req.file.mimetype === 'image/png'    req.file.mimetype === 'image/jpg'    req.file.mimetype === 'image/jpeg'     })     .withMessage('incorrect file type submitted (accepted: PNG, JPG, JPEG)'),   authenticate, postsController.updatePost)  router.delete('/post/:postId', authenticate, postsController.deletePost)  router.post('/post/comments/:postId',   body('comment').isLength({min: 15}).trim()     .withMessage('Comment is too short, min: 15 characters'),   authenticate, postsController.postComment)  module.exports = router </pre>
------------------------------	--



### 5.3.9 Adding user functionality to post model

Now that the routes have access to the user, the user can now be linked to posts, for when a post is created to link a user to that post, as well when user verification for the post author is needed to alter the post, and for retrieving posts to tell who the author is.

<p><b>controllers/post-controller.js</b></p> <p><b>Retrieve &amp; retrieve list</b></p>	<pre> exports.getPosts = async (req, res, next) =&gt; {   try {     const page = +req.params.page    1     // . find() is a method that mongoose creates for     // its schemas (returns all documents by default)     const totalItems = await     Post.find().countDocuments()     const posts = await Post.find()     // populate() grabs the related data to the     // reference collection     // here we are grabbing the user document that     // belongs to the post     .skip((page - 1) * POSTS_PER_PAGE)     // limit 3 posts per page     .limit(POSTS_PER_PAGE)     .populate('author')      // . . .other code   } catch (err) {     return throwError(err, 500, next)   } }  exports.getPost = async (req, res, next) =&gt; {   try {     const postId = req.params.postId      const post = await Post.findById(postId)     .populate('author')      // . . .other code   } catch (err) {     return throwError(err, 500, next)   } } </pre>
---	--

Using the `populate()` method, mongoose will go and retrieve the data related to the `objectId` that mongodb creates in its place. If a user with id 1 who has the username `johnSmith` creates a post, the `author` field in the post will hold id of 1, and when the `populate` method is used, mongoose will retrieve that user and replace the `author` field with a nested object of that user.

<b>controllers/post-controller.js</b>  <b>Create</b>	<pre> exports.postPost = async (req, res, next) =&gt; {   try {     const errors = validationResult(req);     if (!errors.isEmpty()) {       return res.status(422).json({ errors: errors.array() });     }      // multer or json parse will put related data onto req.body     const title = req.body.title     const body = req.body.body     // multer grabs the image and puts it onto the req.file object     const image = req.file.path.replace("\\", "/").replace('public', '')      const userId = req.userId     const author = await User.findById(userId)      if (!author)       return throwError('user not found', 404, next)      const post = await new Post({       title: title,       body: body,       imageUrl: image,       author: author._id,     }).save()      author.posts.push()     await author.save()      res.status(201).json({       message: 'Post Created Successfully',       post: post,     })   } catch (err) {     throwError(err, 500, next)   } } </pre>
--	---

The create method will now check for a user, and if a user with the same id as the one in the token is found it will link that user and post together.

<b>controllers/post-controller.js</b>  <b>Update</b>	<pre> exports.updatePost = async (req, res, next) =&gt; {   try {     const errors = validationResult(req);     if (!errors.isEmpty()) {       return res.status(422).json({ errors: errors.array() });     }      const userId = req.userId </pre>
--	---

	<pre> const author = await User.findById(userId)  if (!author)   return throwError('user not found', 404, next)  const postId = req.params.postId const post = await Post.findById(postId)  if (!post)   return throwError('post not found', 404, next) if (post.author.toString() !== userId)   return throwError('user is not author of post', 401, next)  // if multer parsed info then grab that if not then grab post elements const title = req.body.title    post.title const body = req.body.body    post.body  let imageUrl = req.body.image    post.imageUrl if (req.file) {   imageUrl = req.file.path.replace("\\", "/").replace('public', '') } // remove old image if it was updated to a new image if (imageUrl !== post.imageUrl)   clearImage(post.imageUrl, next)  post.title = title post.body = body post.imageUrl = imageUrl  await post.save()  res.status(200).json({   message: 'Post Updated Successfully',   post: post, }) } catch (err) {   throwError(err, 500, next) } </pre>
--	---

The update method will now check for a user, if that user is found it will then check that the user is the same as the author of the post, and then update the post.

<b>controllers/post-controller.js</b>  <b>Delete</b>	<pre> exports.deletePost = async (req, res, next) =&gt; {   try {     const userId = req.userId     const author = await User.findById(userId)      if (!author)       return throwError('user not found', 404, next) </pre>
--	--

	<pre> const postId = req.params.postId const post = await Post.findById(postId)  if (!post)   return throwError('post not found', 404, next) if (post.author.toString() !== userId)   return throwError('user is not author of post', 401, next)  await clearImage(post.imageUrl, next)  await Post.deleteOne({_id: postId})  res.status(200).json({   message: 'Post Deleted Successfully',   post: post }) } catch (err) {   return throwError(err, 500, next) } } </pre>
--	---

The delete method like the update one, will now check for a user, if that user is found it will then check that the user is the same as the author of the post, and then delete the post.

### 5.3.10 Creating comment model

Similar to the other two models, create a new comment.js file in the models directory. The comment model will be simple, it will just hold a string which will be the comment itself, as well as two foreign keys one for the post it is commented on, and one for the user who made the comment, as well as timestamps of when the comment was made.

modes/comment.js	<pre> const mongoose = require('mongoose') const Schema = mongoose.Schema  const commentSchema = new Schema({   comment: {     type: String,     required: true   },   post: {     type: Schema.Types.ObjectId,     ref: 'Post',     required: true   },   author: {     type: Schema.Types.ObjectId,     ref: 'User', </pre>
------------------	---

	<pre>         required: true       }     }, {timestamps: true})      commentSchema.options.toJSON = {       transform: function (doc, ret, options) {         ret.id = ret._id         delete ret.post         delete ret.updatedAt         delete ret._id         delete ret.__v         return ret       }     }     module.exports = mongoose.model('Comment',     commentSchema) </pre>
--	---

Now that a comment model exists go back to the post model and add a comments field that will be a list of references to the comment model.

modes/post.js	<pre> const mongoose = require('mongoose') const Schema = mongoose.Schema  const postSchema = new Schema({   title: {     type: String,     required: true   },   imageUrl: {     type: String,     required: true   },   body: {     type: String,     required: true   },   author: {     type: Schema.Types.ObjectId,     ref: 'User',     required: true   },   comments: [{     type: Schema.Types.ObjectId,     ref: 'Comment'   }] }, {timestamps: true}) </pre>
---------------	---

### 5.3.11 Adding routes & validating user data for comment model

The comment will only use one endpoint, which is used for creating a comment, it will take a postid as an extra url arg, and require that the user is logged in and passes a token through the auth header. Since the comments

are closely linked to the post, the endpoint routing and logic will be contained in the post controller and router files.

routes/post-routes.js	<pre>// . . . other code router.post('/post/comments/:postId',   body('comment').isLength({min: 15}).trim()     .withMessage('Comment is too short, min: 15 characters'),   authenticate, postsController.postComment) // . . . module export</pre>
-----------------------	---

controllers/post-controller.js	<pre>exports.postComment = async (req, res, next) =&gt; {   try {     const errors = validationResult(req);     if (!errors.isEmpty()) {       return res.status(422).json({ errors: errors.array() });     }      const postId = req.params.postId     const userId = req.userId     const commentBody = req.body.comment     const post = await Post.findById(postId)      if (!post)       return throwError('post not found', 404, next)      const comment = await new Comment({       comment: commentBody,       post: postId,       author: userId     }).save()      post.comments.push(comment)     await post.save()      res.status(200).json({       message: 'Comment Created Successfully',       comment: comment,       postId: postId     })   } catch (err) {     throwError(err, 500, next)   } }</pre>
--------------------------------	---

### 5.3.12 Adding comment functionality to post model

The final step is to now add logic to retrieve the comments when a post is being retrieved. This can simply be done with the populate method, but it requires an object arg, as the author of comment also has to be populated.

controllers/post-controller.js  retrieve	<pre> exports.getPost = async (req, res, next) =&gt; {   try {     const postId = req.params.postId      const post = await Post.findById(postId)       .populate('author')       .populate({         path: 'comments',         populate: {           path: 'author'         }       })      if (!post)       return throwError('post not found', 404, next)      res.status(200).json({       message: 'Post retrieved Successfully',       post: post     })   } catch (err) {     return throwError(err, 500, next)   } } </pre>
--	---

## 5.4 Github Repos for code walkthroughs

Now that all three back-ends are completely built in case there is an error in any of the steps or if you would rather look at the whole code. All three complete back-ends can be found on my github.

- Django: (<https://github.com/HuloM/Django-Blog>)
- Flask: (<https://github.com/HuloM/flask-blog-rest>)
- NodeJS/Express: (<https://github.com/HuloM/nodejs-express-blog-rest>)

## 6. Front-End Setup

The front-end will be what the user interacts with, which typically consists of html, css, and pure javascript (no frameworks or dependencies). However, there are now many frameworks that each have their own approach, that make the process of front-end development easy and produce a good user experience.

For this app, there will only be one front-end across all 3 tech stacks, as it will reduce the complexity and time requirement. For this project, the front-end will be using **react** library. This was chosen as react is one of if not the most popular which means there are tons of jobs that want to use react and also resources to learn or talk about react. However, that does not mean this is the only and best solution, there are many others, including **svelte** (<https://svelte.dev/>), which is a very new and popular rising framework, similar to react, however instead of using the virtual DOM, it just updates the real DOM whenever there are state changes, **angular** (<https://angular.io/>), another popular framework that is kind of an all-in-one framework, and many others.

Other front-end related tools that are being used is **TailwindCSS** (<https://tailwindcss.com/>), which is a CSS framework that provides many pre-made css classes so that they don't have to be made from scratch. This however is not a tutorial on CSS so while the styles will be in the code there won't be a lot of explanation for it, if you are interested in looking at tailwindCSS documentation the link is <https://tailwindcss.com/docs/installation>.

### Front-end





## 6.1 Code Walkthrough (React)

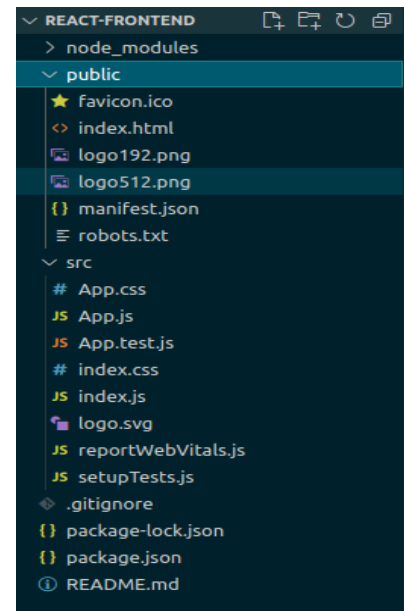
### 6.1.1 Creating base app

While it is ok to create an app from scratch, the react community has created a template that can be used by anyone, using either **npx create-react-app <app name>** or **npm init react-app <app name>**, a simple react app will be generated.

```
matt@ubuntu:~$ npm init react-app react-frontend  
Creating a new React app in /home/matt/react-frontend.
```

What should be present in the template app:

Running **npm start** from within the root directory of the project will start a local server that can be visited on localhost:3000.



If Tailwind is being used it will also need to be installed through npm, **npm install tailwindcss postcss autoprefixer**. Also, **npx tailwindcss init**, which will create a config file for tailwind. The full step by step can be found here -> <https://tailwindcss.com/docs/guides/create-react-app>.

After the init command is run, edit the `tailwind.config.js` file.

<code>tailwind.config.js</code>	<pre>module.exports = {   content: [     './src/**/*..{js,jsx,ts,tsx}',   ],   theme: {     extend: {},   },   plugins: [], }</pre>
---------------------------------	---

Include the following in the `index.css` file to be able to use tailwind.

<code>index.css</code>	<pre>@tailwind base; @tailwind components; @tailwind utilities;  @layer components {   .card {     background-color: theme('colors.white');     color: theme('colors.white');     border-radius: theme('borderRadius.lg');     padding: theme('spacing.6');     box-shadow: theme('boxShadow.xl');   } }</pre>
------------------------	--

### 6.1.2 Removing template code

Currently the `app.js` and `index.js` files (which are the two main files of a react app) have some preloaded things that are not wanted. Adjust them to the following.

<code>src/app.js</code>	<pre>import './App.css'  function App() {   return (&lt;div&gt;&lt;/div&gt;) }  export default App</pre>
<code>src/index.js</code>	<pre>import React from 'react' import ReactDOM from 'react-dom/client' import './index.css' import App from './App'  const root = ReactDOM.createRoot(document.getElementById('root')) root.render(&lt;App /&gt;)</pre>

The App.js file will contain most of the components of the page, including logic to change and update the state. While the index.js will hook onto the index.html (which is what is the html file that is served to the user), and attach the App component to the root div.

### 6.1.3 Auth context

In react apps, “data is passed top-down (parent to child) via props”, passing props through components allows for data to be passed through the app. However, this can be a bit annoying to deal with passing data back and forth through apps especially if two components with different parents need to interact with one another. React's solution to this is Contexts, which provides data to the entire app through the provider that will wrap around the app component in the index.js file. React recommends to not use context when data will be changed frequently as it was not made to handle frequent updates. An alternative to this would be Redux (<https://react-redux.js.org/>) which is built for frequent updates to the state.

Before creating the provider, the first thing that needs to be created is the context. The context will act as a sort of interface where only an object is defined with keys, but the values will be default values that will later be overwritten with the provider. Start by creating a new context directory in the src directory.

```
matt@ubuntu:~/Documents/react-frontend/src$ mkdir context
matt@ubuntu:~/Documents/react-frontend/src$ ls
App.css  App.js  App.test.js  components  context  index.css  index.js
```

Within the context directory, create a new AuthContext directory

```
matt@ubuntu:~/Documents/react-frontend/src/context$ mkdir AuthContext
matt@ubuntu:~/Documents/react-frontend/src/context$ ls
AuthContext
```

Finally create a new auth-context.js file.

src/context/ AuthContext/ auth-context.js	<pre> import React from 'react'  const AuthContext = React.createContext({   token: '',   username: '',   userId: '',   isLoggedIn: false,   authError: '',   ClickAuthModalError: () =&gt; {},   UserSignupHandler: (authdata) =&gt; {},   UserLoginHandler: (authData) =&gt; {},   LogoutUserHandler: () =&gt; {} })  export default AuthContext </pre>
---	---

Now with the context, rather than using the fetch api, a new package will be used to call the back-end, axios, **npm install axios**. The next thing to do is to create a provider with all of the logic for each of the context values.

src/context/ AuthContext / AuthContext Provider.js	<pre> import React, {useState, useEffect, useCallback} from 'react'  import AuthContext from './auth-context'  const axios = require('axios')  const AuthProvider = props =&gt; {   // state variables that hold user related data if they are   // logged in.    const [isLoggedIn, setIsLoggedIn] = useState(false)   const [token, setToken] = useState('')   const [username, setUsername] = useState('')   const [userId, setUserId] = useState('')   const [authError, setAuthError] = useState('')    const ENDPOINT_URL = '/api'    const UserSignupHandler = async authdata =&gt; {     setAuthError('')      // creates a formData with all of the form inputs that     // can be passed to the endpoint     const formData = new FormData()     formData.append('username', authdata.username)     formData.append('first_name', authdata.first_name)     formData.append('last_name', authdata.last_name)     formData.append('email', authdata.email)     formData.append('password', authdata.password)     formData.append('confirmPassword',     authdata.confirmPassword)     // sends a PUT request to the signup endpoint that will     // use the form data to create a new user </pre>
--	---

```

    const response = await
    axios.put(`${ENDPOINT_URL}/signup`, formData, {
      headers: {
        'Content-Type': 'multipart/form-data'
      }
    })
    const data = await response.data
    if (response.status !== 201) {
      if (response.status !== 422)
      {
        setAuthError('Error signing up user, please try
again later')
        return
      }
      setAuthError(data.message)
    }
  }

  const UserLoginHandler = async authData => {
    setAuthError('')

    const formData = new FormData()
    formData.append('email', authData.email)
    formData.append('password', authData.password)

    const response = await
    axios.post(`${ENDPOINT_URL}/login`, formData, {
      headers: {
        'Content-Type': 'multipart/form-data'
      }
    })
    const data = await response.data
    if (response.status !== 201) {
      if (response.status !== 422 && response.status !==
401)
      {
        setAuthError('Error logging user in, please try
again later')
        return
      }
      setAuthError(data.message)
      return
    }

    // sets cached data so the user can retain their login
    credentials after exiting the website
    const milliseconds = 60 * 60 * 1000
    localStorage.setItem('token', `${data.token}`)
    // cached data will no longer be available after 1hr
    since last login or re-entry
    const expDate = new Date(new Date().getTime() +
milliseconds)
    localStorage.setItem('expiryDate',
expDate.toISOString())

    setToken(`${data.token}`)
    setUsername(data.username)
    setUserId(data.userId)

    localStorage.setItem('username', data.username)
    localStorage.setItem('userId', data.userId)
  }

```

```

        AutoLogoutUserHandler(milliseconds)

        setIsLoggedIn(true)
    }

    const AutoLogoutUserHandler = useCallback((milliseconds) =>
    {
        setTimeout(() => {
            LogoutUserHandler()
        }, milliseconds)
    }, [])

    const LogoutUserHandler = () => {
        // when user logs out, remove all cached data and reset
        states to default
        setIsLoggedIn(false)
        setToken('')
        setUsername('')
        setUserId('')
        localStorage.removeItem('token')
        localStorage.removeItem('username')
        localStorage.removeItem('userId')
        localStorage.removeItem('expiryDate')
    }

    const ClickAuthModalError = () => {
        setAuthError('')
    }

    useEffect(() => {
        // when first initialized, look for cached data to log
        user back in
        const cookieToken = localStorage.getItem('token')
        const expiryDate = localStorage.getItem('expiryDate')
        if (!cookieToken || !expiryDate) {
            return
        }
        if (new Date(expiryDate) <= new Date()) {
            LogoutUserHandler()
            return
        }
        const cookieUsername = localStorage.getItem('username')
        const cookieUserId = localStorage.getItem('userId')
        setToken(cookieToken)
        setUsername(cookieUsername)
        setUserId(cookieUserId)
        const milliseconds = 60 * 60 * 1000

        const expDate =
            new Date(new Date().getTime() +
milliseconds).toISOString()
        localStorage.setItem('expiryDate', expDate)

        setIsLoggedIn(true)
        AutoLogoutUserHandler(milliseconds)
    }, [AutoLogoutUserHandler])

    // pass auth context values to AuthContext provider
    const authContext = {
        token,
        username,
        userId,
    }

```

	<pre>         isLoggedIn,         authError,         ClickAuthModalError,         UserSignupHandler,         UserLoginHandler,         LogoutUserHandler,       ]     }      return (       &lt;AuthContext.Provider value={authContext}&gt;         {props.children}       &lt;/AuthContext.Provider&gt;     )   } }  export default AuthProvider </pre>
--	---

The final thing to do is expose the auth context provider to the app by wrapping the app component in the index.js file.

src/index.js	<pre> import ReactDOM from 'react-dom/client' import './index.css' import App from './App' import AuthContextProvider from './context/AuthContext/AuthContextProvider'  const root = ReactDOM.createRoot(document.getElementById('root')) root.render(   &lt;AuthContextProvider&gt;     &lt;App/&gt;   &lt;/AuthContextProvider&gt; ) </pre>
--------------	---

### 6.1.4 Post context

Similar to the auth context, another context that is going to be used is a post context. This context is made to keep all of the post request logic in one place as well as providing access to the post data across the entire app.

First create a new PostContext directory in the context directory.

```

natt@ubuntu:~/Documents/react-frontend/src/context$ mkdir PostContext
natt@ubuntu:~/Documents/react-frontend/src/context$ ls
AuthContext  PostContext

```

Afterwards create a new post-context.js file inside the new directory.

src/context/ PostContext/ post-context.js	<pre> import React from 'react'  const PostContext = React.createContext({   page: 0,   totalPages: 0,   posts: [],   individualPost: {},   postError: '',   onCreatePostHandler: (postData, token) =&gt; {},   onCreateCommentHandler: (postId, commentData, token) =&gt; {},   onRetrievePostHandler: (postId) =&gt; {},   onRetrievePostsHandler: (page) =&gt; {},   onUpdatePostHandler: (postId, postData, token) =&gt; {},   onDeletePostHandler: (postId, token) =&gt; {}, })  export default PostContext </pre>
---	---

Then create a PostContextProvider.js file similar to how the auth-context and AuthContextProvider.js files are.

src/context/ PostContext / PostContext Provider.js	<pre> import React, {useState} from 'react'  import PostContext from './post-context'  const axios = require('axios')  const PostProvider = props =&gt; {   // state variables that hold post related data    const [page, setPage] = useState(1)   const [totalPages, setTotalPages] = useState(0)   const [posts, setPosts] = useState([])   const [individualPost, setIndividualPost] = useState({     title: '',     imageUrl: '',     body: '',     author: {       username: '',       id: ''     },     createdAt: '',     comments: []   })   const [postError, setPostError] = useState('')    const ENDPOINT_URL = '/api'    // sends a POST request to the posts endpoint   // that will use the form data to create a new post   const onCreatePostHandler = async (postData, token) =&gt; {     const formData = new FormData()     formData.append('title', postData.titleInput)     formData.append('body', postData.bodyInput) </pre>
--	---



```

        formData.append('image', postData.image)

        const response = await
    axios.post(`${ENDPOINT_URL}/posts/`, formData, {
        headers: {
            Authorization: token,
            'Content-Type': 'multipart/form-data'
        }
    })
    const data = response.data
    if (response.status !== 200) {
        setPostError(data.message)
        return
    }
    await onRetrievePostsHandler(page)
}

// sends a GET request to the post endpoint
// that will retrieve a single post
const onRetrievePostHandler = async postId => {
    const response = await
    axios.get(`${ENDPOINT_URL}/post/${postId}`)

    const data = response.data

    if (response.status !== 200) {
        setPostError(data.message)
        return
    }
    setIndividualPost(data.post)
}

// sends a GET request to the posts endpoint
// that will retrieve a paginated list of posts
const onRetrievePostsHandler = async (pageNum) => {
    setPage(pageNum)

    const response = await
    axios.get(`${ENDPOINT_URL}/posts/${pageNum}`)

    const data = response.data

    setPosts(data.posts)
    setTotalPages(data.totalPages)
}

// sends a PUT request to the post endpoint
// that will update a post
const onUpdatePostHandler = async (postId, postData, token)
=> {
    const formData = new FormData()
    formData.append('title', postData.titleInput)
    formData.append('body', postData.bodyInput)
    formData.append('image', postData.image)
    const response = await
    axios.put(`${ENDPOINT_URL}/post/${postId}`, formData, {
        headers: {
            Authorization: token,
            'Content-Type': 'multipart/form-data'
        }
    })
    const data = await response.data

```

```

        if (response.status !== 200) {
            setPostError(data.message)
            return
        }
        setIndividualPost(data.post)
    }

    // sends a DELETE request to the post endpoint
    // that will delete a post
    const onDeletePostHandler = async (postId, token) => {
        const response = await
        axios.delete(`${ENDPOINT_URL}/post/${postId}`, {
            headers: {
                Authorization: token,
            }
        })
        const data = await response.data
        if (response.status !== 200) {
            setPostError(data.message)
            return
        }
        await onRetrievePostsHandler(page)
    }

    // sends a PUT request to the post/comments endpoint
    // that will create a comment on the post passed in
    const onCreateCommentHandler = async (postId, commentData,
    token) => {
        const formData = new FormData()
        formData.append('comment', commentData)

        const response = await
        axios.post(`${ENDPOINT_URL}/post/comments/${postId}`,
        formData, {
            headers: {
                Authorization: token,
                'Content-Type': 'multipart/form-data'
            }
        })
        const data = await response.data
        if (response.status !== 200) {
            setPostError(data.message)
        }
        await onRetrievePostHandler(postId)
    }

    // pass post context values to PostContext provider
    const postContext = {
        page,
        totalPages,
        posts,
        individualPost,
        postError,
        onCreatePostHandler,
        onCreateCommentHandler,
        onRetrievePostHandler,
        onRetrievePostsHandler,
        onUpdatePostHandler,
        onDeletePostHandler,
    }

    return (
        <PostContext.Provider value={postContext}>

```

	<pre>         {props.children}       &lt;/PostContext.Provider&gt;     )   }    export default PostProvider </pre>
--	--

Finally wrap the provider around the app component in the index.js file.

src/index.js	<pre> import ReactDOM from 'react-dom/client' import './index.css' import App from './App' import AuthContextProvider from './context/AuthContext/AuthContextProvider' import PostContextProvider from './context/PostContext/PostContextProvider'  const root = ReactDOM.createRoot(document.getElementById('root')) root.render(   &lt;AuthContextProvider&gt;     &lt;PostContextProvider&gt;       &lt;App/&gt;     &lt;/PostContextProvider&gt;   &lt;/AuthContextProvider&gt; ) </pre>
--------------	--

### 6.1.5 Creating the header & nav

To make navigation easy for this app there will be a header that will have the user's username. As well as a post button, that will pull up the main posts page. A signup button which will be available when no user is logged in on the client, when pressed will bring up a form for the user to sign up a new account. A login button which will be available when no user is logged in on the client, when pressed will bring up a form for the user to login with their account to access more of the app (creating posts and comments). Lastly, a logout button that will be available to the user once they have logged in, that will remove any cached data on the client and will set the users log in state to false.

To create this nav component, first create a new components directory inside the src directory.

```

matt@ubuntu:~/Documents/react-frontend/src$ mkdir components
matt@ubuntu:~/Documents/react-frontend/src$ ls
App.css  App.js  App.test.js  components  index.css  index.js

```

Within that components directory create a new UI directory.

```
matt@ubuntu:~/Documents/react-frontend/src/components$ mkdir UI
matt@ubuntu:~/Documents/react-frontend/src/components$ ls
UI
```

Within the UI directory create a new Header directory and a Header.js file.

```
matt@ubuntu:~/Documents/react-frontend/src/components/UI$ mkdir Header
matt@ubuntu:~/Documents/react-frontend/src/components/UI$ ls
Header
```

src/ components/ UI/Header/ Header.js	<pre>import {Disclosure} from '@headlessui/react' import {useContext, useState} from 'react' import authContext from '../../context/AuthContext/auth-context'  function classNames(...classes) {   return classes.filter(Boolean).join(' ') }  const Header = props =&gt; {   const ctx = useContext(authContext)   const [navItems, setNavItems] = useState([     {name: 'Posts', href: '#', current: true, requireLogin: null, onclick: props.onPostButtonClick},     {name: 'Sign up', href: '#', current: false, requireLogin: false, onclick: props.onSigninButtonClick},     {name: 'Login', href: '#', current: false, requireLogin: false, onclick: props.onLoginButtonClick},     {name: 'Logout', href: '#', current: false, requireLogin: true, onclick: ctx.LogoutUserHandler},   ])    const handleButtonClick = event =&gt; {     const buttonClicked = event.target.text     setNavItems(prevState =&gt; {       for (const navItem of prevState) {         if (navItem.name === buttonClicked) {           navItem.current = true           navItem.onclick()         }         else           navItem.current = false       }       return [...prevState]     })   }    return (     &lt;Disclosure as="nav" className="bg-gray-800"&gt;       &lt;div className="max-w-7xl mx-auto px-2 sm:px-6 lg:px-8"&gt;</pre>
--	--

	<pre>         &lt;div className="relative flex items-center justify-between h-16"&gt;           &lt;h2 className='text-white text-2xl'&gt;{ctx.isLoggedIn &amp;&amp; ctx.username !== '' &amp;&amp; ctx.username}&lt;/h2&gt;           &lt;div className="absolute right-0 space-x-4"&gt;             {navItems.map((item) =&gt; (               (props.isLoggedIn === item.requireLogin    item.requireLogin === null) &amp;&amp;               &lt;a key={item.name} href={item.href} onClick={handleButtonClick}                 className={classNames(                   item.current ? 'bg-gray-900 text-white' :                   'text-gray-300 hover:bg-gray-700 hover:text-white',                   'px-3 py-2 rounded-md text-sm font-medium')}                 aria-current={item.current ? 'page' : undefined}&gt;                   {item.name}                 &lt;/a&gt;               )             )           &lt;/div&gt;         &lt;/div&gt;       &lt;/Disclosure&gt;     )   }    export default Header </pre>
--	---

The header will use the auth context to check if there is currently a user logged in to update what buttons it should display on the nav bar. The signup and login buttons will have no use to someone while they are logged in, and while the logout button will have no use to someone not logged in but the opposite should be available (login/signup when no user logged in, logout when user logged in). For a few buttons, the on click methods will be passed through props from the parent app.js component

Now that the header component is created, add it to the main App.js file.

src/app.js	<pre> import './App.css' import Header from './components/UI/Header/Header' import {useState, useContext} from 'react' </pre>
------------	---

```
import authContext from '../context/AuthContext/auth-context'

function App() {
  const ctx = useContext(authContext)

  const OpenLoginFormHandler = () => console.log('login form')

  const OpenSigninFormHandler = () => console.log('signup form')

  const OpenPostFormHandler = () => console.log('post form')

  return (
    <div className='bg-gray-600 h-screen'>
      <Header onLoginButtonClick={OpenLoginFormHandler}
onSigninButtonClick={OpenSigninFormHandler}
onPostButtonClick={OpenPostFormHandler}
isLoggedIn={ctx.isLoggedIn}/>
    </div>
  )
}

export default App
```

Running **npm start**, should now bring up a local live server of react where the header will now be included.

### 6.1.6 Login & signup forms

The next step is to set up forms for the user to signup and login. With these forms, as they require user inputs, it should be validated on the client side to provide the user with some instant feedback on if their input fits with what the back-end is expecting (the back-end should also validate the data).

Providing this feedback can be a bit cumbersome to write for each input, especially for the signup that has 6 different inputs. To deal with this, a new custom hook will be created. Hooks in react, can include state or effect which have been previously looked at. However, custom hooks can be created to extract logic into reusable functions.

To create the custom hook, create a new hooks directory in the src directory, and create a use-input.js file inside the new directory.

src/hooks/

```
import {useState} from 'react'
```

use-input.js	<pre> const useInput = (validator) =&gt; {   const [enteredInput, setEnteredInput] = useState('')   const [enteredInputTouched, setEnteredInputTouched] =     useState(false)    const enteredInputIsValid = validator(enteredInput)   const inputIsValid = !enteredInputIsValid &amp;&amp;     enteredInputTouched    const inputChangeHandler = event =&gt; {     setEnteredInput(event.target.value)   }    const inputBlurHandler = () =&gt; {     setEnteredInputTouched(true)   }    const resetOnFormSubmitHandler = () =&gt; {     setEnteredInput('')     setEnteredInputTouched(false)   }    return {     enteredInput,     enteredInputIsValid,     inputIsValid,     inputChangeHandler,     inputBlurHandler,     resetOnFormSubmitHandler   } }  export default useInput </pre>
--------------	--

This input hook was something learned from a udeemy course from [Maximilian Schwarzmüller](#), plenty of great content to learn web development.

This input hook takes a validator arg, which is a function that will validate the input provided, and will set a bool variable that will eventually change the input box to red to signify the input is not valid, or regular if it is. This can be taken a step further and add text that appears to tell the user what is wrong.

Now that the input hook is created, the next thing is to create a new SignupForm directory with a SignupForm.js file in the new directory.

src/components	<pre> import {useCallback, useContext, useState} from 'react' import useInput from '../../hooks/use-input' </pre>
----------------	---

# /SignupForm/ SignupForm.js

```

import authContext from
'../../context/AuthContext/auth-context'

const SignupForm = props => {
  let formIsValid = false
  const ctx = useContext(authContext)

  const [confirmPassword, setConfirmPassword] =
useState('')
  const confirmPasswordChangeHandler = event => {
    setConfirmPassword(event.target.value)
  }

  const {
    enteredInput: usernameInput,
    enteredInputIsValid: usernameIsValid,
    inputIsValid: usernameIsValid,
    inputBlurHandler: usernameBlurHandler,
    inputChangeHandler: usernameChangeHandler,
    resetOnFormSubmitHandler: resetUsername
  } = useInput(useCallback(input => input.trim() !== '',
[]))

  const {
    enteredInput: firstnameInput,
    enteredInputIsValid: firstnameIsValid,
    inputIsValid: firstnameIsValid,
    inputBlurHandler: firstnameBlurHandler,
    inputChangeHandler: firstnameChangeHandler,
    resetOnFormSubmitHandler: resetFirstname
  } = useInput(useCallback(input => input.trim() !== '',
[]))

  const {
    enteredInput: lastnameInput,
    enteredInputIsValid: lastnameIsValid,
    inputIsValid: lastnameIsValid,
    inputBlurHandler: lastnameBlurHandler,
    inputChangeHandler: lastnameChangeHandler,
    resetOnFormSubmitHandler: resetLastname
  } = useInput(useCallback(input => input.trim() !== '',
[]))

  const {
    enteredInput: emailInput,
    enteredInputIsValid: emailIsValid,
    inputIsValid: emailIsValid,
    inputBlurHandler: emailBlurHandler,
    inputChangeHandler: emailChangeHandler,
    resetOnFormSubmitHandler: resetEmail
  } = useInput(useCallback(input => input.trim() !== '' &&
input.includes('@') && input.length > 4, []))

  const {
    enteredInput: passwordInput,
    enteredInputIsValid: passwordIsValid,
    inputIsValid: passwordIsValid,
    inputBlurHandler: passwordBlurHandler,
    inputChangeHandler: passwordChangeHandler,
    resetOnFormSubmitHandler: resetPassword
  } = useInput(useCallback(
input => input.trim() !== '' && input.length > 8 &&

```



```

input === confirmPassword, [confirmPassword]
  ))

  const FormSubmitHandler = event => {
    event.preventDefault()
    console.log('test')
    const form = {
      email: emailInput,
      first_name: firstnameInput,
      last_name: lastnameInput,
      username: usernameInput,
      password: passwordInput,
      confirmPassword,
    }
    ctx.UserSignupHandler(form)
    resetUsername('')
    resetEmail('')
    resetFirstname('')
    resetLastname('')
    resetPassword('')
    setConfirmPassword('')
  }

  if (emailIsValid && usernameIsValid && passwordIsValid
    && firstnameIsValid && lastnameIsValid)
    formIsValid = true

  return (
    <div className='pt-4 flex h-fit'>
      <form className='card bg-gray-500
justify-between grow w-96' onSubmit={FormSubmitHandler}>
        <div>
          <h2 className='font-bold text-center
text-white'>Create a New Account</h2>
        </div>
        {ctx.authError !== '' && <div
className='bg-gray-800 text-center justify-center mt-3
rounded'>
          <p
className='text-rose-600'>{ctx.authError}</p>
        </div>}
        <div className='py-2'>
          <label className='font-bold'>
            <span
className={` ${usernameIsValid &&
'text-red-700'} `}>Username</span>
            <input type='text'
className={` form-input rounded block w-full text-black
${usernameIsValid &&
'border-red-700 border-2'} `}
onChange={usernameChangeHandler} value={usernameInput}
onBlur={usernameBlurHandler}/>
          </label>
        </div>
        <div className='py-2'>
          <label className='font-bold'>
            <span
className={` ${firstnameIsValid && 'text-red-700'} `}>First
Name</span>
            <input type='text'
className={` form-input rounded block w-1/2 text-black
${firstnameIsValid &&

```

```

'border-red-700 border-2'}}`

onChange={firstnameChangeHandler} value={firstnameInput}
onBlur={firstnameBlurHandler}/>
    <span
      className={` ${lastnameIsValid && 'text-red-700'}}`>Last
      Name</span>
    <input type='text'
      className={`form-input rounded block w-1/2 text-black
        ${lastnameIsValid &&
          'border-red-700 border-2'}}`

      onChange={lastnameChangeHandler} value={lastnameInput}
      onBlur={lastnameBlurHandler}/>
    </label>
  </div>
  <div className='py-2'>
    <label className='font-bold'>
      <span className={` ${emailIsValid && 'text-red-700'}}`>Email</span>
      <input type='email'
        className={`form-input rounded block w-full text-black
          ${emailIsValid &&
            'border-red-700 border-2'}}`

        onChange={emailChangeHandler} value={emailInput}
        onBlur={emailBlurHandler}/>
      </label>
    </div>
    <div className='py-2'>
      <label className='font-bold'>
        <span
          className={` ${passwordIsValid &&
            'text-red-700'}}`>Password</span>
        <input type='password'
          className={`form-input rounded block w-full text-black
            ${passwordIsValid &&
              'border-red-700 border-2'}}`

          onChange={passwordChangeHandler} value={passwordInput}
          onBlur={passwordBlurHandler}/>
        </label>
      </div>
      <div className='py-2'>
        <label className='font-bold'>
          <span className='text-white'>Confirm
          Password</span>
          <input type='password'
            className='form-input rounded block w-full text-black'
            value={confirmPassword}

            onChange={confirmPasswordChangeHandler}/>
          </label>
        </div>
        <button disabled={!formIsValid}
          type='submit'
          className={`rounded-full
            ${formIsValid ? 'bg-gray-900
              hover:bg-blue-700' : 'bg-gray-600'} text-white px-4 py-2
              mt-1`>
          Sign up
        </button>

```

	<pre>         &lt;/form&gt;       &lt;/div&gt;     )   }   export default SignupForm </pre>
--	---

The form will only be able to be submitted when all inputs are valid otherwise the submit button is disabled and grayed out. The signup form will call the auth context signup handler, which will send a request to the back-end to create a new user, when the user submits the form.

Now that a user can sign up, a login form will need to be created to allow the user to login and gain access to parts of the app that require the user to be logged in.

<b>src/components</b> <b>/LoginForm/</b> <b>LoginForm.js</b>	<pre> import {useContext, useCallback} from 'react' import authContext from '../../context/AuthContext/auth-context' import useInput from '../../hooks/use-input'  const LoginForm = () =&gt; {   const ctx = useContext(authContext)   let formIsValid = false    const {     enteredInput: passwordInput,     enteredInputIsValid: passwordIsValid,     inputIsValid: passwordIsInvalid,     inputBlurHandler: passwordBlurHandler,     inputChangeHandler: passwordChangeHandler,     resetOnFormSubmitHandler: resetPassword   } = useInput(useCallback(     input =&gt; input.trim() !== '' &amp;&amp; input.length &gt; 8   ))    const {     enteredInput: emailInput,     enteredInputIsValid: emailIsValid,     inputIsValid: emailIsInvalid,     inputBlurHandler: emailBlurHandler,     inputChangeHandler: emailChangeHandler,     resetOnFormSubmitHandler: resetEmail   } = useInput(useCallback(input =&gt; input.trim() !== '' &amp;&amp;     input.includes('@') &amp;&amp; input.length &gt; 4, []))    const FormSubmitHandler = event =&gt; {     event.preventDefault()     const email = emailInput     const password = passwordInput     ctx.UserLoginHandler({email: email, password: password})   }   if (emailIsValid &amp;&amp; passwordIsValid) </pre>
--	---



Now that both the sign up and login forms are created, logic to change the content of the main body between the posts (this will be created later), as well as the forms is needed to make a seamless user experience.

src/app.js	<pre> import './App.css' import Header from './components/UI/Header/Header' import {useState, useContext} from 'react' import LoginForm from './components/LoginForm/LoginForm' import SignupForm from './components/SignupForm/SignupForm' import authContext from './context/AuthContext/auth-context'  function App() {   const ctx = useContext(authContext)   const [content, setContent] = useState()   const [contentStateString, setContentStateString] =     useState('')    const OpenLoginFormHandler = () =&gt; {     setContent(&lt;LoginForm/&gt;)     setContentStateString('Login form')   }    const OpenSignupFormHandler = () =&gt; {     setContent(&lt;SignupForm/&gt;)     setContentStateString('Sign up form')   }    const OpenPostFormHandler = () =&gt; {     console.log('posts')   }    return (     &lt;div className='bg-gray-600 h-screen'&gt;       &lt;Header onLoginButtonClick={OpenLoginFormHandler} onSignupButtonClick={OpenSignupFormHandler} onPostButtonClick={OpenPostFormHandler} isLoggedIn={ctx.isLoggedIn}/&gt;       &lt;div className='grid grid-flow-row auto-rows-auto gap-20 bg-gray-600 justify-center'&gt;         &lt;div&gt;           {content}         &lt;/div&gt;       &lt;/div&gt;     &lt;/div&gt;   ) }  export default App </pre>
------------	--

### 6.1.7 Main post page & post form

Now that a user can signup and login, the next thing is to allow them to view and upload posts.

First create a new Posts directory in the components directory. Within that Posts directory create a new Posts.js file as well as a Post directory that will have a Post.js file.

src/ components/ Posts/Post/ Post.js	<pre> import {useContext, useState} from 'react' import authContext from '../../../../../context/AuthContext/auth-context' import postContext from '../../../../../context/PostContext/post-context'  const Post = props =&gt; {   const [showPost, setShowPost] = useState(false)   const authCtx = useContext(authContext)   const postCtx = useContext(postContext)   const post = props.post    const handleShowPost = () =&gt; {     setShowPost(true)   }    const handleClosePost = () =&gt; {     setShowPost(false)   }    const onPostDeleteHandler = async () =&gt; {     setShowPost(false)     postCtx.onRetrievePostsHandler()   }    return (     &lt;&gt;       &lt;a onClick={handleShowPost} href={'#'}&gt;         &lt;div className='card bg-gray-500 flex justify-between grow my-2 gap-x-40' onClick={handleShowPost}&gt;           &lt;span className='text-left'&gt;{post.title}&lt;/span&gt;           &lt;div className='top-0 right-0 text-right'&gt;             &lt;span&gt;{post.author.username}&lt;/span&gt;             &lt;div&gt;{new Date(post.createdAt).toLocaleString().split(',')[0]}&lt;/div&gt;           &lt;/div&gt;         &lt;/div&gt;       &lt;/a&gt;     &lt;/&gt;   ) }  export default Post </pre>
---	--

The Post component is used to display a clickable list of posts that when clicked will pull up that post's full details as well as its comments in a modal (a lightbox that will lay on top of everything making the rest of the

app intractable until it is closed). These will be created using a map function on the list retrieved from the Posts component.

Now to create the Posts.js component.

src/ components/ Posts/ Posts.js	<pre> import Post from '../Post/Post' import {useContext, useEffect} from 'react' import postContext from '../../context/PostContext/post-context'  const Posts = () =&gt; {   const postCtx = useContext(postContext)    useEffect(() =&gt; {     postCtx.onRetrievePostsHandler(1)   }, [])    const onNextClickHandler = () =&gt; {     postCtx.onRetrievePostsHandler(postCtx.page + 1)   }    const onPreviousClickHandler = () =&gt; {     postCtx.onRetrievePostsHandler(postCtx.page - 1)   }    return (     &lt;div className='flex-container h-fit'&gt;       {postCtx.posts.map(post =&gt; (         &lt;Post post={post} key={post.id}/&gt;       ))}       &lt;div className="flex justify-center"&gt;         {postCtx.page &gt; 1 &amp;&amp; (           &lt;button className='rounded-full bg-gray-900 hover:bg-blue-700 text-white px-4 py-2 mt-1 mx-1'             onClick={onPreviousClickHandler}&gt;             Previous           &lt;/button&gt;         )}         {postCtx.page &lt; postCtx.totalPages &amp;&amp; (           &lt;button className='rounded-full bg-gray-900 hover:bg-blue-700 text-white px-4 py-2 mt-1'             onClick={onNextClickHandler}&gt;             Next           &lt;/button&gt;         )}       &lt;/div&gt;     &lt;/div&gt;   ) }  export default Posts </pre>
---	---

The Posts module which will be connected to the App component is in charge of displaying the list of posts as well as providing a next or previous button depending on the current page the user is on and if more pages exist.

Now that there is a way to display posts, the user should now be able to create new posts (if they are logged in). Create a new PostForm directory in the Posts/Post directory and with it a new PostForm.js file.

<b>src/ components/ Posts/Post/ PostForm/ PostForm.js</b>	<pre> import {useState, useCallback, useContext} from 'react' import useInput from '../../hooks/use-input' import authContext from '../../context/AuthContext/auth-context' import postContext from '../../context/PostContext/post-context'  const PostForm = props =&gt; {   const authCtx = useContext(authContext)   const postCtx = useContext(postContext)   const [openForm, setOpenForm] = useState(false)   let formIsValid = false   const {     enteredInput: titleInput,     enteredInputIsValid: titleIsValid,     inputIsValid: titleIsValid,     inputBlurHandler: titleBlurHandler,     inputChangeHandler: titleChangeHandler,     resetOnFormSubmitHandler: resetTitle   } = useInput(useCallback(input =&gt; input.trim() !== '' &amp;&amp; input.length &gt; 10, []))    const {     enteredInput: bodyInput,     enteredInputIsValid: bodyIsValid,     inputIsValid: bodyIsValid,     inputBlurHandler: bodyBlurHandler,     inputChangeHandler: bodyChangeHandler,     resetOnFormSubmitHandler: resetBody   } = useInput(useCallback(input =&gt; input.trim() !== '' &amp;&amp; input.length &gt; 5, []))    const [image, setImage] = useState('')    const handleOpenForm = () =&gt; {     setOpenForm(true)   }    const onImageUploadChange = event =&gt; {     console.log(event.target.files[0])     setImage(event.target.files[0])   }    const handleFormSubmit = async event =&gt; {     event.preventDefault()     const post = {       titleInput,       bodyInput,       image     }     postCtx.onCreatePostHandler(post, authCtx.token)     setOpenForm(false)     resetBody()     resetTitle()     setImage('')   } } </pre>
---	---



```

const handleCancelSubmit = event => {
  event.preventDefault()
  setOpenForm(false)
  resetBody()
  resetTitle()
  setImage('')
}
if (titleIsValid && bodyIsValid && image !== '')
  formIsValid = true
return (
  <div className="pt-4 flex h-fit">
    {!openForm &&
      <button onClick={handleOpenForm}
        className="rounded-full bg-gray-900
        hover:bg-blue-700 text-white px-4 py-2 text-center
        grow">Create
          New Post</button>
    }
    {openForm && <form onSubmit={handleFormSubmit}
      className="card bg-gray-500 justify-between grow w-96">
      <div>
        <h2 className="font-bold text-center
        text-white">Create New Post</h2>
      </div>
      {authCtx.authError !== '' && <div
        className="bg-gray-800 text-center justify-center mt-3
        rounded">
        <p
          className="text-rose-600">{authCtx.authError}</p>
        </div>}
      <div className="py-2">
        <label className="font-bold">
          <span className={` ${titleIsValid &&
            'text-red-700'} `}>Title:</span>
          <input type="text"
            className={` form-input rounded block w-full text-black
              ${titleIsValid &&
                'border-red-700 border-2'} `}
            onChange={titleChangeHandler}
            value={titleInput} onBlur={titleBlurHandler}/>
        </label>
      </div>
      <div className="py-2">
        <label className="font-bold">
          <span className={` ${bodyIsValid &&
            'text-red-700'} `}>Body:</span>
          <textarea type="textarea"
            className={` form-textarea mt-1 w-full rounded block text-black
              ${bodyIsValid &&
                'border-red-700 border-2'} `}
            onChange={bodyChangeHandler}
            value={bodyInput} onBlur={bodyBlurHandler} rows="5"/>
        </label>
      </div>
      <div className="py-2">
        <label className="font-bold">
          <span>Image:</span>
          <input type="file" className={` w-full `}
            onChange={onImageUploadChange}/>
        </label>
      </div>
    </div>
  )

```

	<pre>         &lt;/div&gt;         &lt;button type="submit" disabled={!formIsValid}             className={`rounded-full                 \${formIsValid ? 'bg-gray-900 hover:bg-blue-700' : 'bg-gray-600'} text-white px-4 py-2 mt-1`} &gt;Submit         &lt;/button&gt;         &lt;button onClick={handleCancelSubmit}             className="rounded-full bg-gray-900 hover:bg-blue-700 text-white px-4 py-2 mt-1" &gt;Cancel         &lt;/button&gt;     &lt;/form&gt; &lt;/div&gt;     ) }  export default PostForm </pre>
--	--

Similar to the login and sign up forms, this post form uses the custom input hook, and when all of the data has been validated, it will allow the user to submit it which uses the post context create handler to send a POST request to the back-end to create a new post.

Now that the Posts and PostForm components are created, simply add them to the app.js file, the Posts component will be set as the default content state, and the PostForm will require the user is logged in to be able to see and interact with.

src/app.js	<pre> import './App.css' import Header from './components/UI/Header/Header' import Posts from './components/Posts/Posts' import PostForm from './components/Posts/PostForm/PostForm' import {useState, useContext} from 'react' import LoginForm from './components/LoginForm/LoginForm' import SignupForm from './components/SignupForm/SignupForm' import authContext from './context/AuthContext/auth-context'  function App() {   const ctx = useContext(authContext)   const [content, setContent] = useState(&lt;Posts/&gt;)   const [contentStateString, setContentStateString] =     useState('posts')    const OpenLoginFormHandler = () =&gt; {     setContent(&lt;LoginForm/&gt;)     setContentStateString('Login form')   }    const OpenSignupFormHandler = () =&gt; {     setContent(&lt;SignupForm/&gt;)     setContentStateString('Sign up form')   } </pre>
------------	---

	<pre> const OpenPostFormHandler = () =&gt; {   setContent(&lt;Posts/&gt;)   setContentStateString('posts') } return (   &lt;div className='bg-gray-600 h-screen'&gt;     &lt;Header onLoginButtonClick={OpenLoginFormHandler} onSignupButtonClick={OpenSignupFormHandler} onPostButtonClick={OpenPostFormHandler} isLoggedIn={ctx.isLoggedIn}/&gt;     &lt;div className='grid grid-flow-row auto-rows-auto gap-20 bg-gray-600 justify-center'&gt;       &lt;div&gt;         {content}         {ctx.isLoggedIn &amp;&amp; contentStateString === 'posts' &amp;&amp; &lt;PostForm/&gt;}       &lt;/div&gt;     &lt;/div&gt;   &lt;/div&gt; ) }  export default App </pre>
--	--

### 6.1.8 Individual post, update, delete, retrieve & comments, commentForm section

To display individual posts, a modal (lightbox that prevents interaction with the rest of the app until modal is closed) will be used to overlay on top of the page to give the user a nice simple box to be able to read the posts from. Complete with a comment section as well as the ability to edit or delete said post (this will only be available if the current logged in user is the author of the post).

To begin, create a new Modal directory within the components/UI directory as well as a Modal.js file inside the new directory.

src/ components/ UI/ Modal/ Modal.js	<pre> import {createPortal} from 'react-dom' import {Fragment} from 'react'  const Backdrop = props =&gt; {   return &lt;div className='fixed top-0 left-0 z-20 h-screen w-screen block bg-black/75' onClick={props.onClick}/&gt; } const ModalOverlay = props =&gt; {   return ( </pre>
--	--

	<pre>         &lt;div className='fixed p-8 bg-white max-w-lg m-auto z-50 rounded-xl top-20 inset-x-0 shadow-md'&gt;           &lt;div&gt;{props.children}&lt;/div&gt;         &lt;/div&gt;       )     }      const Modal = props =&gt; {       return (         &lt;Fragment&gt;           {createPortal(             &lt;Backdrop onClick={props.onClick}/&gt;,             document.getElementById('overlays')           )}           {createPortal(             &lt;ModalOverlay&gt;{props.children}&lt;/ModalOverlay&gt;,             document.getElementById('overlays')           )}         &lt;/Fragment&gt;       )     }      export default Modal </pre>
--	--

Notice the `createPortal` method, React allows developers to create portals which will render components outside of the root DOM node that the rest of the app lives in. In the `index.html` file found in the `public` folder, add a new `div` alongside the root `div` with the `id overlays`

public/ index.html	<pre> &lt;!DOCTYPE html&gt; &lt;html lang="en"&gt;   &lt;head&gt;     &lt;meta charset="utf-8" /&gt;     &lt;link rel="icon" href="%PUBLIC_URL%/favicon.ico" /&gt;     &lt;meta name="viewport" content="width=device-width, initial-scale=1" /&gt;     &lt;meta name="theme-color" content="#000000" /&gt;     &lt;meta       name="description"       content="Web site created using create-react-app"     /&gt;     &lt;link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png"   /&gt;     &lt;!--       manifest.json provides metadata used when your web app is       installed on a       user's mobile device or desktop. See       https://developers.google.com/web/fundamentals/web-app-manifest       /     --&gt;     &lt;link rel="manifest" href="%PUBLIC_URL%/manifest.json" /&gt;     &lt;!--       Notice the use of %PUBLIC_URL% in the tags above.       It will be replaced with the URL of the `public` folder       during the build.       Only files inside the `public` folder can be referenced </pre>
-----------------------	---

	<p>from the HTML.</p> <p>Unlike <code>"/favicon.ico"</code> or <code>"favicon.ico"</code>, <code>"%PUBLIC_URL%/favicon.ico"</code> will work correctly both with client-side routing and a non-root public URL. Learn how to configure a non-root public URL by running <code>`npm run build`</code>.</p> <pre>--&gt; &lt;title&gt;React App&lt;/title&gt; &lt;/head&gt; &lt;body&gt;   &lt;noscript&gt;You need to enable JavaScript to run this app.&lt;/noscript&gt;   &lt;div id="overlays"&gt;&lt;/div&gt;   &lt;div id="root"&gt;&lt;/div&gt; &lt;!--   This HTML file is a template.   If you open it directly in the browser, you will see an empty page.    You can add webfonts, meta tags, or analytics to this file.   The build step will place the bundled scripts into the &lt;body&gt; tag.    To begin the development, run `npm start` or `yarn start`.   To create a production bundle, use `npm run build` or `yarn build`. --&gt; &lt;/body&gt; &lt;/html&gt;</pre>
--	--

Now that the modal is created, create a new `IndividualPost` directory in the `components/Posts` directory, but before creating the `IndividualPost` component, the other components (edit post form, comment section, comment form) will be created since they will all reside within the `IndividualPost` component.

Start by creating a new `EditPostForm` directory in the `IndividualPost` directory and `EditPostForm.js` file in the new directory.

src/ components/ Posts/ IndividualPost/ EditPostForm/ EditPostForm.js	<pre>import {useState, useCallback, useContext} from 'react' import useInput from '../../hooks/use-input' import authContext from   '../../context/AuthContext/auth-context' import postContext from   '../../context/PostContext/post-context'  const EditPostForm = props =&gt; {   const authCtx = useContext(authContext)   const postCtx = useContext(postContext)</pre>
--	---

```

const {
  enteredInput: titleInput,
  enteredInputIsValid: titleIsValid,
  inputIsValid: titleIsValid,
  inputBlurHandler: titleBlurHandler,
  inputChangeHandler: titleChangeHandler,
  resetOnFormSubmitHandler: resetTitle
} = useInput(useCallback(input => input.trim() !== '' &&
input.length > 10, []))

const {
  enteredInput: bodyInput,
  enteredInputIsValid: bodyIsValid,
  inputIsValid: bodyIsValid,
  inputBlurHandler: bodyBlurHandler,
  inputChangeHandler: bodyChangeHandler,
  resetOnFormSubmitHandler: resetBody
} = useInput(useCallback(input => input.trim() !== '' &&
input.length > 5, []))

const [image, setImage] = useState('')

const onImageUploadChange = event => {
  console.log(event.target.files[0])
  setImage(event.target.files[0])
}

const handleFormSubmit = async event => {
  event.preventDefault()
  const post = {
    titleInput,
    bodyInput,
    image
  }
  postCtx.onUpdatePostHandler(props.id, post,
authCtx.token)
  resetBody()
  resetTitle()
  setImage('')
  props.onCloseForm()
}

const handleCancelSubmit = event => {
  event.preventDefault()
  resetBody()
  resetTitle()
  setImage('')
  props.onCloseForm()
}

return (
  <div className='pt-4 flex h-fit'>
    <form onSubmit={handleFormSubmit}
className='card bg-gray-500 justify-between grow w-96'>
      <div>
        <h2 className='font-bold text-center
text-white'>Update Post</h2>
      </div>
      {authCtx.authError !== '' && <div
className='bg-gray-800 text-center justify-center mt-3
rounded'>
        <p

```

```

className='text-rose-600'>{authCtx.authError}</p>
</div>
<div className='py-2'>
  <label className='font-bold'>
    <span className={` ${titleIsValid}
    && 'text-red-700'} `>>Title:</span>
    <input type="text"
    className={`form-input rounded block w-full text-black
    ${titleIsValid} &&
    'border-red-700 border-2'} `>
    onChange={titleChangeHandler} defaultValue={props.title}
    onBlur={titleBlurHandler}/>
  </label>
</div>
<div className='py-2'>
  <label className='font-bold'>
    <span className={` ${bodyIsValid} &&
    'text-red-700'} `>>Body:</span>
    <textarea className={`form-textarea
    mt-1 w-full rounded block text-black
    ${bodyIsValid} &&
    'border-red-700 border-2'} `>
    onChange={bodyChangeHandler} defaultValue={props.body}
    onBlur={bodyBlurHandler}
    rows='5' />
  </label>
</div>
<div className='py-2'>
  <label className='font-bold'>
    <span>Image:</span>
    <br/>
    <span className='text-sm'>Leave
    empty if you do not want to change the image</span>
    <input type="file"
    className={`w-full`}
    onChange={onImageUploadChange}/>
  </label>
</div>
<button type='submit'
  className='rounded-full bg-gray-900 hover:bg-blue-700
  text-white px-4 py-2 mt-1'>Submit</button>
  <button onClick={handleCancelSubmit}
  className='rounded-full bg-gray-900 hover:bg-blue-700
  text-white px-4 py-2 mt-1'>Cancel</button>
</form>
</div>
)
}

export default EditPostForm

```

This form, very similar to the normal post form, will instead fill the inputs (not image) with the post that is being updated 's current body and title, to allow the user to see what it currently is and if they only need to make small changes. The post's info will be sent through a props field.

The next thing to add is the comments section. This similar to the posts list will be a list of cards populated with the comment body, author username, and date created field. Create a new Comments directory in the Components directory, and with it a Comments.js, Comment directory and Comment.js in the Comment directory, in the Comments directory.

src/ components/ Comments/ Comment/ Comment.js	<pre> const Comment = props =&gt; {   const comment = props.comment    return (     &lt;div className='p-8 bg-white max-w-lg m-auto z-50 rounded-xl top-20 inset-x-0 shadow-lg border border-2 border-gray-500'&gt;       &lt;p className='scrollbar-thin text-left overflow-y-auto max-h-24 text-justify scroll-smooth scroll-m-auto'&gt;{comment.comment}&lt;/p&gt;       &lt;h2 className='text-right font-bold'&gt;{comment.author.username}&lt;/h2&gt;     &lt;/div&gt;   ) }  export default Comment </pre>
--	---

src/ components/ Comments/ Comments.js	<pre> import Comment from './Comment/Comment'  const Comments = props =&gt; {   const comments = props.comments    return (     &lt;div&gt;       &lt;h2 className='text-left pt-10'&gt;Comments&lt;/h2&gt;       &lt;ul className='list-none'&gt;         {comments.map(comment =&gt; (           &lt;li&gt;&lt;Comment comment={comment} key={comment.id}/&gt;&lt;/li&gt;         ))}       &lt;/ul&gt;     &lt;/div&gt;   ) }  export default Comments </pre>
---	--



The final thing to do before creating the IndividualPost component is to create a comment form for users to create comments on any post. Create a new CommentForm directory in the Components/Posts/IndividualPost directory and with it a CommentForm.js file. This form will be simple only needing the user to create a single body input

<b>src/ components/ Posts/ IndividualPost/ CommentForm/ CommentForm.js</b>	<pre> import {useContext, useCallback} from 'react' import authContext from '../../../../../context/AuthContext/auth-context' import postContext from '../../../../../context/PostContext/post-context' import useInput from '../../../../../hooks/use-input'  const CommentForm = props =&gt; {   const authCtx = useContext(authContext)   const postCtx = useContext(postContext)    const {     enteredInput: commentInput,     enteredInputIsValid: commentIsValid,     inputIsValid: commentIsValid,     inputBlurHandler: commentBlurHandler,     inputChangeHandler: commentChangeHandler,     resetOnFormSubmitHandler: resetComment   } = useInput(useCallback(input =&gt; input.trim() !== '' &amp;&amp; input.length &gt; 5, []))    const handleFormSubmit = async event =&gt; {     event.preventDefault()     const postId = postCtx.individualPost.id     postCtx.onCreateCommentHandler(postId, commentInput, authCtx.token)     resetComment()     props.onCloseForm()   }    const handleCancelSubmit = event =&gt; {     event.preventDefault()     resetComment()     props.onCloseForm()   }    return (     &lt;div className="pt-4 flex h-fit"&gt;       &lt;form onSubmit={handleFormSubmit} className="card bg-gray-500 justify-between grow w-96"&gt;         &lt;div&gt;           &lt;h2 className="font-bold text-center text-white"&gt;Create a Comment&lt;/h2&gt;         &lt;/div&gt;         {authCtx.authError !== '' &amp;&amp; &lt;div className="bg-gray-800 text-center justify-center mt-3 rounded"&gt; </pre>
--	--

	<pre>         &lt;p           className="text-rose-600"&gt;{authCtx.authError}&lt;/p&gt;         &lt;/div&gt;         &lt;div className="py-2"&gt;           &lt;label className="font-bold"&gt;             &lt;span               className={` \${commentIsValid} &amp;&amp;                 'text-red-700'}`&gt;Comment:&lt;/span&gt;             &lt;input type="text"               className={`form-input rounded block w-full text-black                 \${commentIsValid} &amp;&amp;                 'border-red-700 border-2'}`&gt;              onChange={commentChangeHandler} value={commentInput}             onBlur={commentBlurHandler}/&gt;           &lt;/label&gt;         &lt;/div&gt;         &lt;button type="submit"           disabled={!commentIsValid} className={`rounded-full             \${commentIsValid ? 'bg-gray-900             hover:bg-blue-700' : 'bg-gray-600'} text-white px-4 py-2             mt-1`}&gt;           Submit         &lt;/button&gt;         &lt;button onClick={handleCancelSubmit}           className="rounded-full             bg-gray-900 hover:bg-blue-700 text-white px-4 py-2             mt-1"&gt;Cancel         &lt;/button&gt;       &lt;/form&gt;     &lt;/div&gt;   ) }  export default CommentForm </pre>
--	--

Finally now that all of the individual components for the IndividualPost component are created, create a new IndividualPost.js file in the IndividualPost directory.

<b>src/ components/ Posts/ IndividualPost/ IndividualPost.js</b>	<pre> import Comments from '../..//Comments/Comments' import authContext from '../../..//context/AuthContext/auth-context' import {useContext, useEffect, useState} from 'react' import postContext from '../../..//context/PostContext/post-context' import Modal from '../..//UI/Modal/Modal' import EditPostForm from './EditPostForm/EditPostForm' import CommentForm from './CommentForm/CommentForm'  const IndividualPost = props =&gt; {   const authCtx = useContext(authContext)   const postCtx = useContext(postContext)    const [openEditForm, setOpenEditForm] = useState(false) </pre>
--	--

```

    const [openCommentForm, setOpenCommentForm] =
    useState(false)

    useEffect(() => {
      postCtx.onRetrievePostHandler(props.id)
    }, [])

    const onDeleteClickHandler = () => {
      postCtx.onDeletePostHandler(props.id,
      authCtx.token)
      props.onDelete()
    }

    const OpenEditFormHandler = () => {
      setOpenEditForm(true)
    }
    const CloseEditFormHandler = () => {
      setOpenEditForm(false)
    }
    const OpenCommentFormHandler = () => {
      setOpenCommentForm(true)
    }
    const CloseCommentFormHandler = () => {
      setOpenCommentForm(false)
    }

    return (
      <>
        {openEditForm && <Modal><EditPostForm
        id={props.id} onCloseForm={CloseEditFormHandler}
          title={postCtx.individualPost.title}
          body={postCtx.individualPost.body}/></Modal>}
        <div className="text-black text-2xl text-center
        font-bold">
          {postCtx.individualPost.title}
        </div>
        <div className="flex justify-center max-h-96
        w-fit">
          {/*Django: 'http://localhost:8080' Flask:
          'http://localhost:8080/static/' NodeJS:
          'http://localhost:8080'*/}
          <img src={'http://localhost:8080' +
          postCtx.individualPost.imageUrl} alt="alt"/>
          </div>
          <div className="scrollbar-thin overflow-y-auto
          max-h-96 text-justify scroll-smooth scroll-m-auto">
            <p
              className="m-auto">{postCtx.individualPost.body}</p>
            </div>
            <div className="text-right font-bold">
              <h2>
                {postCtx.individualPost.author.username}
              </h2>
              {new
                Date(postCtx.individualPost.createdAt).toLocaleString().sp
                lit(',',')[0]}
            </div>
            {postCtx.individualPost.author.id ===
            parseInt(authCtx.userId) && <div className="flex
            justify-center">
              <button className={`rounded-full

```

	<pre> bg-gray-900 hover:bg-blue-700 text-white px-3 py-1 mt-1 mr-1`}          onClick={OpenEditFormHandler}&gt;         Edit       &lt;/button&gt;       &lt;button className={`rounded-full bg-gray-900 hover:bg-blue-700 text-white px-3 py-1 mt-1 ml-1`}          onClick={onDeleteClickHandler}&gt;         Delete       &lt;/button&gt;     &lt;/div&gt;   &lt;/div&gt;   {authCtx.userId !== '' &amp;&amp; !openCommentForm &amp;&amp; &lt;button onClick={OpenCommentFormHandler}   className={`rounded-full bg-gray-900 hover:bg-blue-700 text-white px-3 py-1 mt-1 ml-1`}&gt;     Add Comment&lt;/button&gt;     {openCommentForm &amp;&amp; &lt;CommentForm onCloseForm={CloseCommentFormHandler}/&gt;}     {postCtx.individualPost.comments !== undefined &amp;&amp; postCtx.individualPost.comments.length &gt; 0 &amp;&amp;     &lt;Comments comments={postCtx.individualPost.comments}/&gt;}   &lt;/div&gt; &lt;/&gt; ) }  export default IndividualPost </pre>
--	---

This component, will be opened when a user clicks on one of the posts within the main posts list page, which will then send a request to the back-end to retrieve the more detailed single post complete with comments, image, body, and it will allow authors of posts to update or delete their posts, as well as allow other users to create comments on that post that anyone can see. Add in some logic on the Post.js component that will open the modal when a user clicks on a post.

<b>src/ components/ Posts/Post/ Post.js</b>	<pre> import {useContext, useState} from 'react' import Modal from '../../UI/Modal/Modal' import IndividualPost from '../IndividualPost/IndividualPost' import authContext from '../../context/AuthContext/auth-context' import postContext from '../../context/PostContext/post-context'  const Post = props =&gt; {   const [showPost, setShowPost] = useState(false)   const postCtx = useContext(postContext)   const post = props.post </pre>
---	--

```

const handleShowPost = () => {
  setShowPost(true)
}

const handleClosePost = () => {
  setShowPost(false)
}

const onPostDeleteHandler = async () => {
  setShowPost(false)
  postCtx.onRetrievePostsHandler()
}

return (
  <>
    {showPost && <Modal
onClick={handleClosePost}><IndividualPost id={post.id}
onDelete={onPostDeleteHandler}/></Modal>}
    <a onClick={handleShowPost} href={'#'}>
      <div className='card bg-gray-500 flex
justify-between grow my-2 gap-x-40' onClick={handleShowPost}>
        <span
className='text-left'>{post.title}</span>
        <div className='top-0 right-0 text-right'>
          <span>{post.author.username}</span>
          <div>{new
Date(post.createdAt).toLocaleString().split(',')[0]}</div>
        </div>
      </div>
    </a>
  </>
)
}

export default Post

```

With that the front-end should be completed, try to start up the back-end on port 8080, as well as this front-end using **npm start**, and create a new user, login, create some new posts, comments, edit posts, delete, etc.

As always the full code can be found on my github:

- React: (<https://github.com/HuloM/React-Blog-Frontend>)

## 7. Cloud Setup

Now that the completed app is finished and working the only thing left is to set up the app for production and eventually deploy it somewhere. For deployment there are a few new tools that will be used, the first is going to be the cloud provider, which is where an instance with the project will live to allow for it to be accessed anywhere and not just in a local setting. As well as docker and github which will help set up a CI/CD pipeline to AWS. The pipeline works as follows,

- Code is updated and pushed to a repo (this will typically be your own personal branch that will later be merged with other branches, once the branches merge and push to a main branch the rest of the pipeline would start)
- Github actions will then run a deploy file included in the repo, that will run tests, invoke docker to create images
- Docker will create separate images for each of the components of the app
- Github actions will then deploy the app to a premade AWS beanstalk environment, where a docker-compose file will be run

### 7.1 AWS Beanstalk Deploy with Github Actions & Docker

#### 7.1.1 Download Docker

Docker is a tool that allows developers to create images and with those images create containers, images are basically blueprints that can be used to create containers that will hold code as well as dependencies to run the code, any number of containers can be created from an image. Docker provides its own images maintained by the docker team, as well as community images made by independent developers for everyone to use (<https://hub.docker.com/search>).

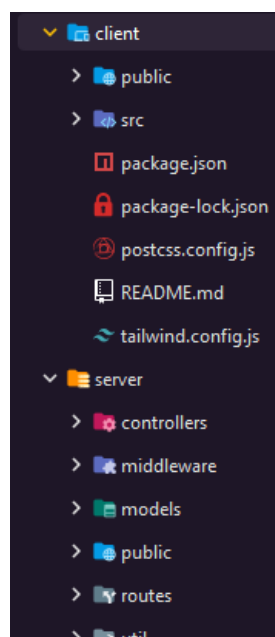
Link to download: <https://www.docker.com/products/personal/>

### 7.1.2 Staging app for deployment

To get the app ready for deploying a few things need to happen, first both components the front and back end should be in the same root folder.

Start by bringing the react app into a folder named client, the react app should consist of the public and src directories, as well as the package.json, tailwind config, postcss config files.

For this demonstration, the node js back end will be used, the only difference between setting up a CI/CD pipeline for this back-end and the others are just a few changes to dockerfiles and the app itself, completed apps for CI/CD for the other tech stacks will be provided in a github repo. For the back-end include the various directories along with an app.js, and package.json file. The .env file will be needed for setting up local development containers, however for production AWS will hold the environment variables, and allow the app to use them.



Along with setting this up a few changes to the fetch requests in the react app will need to be made so that it isn't looking for a localhost back-end but will call the one through another service that will relay it to the back-end. This change will be simple in the AuthContextProvider.js and the PostContextProvider.js simply change the ENDPOINT\_URL constant to '/api'.

src/context/ PostContext/ PostContext Provider.js	<code>const ENDPOINT_URL = '/api'</code>
src/context/ AuthContext / AuthContext Provider.js	<code>const ENDPOINT_URL = '/api'</code>

This is done for a few reasons, rather than calling localhost, which will try to find it on the local machine, it will now send a request through nginx which is a service that will essentially route the user to the correct service depending on the end of the url if the end is / it will be routed to the frontend and if it is /api it will route to the back-end.

As well as create a new script in the package.json file that will be explicitly for running nodemon for the development environment and then a new start script for running the app for the production environment.

server/package.json	<code>"scripts": {   "start": "node app.js",   "dev": "nodemon" }</code>
---------------------	--



### 7.1.3 Creating docker development files

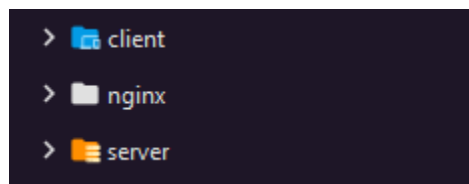
When working with docker, dockerfiles hold the instructions for building new images. When the docker build command is used, docker will read these instructions and execute them to create custom images.

(<https://docs.docker.com/engine/reference/builder/>)

The docker development files will be slightly different from the deployment files as they are intended to quickly spin up a local environment for the developer to use.

For starters since this will be a multi container app, another container will need to be made which will be for nginx, that will help route the user to the appropriate app or api depending on the call made to the url. If the user were to make a request **mywebsite.com/** nginx would serve up the react project which would bring the user to the main html page. If the user were to make a request to **mywebsite.com/api/** nginx would go and call the back-end, remove the api/ bit and call the endpoint provided afterwards (i.e. **mywebsite.com/api/posts/1**, would retrieve the first page of posts.

To do this create a new nginx directory in the root directory of the project fold alongside the other two projects.



Create a Dockerfile.dev file in the client directory and include the following.

client/Dockerfile .dev	<pre># gets a specific node image from docker hub FROM node:16-alpine  # creates a new directory and sets it as the current # directory for all other commands WORKDIR '/app'</pre>
---------------------------	---

	<pre> # copy package.json file to current working directory # this is done so that docker will use the cached version # of package.json and npm install unless actual changes are # made to package.json to build image faster COPY ./package.json ./ RUN npm install  # copy over the rest of the files COPY . .  # use npm run start command to start the server CMD ["npm", "run", "start"] </pre>
--	---

Create a Dockerfile in the server directory and include the following.

server/Dockerfile .dev	<pre> # gets a specific node image from docker hub FROM node:14.14.0-alpine  # creates a new directory and sets it as the current # directory for all other commands WORKDIR '/app'  # copy package.json file to current working directory # this is done so that docker will use the cached version # of package.json and npm install unless actual changes are # made to package.json to build image faster COPY ./package.json ./ RUN npm install  # copy over the rest of the files COPY . .  # use npm run dev command to use nodemon for development # server CMD ["npm", "run", "dev"] </pre>
---------------------------	--

In the nginx directory, create a default.conf file. This file will hold the config options for the nginx server which will hold details routing the server to either the react front-end or node js back-end depending on the extension of the url they use.

nginx/default .conf	<pre> # creating an upstream server named client # pointing at the client with port 3000 upstream client {     server client:3000; }  # creating an upstream server named api </pre>
------------------------	--

	<pre> # pointing at the api with port 3000, # the name server is not used to avoid confusion upstream api {     server api:8080; }  # nginx server config, listens on port 80 # the client max body is to allow the user to submit # forms with enough space to allow most images to be submitted  # location / will tell nginx to pass all requests with / to the # client (front-end) # location /api will tell nginx to pass all requests with /api # to api (back-end)  # location /ws is to support web sockets server {     listen 80;     client_max_body_size 10M;      location / {         proxy_pass http://client;     }      location /ws {         proxy_pass http://client;         proxy_http_version 1.1;         proxy_set_header Upgrade \$http_upgrade;         proxy_set_header Connection "Upgrade";     }      location /api {         rewrite /api/(.*) /\$1 break;         proxy_pass http://api;     } } </pre>
--	---

Create a new Dockerfile.dev in the nginx directory and include the following.

nginx/Dockerfile .dev	<pre> # nginx image FROM nginx  # copy our custom conf file to overwrite existing nginx # conf file COPY ./default.conf /etc/nginx/conf.d/default.conf </pre>
--------------------------	---

Now that all services have dockerfiles for development, the only thing left is to create a docker-compose file. Compose is a tool for docker that makes it simple for running multi container applications, the file is a yaml file that allows the user to define multiple services and any additional required configurations such as port mapping, environment variables

volumes, etc (<https://docs.docker.com/compose/>). Create a new `docker-compose-dev.yml` file in the root directory of the project.

<code>docker-compose-dev.yml</code>	<pre> version: '3' services:   nginx:     restart: always     build:       dockerfile: Dockerfile.dev       context: ./nginx     ports:       - '8080:80'   api:     build:       dockerfile: Dockerfile.dev       context: ./server     volumes:       - /app/node_modules       - ./server:/app     environment:       - MONGODB_HOST       - JWT_SECRET_KEY   client:     build:       dockerfile: Dockerfile.dev       context: ./client     volumes:       - /app/node_modules       - ./client:/app     environment:       - WDS_SOCKET_PORT=0 </pre>
-------------------------------------	---

This compose file contains 3 services. The nginx service which is set to restart always if it were to fail, as well as specifying the docker file and the nginx directory for building the container, lastly a simple port mapping connecting the local machine's port 8080 to the exposed port 80 of the nginx service. The api service which builds the back-end, specifies the dockerfile and directory, as well as uses environment variables which can be set so that they will use the environment variables of the local machine to not expose it to the public, and lastly some volumes to give persistent data to containers, but can also be used to bind containers so that changes made to local bound files can be seen in real time in the container. Lastly the client which builds the react app, specifies the dockerfile and directory of the client, some volumes, and an environment variable

Now that the entire docker development is set up the only thing to do is to run it, use **`docker compose -f docker-compose-dev.yml up`**, this

command will create the multi container app and start it, the -f tag is used to specify the docker-compose file as docker will look for a docker-compose.yml file by default and the development compose file is named slightly different.

```
$ docker compose -f docker-compose-dev.yml up
Network nodejs-express-blog-rest_default Creating
Network nodejs-express-blog-rest_default Created
Container nodejs-express-blog-rest-nginx-1 Creating
Container nodejs-express-blog-rest-api-1 Creating
Container nodejs-express-blog-rest-client-1 Creating
Container nodejs-express-blog-rest-nginx-1 Created
Container nodejs-express-blog-rest-api-1 Created
```

Running this command for the first time might result in some more instructions being run and output text to the console. But if the end result shows something similar to the following then the app will be successfully set up, and accessible on localhost with port 8080 (localhost:8080).

```
nodejs-express-blog-rest-api-1 | > nodejs-express-blog-rest@0.0.0 dev /app
nodejs-express-blog-rest-api-1 | > nodemon
nodejs-express-blog-rest-api-1 |
nodejs-express-blog-rest-api-1 | [nodemon] 2.0.15
nodejs-express-blog-rest-api-1 | [nodemon] to restart at any time, enter `rs`
nodejs-express-blog-rest-api-1 |
nodejs-express-blog-rest-api-1 | [nodemon] watching path(s): *.*
nodejs-express-blog-rest-api-1 | [nodemon] watching extensions: js,mjs,json
nodejs-express-blog-rest-api-1 | [nodemon] starting 'node app.js'
nodejs-express-blog-rest-client-1 |
nodejs-express-blog-rest-client-1 | > react-blog-frontend@0.1.0 start
nodejs-express-blog-rest-client-1 | > react-scripts start
nodejs-express-blog-rest-client-1 |
nodejs-express-blog-rest-api-1 | Server Active
```

### 7.1.4 Creating docker production files

Now that the app is complete there is a development docker environment, the next step is to make a production ready docker environment.

Similar to the development set up, new docker files will be created that do similar things as the development ones, but instead make the applications ready for being deployed to a server for real world users to use.

Create a Dockerfile file in the client directory and include the following.

client/Dockerfile	<pre>FROM node:16-alpine as builder  WORKDIR '/app'  COPY ./package.json ./ RUN npm install  COPY . .  RUN npm run build  FROM nginx EXPOSE 3000 COPY ./nginx/default.conf /etc/nginx/conf.d/default.conf COPY --from=builder /app/build /usr/share/nginx/html</pre>
-------------------	--

This dockerfile is a bit different from other ones that have been made, as react will need its own nginx that will be in charge of serving the built react files, while the other one is in charge of routing the user to the correct service from the url. So with this dockerfile, first is to create an image from node base image, and then do the normal stuff but instead of the command running the server it will build the react project, which react will then provide a build directory with all of the necessary files, which is then copied into its own nginx service. Afterwards nginx will expose port 3000 which is where the react app can be visited.

Since nginx is being used for react a new default.conf file will need to be included in its directory as well, to keep things simple to understand create a new nginx directory inside the client directory and create a new default.conf file in the new nginx directory.

client/nginx/ default.conf	<pre>server {     listen 3000;</pre>
-------------------------------	--------------------------------------

	<pre>location / {     root /usr/share/nginx/html;     index index.html index.htm;     try_files \$uri \$uri/ /index.html; }</pre>
--	---

This conf file sets up a server that points to the directory where the build folder was copied to from the image, and then sets the index page to index.html which is the html page that react provides.

Create a Dockerfile file in the client directory and include the following.

server/Dockerfile	<pre># gets a specific node image from docker hub FROM node:14.14.0-alpine  # creates a new directory and sets it as the current # directory for all other commands WORKDIR '/app'  # copy package.json file to current working directory # this is done so that docker will use the cached version # of package.json and npm install unless actual changes # are made to package.json to build image faster COPY ./package.json ./ RUN npm install  # copy over the rest of the files COPY . .  # use npm run start command to start server for # deployment CMD ["npm", "run", "start"]</pre>
-------------------	---

This dockerfile is similar to the dev file however the command run is now npm run start, which instead of using nodemon to start the server will instead use node to start the server. The difference between nodemon and node starting the server is that with nodemon changes made to the app will automatically restart the server and allow developers to quickly see the changes made, while node will not do this.

Create a new Dockerfile in the nginx directory and include the following.

nginx/Dockerfile	<pre># nginx image FROM nginx  # copy our custom conf file to overwrite existing nginx conf file COPY ./default.conf /etc/nginx/conf.d/default.conf</pre>
------------------	---

This file is identical to the Dockerfile.dev file, however it is a good idea to keep them as separate files in case there will be differences for development and deployment down the line.

The final step is to make a docker-compose.yml file in the root directory of the project, which will create the multi-container docker app that will be ready to be deployed.

docker-compose .yml	<pre>version: "3" services:   client:     image: "&lt;docker username&gt;/blog-client"     hostname: client   server:     image: "&lt;docker username&gt;/blog-server"     hostname: api     environment:       - MONGODB_HOST       - JWT_SECRET_KEY   nginx:     image: "&lt;docker username&gt;/blog-nginx"     hostname: nginx     ports:       - "80:80"</pre>
------------------------	---

This is a slightly different docker compose file, as rather than specifying build fields with docker file and directories, the builds will have already happened, and the images resulting have been pushed onto the docker hub under the username of the person that created it. In this example, the docker username should be the one that was used to sign up when making an account and the other part can be whatever is found suitable to explain the purpose of the image. To create these images the following



commands can be used as long as the CLI is currently logged into a docker account using **docker login** and providing credentials. Afterwards the two commands would be **docker build -t <docker username>/<image name> ./<directory with dockerfile>** and **docker push <docker username>/<image name>**. However doing this for all can be cumbersome so there is a simple way to automate it along with automating the aws setup using github.

### 7.1.5 Creating AWS account

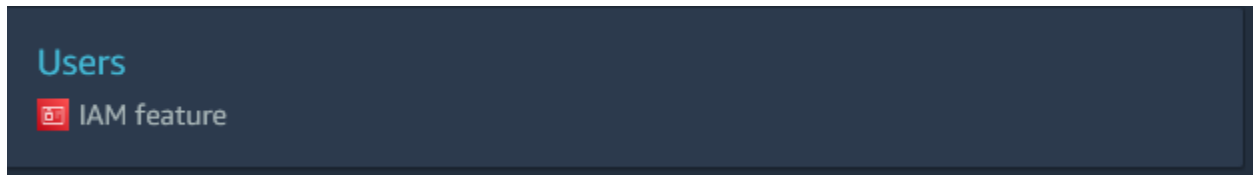
Before being able to set up the automation a few things need to be done beforehand. The first of which is creating an AWS account, something to note is that when creating an AWS account a credit will need to be used to make an account. However when first registering for an account that account will have a 12-month free tier period where it will access to some services at no cost, not all services so be careful when creating new services and make sure the selected services come at no cost. Some of these services come with a monthly limit of operations, if the operations exceed the limit then any additional operations will be billed at the normal rates so make sure to check in the **billing dashboard** what the current usages are.

To create an account visit <https://aws.amazon.com/> and create a new account.

### 7.1.6 Setting up IAM user

In AWS, there is a free service known as Identity and Access Management or IAM, which serves to give the account owner the ability to create sub accounts that hold specified permissions. It is a good idea to use IAM users for services as it separates out the permissions for different groups; i.e. developer team will only have access to only the permissions needed for them, while the devops team will only have the permissions needed for their tasks, as well as also keeping the root user (which has complete admin access to everything) secure by not exposing the keys of the root user but instead using the iam users' keys.

To create an IAM user type users in the search bar at the top and click on the users feature as shown below.



Once in the users dashboard click the Add user button.



Create a username that reflects the purpose of the user as well as checking the box for access keys, password is not needed but can be included.

### Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name\*

[+ Add another user](#)

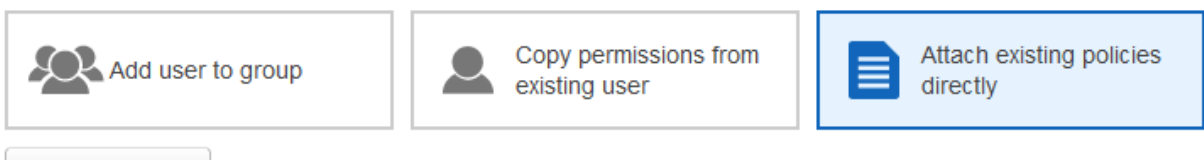
### Select AWS access type

Select how these users will primarily access AWS. If you choose only programmatic access, it does NOT prevent users from accessing the console using an assumed role. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)


- Select AWS credential type\* ☒ **Access key - Programmatic access**  
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.
- ☐ **Password - AWS Management Console access**  
Enables a **password** that allows users to sign-in to the AWS Management Console.

Afterwards click on the far right box for permissions for existing policies to bring up aws default policies.









### ▼ Set permissions



For the policies added onto the user there will be two, the first is the AWS beanstalk admin access policy, which can be found by typing in AWSElastic in the search bar.

Filter policies <span>Q awsElastic</span> <span>Showing 25 results</span>			
	Policy name	Type	Used as
<input checked="" type="checkbox"/>	 AdministratorAccess-AWSElasticBeanstalk	AWS managed	Permissions policy (1)

The second will be full access for EC2, which can be found by typing in ec2 in the search bar.

Filter policies <span>Q ec2</span> <span>Showing 26 results</span>			
	Policy name	Type	Used as
<input type="checkbox"/>	 AmazonEC2ContainerRegistryFullAccess	AWS managed	None
<input type="checkbox"/>	 AmazonEC2ContainerRegistryPowerUser	AWS managed	None
<input type="checkbox"/>	 AmazonEC2ContainerRegistryReadOnly	AWS managed	None
<input type="checkbox"/>	 AmazonEC2ContainerServiceAutoscaleRole	AWS managed	None
<input type="checkbox"/>	 AmazonEC2ContainerServiceEventsRole	AWS managed	None
<input type="checkbox"/>	 AmazonEC2ContainerServiceforEC2Role	AWS managed	None
<input type="checkbox"/>	 AmazonEC2ContainerServiceRole	AWS managed	None
<input checked="" type="checkbox"/>	 AmazonEC2FullAccess	AWS managed	Permissions policy (1)

Finally review that the user has the necessary policies, and create the user. Once the user is created, this page should show up

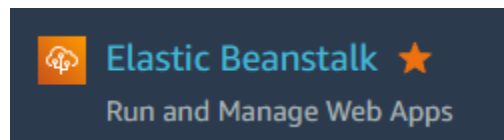
User	Access key ID	Secret access key
 testUser		 <a href="#">Show</a>

Make sure that both keys are kept secure, the secret access key can only be shown once on AWS so make sure it is saved somewhere, if not then a new key will have to be made. Optionally aws also allows the user download the keys in a file if that is needed.

### 7.1.7 Creating beanstalk app/environment

Eventually when github will automatically deploy the docker containers to aws it will need a beanstalk application premade to deploy it to. Elastic beanstalk is another service from AWS that makes it easy and simple to create scalable web apps (<https://aws.amazon.com/elasticbeanstalk/>).

To create a beanstalk application search for elastic beanstalk in the search at the top and click on the elastic beanstalk service.



Click on **Create Application**, and select a name for the application as well as setting the platform to docker.

**Application information**

Application name

SomeApplcation

Up to 100 Unicode characters, not including forward slash (/).

**Platform**

Platform

Docker

Platform branch

Docker running on 64bit Amazon Linux 2

Platform version

3.4.16 (Recommended)

**Application code**

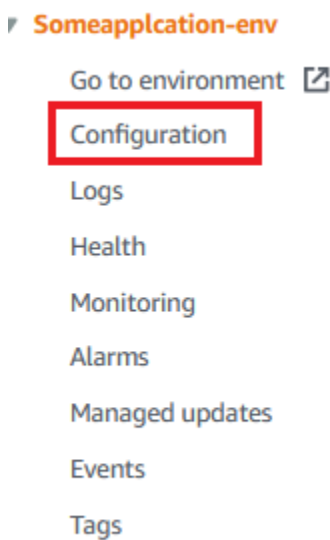
☒ **Sample application**  
Get started right away with sample code.

☐ **Upload your code**  
Upload a source bundle from your computer or copy one from Amazon S3.

Docker is the platform since the app that is going to be deployed to docker will have dockefiles and AWS will look for that to set up the containers. When the application is created it should also create an environment but if it doesn't simply click into the applications dashboard select the application and it should open up a new page to create an environment.

### 7.1.8 Setting up AWS environment variables

In the local applications environment variables were stored in a .env file, with AWS the environment variables are stored in the environment of a beanstalk application and can be edited in the configuration page of an application environment.



While in the configuration page select the software category and click the edit button.

Category	Options	Actions
Software	Log streaming: disabled Proxy server: nginx Rotate logs: disabled X-Ray daemon: disabled	<div>Edit</div>

Finally scroll down to the environment properties section and add in key pair values for the environment variables where the key is the name of the variable.

**Environment properties**  
 The following properties are passed in the application as environment properties. [Learn more](#)

Name	Value
JWT_SECRET_KEY	some string <span>✕</span>
MONGODB_HOST	some url <span>✕</span>

### 7.1.9 Creating Github account

Since github will be in charge of automatically deploying the code to the beanstalk application, a github account will be required to set up a repo that will do that.

Create a github account if needed at <https://github.com/>

### 7.1.10 Creating repo of project

To create a new repository visit your github page and click the new button found in the left area of the web page.



Set a name for the repo and make sure it is set to public.

Owner <sup>\*</sup> Repository name <sup>\*</sup>

HuloM / this is a new repository ✓

Great repository names: Your new repository will be created as this-is-a-new-repository-1special-waddle?

Description (optional)

☒ Public  
Anyone on the internet can see this repository. You choose who can commit.

☐ Private  
You choose who can see and commit to this repository.

Initialize this repository with:  
Skip this step if you're importing an existing repository.

☒ Add a README file  
This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore  
Choose which files not to track from a list of templates. [Learn more.](#)

.gitignore template: None

Choose a license  
A license tells others what they can and can't do with your code. [Learn more.](#)

License: None

This will set `main` as the default branch. Change the default name in your [settings](#).

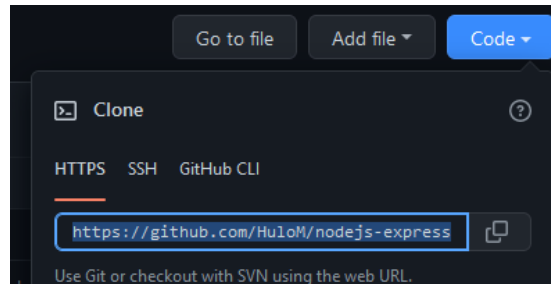
<sup>i</sup> You are creating a public repository in your personal account.

Create repository

Once the repo has been created use the following commands in the root directory of the project to push the current project directory onto the repo.

- **git init**
  - Git init will create a .git file in the directory.
- **git add .**
  - Git add . will stage all the changed or new files.
- **git commit -m "<some message>"**
  - Git commit will create a snapshot of the repository.
- **git remote add origin <Remote repository URL>**
  - Git remote add origin will link the existing project to the repository.

- The url can be found by visiting the repository page and clicking the code button and then the https tab.

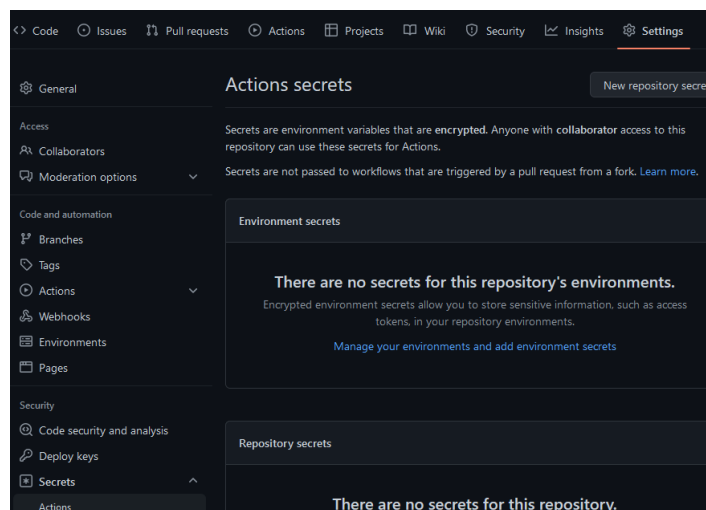


- **git push origin master**
  - This is the last command and it will push the changes up to the remote repository on whatever git platform is being used.

### 7.1.11 Setting up github action secrets

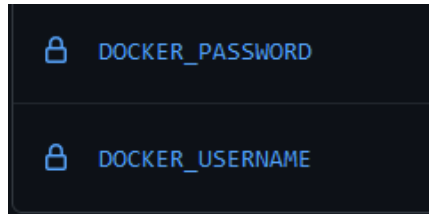
Similar to the concept of using environment variables for sensitive data in publicly visible applications. Github Actions, which is the service that will automate testing and deploying of the application to AWS, has secrets. Secrets are basically environment variables for github deploy files, a secret comes as a key value pair which github deploy files can access using that key name. Github deploy files will also print out logs for all of the operations it does, and as such github will filter out all secrets so they are not exposed.

To set up secrets visit the github repo, and go to the settings page, once there click secrets, then actions, which will then show this page.

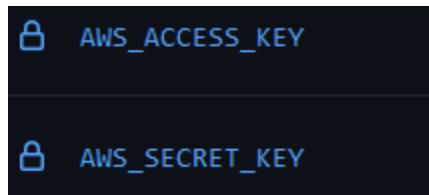




Click on the new repository secret and enter in the key value pairs. For the deploy file, there will need to be four secrets. The first two are the docker username and password to be able to login to a docker account. Create them using **DOCKER\_USERNAME** and **DOCKER\_PASSWORD**.



The last two are the AWS IAM user's access key and secret key. Create them using **AWS\_ACCESS\_KEY** and **AWS\_SECRET\_KEY**.



### 7.1.12 Creating github deploy file

Now create a new directory in the root directory of the project, named `.github` and inside that directory create another directory named `workflows`, lastly inside that directory create a `deploy.yaml` file

<code>.github/ workflows/ deploy.yaml</code>	<pre> name: Deploy Full Stack w Docker-Github to AWS EB on:   push:     branches:       - AWS-Docker-Setup jobs:   build:     runs-on: ubuntu-latest     steps:       - uses: actions/checkout@v2       - uses: docker/login-action@v2         with:           username: \${ secrets.DOCKER_USERNAME }           password: \${ secrets.DOCKER_PASSWORD }       - run: docker build -t &lt;docker username&gt;/react-test -f ./client/Dockerfile.dev ./client       - run: docker run -e CI=true &lt;docker username&gt;/react-test npm test </pre>
--	--

	<pre> - run: docker build -t &lt;docker username&gt;/blog-client ./client - run: docker build -t &lt;docker username&gt;/blog-nginx ./nginx - run: docker build -t &lt;docker username&gt;/blog-server ./server  - run: docker push &lt;docker username&gt;/blog-client - run: docker push &lt;docker username&gt;/blog-nginx - run: docker push &lt;docker username&gt;/blog-server  - name: Generate deployment package   run: zip -r deploy.zip . -x '*.git*'  - name: Deploy to EB   uses: einaregilsson/beanstalk-deploy@v18   with:     aws_access_key: \${ secrets.AWS_ACCESS_KEY }     aws_secret_key: \${ secrets.AWS_SECRET_KEY }     application_name: &lt;name of beanstalk application&gt;     environment_name: &lt;name of beanstalk environment&gt;     existing_bucket_name: &lt;name of bucket (can be found in s3&gt;     region: &lt;region the app is located in (i.e. us-west-2)&gt;     version_label: \${ github.sha }     deployment_package: deploy.zip </pre>
--	--

The first step is that github will wait for pushes on the specified branch, normally this would be like main or master, however this repo has a separate branch that is meant to display the AWS setup.

The next step is doing the jobs, which the first thing it does is check-out the repo, allowing workflows to access the project.

Afterwards it will login to docker and run some tests for the react app (this step can include tests for the api as well however none were set up)

Then it will build the images and push them to the docker account's repo.

It will then create a zip of the project files

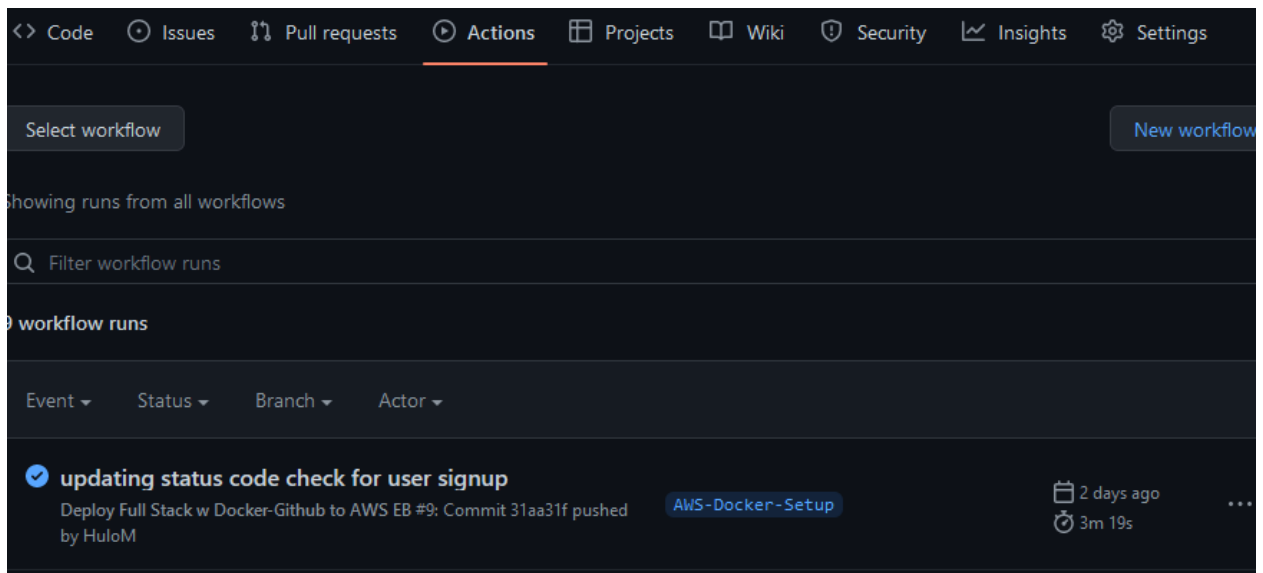
AWS will then use the keys to authenticate the user, look for the beanstalk application with the given application fields, these can be found as the name of the beanstalk application, name of the beanstalk environment,

name of the bucket of the beanstalk application (can be found under s3 service), and the region the application is in.

Lastly AWS will take the `deploy.zip` file and unzip and run the docker-compose file which will then grab the images that were created and run the application.

### 7.1.13 Testing CI/CD deployment to AWS

After the `deploy.yaml` file is complete, push the changes to the remote repository and visit the actions page in the github repository.



This should show an active workflow that will run through that `deploy.yaml` file and do all of the steps in real time to see any issues. If any should arise check to make sure the AWS keys and docker login credential secrets do not contain typos.

Afterwards if all is successful visit the beanstalk app and visit the provided link in the environment page, which should automatically show the react page.

Side note when using mongodb, it is likely that the app will not be able to connect as it has a whitelist of IPs that can connect to it. Since aws

ec2 instances don't have a static IP, mongodb cannot whitelist the app. One way to do this is to set up a vpc peering connection between the mongodb and the beanstalk app, the main issue with this is that mongodb does not allow this for the free or even monthly flat rate (M0, M2, M5) database tiers, it only allows this for the M10 and above which cost \$0.08+/hour of running. The way around this is to whitelist all IPs in the database, obviously the big issue with this is that anyone can connect so ensure that all users of the database use really secure passwords to prevent people from attempting to gain access, this whitelist can also be set for a certain duration if this app is temporary it might be useful to limit the time that all IPs are whitelisted. To do this visit the mongodb atlas page and visit the network access page in the security section, click on add IP, and click on allow access from anywhere, the option to have the ip whitelisted temporarily is down below and can be set up to a week.

---

Atlas only allows client connections to a cluster from entries in the project's IP Access List. Each entry should either be a single IP address or a CIDR-notated range of addresses. [Learn more.](#)

ADD CURRENT IP ADDRESS

ALLOW ACCESS FROM ANYWHERE

**Access List Entry:**

Enter IP Address or CIDR Notation

**Comment:**

Optional comment describing this entry



This entry is temporary and will be deleted in

6 hours ▼

Cancel

Confirm

## 8. Comparing Tech Stacks

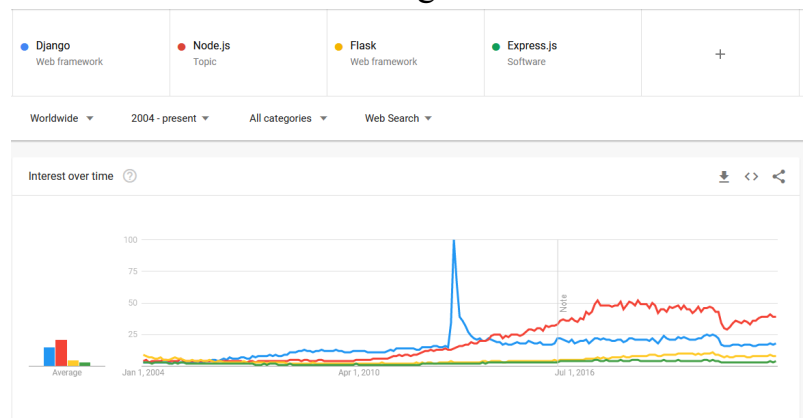
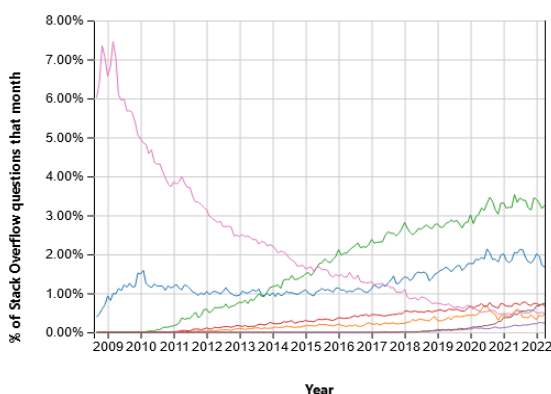
Now that all the walkthroughs for the projects are over, let's take a look at the tech stacks and each of the individual tools/technologies that were used. The comparisons will take a look at how easy complex the technology is, how long it took to learn, What size of projects are suitable (small, large) for the back-ends, and the popularity of different technologies.

A useful site to see popular stacks is <https://stackshare.io/stacks>. Where even some popular companies like pinterest, uber, airbnb, google, facebook, etc. will share what technologies are being used in their tech stacks.

### 8.1 Back-end Comparison

The back-end comparisons will include a look at each of the languages for the frameworks as well as the frameworks themselves. This comparison will combine personal experience in learning and using these languages and frameworks, as well as trend data from stackoverflow and google.

Taking a look at the following chart, provided from stackoverflow, based on the popularity of questions posted about the back-end technologies.



nodeJS seems to be the most popular, alongside express or nextJS as the web framework of choice. With django being almost as popular. Why does popularity matter? For a few reasons, popularity can signify trends among the industry for more preferred technologies or architectures which can result in more jobs being

available for having skills using those technologies. It will also show the amount of support and community that comes with it. For instance a framework like flask, during the code walkthrough there were a few flask specific extension packages used to incorporate other popular packages to be used with flask, such as flask JWT, or the migration extension, or the sqlalchemy extension, which were created by the community. Which without would have taken more involvement to add into the project than the simple few lines that made them work.

### 8.1.1 Advantages & Disadvantages (Django)

Django offers a complete batteries included framework, with a lot of builtin features to help make the development process quick and simple. These features include built-in ORM tools, authentication services, validation, admin layer for viewing the entries in the different model tables, database migration handling, etc. So naturally it can appear to be fairly easy to get into and use as it comes with a lot of the heavy lifting already included. However, in the development of the django back-end, it became apparent how everything was interconnected, requiring more knowledge of the main components of django (views, models, url mapping) as well as learning the rest framework components (serializers, authentication, viewsets).

Pros	Cons
<ul style="list-style-type: none"> <li>• Good scalability <ul style="list-style-type: none"> <li>◦ While there aren't exact numbers on how high of traffic django can handle, there are highly popular sites that see enormous amounts of traffic everyday that use django (instagram, bitbucket, pinterest)</li> </ul> </li> <li>• Helps development process go faster <ul style="list-style-type: none"> <li>◦ Being able to work on multiple modules at the same time helps reduce overall time</li> </ul> </li> <li>• Batteries-included <ul style="list-style-type: none"> <li>◦ Reducing amount of coding developers have to do</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• Learning Django as a new developer can be difficult <ul style="list-style-type: none"> <li>◦ Without any background in web development jumping straight into django can be confusing as it has a lot of components</li> </ul> </li> <li>• Not a good choice for small projects <ul style="list-style-type: none"> <li>◦ With django's batteries included code, it can encumber small projects that don't need all of that complexity</li> </ul> </li> <li>• Mastering the Django framework <ul style="list-style-type: none"> <li>◦ Learning the framework and being able to master it can differ, and while learning is faster and</li> </ul> </li> </ul>

<ul style="list-style-type: none"> <li>● Built-in security features <ul style="list-style-type: none"> <li>○ Security can be really important when it comes to web apps, as they can contain personal user data.</li> <li>○ Protects against common attacks like SQL injection, Cross-site request forgery, or Cross-site scripting injection</li> </ul> </li> <li>● REST framework <ul style="list-style-type: none"> <li>○ With the addition of the rest framework, it enables django to become a RESTful server rather than using the MVT approach</li> </ul> </li> <li>● Good documentation <ul style="list-style-type: none"> <li>○ It is always a good thing when a technology has documentation to help developers understand it.</li> </ul> </li> </ul>	<p>can take a few weeks it can take a little longer to be able to fully understand and use all of what django can offer.</p>
---	--

In short, django offers a very nice all in one package, offering its MVT architecture or using the rest framework to make it more aligned to the REST architecture, complete with its authentication, ORM, and various security implementations. Django is a great framework to use, for this small of a project, it would not be recommended to use as it is a lot easier to set it up using flask (if python is the preferred language) or express (if javascript is the preferred language).

### 8.1.2 Advantages & Disadvantages (Flask)

Flask, unlike django, is a minimalist framework offering just what is needed to get something up and running. Flask is very simple to learn and very beginner friendly as there is not a lot that needs to be learned. Flask can also scale fairly well with the use of cloud computing, which will handle scaling the app, but the key thing is that small applications will be easier to deploy onto as many servers as needed.

Pros	Cons
<ul style="list-style-type: none"> <li>• Beginner friendly/easy to learn <ul style="list-style-type: none"> <li>◦ The amount of time to learn this framework will be a lot less than something like django</li> </ul> </li> <li>• Scales well with cloud services <ul style="list-style-type: none"> <li>◦ In conjunction with something like AWS and beanstalk service a small flask app can be scaled</li> </ul> </li> <li>• Great for smaller apps <ul style="list-style-type: none"> <li>◦ With the simplicity that comes with flask, it is easy and fast to set up a small application or APIs</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• No standard for development <ul style="list-style-type: none"> <li>◦ Unlike django that comes with a standard structure for all apps, the flask application is up to the development team to create.</li> <li>◦ This means that someone that takes the time to learn django will have an easier time adjusting to a new django project than a flask experienced developer would adjust to an equally sized flask project</li> </ul> </li> <li>• Less support than bigger frameworks <ul style="list-style-type: none"> <li>◦ While there is a flask community, it is however not as big as the community surrounding django or node js and its various web frameworks</li> </ul> </li> </ul>

In short, flask is a great framework for beginner developers looking to expand into web development. Offering full control over the entire app development, something that django does not offer as it is more rigid in its structure and has a lot of the setup obfuscated to the user. While the community is small it is not none, there are many good third party packages that help to build flask into a competitor in the framework game.

### 8.1.3 Advantages & Disadvantages (NodeJS/Express)

Unlike both django and flask with python, javascript does not normally run outside of a browser. The way that it is possible to have server side javascript is by using nodeJS, nodeJS can be used alone to create a rest api or a static web page server. However, nodeJS does not have a lot of the functionality to do it cleanly, that's where something like express comes into play, express is the web framework of the backend while nodeJS will simply act as the server.

NodeJS has many benefits, such as being a part of one of the largest developer communities as it is the most popular technology as shown



(<https://insights.stackoverflow.com/survey/2020>), with all this popularity comes one of the most largest communities developing third party packages and tools for nodeJS development, which likely exceeds any other. NodeJS also offers good scalability with its event loop mechanism. Another good benefit is with using nodeJS for the backend along with a web framework, it enables an app to be comprised of almost entirely javascript, using a front-end framework like react or vue will mean that javascript is the main language of the application making it possible for developers to only need to know javascript, and some html/css.

### NodeJS

Pros	Cons
<ul style="list-style-type: none"> <li>• Large community of developers <ul style="list-style-type: none"> <li>◦ With the massive popularity comes a massive community of developers creating third party software for making the development of apps more simple</li> </ul> </li> <li>• Complete JS code base <ul style="list-style-type: none"> <li>◦ With a server side solution in the form of nodeJS it is possible to have the code base be entirely comprised of JS making it easier to hire and manage teams and development</li> </ul> </li> <li>• Good scalability <ul style="list-style-type: none"> <li>◦ Easy to deploy in containers to meet traffic demands</li> </ul> </li> <li>• Event loop <ul style="list-style-type: none"> <li>◦ Makes writing code a lot more simpler since concurrency doesn't need to be concerned about</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• Not good for CPU-intensive apps <ul style="list-style-type: none"> <li>◦ Since nodejs is a single-threaded app, CPU intensive applications that require a lot of processing are not ideal as multi threaded apps would be better.</li> <li>◦ The nodejs team has addressed this by adding a worker threads module, however it is likely still better to use other languages and their frameworks for these types of apps</li> </ul> </li> <li>• Event loop <ul style="list-style-type: none"> <li>◦ While the event loop is a pro, it can also be a con, as the dev has to know that NodeJS runs on a single thread so the dev has to make sure there is nothing that can block the single thread (i.e. infinite loop, synchronous calls)</li> </ul> </li> </ul>

### Express

Pros	Cons
<ul style="list-style-type: none"> <li>• Easy to learn             <ul style="list-style-type: none"> <li>◦ Similar to flask, express being a minimalist framework makes it easy and simple to learn on top of js syntax also being easy to understand</li> </ul> </li> <li>• Similar pros to nodejs</li> </ul>	<ul style="list-style-type: none"> <li>• Similar cons to nodejs</li> </ul>

#### 8.1.4 Brief look at other popular back-end frameworks

These brief looks, will take a look at other popular frameworks, however they are not intended to be as extensive as the frameworks used in the tech stacks, just a simple introduction with a link to their documentation.

##### 8.1.4.1 Fastify

“Fastify is a web framework highly focused on providing the best developer experience with the least overhead and a powerful plugin architecture, inspired by Hapi and Express. As far as we know, it is one of the fastest web frameworks in town.”

<https://www.fastify.io/>

##### 8.1.4.2 NestJS

“Nest (NestJS) is a framework for building efficient, scalable Node.js server-side applications. It uses progressive JavaScript, is built with and fully supports TypeScript (yet still enables developers

to code in pure JavaScript) and combines elements of OOP (Object Oriented Programming), FP (Functional Programming), and FRP (Functional Reactive Programming).”

<https://nestjs.com/>

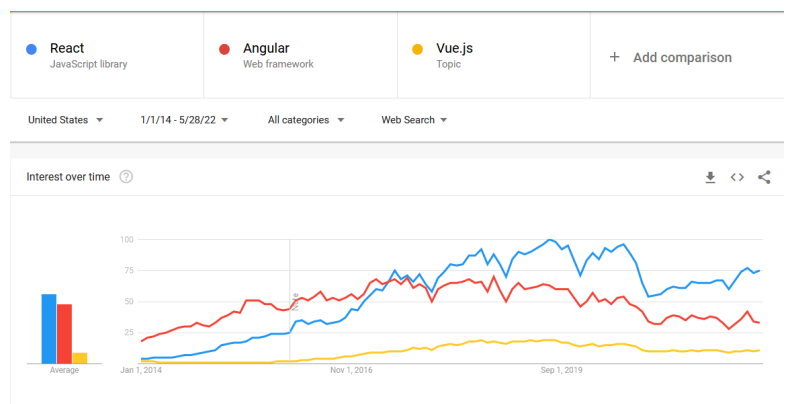
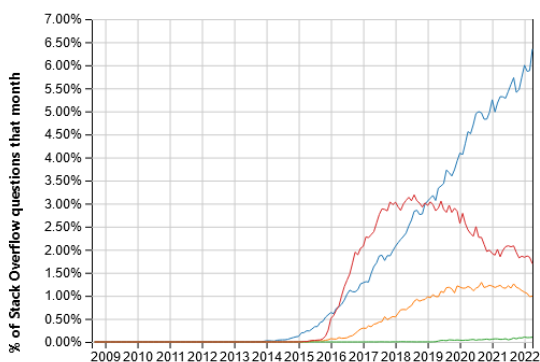
### 8.1.4.3 ASP.NET

“ASP.NET is an open-source, server-side web-application framework designed for web development to produce dynamic web pages. It was developed by Microsoft to allow programmers to build dynamic web sites, applications and services. ASP.NET is built on the Common Language Runtime (CLR), allowing programmers to write ASP.NET code using any supported .NET language.”([wikipedia](#))

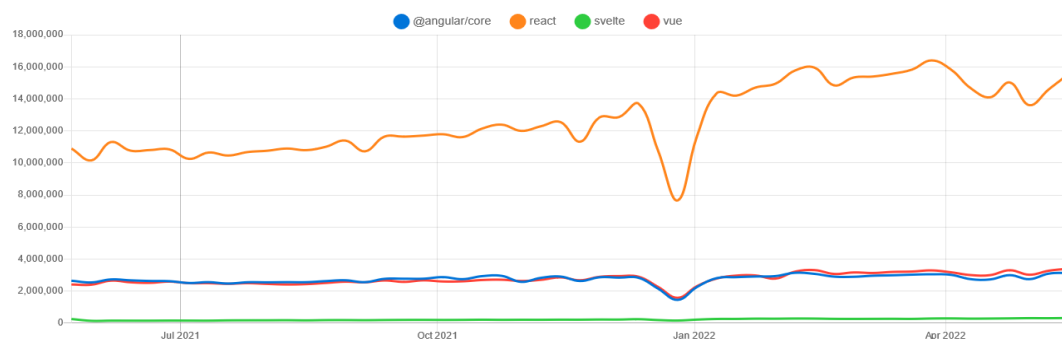
<https://dotnet.microsoft.com/en-us/apps/aspnet>

## 8.2 Front-end comparison

Since there is only one front-end framework, that will be the only comparison however taking a look at stackoverflow and google trends.



Downloads in past 1 Year



It is clear to see that currently react is the most favored, however, svelte which is a new framework that just launched in 2019 has gained some traction and is being favored among developers as seen (<https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190>, provided by <https://gist.github.com/tkrotoff>) with some data collection from surveys and other trend reports..

### 8.2.1 Advantages & Disadvantages (React)

React is a component with the ability to have states. React has a lot of benefits, one being a component based ui library, makes it easy to organize projects, even if they have dozens to hundreds of components, along with the reusability of the components. Other benefits such as being able to have a state and react automatically updating only the dom elements with that state whenever the state is updated. It is also not too terrible to learn aside from the JSX syntax which is basically getting used to using html elements as javascript variables and return types, as well as getting the hang of the different hooks react has to offer, state, context, portals, etc.

Pros	Cons
<ul style="list-style-type: none"> <li>● Reusability <ul style="list-style-type: none"> <li>○ Components in react act like functions they can be called endlessly and be used dynamically</li> </ul> </li> <li>● Updating only what is needed to <ul style="list-style-type: none"> <li>○ When a state changes and react has to re-render the html page, rather than re-rendering the whole page it will just re-render the dom element where the state lies</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>● Needing something like redux for a state container</li> <li>● Learning curve can be jarring for new developers</li> </ul>

## 8.2.2 Brief look at other popular front-end frameworks

These brief looks, will take a look at other popular frameworks, however they are not intended to be as extensive as the frameworks used in the tech stacks, just a simple introduction with a link to their documentation.

### 8.2.2.1 VueJS

“Vue is a JavaScript framework for building user interfaces. It builds on top of standard HTML, CSS and JavaScript, and provides a declarative and component-based programming model that helps you efficiently develop user interfaces, be it simple or complex.”

<https://vuejs.org/>

### 8.2.2.2 Angular

“Angular is a TypeScript-based free and open-source web application framework led by the Angular Team at Google and by a community of individuals and corporations. Angular is a complete rewrite from the same team that built AngularJS.” ([wikipedia](#))

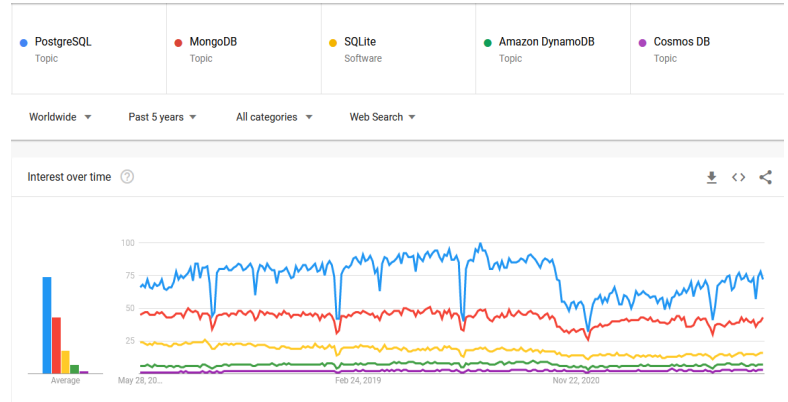
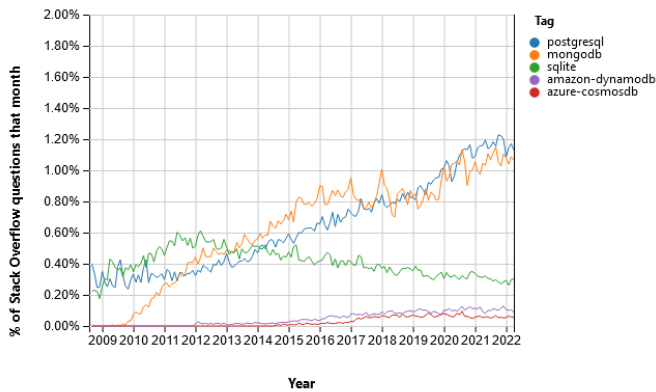
<https://angular.io/>

## 8.3 Database comparison

When looking at the differences between a SQL and NoSQL database, the main difference is that NoSQL databases will typically be document based where a document is similar to a json format with key value pairs and the ability to nest objects or arrays in a key. These documents will act as the rows of an SQL database, the key value pairs will be similar to columns, both will typically have an index as well as a primary key.

### 8.3.1 Advantages & Disadvantages (SQL based)

For the SQL based databases the ones used in two of the tech stacks were SQLite and PostgreSQL.



Looking at the graphs it is easy to see that both postgres and mongodb have become the most popular options, one for SQL and one for NoSQL.

#### 8.3.1.1 PostgreSQL

Postgres has become an iconic enterprise level database. As seen (<https://stackshare.io/postgresql>), companies like netflix, instagram, uber, spotify, all use postgres as part of their stacks. This is due to the features and stability it provides, while it was true that databases like mysql performed better in terms of speed, it lacked the same level of security and stability as postgres, and over time as mysql would add these features it became as slow as postgres. So in short the more features postgres provided gave it the boost it needed to become the more favored option, as well as being open source comes with its set of attractions.

Pros	Cons
<ul style="list-style-type: none"> <li>Flexible database <ul style="list-style-type: none"> <li>Can create new data types, or change existing ones to suit needs</li> </ul> </li> <li>Open-source</li> </ul>	<ul style="list-style-type: none"> <li>No columnar table support <ul style="list-style-type: none"> <li>Trying to read just a single column will require the full table to be parsed, which can be an issue if the table</li> </ul> </li> </ul>

<ul style="list-style-type: none"> <li>○ Can modify the source code to your needs</li> <li>● Simple to use and set up</li> <li>● Can be set up with AWS RDS</li> </ul>	has dozens of columns
--	-----------------------

In short, postgres is good for use in large scale applications that sees heavy traffic daily.

### 8.3.1.2 SQLite

SQLite, like its name suggests, is a lightweight but fast database, it requires less memory to run than other databases. This all however comes with a caveat, sqlite is meant for small projects and small sets of data, so it doesn't scale too well but for simple projects like this or other demo projects it would be great for use.

Pros	Cons
<ul style="list-style-type: none"> <li>● Fast</li> <li>● no setup needed <ul style="list-style-type: none"> <li>○ Simply download and run an executable to use sqlite</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>● Low-volume websites <ul style="list-style-type: none"> <li>○ Sqlite is only ok to handle low to medium traffic</li> <li>○ “any site that gets fewer than 100K hits/day should work fine with SQLite”</li> <li>○ Sites like twitter or youtube likely get tens to hundreds of millions of hits per day</li> </ul> </li> <li>● Limited database size <ul style="list-style-type: none"> <li>○ The max size of a sqlite database can be 281 terabytes, which is a lot but something like youtube that has billions of users would likely max that out quickly.</li> </ul> </li> </ul>

In short, SQLite is a great database to use for training or learning as it is simple to set up and is fast when it comes to low to medium traffic which is likely in development environments. However, SQLite can be used in commercial applications depending on the size and scale of the application, something as simple as a personal blog website.

## 8.3.2 Advantages & Disadvantages (NoSQL based)

### 8.3.2.1 MongoDB

MongoDB, unlike the other two databases shown in the other tech stacks, is a NoSQL or non-relational database. However that isn't to say that it can have related data, with a NoSQL db its possible to retrieve the data from different documents with just a single id embedded in the different documents that links them (userid, orderid, etc.). MongoDB is also flexible in that a document is not inherently tied to a specific schema, making it possible to change document structure as you go without having to add new data to new documents.

Pros	Cons
<ul style="list-style-type: none"> <li>● Flexible structure <ul style="list-style-type: none"> <li>○ Document schema can change as needed without needing to worry about old documents</li> <li>○ Documents can include or not include fields that other documents in the same collection have (i.e. if a user included a name or not, the field can be included or not in the document)</li> </ul> </li> <li>● Embed linked documents <ul style="list-style-type: none"> <li>○ Documents that have some key that links them together can be embedded within a main document rather than using a join operation</li> </ul> </li> <li>● Sharding <ul style="list-style-type: none"> <li>○ When a load on one server is too much, mongodb can split that load into multiple servers</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>● Data repetition <ul style="list-style-type: none"> <li>○ Sometimes different collections will contain redundant data in documents</li> <li>○ Leading to larger storage</li> </ul> </li> <li>● Document limits <ul style="list-style-type: none"> <li>○ Documents are limited to 16MB of storage each, as well as only allowing 100 levels of nesting in documents (each array or object adds a layer of nesting)</li> </ul> </li> </ul>

In short, mongodb is great to use when there is no defined schema as it is very flexible with documents in collections. It can also



be useful for high availability needs as its sharding will scale it out to meet the demand. Given that mongodb also has paid tiers for larger applications, it can also depend on the expenses available as the larger tiers are per hour rate. That being said there is a free tier and also a flat rate/month for slightly larger than the free tier (\$9 or \$25) which offer a good amount of storage for small business, educational or development purposes.

### **8.3.3 Brief look at other popular SQL & NoSQL databases**

#### **8.3.3.1 DynamoDB**

“Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB lets you offload the administrative burdens of operating and scaling a distributed database so that you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling. DynamoDB also offers encryption at rest, which eliminates the operational burden and complexity involved in protecting sensitive data.”

([https://en.wikipedia.org/wiki/Amazon\\_DynamoDB](https://en.wikipedia.org/wiki/Amazon_DynamoDB))

<https://aws.amazon.com/dynamodb/>

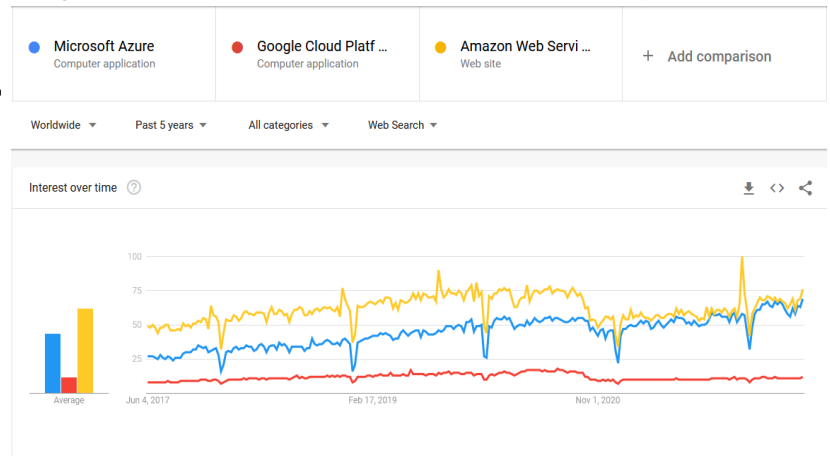
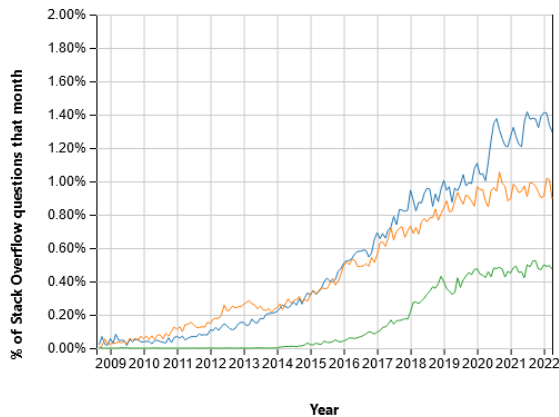
#### **8.3.3.2 MySQL**

“MySQL is an open-source relational database management system. A relational database organizes data into one or more data tables in which data may be related to each other. In addition to relational databases and SQL, an RDBMS like MySQL works with an operating system to implement a relational database in a computer's storage system, manages users, allows for network access and facilitates testing database integrity and creation of backups.”([wikipedia](#))

<https://www.mysql.com/>

## 8.4 Cloud provider comparison

Currently there are three main cloud providers dominating the space, amazon web services (most likely the most favored), microsoft azure (second favored), google cloud (not nearly as large as the other two but has some traffic).



At the core level all of these cloud providers provide a virtual machine service which is where the code will live AWS EC2, Azure VMs, google cloud compute engine. There are also serverless services, for running code without requiring a server for APIs such as AWS lambda, Azure functions, Google Cloud functions. Typically there will also be a set of services that provide cloud managed database engines, AWS RDS/DynamoDB, Azure cosmoDB/SQL database, google cloud SQL/Bigtable. There will also be a set of security tools, as well as scaling tools to make apps have top level security as well as being able to scale apps to meet any demand of traffic. There are many other services for all aspects of developing any type of application but these are some common ones. All 3 cloud platforms are useable for any project, however it should be noted that AWS is the most popular due to its extensive list of features for anything, its large community, extensive docs, and for pricing, AWS will give new users a year of free tier to use specific services for limited amounts of operations or time per month for a full year, while azure and google cloud will only give \$100-300 worth of credits which can be used to last a while but can also be gone in an instant.

### 8.4.1 Advantages & Disadvantages (AWS)

AWS provides a whole host of features that helps developers, IT, devops, analytics, debugging, and finance departments (probably more as well) do their jobs. AWS is even really useful just for learning to develop and deploy web apps into the real world even as an educational project. AWS offers a pay as you go structure basically only billing for services used and for the time they are used. AWS is well documented providing simple tutorials for doing.

Pros	Cons
<ul style="list-style-type: none"> <li>• Documentation <ul style="list-style-type: none"> <li>◦ AWS provides plenty of documentation on how to set things up as well as use services</li> </ul> </li> <li>• Community <ul style="list-style-type: none"> <li>◦ The AWS community is large enough that there is likely a solution to a problem already discovered and if not then there's likely someone that has the answer.</li> </ul> </li> <li>• Pay as you go <ul style="list-style-type: none"> <li>◦ Pay only for the resources being used at that time</li> </ul> </li> <li>• Fast deploy <ul style="list-style-type: none"> <li>◦ Rather than provisioning a server at some data center AWS can set up a new server in minutes rather than days</li> </ul> </li> <li>• Security and reliability</li> </ul>	<ul style="list-style-type: none"> <li>• Costs money, sometimes without the user knowing <ul style="list-style-type: none"> <li>◦ Without being careful about resources used it is possible to see sudden bills for services used</li> <li>◦ This likely wouldn't be an issue for companies that are looking to use cloud but for small businesses and students/hobbyists it can end with paying some money</li> </ul> </li> </ul>

### 8.4.2 Brief look at other popular cloud providers

#### 8.4.2.1 Microsoft Azure

“Microsoft Azure, often referred to as Azure, is a cloud computing service operated by Microsoft for application management via Microsoft-managed data centers. It provides software as a service, platform as a service and infrastructure as a service and supports many

different programming languages, tools, and frameworks, including both Microsoft-specific and third-party software and systems.”

([wikipedia](#))

<https://azure.microsoft.com/>

#### **8.4.2.2 Google cloud**

“Google Cloud Platform (GCP), offered by Google, is a suite of cloud computing services. Alongside a set of management tools, it provides a series of modular cloud services including computing, data storage, data analytics and machine learning. Registration requires a credit card or bank account detail. Google Cloud Platform provides infrastructure as a service, platform as a service, and serverless computing environments.” ([wikipedia](#))

### **8.5 Brief look at other popular tech stacks**

#### **8.5.1 MEAN/MEVN**

These 2 tech stacks are essentially similar to one of the tech stacks featured in the project, the MERN tech stack, the only difference with all 3 of these stacks is the front-end framework being used. MEVN uses vue, and MEAN, which uses angular, both of these frameworks are popular as well.

<https://www.geeksforgeeks.org/introduction-to-mean-stack/>

<https://www.geeksforgeeks.org/what-is-mevn-stack/>

#### **8.5.2 LAMP**

The LAMP stack consists of Linux (server operating system), Apache (web server), MySQL (database), PHP (programming language).

<https://www.ibm.com/cloud/learn/lamp-stack-explained>

## 9. Conclusion

The total amount of time spent on this project is about 6 months which includes learning python and javascript mostly from scratch, learning all of the frameworks, and other tools used (docker, postman, aws). The overall goal of this project was to see if it was possible to pivot from game development over to web development in about the 6 month period as well as expose a little bit of devops into the mix. Overall through the many many resources used to get to the point where this was doable it was also learned that programming has become this overall large community of developers that help one another whether it be it from stackoverflow, packages made for tedious tasks which are meant to cut development time down, or the average youtube/udemy videos. With all of these resources it has opened up the opportunity for anyone to get into programming as a hobby or job and has made a big impact on.

As for specific tools/technologies for using in a tech stack. Honestly, use whatever is the most comfortable for you as a developer because while efficiency will come into play it is better overall for the development cycle to be as smooth and fast as possible than for the deployment of the app to be as efficient as possible, optimizations can be implemented later on.

Some favored tools for development not mentioned in this document but can be useful, jetbrains (good suite of IDEs for any development needs, intellisense, plenty of extensions, unfortunately is a subscription based unless using community versions which have less functionality). VMs, virtual machines for linux are beneficial for development as it is easier to set up a work environment. WSL, windows subsystem for linux, is also a good alternative to vms as it provides a linux system that can be accessible through a cli and run like a linux machine.

Some favored frameworks for web development. Svelte is a new framework just released around 2019 and it has gained some popularity, it could someday be a top contender and worth taking a look at.

Thank you for reaching the end of this paper, hopefully some useful knowledge was gained from reading this. If there is anything that is incorrect please reach out on [linkedin](#), and let me know I always welcome criticism.