A Day in the Life of a Developer — Building a Dashboard App with SQL, Node.js, Django, and Next.js

Andrew Baisden · Follow
Published in JavaScript in Plain English
32 min read · Jan 25, 2024

Introduction

Today we will be getting a quick insight into what it's like to work on projects as a developer as part of a team inside of a company. We are going to be building a dashboard application which has two backends. One will be created using Node.js and the other using Django. The frontend will be created using Next.js and Tailwind. The dashboard is used for adding animes to a database. You can perform full CRUD actions, and filter by release year and the data will be displayed in a chart.

This project will be divided into these 3 sections:

1. Jira project workflow
2. Building the backend
3. Building the frontend

You can find the full codebase on GitHub here Anime Dashboard App.

With our introduction out of the way let's move on to the Jira project workflow.

Jira project workflow

The Product Owner and the Development Team, are often facilitated by the Scrum Master when setting up a project. This stage can be different for each company and this is just one example of how it could be done.

1. Creating a Jira project
2. Create epics, stories, bugs, and tasks in the backlog section and then decide which ones to add to the next sprint
3. Use confluence for documentation and project management
4. Start the sprint

A developer could have several tools installed which could include:

- A GitHub repo for the project
- Slack or alternative
- Jira or alternative
- IDE or Code editor and development environment setup for work
- API testing tool for the backend (Postman, Insomnia, REST Client, Swagger etc…)

Now let's see a quick example of what a developer's workflow could look like in a project in the next section.

Developer workflow

This example could apply to any developer who is working on the team. They would need to:

1. Pick up a Jira ticket from the current sprint or backlog and then assign the ticket to yourself and work on it by changing its status from to do to in progress
2. Create a new development branch for the ticket with a naming convention that matches the ticket so it is easy to find for example feature-123, bug-123, task-123 etc…
3. Work on the story and when you are done push your changes to the repo so that it goes through the CI/CD workflow by creating a pull request (PR) and changing the status to "in review"
4. Add the PR to the Jira story as a web link or similar
5. Share the story you worked on in the team chat app like Slack with the appropriate message for another developer to review your changes

For example "PR for https://project/browse/feature-123 is ready for review"

Another developer will check the PR, approve it and merge it if all the tests pass and there are no errors.

Now repeat the process by returning to step 1 and picking up another ticket to work on.

With that brief explanation out of the way let's move on to the prerequisites for this project.

Prerequisites

- PostgreSQL installed
- MySQL installed
- SQLite installed
- Database Management System installed (Azure Data Studio or alternative)
- Node and npm installed
- Python installed with virtualenv or an alternative

Now with our prerequisites done let's begin!

Building the backend

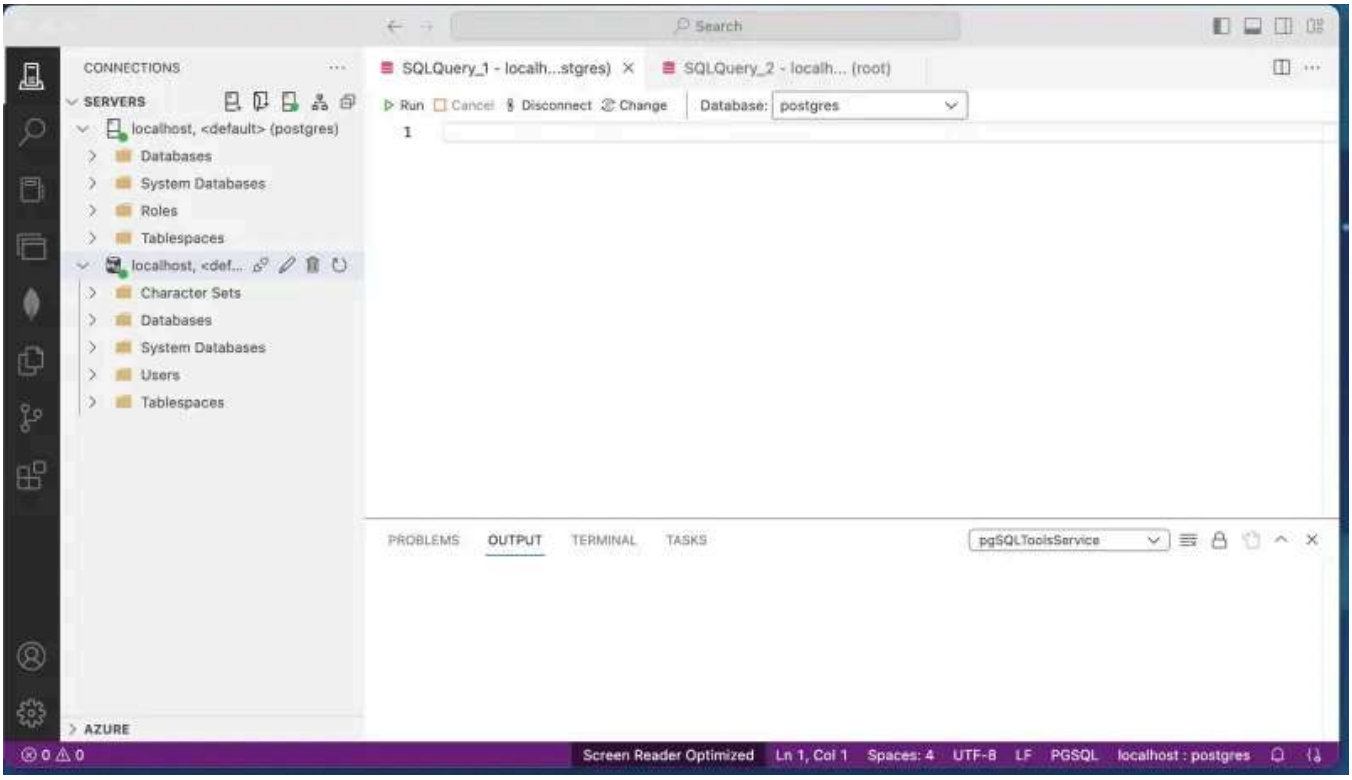
Let's just assume that a backend developer has picked up a ticket from Jira for setting up the backend architecture. The first stage would be to set up the SQL databases.

Setup a SQL database

Our application will only need one database however I have included steps for creating a database for PostgreSQL, MySQL and SQLite so you can use whichever one you want.

PostgreSQL and MySQL database setup

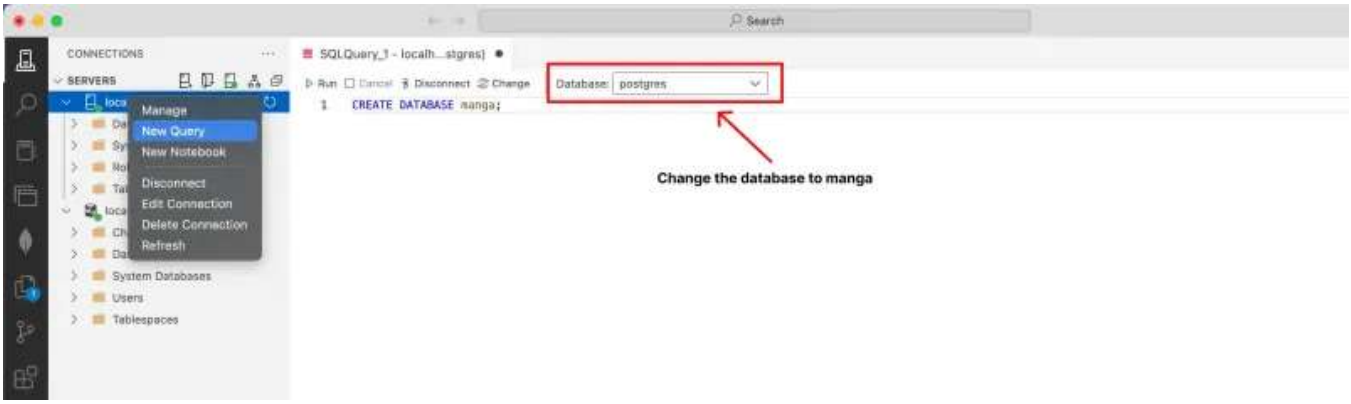
Connect to the PostgreSQL and MySQL databases using Azure Data Studio or the database management tool of your choice. This is how my setup looks in Azure Data Studio if you use Visual Studio Code you will get used to the design easily because they are both made by Microsoft.



This is how you would create a database using Azure Data Studio. Please make sure that you change the database from Postgres or MySQL after you run the SQL query below. We want to create our tables in the manga database, not the default.

Now create a SQL database by running the query below in your database application of choice:

```
CREATE DATABASE manga;
```



Next, select the database you just made and create a table that has a primary key. The SQL for primary keys differs for both PostgreSQL and MySQL however the SQL for all other properties remains the same.

These SQL queries essentially create two tables called `anime` and `anime_details` inside of our manga database. The column and data types are self-explanatory as you can see below. Both tables have a column called `anime_id` which we will use later for joining both tables together using a SQL join clause.

Copy, paste and run the SQL queries into their corresponding database:

PostgreSQL table setup

```
CREATE TABLE anime (
id SERIAL PRIMARY KEY,
anime_id VARCHAR(10),
anime_name VARCHAR(100),
anime_release VARCHAR(10)
);
```

```
CREATE TABLE anime_details (
id SERIAL PRIMARY KEY,
anime_id VARCHAR(10),
anime_genre VARCHAR(50),
anime_rating INT
);
```

MySQL table setup

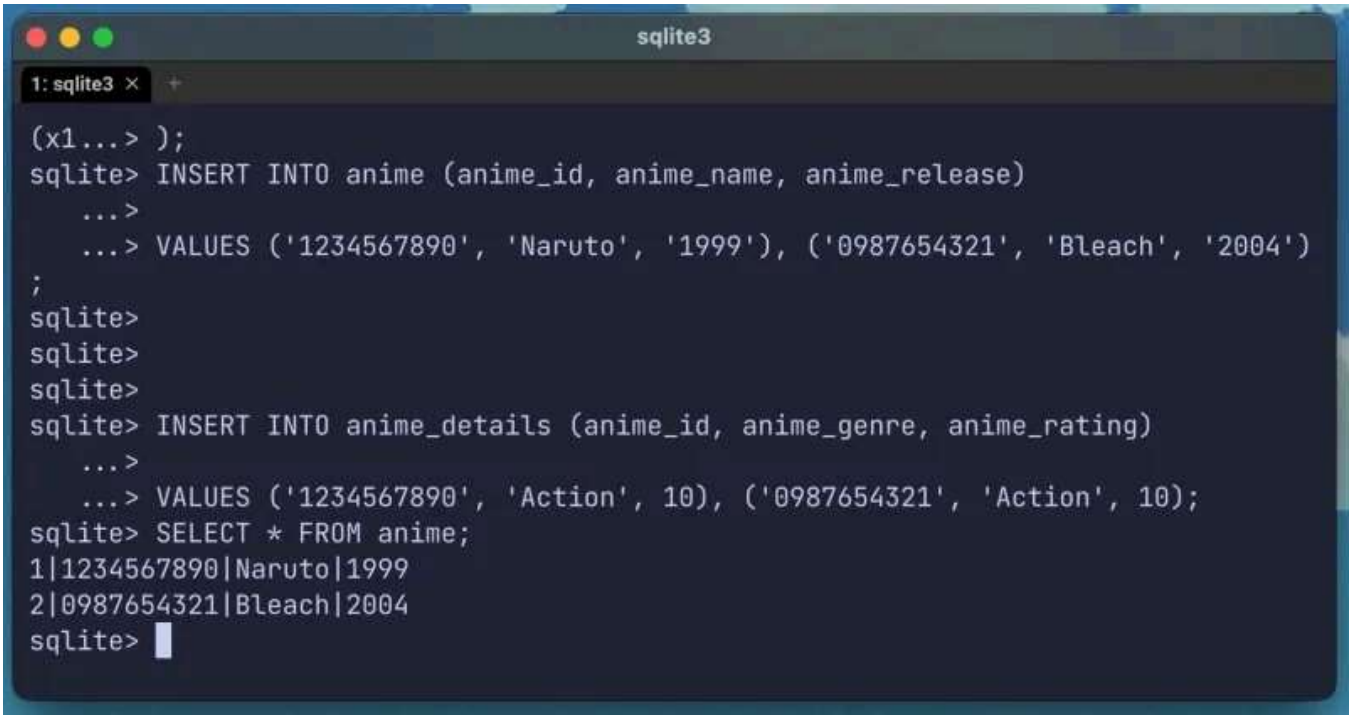
```
CREATE TABLE anime (
id INT AUTO_INCREMENT PRIMARY KEY,
anime_id VARCHAR(10),
anime_name VARCHAR(100),
anime_release VARCHAR(10)
);
```

```
CREATE TABLE anime_details (
id INT AUTO_INCREMENT PRIMARY KEY,
anime_id VARCHAR(10),
anime_genre VARCHAR(50),
anime_rating INT
);
```

In the next section, we will repeat the same process but this time for an SQLite database. Before we begin create a folder on your computer for the dashboard project called **anime-dashboard-app**.

SQLite database setup

Azure Data Studio does not currently support SQLite databases. So you can either use the command line to manage your database or find a GUI application that supports SQLite. This is how the SQLite database looks when using the command line:



Use your terminal application to `cd` into the **anime-dashboard-app** folder and then use the command line to create an SQLite database using the code below:

```
sqlite3 manga.db
```

We now need to create some tables for our SQLite database. They are going to be the same as the ones we created for our PostgreSQL and MySQL databases however the SQL query will be different. Copy and paste this code into your terminal to run the query:

```
CREATE TABLE anime (
id INTEGER PRIMARY KEY AUTOINCREMENT,
anime_id VARCHAR(10),
anime_name VARCHAR(100),
anime_release VARCHAR(10)
);
```

```
CREATE TABLE anime_details (
id INTEGER PRIMARY KEY AUTOINCREMENT,
anime_id VARCHAR(10),
anime_genre VARCHAR(50),
anime_rating INT
);
```

With our database and tables set, the next stage will be to add some data to our databases which we are going to do in the next section.

Creating SQL statements for our SQL databases

It is now time for us to create the SQL queries which we will be using to give us full CRUD (Create, Read, Update, Delete) functionality in our databases. These SQL queries will work with all of our SQL databases.

Let's start with the SQL query for creating data.

CREATE

The SQL query below inserts data into both tables. All you have to do is run the query in the databases we created like before:

```
INSERT INTO anime (anime_id, anime_name, anime_release)
VALUES ('1234567890', 'Naruto', '1999'), ('0987654321', 'Bleach', '2004');
```

```
INSERT INTO anime_details (anime_id, anime_genre, anime_rating)
VALUES ('1234567890', 'Action', 10), ('0987654321', 'Action', 10);
```

Now that we have data in our databases let's check to see if we can access them with the read statement queries up next.

READ

If you run these SQL queries in our databases you should be able to see the data that we added in the tables:

```
SELECT * FROM anime;
```

```
SELECT * FROM anime_details;
```

And here is how we will read the data using a condition like the ID:

```
SELECT * FROM anime WHERE id = '1';
```

```
SELECT * FROM anime_details WHERE id = '1';
```

With this code below, we can do an inner join and combine both tables on the anime_id key. This SQL query selects unique keys for the table so there is no duplication:

```
SELECT anime.anime_id, anime_name, anime_genre, anime_release anime_rating FROM
INNER JOIN anime_details
ON anime.anime_id = anime_details.anime_id;
```

If we wanted everything including duplicates we could use the wildcard:

```
SELECT * FROM anime
INNER JOIN anime_details
ON anime.anime_id = anime_details.anime_id;
```

Up next will be our updated SQL queries.

UPDATE

To update data in the tables we could use these SQL queries in our database:

```
UPDATE anime
SET anime_name = 'Dragonball Z', anime_release = '1989'
WHERE anime_id = '1234567890';
```

```
UPDATE anime_details
SET anime_genre = 'Adventure', anime_rating = 9
WHERE anime_id = '1234567890';
```

Now we will learn how to do updates using inner joins.

Performing an INNER JOIN

Inner joins allow us to combine two tables by referencing records that have matching values in the same tables.

This is how we would do an INNER JOIN in PostgreSQL:

```
UPDATE anime
SET anime_name = 'Attack on Titan'
FROM anime_details
WHERE anime.anime_id = anime_details.anime_id
AND anime.anime_id = '0987654321';
```

```
UPDATE anime_details
SET anime_genre = 'Fantasy'
```

```
WHERE anime_id = '0987654321';
```

And this is how we would do an INNER JOIN in MySQL:

```
UPDATE anime
INNER JOIN anime_details
ON anime.anime_id = anime_details.anime_id
SET anime.anime_name = 'Attack on Titan', anime_details.anime_genre = 'Fantasy'
WHERE anime.anime_id = '0987654321';
```

SQLite does not support the use of INNER JOIN directly in the UPDATE syntax so this is how we would achieve the desired result by using a subquery instead:

```
UPDATE anime
SET anime_name = 'Attack on Titan'
WHERE anime_id = '0987654321';
```

```
UPDATE anime_details
SET anime_genre = 'Fantasy'
WHERE anime_id = '0987654321';
```

Lastly, let's see how we can delete data from our databases.

DELETE

To delete data from a table use this type of SQL query:

```
DELETE FROM anime
WHERE anime_id = '1234567890';
```

We are done with the database section up next is the section where we are going to build our backend API.

Build a backend API with full CRUD

This section we be split into two. First, we shall create a Node.js backend and then one for Django.

Node.js backend project setup

For creating the backend and connecting it to our SQL databases we will be using Prisma for setting up relational databases.

You should still be inside the folder for **anime-dashboard-app** if not make sure that you are and then set up a project with Prisma using these commands:

```
mkdir manga-backend-express
cd manga-backend-express
npm init -y
npm i express nodemon dotenv cors prisma
```

Now get the Prisma CLI running with this command:

```
npx prisma
```

And now lets setup our Prisma project with this command:

```
npx prisma init
```

Now open the project in your code editor, if you use VS Code and have it setup to open from the terminal you can use this command:

```
code .
```

Ok great, next we will connect to our SQL database using Prisma.

Connecting to the SQL databases with Prisma

Use the following setups to connect to the different databases. It is only possible to connect to one of these databases and as long as you completed all the steps earlier you should have 3 databases setup with data. So choose one of the databases to connect to in the setups below:

PostgreSQL setup

The file is located at `prisma/schema.prisma`

```
datasource db {
provider = "postgresql"
url = env("DATABASE_URL")
}
```

The `.env` should be in the root folder and it will hold our environment variables. This is the connection string that we will use to connect to the database. It is possible that your configuration could be different so locate the correct path:

```
DATABASE_URL="postgresql://postgres:@localhost:5432/manga?schema=public"
```

MySQL setup

The file is located at `prisma/schema.prisma`

```
datasource db {
provider = "mysql"
url = env("DATABASE_URL")
}
```

The `.env` should be in the root folder and it will hold our environment variables. This is the connection string that we will use to connect to the database. It is possible that your configuration could be different so locate the correct path:

```
DATABASE_URL="mysql://root:@localhost:3306/manga"
```

SQLite setup

The file is located at `prisma/schema.prisma`

```
datasource db {
provider = "sqlite"
url = env("DATABASE_URL")
}
```

The `.env` should be in the root folder and it will hold our environment variables. This is the connection string that we will use to connect to the database. It is possible that your configuration could be different so locate the correct path:

Create a folder inside of the **manga-backend** directory called `database` and then move the SQLite database we created earlier called `manga.db` and put it inside of the `database` folder.

```
DATABASE_URL="file:../database/manga.db"
```

We are making good progress so let's move on to the next section where we will create our database schema.

Creating the database schema

We have already created our database so we can use the commands below to create our models and schema automatically. Run these commands inside of our **manga-backend** project to create a schema for our database based on the tables that we created for our databases.

```
npx prisma db pull
```

```
npx prisma generate
```

Alternatively, we can use Prisma Migrate which means that we have to create the database models and schema manually.

Creating the Express server with CRUD endpoints

First update the `.env` file by adding the port and development environment variables:

```
PORT="8000"
ENVIRONMENT="development"
```

```
Now create an index.js file in the root folder for manga-backend-express and ac
```

```
const express = require('express');
const { PrismaClient } = require('@prisma/client');
require('dotenv').config();
const cors = require('cors');
const prisma = new PrismaClient();
```

```
const app = express();

app.use(cors());
app.use(express.urlencoded({ extended: false }));
app.use(express.json());

// GET ALL

app.get('/api/anime', async (req, res) => {
  const allAnime = await prisma.anime.findMany();
  console.log(allAnime);
```

```
  res.json(allAnime);
});

// GET ONE

app.get('/api/anime/:id', async (req, res) => {
  const { id } = req.params;

  const anime = await prisma.anime.findUnique({
    where: { id: Number(id) },
  });

  res.json(anime);
  console.log(anime);
});

// POST

app.post('/api/anime', async (req, res) => {
  const { anime_id, anime_name, anime_release } = req.body;

  const result = await prisma.anime.create({
    data: {
      anime_id,
      anime_name,
      anime_release,
    },
  });

  res.json({ msg: 'Anime added' });
});

// UPDATE

app.put('/api/anime/:id', async (req, res) => {
  const { id } = req.params;
  const { anime_id, anime_name, anime_release } = req.body;

  const post = await prisma.anime.update({
    where: { id: Number(id) },

    data: {
      anime_id,
      anime_name,
      anime_release,
    },
  });

  res.json({ msg: `Anime ${id} updated` });
});

// DELETE

app.delete('/api/anime/:id', async (req, res) => {
  const { id } = req.params;

  const post = await prisma.anime.delete({
    where: { id: Number(id) },
  });
```

```
  res.json({ msg: `Anime ${id} deleted` });
});

// GET: Inner join (using raw SQL query syntax)

app.get('/api/joinedtables', async (req, res) => {
  const users = await prisma.$queryRaw`
SELECT * FROM anime
INNER JOIN anime_details
ON anime.anime_id = anime_details.anime_id;
`;

  res.json(users);
  console.log(users);
});

const port = process.env.PORT || 8000;

if (process.env.ENVIRONMENT === 'development') {
  app.listen(port, () =>
    console.log(`Server running on port ${port},
http://localhost:${port}`)
  );
}
```

Lastly, add these run scripts to the `package.json` file so that we can run our server. We have one that uses node and another for our development server that uses nodemon so our server can auto refresh without having to stop and start the server each time we make changes.

```
"scripts": {
"start": "node index.js",
"dev": "nodemon index.js"
},
```

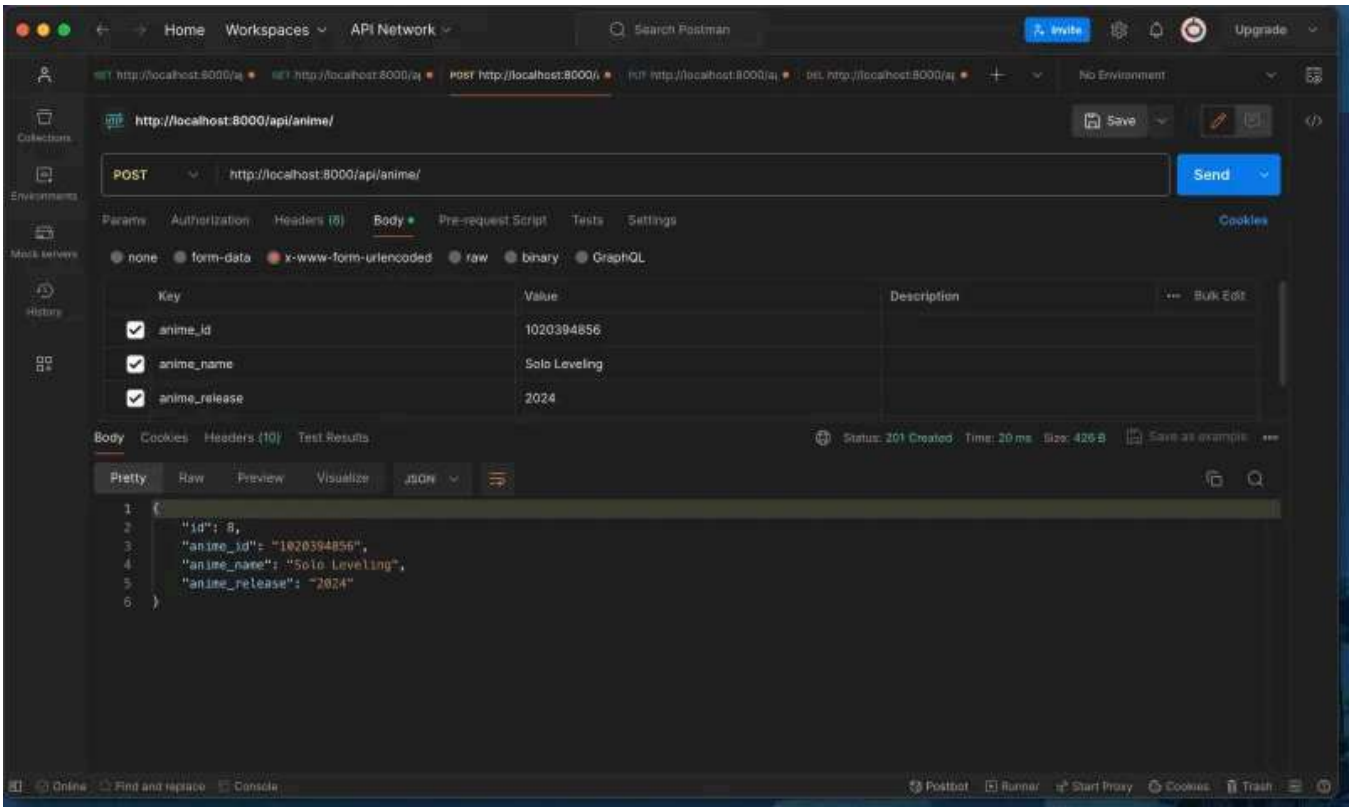
Run the backend with either one of these commands and use an API testing tool like Postman to test the endpoints:

```
npm run start
npm run dev
```

Take a look at the examples below to see the data and endpoints.

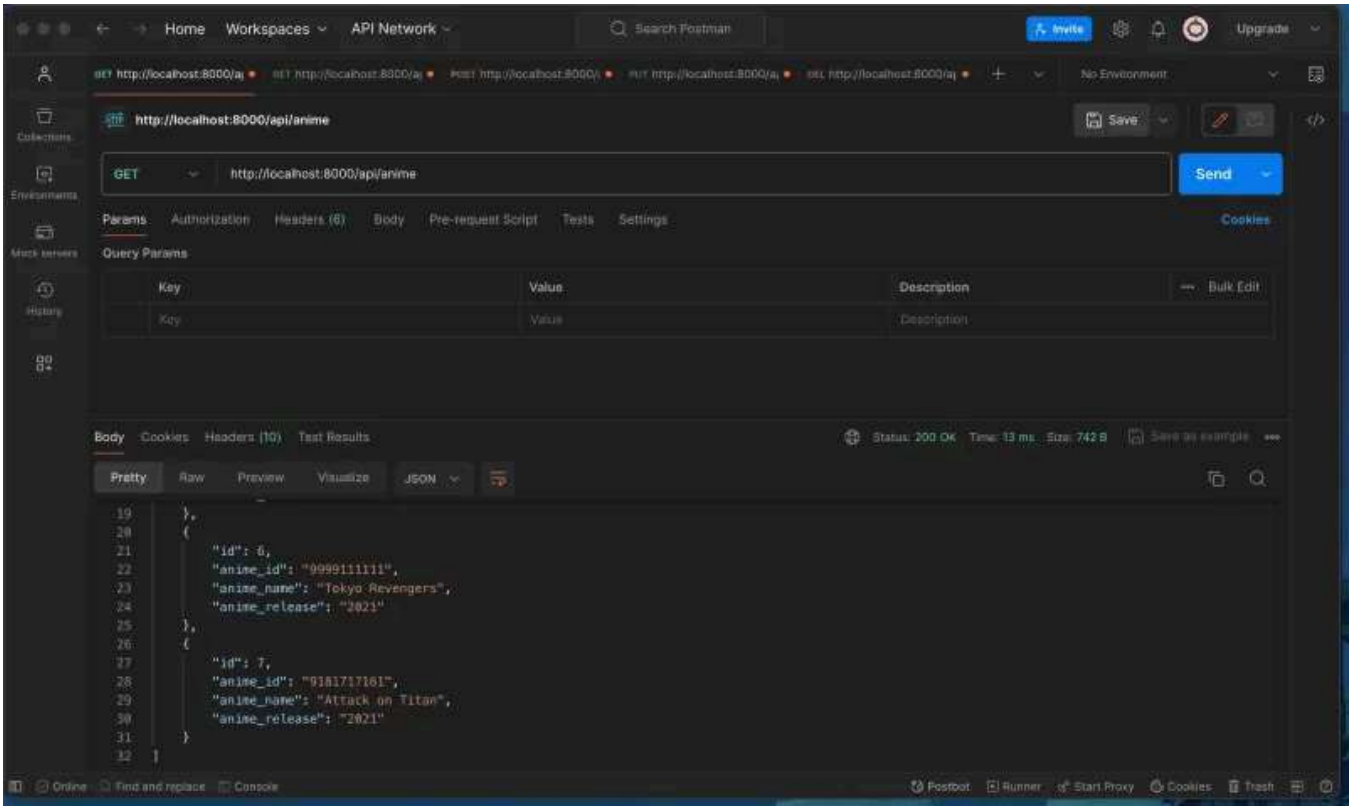
POST data endpoint

Endpoint: http://localhost:8000/api/anime/



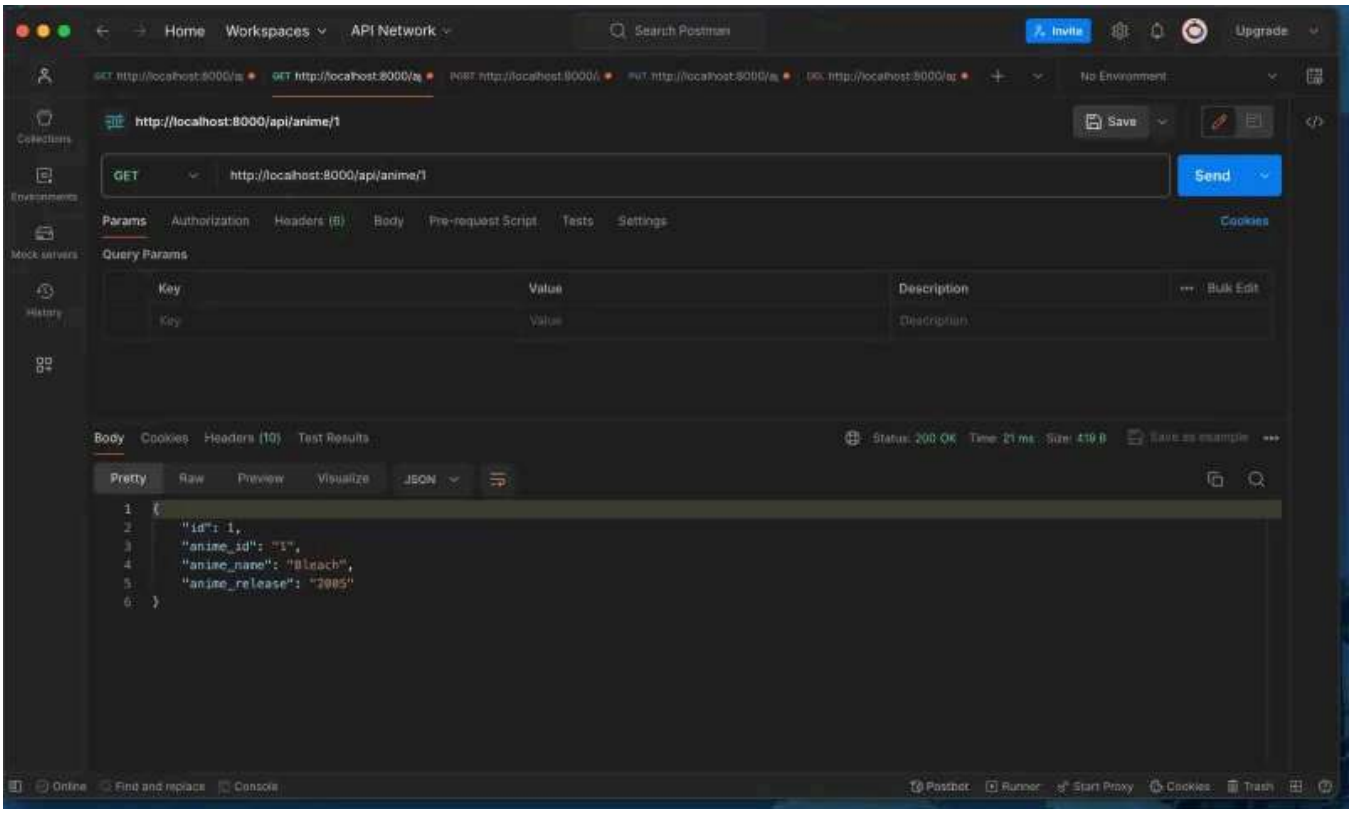
GET all data endpoint

Endpoint: http://localhost:8000/api/anime



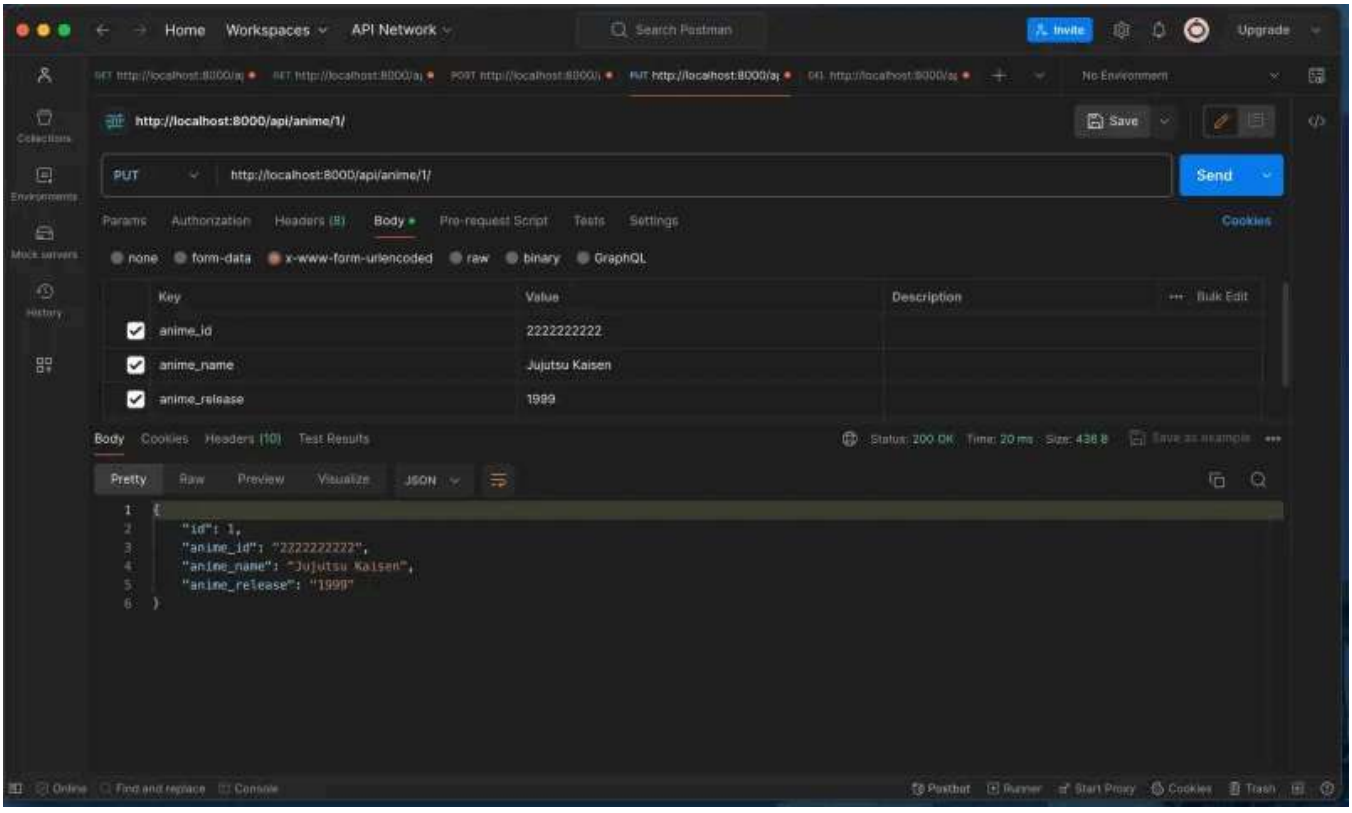
GET by ID data endpoint

Endpoint: http://localhost:8000/api/anime/1



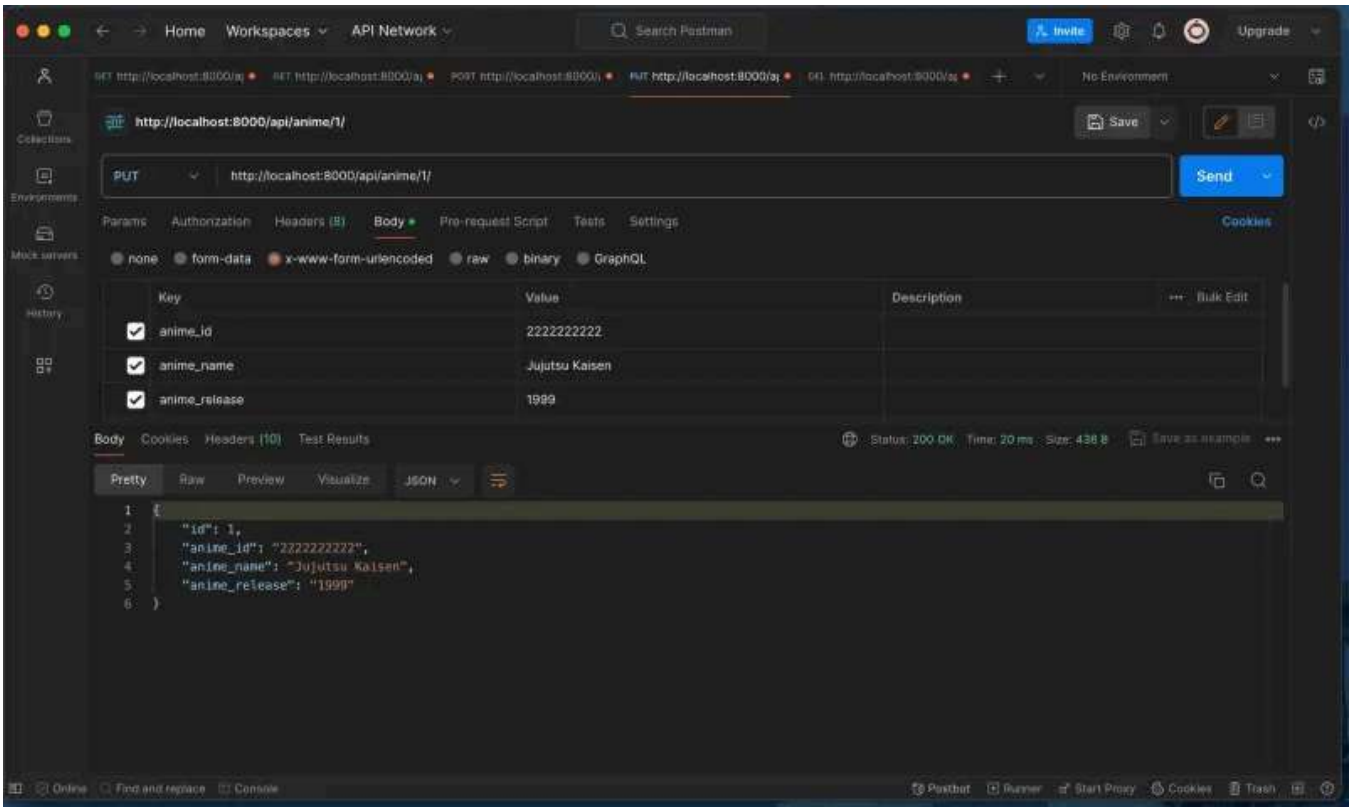
PUT data endpoint

Endpoint: http://localhost:8000/api/anime/1



Delete data endpoint

Endpoint: http://localhost:8000/api/anime/1



Our Express backend should be connected to one of our SQL databases. In the next section, we are going to create a Django backend.

Django backend project setup

Before we begin we should first stop our Express server from running because both servers are going to use port 8000 and only one server can run on that port, not both. So make sure that you are in the root folder for **anime-dashboard-app** and then create a Python virtual environment with these commands using the terminal:

> You might need to use a different Python command for running these upcoming Python code scripts, it all depends on how your development environment is set up. So you might have to use `python` or `python3` to run the commands. The same applies to using `pip` or `pip3` for installing packages.

```
virtualenv manga-backend-django
source manga-backend-django/bin/activate
pip3 install django
```

```
pip3 install djangorestframework
pip install django-cors-headers
```

With these commands, we are setting up a Python virtual environment and installing the packages for Django and the Django Rest Framework. Without a virtual environment, these would be installed globally on our computers.

Run this command from to root of the **anime-dashboard-app** folder to activate the Python virtualenv environment:

```
source manga-backend-django/bin/activate
```

When you want to deactivate the environment you can use this command. We don't need to do that right now.

```
deactivate
```

Connecting to the SQL databases with Django

Connecting to the different databases requires us to install drivers. Install the following drivers for the databases you intend to connect to. We only need to do this for PostgreSQL and MySQL because Django should already work with SQLite databases.

```
## PostgreSQL Databases
pip3 install psycopg2 psycopg2-binary
```

```
## MySQL Databases
pip3 install mysqlclient
```

Now we need to create a Django application so run the commands below to do that in the same folder:

```
cd manga-backend-django
django-admin startproject manga
cd manga
```

Now open the **manga-backend-django** project in your code editor if you have not done so already because we are going to be adding code to the files in our Django codebase.

Next, we need to connect to our database so use the example database connection examples below as a reference for each database and configure them for your ones. Open the `manga/manga/settings.py` file in your project and find the databases section. The connection object should be very similar to the one that we used for our Express app with Prisma.

PostgreSQL

```
DATABASES = {
'default': {
'ENGINE': 'django.db.backends.postgresql',
'NAME': 'manga',
'USER': 'postgres',
'PASSWORD': '',
'HOST': 'localhost',
'PORT': '5432',
}
}
```

MySQL

```
DATABASES = {
'default': {
'ENGINE': 'django.db.backends.mysql',
'NAME': 'manga',
'USER': 'root',
```

```
'PASSWORD': '',
'HOST': 'localhost',
'PORT': '3306',
}
}
```

SQLite

When setting it up for SQLite databases we have to ensure that our SQLite database is in the correct folder. The base directory is the root folder for the project so for example `manga-backend-django/database`. You can just copy the database folder from the **manga-backend-express** project and put it in the root folder for this project.

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

Great now let's work on this next step which will be to do the migration and create a super user for the database.

Doing a Python migration and creating a super user for our database

Make sure that you are inside the `manga` folder with the `manage.py` file and run the following commands. We will use Python migration to set all of the tables for our database:

```
python3 manage.py migrate
```

Ok now let's create a super user for the database so that we can access it. Run this command and go through the steps. You are required to have a Username and Password at least:

```
python3 manage.py createsuperuser
```

Right with that setup completed let's finish it off by creating a Django server with CRUD endpoints.

Creating the Django server with CRUD endpoints

We first need to create an app called API so run this code from the same folder with the `manage.py` file in it to create it:

```
django-admin startapp api
```

Now configure the Django Rest Framework in `settings.py` by adding these packages to the `INSTALLED_APPS = []` array:

```
'api.apps.ApiConfig',
'rest_framework',
'corsheaders',
```

To avoid CORS errors on the front end we will need to add some more configurations to our `settings.py` file. First, add these modules to the existing MIDDLEWARE array in the file:

```
'django.middleware.common.CommonMiddleware',
'corsheaders.middleware.CorsMiddleware',
```

Then add these CORS configurations somewhere in the file:

```
CORS_ALLOWED_ORIGINS = [
    "http://localhost:3000",  # Add your frontend URL here
```

```
  ]
```

```
CORS_ALLOW_METHODS = [
    'DELETE',
    'GET',
    'OPTIONS',
    'PATCH',
    'POST',
    'PUT',
]

CORS_ALLOW_HEADERS = [
    'accept',
    'accept-encoding',
    'authorization',
    'content-type',
    'dnt',
    'origin',
    'user-agent',
    'x-csrftoken',
    'x-requested-with',
]
```

Now we need to create our database models so add this code to the `api/models.py` file:

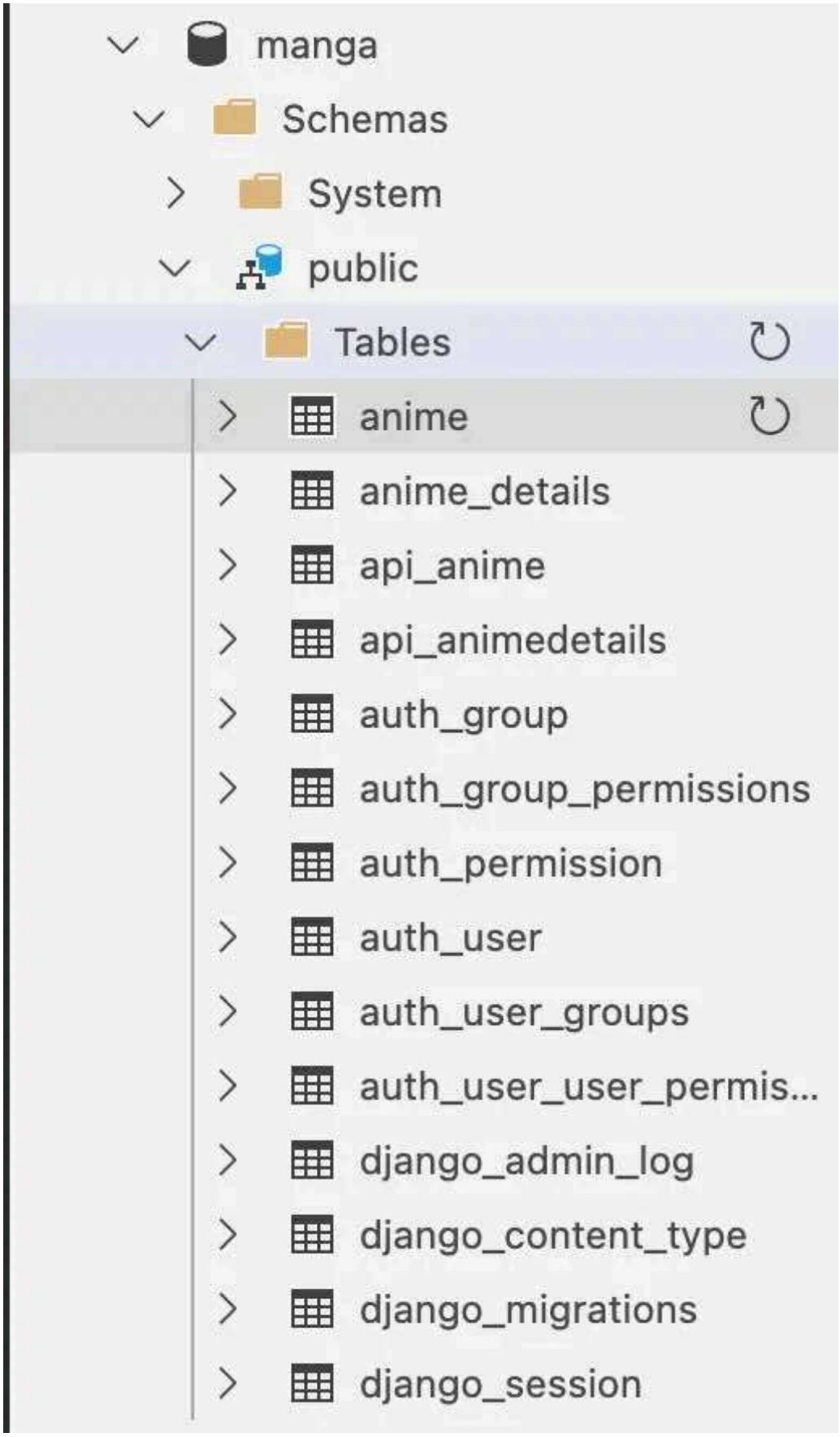
```
from django.db import models
```

```
class Anime(models.Model):
    id = models.AutoField(primary_key=True)
    anime_id = models.CharField(max_length=10, null=True)
    anime_name = models.CharField(max_length=100, null=True)
    anime_release = models.CharField(max_length=10, null=True)

    def __str__(self):
        return self.anime_name

class AnimeDetails(models.Model):
    id = models.AutoField(primary_key=True)
    anime_id = models.CharField(max_length=10, null=True)
    anime_genre = models.CharField(max_length=50, null=True)
    anime_rating = models.IntegerField(null=True)

    def __str__(self):
        return self.anime_id
```

After defining the models, run the following commands to create the necessary database tables:

```
python manage.py makemigrations
python manage.py migrate
```

This will create “api_anime” and “api_animedetails” tables in your database along with a few others. Existing tables for “anime” and “anime_details” already exist so it will create new ones for the models. You can see what that looks like inside of Azure Data Studio here:



The next step is to create the Serializers and API views. Serializers transform complicated data, like querysets and model instances, to native Python datatypes that can be readily displayed as JSON, XML, or other content types. Create a file at `api/serializers.py` and add this code:

```
from rest_framework import serializers
from .models import Anime, AnimeDetails
```

```
class AnimeSerializer(serializers.ModelSerializer):
    class Meta:
        model = Anime
        fields = '__all__'

class AnimeDetailsSerializer(serializers.ModelSerializer):
    class Meta:
        model = AnimeDetails
        fields = '__all__'
```

And now add this code to the `api/views.py` file which is where our database data will be returned:

```
from django.shortcuts import render
# Create your views here.
from rest_framework import viewsets
from .models import Anime, AnimeDetails
from .serializers import AnimeSerializer, AnimeDetailsSerializer
from rest_framework.views import APIView
from rest_framework.response import Response
from django.db import connection
```

```
class AnimeViewSet(viewsets.ModelViewSet):
    queryset = Anime.objects.all()
    serializer_class = AnimeSerializer

class AnimeDetailsViewSet(viewsets.ModelViewSet):
    queryset = AnimeDetails.objects.all()
    serializer_class = AnimeDetailsSerializer

class JoinedTablesView(APIView):
    def get(self, request):
        try:
            with connection.cursor() as cursor:
                cursor.execute("""
                SELECT * FROM api_anime
                INNER JOIN api_animedetails ON api_anime.anime_id =
api_animedetails.anime_id;
                """)
                columns = [col[0] for col in cursor.description]
                data = [dict(zip(columns, row)) for row in
```

```
cursor.fetchall()]
                return Response(data)

        except OperationalError as e:
            # Log the exception or handle it appropriately
            return Response({"error": f"Database error: {e}"},
status=500)
```

Ok, next we will create a file for `api/urls.py` with this code which is where we will be doing the routing for our URL endpoints:

```
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import AnimeViewSet, AnimeDetailsViewSet, JoinedTablesView
```

```
router = DefaultRouter()
router.register(r"anime", AnimeViewSet)
router.register(r"anime-details", AnimeDetailsViewSet)

urlpatterns = [
    path("", include(router.urls)),
    path("joinedtables/", JoinedTablesView.as_view(), name="joined-
tables"),
]
```

Finally, replace the code in the `manga/urls.py` file with this code which contains our root endpoints:

```
from django.contrib import admin
from django.urls import path, include
```

```
urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/", include("api.urls")),  # Include the 'api' app's
URLs
]
```

After defining the views and URLs, we must run the following commands to create the necessary database tables. If you get any errors in the console they are most

likely related to Python indentation, so try to format the files which are causing problems:

```
python3 manage.py makemigrations
python3 manage.py migrate
```

That's it we should be done with this section so, run the Django development server with the code below:

```
python3 manage.py runserver
```

The Django API should be accessible at http://127.0.0.1:8000/api/anime/, http://127.0.0.1:8000/api/anime-details/ and http://127.0.0.1:8000/api/joinedtables/ for performing CRUD operations on your `Anime` and `AnimeDetails` models. As you can see Django has a built-in API testing framework called the Django Rest Framework so you don't need to use an external API testing tool like Postman.

We can also access the database tables in our Database Management System (Azure Data Studio) and perform CRUD on them. Just remember that in our Django application, the database tables will be called `api_anime` and `api_animedetails`.

Note: If you see the errors below when using your API testing with the Django backend then you need to add a trailing slash at the end of your URL endpoint.

> RuntimeError: You called this URL via POST, but the URL doesn't end in a slash and you have APPEND_SLASH set. Django can't redirect to the slash URL while maintaining POST data. Change your form to point to localhost:8000/api/anime/ (note the trailing slash), or set APPEND_SLASH=False in your Django settings.
>
> RuntimeError: You called this URL via PUT, but the URL doesn't end in a slash and you have APPEND_SLASH set. Django can't redirect to the slash URL while maintaining PUT data. Change your form to point to localhost:8000/api/anime/1/ (note the trailing slash), or set APPEND_SLASH=False in your Django settings.

If you encounter any new errors then they are most likely caused by incorrect Python indentation. We have completed this section, now let's work on the last section which is where we are going to build the frontend.

Building the frontend

Ok, this time around let's just assume that a frontend developer has picked up a ticket from Jira for creating the frontend user interface. The first stage would be to scaffold a project using a framework in this case it will be Next.js.

In this example, we will integrate a chart library on the frontend that connects to the backend which has full CRUD functionality.

Here is a quick run-through of what we will be doing:

1. Create a React project using Next.js
2. Setup Jest, and React Testing Library with an option for normal tests and testing coverage
3. Build components and use a TDD workflow

Now that we have an overview of the tasks let's begin.

Create a React project using Next.js

The first thing we need to do is use the command line to get back to the root folder for our project which is **anime-dashboard-app**. Now create a Next.js project and make sure that you select **Yes** for Tailwind CSS because we will need it later. You can use the settings I used here:

```
npx create-next-app manga-client
```

✓ Would you like to use TypeScript? … No / **Yes**
✓ Would you like to use ESLint? … **No** / Yes
✓ Would you like to use Tailwind CSS? … No / **Yes**
✓ Would you like to use `src/` directory? … No / **Yes**
✓ Would you like to use App Router? (recommended) … No / **Yes**
✓ Would you like to customize the default import alias (@/*)? … **No** / Yes

Let's quickly discuss the topic of separation of concerns. In computer science, separation of concerns is a design idea that divides a computer program into different parts. Each part handles a distinct issue, a collection of data that impacts the coding of a computer program. Taking this into consideration we are going to ensure that our folders are well defined and that there is a separate folder which holds all of our tests.

We will now set up the project folder structure, for our components, custom hooks and test files. Every component should have a test. Make sure that you run this script from the root of the **manga-client** folder in your terminal:

```
mkdir __tests__
touch __tests__/FormDelete.test.js
touch __tests__/FormPostUpdate.test.js
touch jest.config.mjs
cd src/app
mkdir components hooks
mkdir components/FormDelete
mkdir components/FormPostUpdate
cd components
touch FormDelete/FormDelete.js
touch FormPostUpdate/FormPostUpdate.js
cd ..
touch hooks/useFetch.js hooks/useFetchSWR.js hooks/usePost.js hooks/useUpdate.j
cd ../..
```

Next, we will install the dependencies for our project so install the required packages with these commands:

```
npm i --save-dev jest @types/jest @testing-library/jest-dom @testing-library/re
```

Ok now if your project is not open in a code editor do so now. The next step is to add these Jest run scripts to the `package.json` file which we will use for testing our components:

```
  "test": "jest --watchAll",
  "coverage": "jest --collect-coverage --collectCoverageFrom='./src/app/component
```

With our test scripts set up now, we need to add this code to the `jest.config.mjs` file which we created so that we have a working Jest configuration ready for when we want to run some tests:

```
import nextJest from 'next/jest.js';

const createJestConfig = nextJest({
  // Provide the path to your Next.js app to load next.config.js and
.env files in your test environment
  dir: './',
});
// Add any custom config to be passed to Jest
/** @type {import('jest').Config} */
const config = {
  // Add more setup options before each test is run
  // setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],
  testEnvironment: 'jest-environment-jsdom',
  coverageThreshold: {
    global: {
      branches: 80,
      functions: 80,
      lines: 80,
      statements: -10,
    },
  },
};

// createJestConfig is exported this way to ensure that next/jest
can load the Next.js config which is async

export default createJestConfig(config);
```

These test run scripts should be working. If you run them now the tests will fail because we have not added any code to our test files yet.

```
npm run test
```

```
npm run coverage
```

Ok, our frontend project has been set up all that is left is to build components and tests and then our anime dashboard application will be complete!

Build components

Starting with our components we will begin by adding the upcoming code to our component files.

Up first will be our form delete component which is inside the components folder. It is located at `FormDelete/FormDelete.js`. This component is self-explanatory it will be used for deleting animes from our database.

```
import { useState } from 'react';
import { useDelete } from '@/app/hooks/useDelete';
```

```
export default function FormDelete() {
  const [id, setId] = useState('');
  const { deleteRequest } = useDelete();
  const handleDeleteForm = async (e) => {
    e.preventDefault();
    if (id === '') {
      console.log('Form needs and id to be submitted');
    } else {
      try {
        deleteRequest(`http://127.0.0.1:8000/api/anime/${id}/`);
        console.log(`Anime ${id} deleted`);
        setId('');
      } catch (error) {
        console.log(error);
      }
    }
  };

  return (
    <>
      <form onSubmit={(e) => handleDeleteForm(e)}>
        <div className="bg-sky-200 flex flex-wrap items-center mb-
2">
          <label className="p-2 w-24">ID</label>
          <input
            type="text"
            value={id}
```

```
            onChange={(e) => setId(e.target.value)}
            className="grow p-2"
            required
          />
        </div>
        <div>
          <button
            type="submit"
            className="bg-sky-600 p-2 text-white cursor-pointer
font-bold rounded-lg"
          >
            Submit
          </button>
        </div>
      </form>
    </>
  );
}
```

Next is the form for adding and updating animes in the database and the file can be located at `FormPostUpdate/FormPostUpdate.js`. Add the code to the file:

```
import { useState } from 'react';
import { usePost } from '../../hooks/usePost';
import { useUpdate } from '../../hooks/useUpdate';
```

```
export default function FormPostUpdate({ crudType }) {
  const [id, setId] = useState('');
  const [animeId, setAnimeId] = useState('');
  const [animeName, setAnimeName] = useState('');
  const [animeRelease, setAnimeRelease] = useState('');
  const { postRequest } = usePost();
  const { updateRequest } = useUpdate();
  const handlePostUpdateForm = async (e, type) => {
    e.preventDefault();

    if (animeId === '' || animeName === '' || animeRelease === '') {
      console.log('Form needs data to be submitted');
    } else {
      try {
        const anime = {
          anime_id: animeId,
          anime_name: animeName,
          anime_release: animeRelease,
        };
        if (type === 'posted') {
          postRequest('http://127.0.0.1:8000/api/anime/', anime);
          console.log(`Anime ${type}`);
```

```
            setId('');
            setAnimeId('');
            setAnimeName('');
            setAnimeRelease('');
          } else if (type === 'updated' && id !== '') {
            updateRequest(`http://127.0.0.1:8000/api/anime/${id}/`,
anime);
            console.log(`Anime ${type}`);
            setId('');
            setAnimeId('');
            setAnimeName('');
            setAnimeRelease('');
          }
        } catch (error) {
          console.log(error);
        }
      }
    };

    return (
      <>
        <form onSubmit={(e) => handlePostUpdateForm(e, crudType)}>
          {crudType === 'updated' ? (
            <div className="bg-sky-200 flex flex-wrap items-center mb-
2">
              <label className="p-2 w-24">ID</label>
              <input
                type="text"
                value={id}
                onChange={(e) => setId(e.target.value)}
                className="grow p-2"
                required
              />
            </div>
          ) : (
            <></>
          )}
          <div className="bg-sky-200 flex flex-wrap items-center mb-
2">
            <label className="p-2 w-24">Anime ID</label>
            <input
              type="text"
              value={animeId}
              onChange={(e) => setAnimeId(e.target.value)}
              className="grow p-2"
              required
            />
          </div>

          <div className="bg-sky-200 flex flex-wrap items-center mb-
2">
            <label className="p-2 w-24">Name</label>
            <input
              type="text"
              value={animeName}
              onChange={(e) => setAnimeName(e.target.value)}
```

```
            className="grow p-2"
            required
          />
        </div>
        <div className="bg-sky-200 flex flex-wrap items-center mb-
2">
          <label className="p-2 w-24">Release</label>
          <input
            type="text"
            value={animeRelease}
            onChange={(e) => setAnimeRelease(e.target.value)}
            className="grow p-2"
            required
          />
        </div>
        <div>
          <button
            type="submit"
            className="bg-sky-600 p-2 text-white cursor-pointer
font-bold rounded-lg"
          >
            Submit
          </button>
        </div>
      </form>
    </>
  );
}
```

Next, let's do the `Page.js` file which is the main file for our entire application. This is where all of our components will reside so replace all of the code with this code here:

```
'use client';
import FormPostUpdate from './components/FormPostUpdate/FormPostUpdate';
import FormDelete from '../app/components/FormDelete/FormDelete';
import { useFetch } from './hooks/useFetch';
import { useFetchSWR } from './hooks/useFetchSWR';
export default function Home() {
  // Uncomment the code below and comment out the "useFetch" code if your want
  // const { data, error, isLoading } = useFetchSWR(
  // 'http://127.0.0.1:8000/api/anime/'
  // );
  // Uses the Fetch API for data fetching
  const { data, error, isLoading } = useFetch(
    'http://127.0.0.1:8000/api/anime/'
  );
  if (error) return <div>An error has occurred.</div>;
```

```
    if (isLoading) return <div>Loading...</div>;
    console.log(data);
```

```
  return (
    <>
      <div className="container mx-auto mt-10">
        <div className="bg-sky-300 rounded-lg p-4 mb-4">
          <h1 className="text-xl mb-4">POST: Add Anime Form</h1>
          <FormPostUpdate crudType="posted" />
        </div>
        <div className="bg-sky-300 rounded-lg p-4 mb-4">
          <h1 className="text-xl mb-4">UPDATE: Update Anime
Form</h1>
          <p className="bg-yellow-100 p-4">
            Select an ID from the list. You can change the data for
Anime ID,
            Name and Release.
          </p>
          <FormPostUpdate crudType="updated" />
        </div>
        <div className="bg-sky-300 rounded-lg p-4 mb-4">
          <h1 className="text-xl mb-4">DELETE: Delete Anime
Form</h1>
          <FormDelete />
        </div>
        <div className="bg-sky-700 rounded-lg p-4 mb-4 text-white">
          <h1 className="text-xl">GET: Anime Data List</h1>
          {data.map((anime) => (
            <div key={anime.id}>
              <ul className="bg-sky-800 mb-4 p-4">
                <li>
                  <h1 className="text-lg font-bold">
{anime.anime_name}</h1>
                </li>
                <li>ID: {anime.id}</li>
                <li>Anime ID: {anime.anime_id}</li>
                <li>Anime Release Year: {anime.anime_release}</li>
              </ul>
            </div>
          ))}
        </div>
      </div>
    </>
  );
}
```

Our main components are complete in the next section we will create some test files.

Setting up our test files

There are going to be two test files for the two form components we made. These test files do not have full test coverage they only test to see if the component has rendered without crashing. You can add more tests if you want to. Our test file is located in the root folder for this project and is named `__tests__`.

Start by adding this test to the form delete file located in `FormDelete.test.js`.

```
import { render } from '@testing-library/react';
import FormDelete from '../src/app/components/FormDelete/FormDelete';
```

```
describe('component renders', () => {
  test('component does not crash', () => {
    render(<FormDelete />);
  });
});
```

And lastly, add this code to the form post update file located in `FormPostUpdate.test.js`.

```
import { render } from '@testing-library/react';
import FormPostUpdate from '../src/app/components/FormPostUpdate/FormPostUpdate
```

```
describe('Form component', () => {
  test('renders without crashiing', () => {
    render(<FormPostUpdate />);
  });
});
```

Let's now create our reusable hooks in the next section for all of our backend CRUD requests which we created earlier.

Creating our custom hooks

The first custom hook we create will be for getting data from our backend.

GET Requests

Two custom hooks are available in this section but we only need to use one of them. This version uses the Fetch API and the file is located in `hooks/useFetch.js` so add this code to the file. It also has a polling setup which means that it can automatically fetch data periodically from our API so that we do not have to manually refresh the page after performing a CRUD action.

It is just a simple function that calls the fetch data function every 2 seconds and can be configured to increase or decrease the time. This is an easy-to-implement feature that does not require a third-party library like WebSockets. Of course, using WebSockets would be a much better implementation but for this simple demo application, it's fine.

```
  import { useState, useEffect } from 'react';
```

```
export function useFetch(url) {
  const [data, setData] = useState([]);
  const [error, setError] = useState(null);
  const [isLoading, setIsLoading] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const json = await fetch(url).then((r) => r.json());
        setIsLoading(false);
        setData(json);
      } catch (error) {
        setError(error);
        setIsLoading(false);
      }
    };
    fetchData();

    // Polling so that the chart with the most recent changes from
the database updates without having to do a page reload anytime a
CRUD request is sent to the database
    const pollInterval = setInterval(() => {
      fetchData(); // Fetch data periodically
    }, 2000); // Poll every 2 seconds (adjust as needed)

    // Clear the interval when the component unmounts
    return () => {
```

```
      clearInterval(pollInterval);
    };
  }, [url]);

  return { data, error, isLoading };
}
```

This alternative version uses SWR and the file is located in `hooks/useFetchSWR` so add the code to it. You can choose which version you want to use by looking at the comments in the main `page.js` file.

```
import useSWR from 'swr';

const fetcher = (url) => fetch(url).then((res) => res.json());

export function useFetchSWR(url) {
  const { data, error, isLoading } = useSWR(url, fetcher);
  return { data, error, isLoading };
}
```

Ok, moving on it's time to work on the post-request hook.

POST Requests

The file is located in `hooks/usePost.js` and this hook is used for adding animes to our database so add this code to the file:

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

export function usePost() {
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);
  const [response, setResponse] = useState(null);
  const postRequest = async (url, formData) => {
    setIsLoading(true);
    setError(null);
    setResponse(null);

    try {
      const response = await fetch(url, {
```

```
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify(formData),
      });

      const responseData = await response.json();

      if (response.ok) {
        setResponse(responseData);
      } else {
        setError(responseData);
      }
    } catch (error) {
      setError(error);
    } finally {
      setIsLoading(false);
    }
  };

  return { isLoading, error, response, postRequest };
}
```

Right, let's move on to the update hook next.

UPDATE Requests

We can find the file in `hooks/useUpdate.js` this hook is for updating our data in the database so add this code to the file:

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

```
export function useUpdate() {
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);
  const [response, setResponse] = useState(null);

  const updateRequest = async (url, formData) => {
    setIsLoading(true);
    setError(null);
    setResponse(null);

    try {
      const response = await fetch(url, {
        method: 'PUT',

        headers: {
          'Content-Type': 'application/json',
```

```
        },
        body: JSON.stringify(formData),
      });

      const responseData = await response.json();

      if (response.ok) {
        setResponse(responseData);
      } else {
        setError(responseData);
      }
    } catch (error) {
      setError(error);
    } finally {
      setIsLoading(false);
    }
  };

  return { isLoading, error, response, updateRequest };
}
```

And lastly lets do our delete hook.

DELETE Requests

The file is located in `hooks/useDelete.js` and this hook is for deleting data in the database so add this code below to the file.

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

```
export function useDelete() {
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);
  const [response, setResponse] = useState(null);

  const deleteRequest = async (url) => {
    setIsLoading(true);
    setError(null);
    setResponse(null);

    try {
      const response = await fetch(url, {
        method: 'DELETE',
        headers: {
          'Content-Type': 'application/json',
        },
      });
```

```
        const responseData = await response.json();

        if (response.ok) {
          setResponse(responseData);
        } else {
          setError(responseData);
        }
      } catch (error) {
        setError(error);
      } finally {
        setIsLoading(false);
      }
    };

    return { isLoading, error, response, deleteRequest };
  }
```

Ok good, there is just one more step to go. Find the `globals.css` file which should be inside of the `src/app` folder in your Next.js project. Just delete all of the CSS in there apart from the tailwind CSS imports at the top of the file. So it should look like this:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

We have to do this so that the styles in there do not conflict with the ones we created for this Tailwind CSS design.

Now let's see if it's all working, first you need to make sure that the database is up and running, and that one of the backends (Node or Django) is running on port 8000. Our frontend needs to connect to the backend so obviously it needs to be running first. Use this run script to start the app afterwards:

```
npm run dev
```

Our app should look like the design here:

And use these for the tests:

```
npm run test
npm run coverage
```

The tests should all pass although you will see that the test coverage score is low. That is because we did not fully test all of the components if you want you can play around with the files and try to get the test coverage up.

All that's left is to add a chart to our application and then it truly will be complete! Let's move on to the final section and get this done!

Integrating a chart library with full CRUD

We now have working backends for Node and Django which connect to SQL databases. We can do full CRUD requests to the backend on the frontend using the forms on the page. Let's now integrate a chart that shows the data from the backend with full CRUD functionality.

The chart library we will use for this example will be Highcharts so first install the package into our Next.js application:

```
npm install highcharts highcharts-react-official
```

Up next we will need to, create a folder, component and test for the chart in the components folder using the mentioned here. Now following our current project structure start by creating a folder and file for our `Chart.js` component. The file directory should be as follows `src/app/components/Chart/Chart.js`. Add this code to our `Chart.js` file and like before the name is self-explanatory this component will be used for displaying our chart along with its data on the page.

```
import { useState, useEffect } from 'react';
import Highcharts from 'highcharts';
import HighchartsReact from 'highcharts-react-official';
import { useFetch } from '@/app/hooks/useFetch';
```

```
export default function Chart() {
  const { error, isLoading, data } = useFetch(
    'http://127.0.0.1:8000/api/anime/'
  );

  const [animeDate, setAnimeDate] = useState('');
  const [filterAnimeDate, setFilterAnimeDate] = useState([]);
  const [hoverData, setHoverData] = useState(null);
  const [chartOptions, setChartOptions] = useState({
    xAxis: {
      categories: [],
    },
    title: {
      text: 'Animes',
      align: 'center',
    },
    series: [{ data: [] }],
    plotOptions: {
      series: {
        point: {
          events: {
            mouseOver(e) {
              setHoverData(e.target.category);
            },
          },
        },
      },
    },
  });
```

```
  useEffect(() => {
    setChartOptions({
      xAxis: {
        categories: data.map((item) => item.anime_name),
      },
      series: [{ data: data.map((item) => parseInt(item.anime_id))
}],
      plotOptions: {
        series: {
          point: {
            events: {
              mouseOver(e) {
                setHoverData(e.target.category);
              },
            },
          },
        },
      },
    });
  }, [data]);

  if (error) return <div>Error</div>;
  if (isLoading) return <div>Loading...</div>;

  const handleFilterAnime = (e) => {
    e.preventDefault();

    const filterDate = data.filter((a) => a.anime_release ===
animeDate);
    console.log(filterDate);
    setFilterAnimeDate(filterDate);

    setAnimeDate('');
  };

  return (
    <>
      <div>
        <HighchartsReact highcharts={Highcharts} options=
{chartOptions} />
        <p className="bg-green-300 p-4 mb-4 mt-4 rounded-lg">
          Hovering over: <span className="font-bold">{hoverData}
</span>
        </p>
      </div>
      <form onSubmit={handleFilterAnime}>
        <div className="bg-sky-300 rounded-lg p-4 mb-4">
          <div>
            <h1 className="text-xl mb-4">Filter by release year</h1>
            <div className="bg-sky-200 flex flex-wrap items-center
mb-2">
              <label className="p-2 w-24">Date:</label>
              <input
                type="text"
                value={animeDate}
                onChange={(e) => setAnimeDate(e.target.value)}
                className="grow p-2"
```

```
                required
              />
            </div>
            <div>
              <button
                type="submit"
                className="bg-sky-600 p-2 text-white cursor-pointer
font-bold rounded-lg"
              >
                Submit
              </button>
            </div>
          </div>
        </div>
      </form>

      <div className="bg-green-600 p-4 rounded-lg">
        <h1 className="text-xl text-white">Release Year</h1>
        {filterAnimeDate === 0 ? (
          <></>
        ) : (
          <>
            {filterAnimeDate.map((anime) => (
              <div key={anime.id} className="mt-4 text-white">
                <ul className="bg-green-700 mb-4 p-4">
                  <li>
                    <h1 className="text-lg font-bold">
{anime.anime_name}</h1>
                  </li>
                  <li>ID: {anime.id}</li>
                  <li>Anime ID: {anime.anime_id}</li>
                  <li>Anime Release Year: {anime.anime_release}</li>
                </ul>
              </div>
            ))}
          </>
        )}
      </div>
    </>
  );
}
```

Ok almost done now create a test file for `Chart.test.js` inside of the `__tests__` folder and add this code for the chart test file:

```
import { render } from '@testing-library/react';
import Chart from '../src/app/components/Chart/Chart';
```

```
describe('it renders', () => {
  test('Chart component loads', () => {
    render(<Chart />);
  });
});
```

Finally, update the `Page.js` file with this code so that our chart component can now be displayed on the page and we are done!

```
'use client';
import FormPostUpdate from './components/FormPostUpdate/FormPostUpdate';
import FormDelete from '../app/components/FormDelete/FormDelete';
import { useFetch } from './hooks/useFetch';
import { useFetchSWR } from './hooks/useFetchSWR';
import Chart from './components/Chart/Chart';
```

```
export default function Home() {
  // Uncomment the code below and comment out the "useFetch" code if
your want to use SWR for data fetching -->
https://swr.vercel.app/docs/with-nextjs
  // const { data, error, isLoading } = useFetchSWR(
  // 'http://127.0.0.1:8000/api/anime/'
  // );
  // Uses the Fetch API for data fetching
  const { data, error, isLoading } = useFetch(
    'http://127.0.0.1:8000/api/anime/'
  );

  if (error) return <div>An error has occurred.</div>;
  if (isLoading) return <div>Loading...</div>;
  console.log(data);

  return (
    <>
      <div className="container mx-auto mt-10 grid gap-4 lg:grid-
cols-2 md:grid-cols-2 sm:grid-cols-1">
        <div>
          <div className="bg-sky-300 rounded-lg p-4 mb-4">
            <h1 className="text-xl mb-4">POST: Add Anime Form</h1>
            <FormPostUpdate crudType="posted" />
          </div>
          <div className="bg-sky-300 rounded-lg p-4 mb-4">
            <h1 className="text-xl mb-4">UPDATE: Update Anime
Form</h1>
            <p className="bg-yellow-100 p-4">
              Select an ID from the list. You can change the data
for Anime ID,
```

```
              Name and Release.
            </p>
            <FormPostUpdate crudType="updated" />
          </div>
          <div className="bg-sky-300 rounded-lg p-4 mb-4">
            <h1 className="text-xl mb-4">DELETE: Delete Anime
Form</h1>
            <FormDelete />
          </div>
          <div className="bg-sky-700 rounded-lg p-4 mb-4 text-
white">
            <h1 className="text-xl">GET: Anime Data List</h1>
            {data.map((anime) => (
              <div key={anime.id}>
                <ul className="bg-sky-800 mb-4 p-4">
                  <li>
                    <h1 className="text-lg font-bold">
{anime.anime_name}</h1>
                  </li>
                  <li>ID: {anime.id}</li>
                  <li>Anime ID: {anime.anime_id}</li>
                  <li>Anime Release Year: {anime.anime_release}</li>
                </ul>
              </div>
            ))}
          </div>
        </div>
        <div>
          <Chart />
        </div>
      </div>
    </>
  );
}
```

Alright! We should have a working chart with full CRUD functionality that has real-time updates and another form for filtering the anime by dates and the data is shown on the page. You might need to reload the page so it updates with the new changes we made before it starts working properly. All 3 tests should be passing and this is what the final design looks like:

Congratulations you have just built a full-stack dashboard application and completed this project 🎉

Conclusion

So today we had a run-through of what it could be like to work as a developer. We had a brief introduction to Agile and working with Jira. And we managed to build two backends one using Node.js and Express and the other using Python and Django. We used our API testing tool (in this case Postman) to perform CRUD (Create, Read, Update, Delete) requests. Then we connected the backends to our frontend which we created using Next.js. This allowed us to see the CRUD functionality working on the frontend. Lastly, we implemented a chart that shows the data in our database completing our dashboard application.

Building software can require a lot of work and phases and what we saw today was just a small snapshot of a developer's workflow. There are many more areas to consider like having frequent team meetings, and code reviews every time you make changes, etc... To improve your knowledge in this area even more you could try setting up a Jira or alternative workflow with a full CI/CD pipeline as this can be good practice for beginners who have yet to work professionally in a company team. In closing, there are infinite ways for a developer and teams to work and this is just one example.

In Plain English 🚀

Thank you for being a part of the ***In Plain English*** *community! Before you go:*

- Be sure to **clap** and **follow** the writer 👏
- Follow us: **X** | **LinkedIn** | **YouTube** | **Discord** | **Newsletter**
- Visit our other platforms: **Stackademic** | **CoFeed** | **Venture**
- More content at **PlainEnglish.io**

Webdev JavaScript Beginner Programming Python

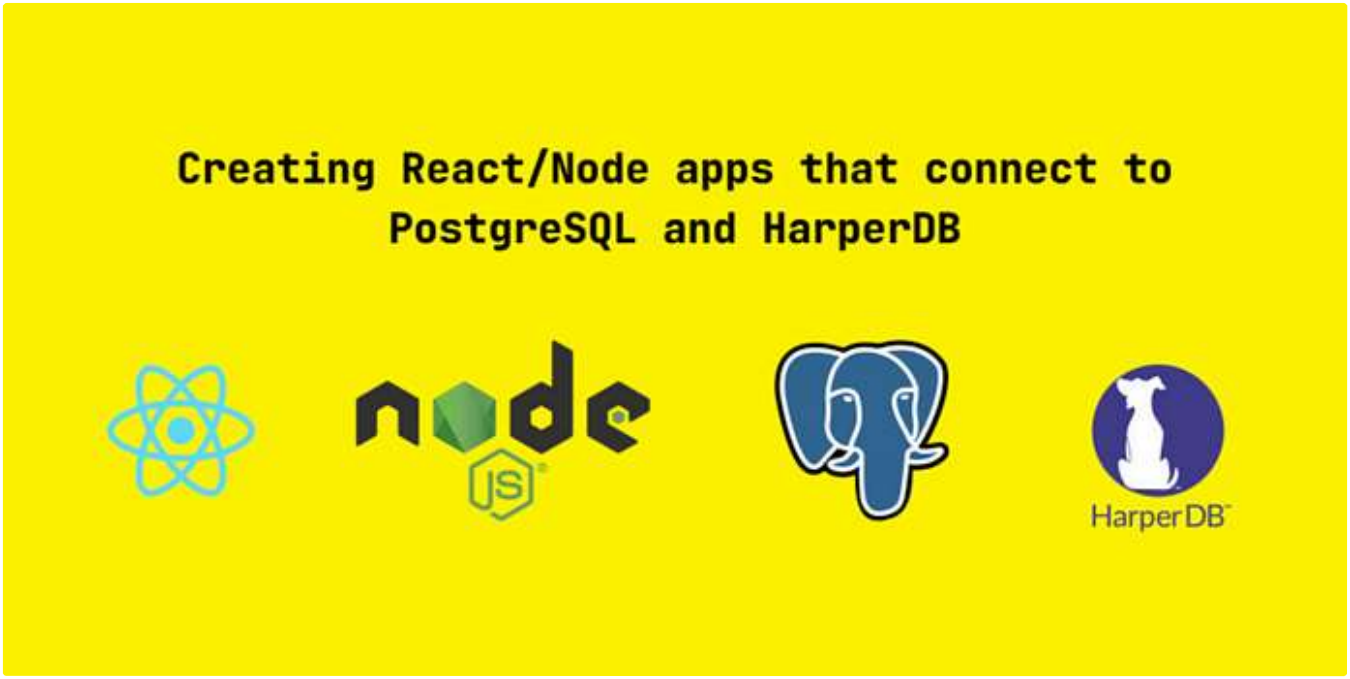
Follow

Written by Andrew Baisden

1.1K Followers · Writer for JavaScript in Plain English

👨‍💻 Software Developer 📝 Tech Writer @LogRocket 📹 Content Creator All things tech and programming 💻

More from Andrew Baisden and JavaScript in Plain English



Andrew Baisden

Creating React/Node apps that connect to PostgreSQL and HarperDB

I'm sure that most of you are already more than familiar with the MERN stack. Having a React front end with a Node/Express back end that...

17 min read · Jan 12, 2021

73 2

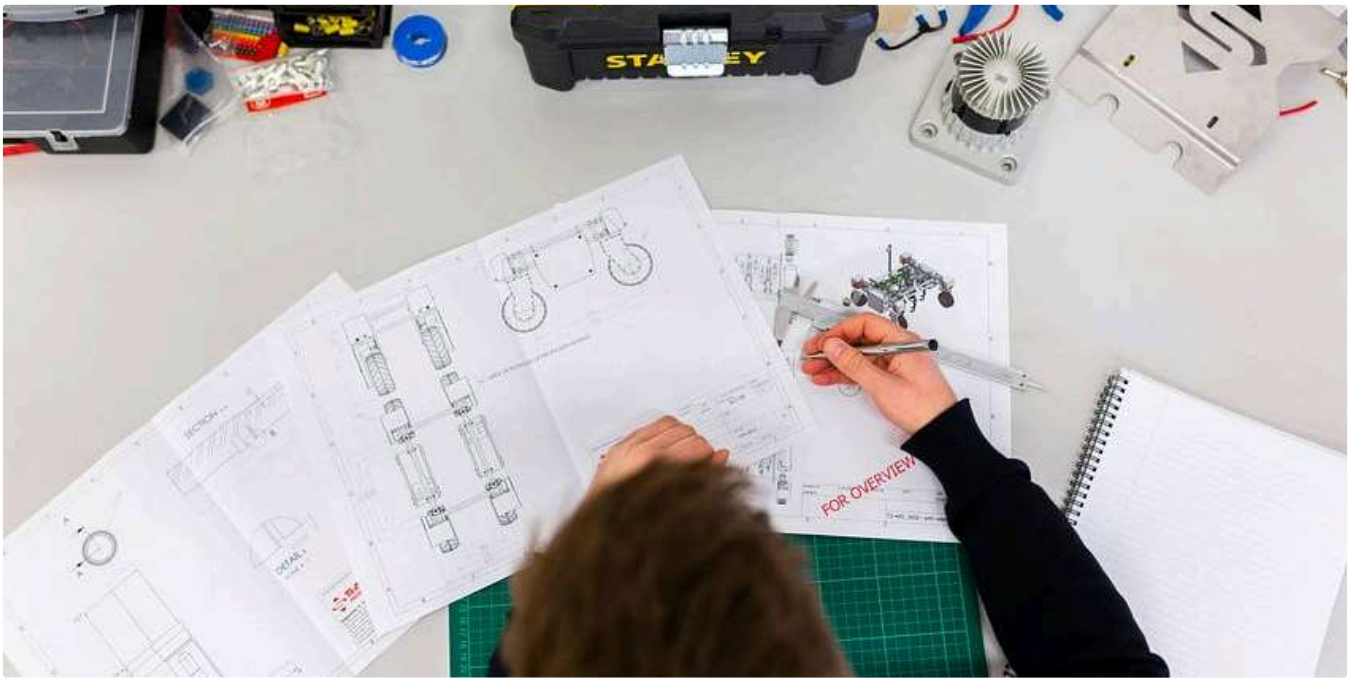
Rehan Pinjari in JavaScript in Plain English

15 Time-Saving Websites Every Developer Needs

Ever thought that there aren't enough hours in the day for all your never-ending development tasks? You are not alone.

5 min read · Apr 29, 2024

1K 13



Selcuk Ozdemir in JavaScript in Plain English

Write a React Component Like a Pro

In the world of React, writing components is an art. It's not just about making them work—it's about making them work well. Today, we're...

3 min read · May 1, 2024

947 15

Andrew Baisden

100 AWESOME Developers to follow on Twitter

The Twitter community is amazing and there are so many awesome people to follow. If you are a Developer then having a Twitter is one of...

8 min read · Jan 12, 2021

152

See all from Andrew Baisden

See all from JavaScript in Plain English

Recommended from Medium

Rehan Pinjari in JavaScript in Plain English

10 Cutting-Edge Web Dev Resources You Should Be Using Now

Hey, developers! The web is a live thing, growing constantly, and it demands new skills and technologies. Whether you're an experienced...

5 min read · May 13, 2024

105

Daniel Craciun in Level Up Coding

Stop Using UUIDs in Your Database

How UUIDs can Destroy SQL Database Performance

✦ · 4 min read · May 16, 2024

403 41

Lists

Coding & Development

11 stories · 623 saves

General Coding Knowledge

20 stories · 1247 saves

Stories to Help You Grow as a Software Developer

19 stories · 1077 saves

Predictive Modeling w/ Python

20 stories · 1219 saves

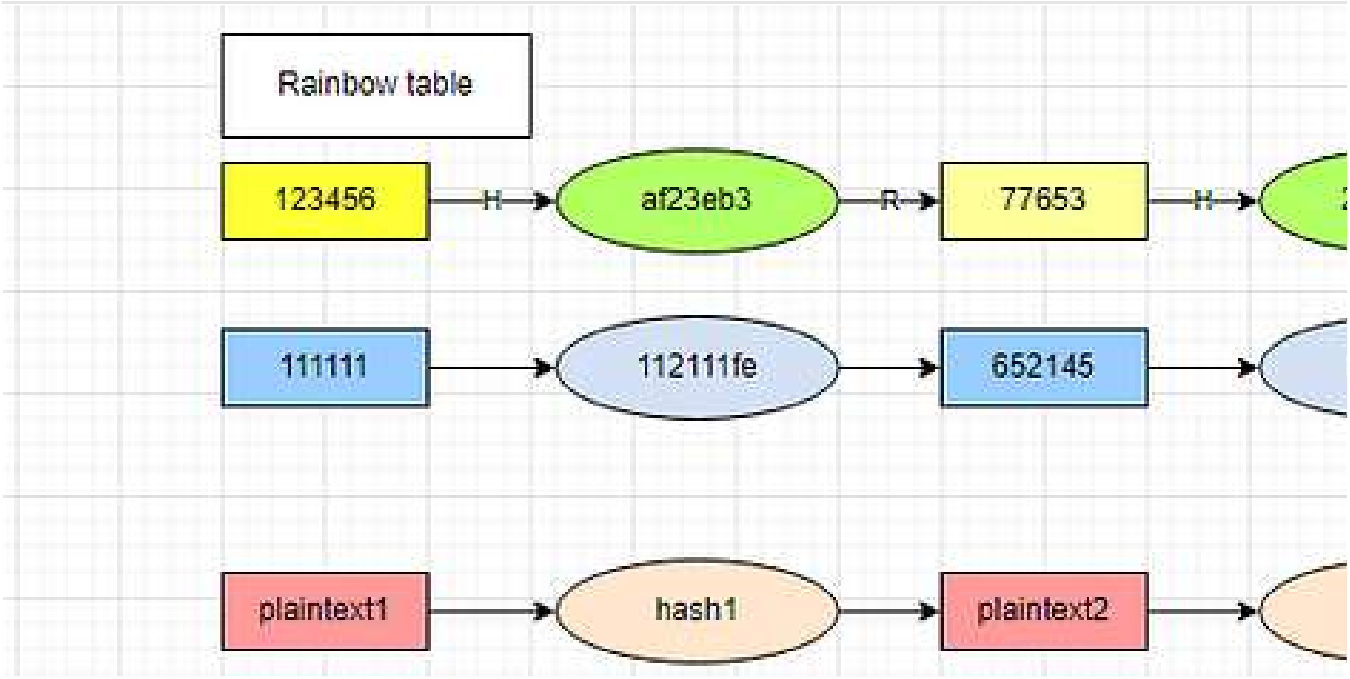
Stanley Udegbunam in Writing Solopreneur

11 Stupid Simple Passive Income Ideas for Programmers

Actionable Steps for Entrepreneurial Developers

✦ · 7 min read · Mar 13, 2024

868 17



LORY

A basic question in security Interview: How do you store passwords in the database?

Explained in 3 mins.

· 7 min read · May 12, 2024

1.8K 28

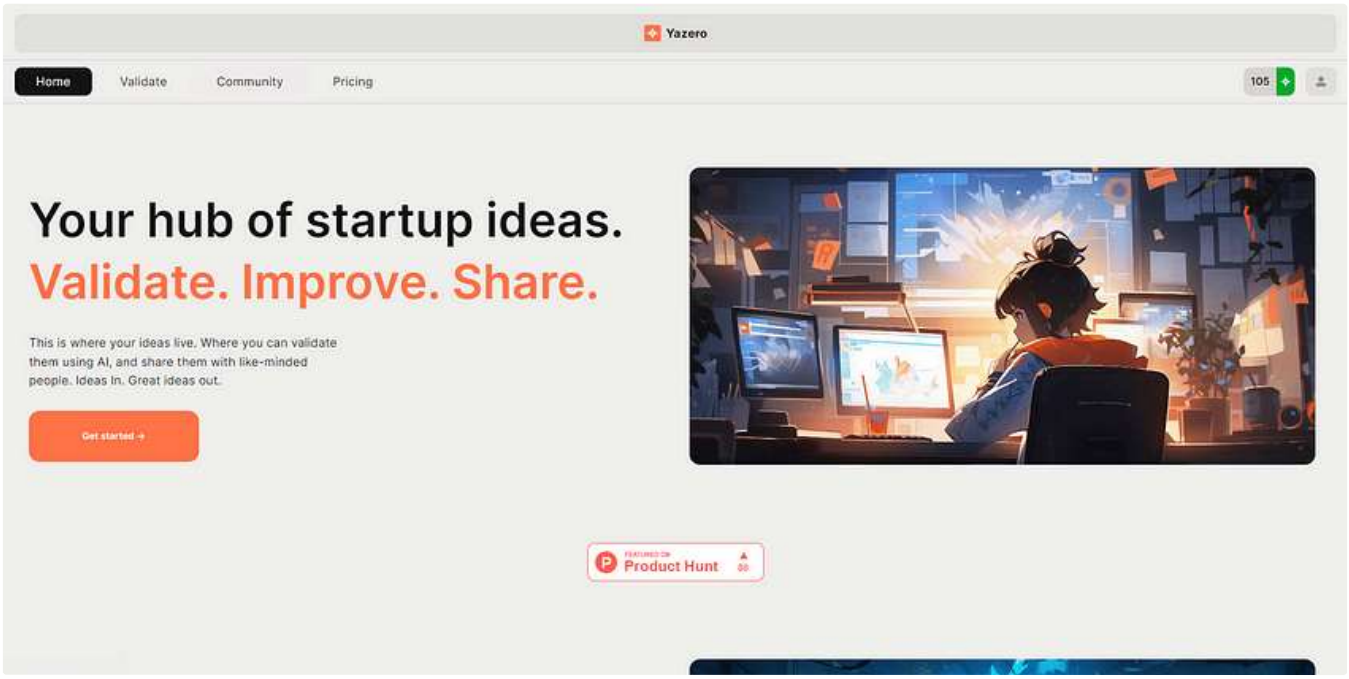
Oliver

20 Must-Have VSCode Extensions for Web Development In 2024

As a senior ReactJS developer, I've come across various tools and technologies that have significantly enhanced my productivity and...

4 min read · Apr 21, 2024

68



Artem Shelamanov in Python in Plain English

How I Built My First AI Startup (With No Experience)

My detailed journey with advices on building startups.

9 min read · Apr 30, 2024

1.6K 39

See more recommendations