VPC Technical spec

Introduction

One of our goals for the 30th anniversary of the Venice Project Center was to give their website a much needed web overhaul. This web overhaul is justified mostly because we wanted to document the process to make it easier to maintain. Additionally we wanted to move all web hosting capabilities to AWS so that the VPC only has to care about a single account.

Tech stack

Webstorm

We would recommend you develop this website with the webstore IDE from jetbrains. The reason for this is because webstore comes with an incredible set of tools. It has one of the best debuggers available that makes spotting problems on the server side incredibly easy. It also comes with a database viewer to make it easy for you to view data changes.

Typescript

Our tech stack is built on top of the typescript programing language which is a superset of the javascript programing language. This means that all valid javascript code is also valid typescript code. Additionally Typescript is statically typed meaning all variables have to be typed, meaning that you get a level of safety that you don't get with Javascript. I would highly recommend learning some basic typescript before working on maintaining the website.

Typescript docs

React

At the core of our app we used the React rendering library because it is by far the most popular front end development library and therefore has the widest ecosystem. React is a component based framework meaning that you group your code into reusable elements that can then be placed through the app. React uses the JSX syntax to create a powerful hybrid of html and javascript code to allow you to group your logic and rendering together.

React docs

Next JS

On top of React we are using the Next.js which is a framework built on top of React. The main powers of next is that we have built in server side rendering meaning that our web pages are built on the server and then sent as full html to the client. This is done to increase web performance and searchability. Additionally it provides an easy way to make new pages. To make a new page all you need to do is make a new directory and make a page.tsx file in that directory. Finally, Next.js provides an easy way to define api routes making it easy to develop front and back end in a single project.

Next js docs

Tailwind

Tailwind is a set of convenient css classes that you can add to your elements. There are three reasons to use Tailwind instead of normal css. The first being speed of development once you get the hang of tailwind it is way faster to style with than normal css. The second reason is that it is far easier to make mobile first designs because tailwind is a mobile first framework. Finally tailwind is better for performance because the classes have been optimized and unused classes have been removed when the project was built.

Git + github for source control

If a team needs access to the Github Repository they should log into Venice-project-center(venice.wpi@gmail.com) and access the GitHub account. They can then clone the repository and make changes with their personal GitHub. It is highly recommended that if the infrastructure needs to be changed, it should be done by a skilled developer. If the team has to make modifications to the website source code, they should first get access to the git repository **v23e-website**. If changes need to be made to the hosting infrastructure, it should be changed via the terraform config files in the **v23e-website** repository. The reason for this is that the Terraform files act as a way to document and power the Venice Project Center hosting. Therefore, it should be easier to preserve and maintain settings for hosted content.

Prisma as an ORM

We used prisma as our orm. To utilize it, declare your database Schema in the schema.prisma file. Then by running prisma migrate dev —name=name of migration you are able to update your database. Additionally prisma will generate convenient classes for you to work within your code. I would highly recommend reading the prisma docs if you plan on updating the database schema.

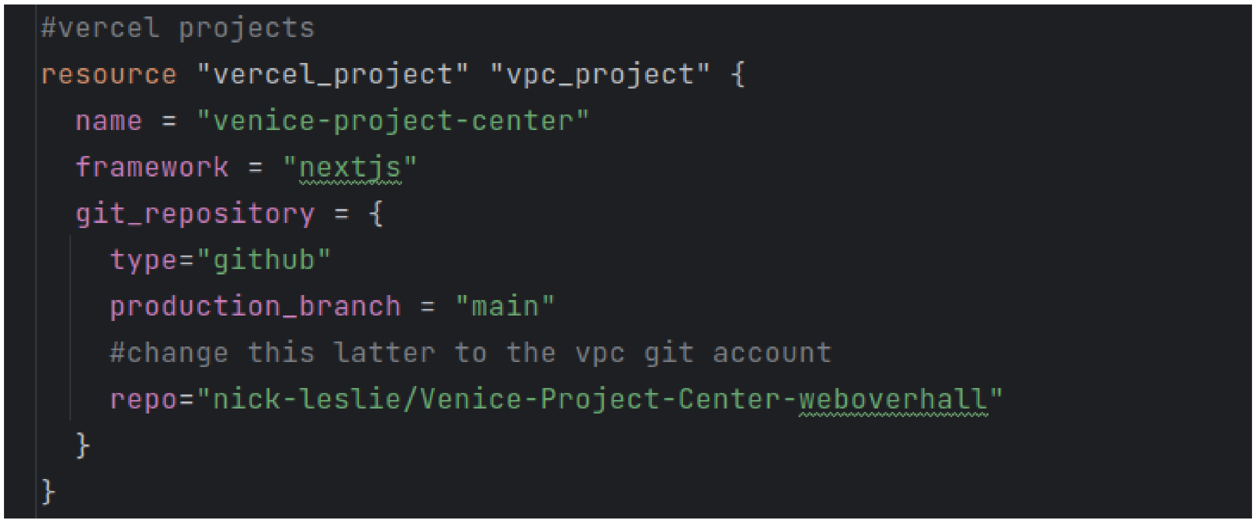
Infrastructure

Introduction to infrastructure

Hosting code is as complicated a process as writing it. We did everything that we can to make updating the code as painless an expernce as posible. To make changes to the production cite is to push changes to main. This is posible only becuase of the built in continious integration of vercel. Although changing infrastructure is hard and not somehting that should taken lightly. If you want to add new things to the existing web infrastructure make sure you do your reasrch. While cloud hosting has changed the way developers thing about hosting there code by allwoing them to host anything on a amzaons billion dolor servers. However this power should not be taken lightly. If done wrong you can get hit with an extreamly large bill. So if your trying to update infrastructure take and your time do your reserch.

Terraform

Terraform is an infrastructure as code tool that alows you settup your infrstructure using a delcaitive code syntax.



```
#vercel projects
resource "vercel_project" "vpc_project" {
  name = "venice-project-center"
  framework = "nextjs"
  git_repository = {
    type="github"
    production_branch = "main"
    #change this latter to the vpc git account
    repo="nick-leslie/Venice-Project-Center-weboverhall"
  }
}
```

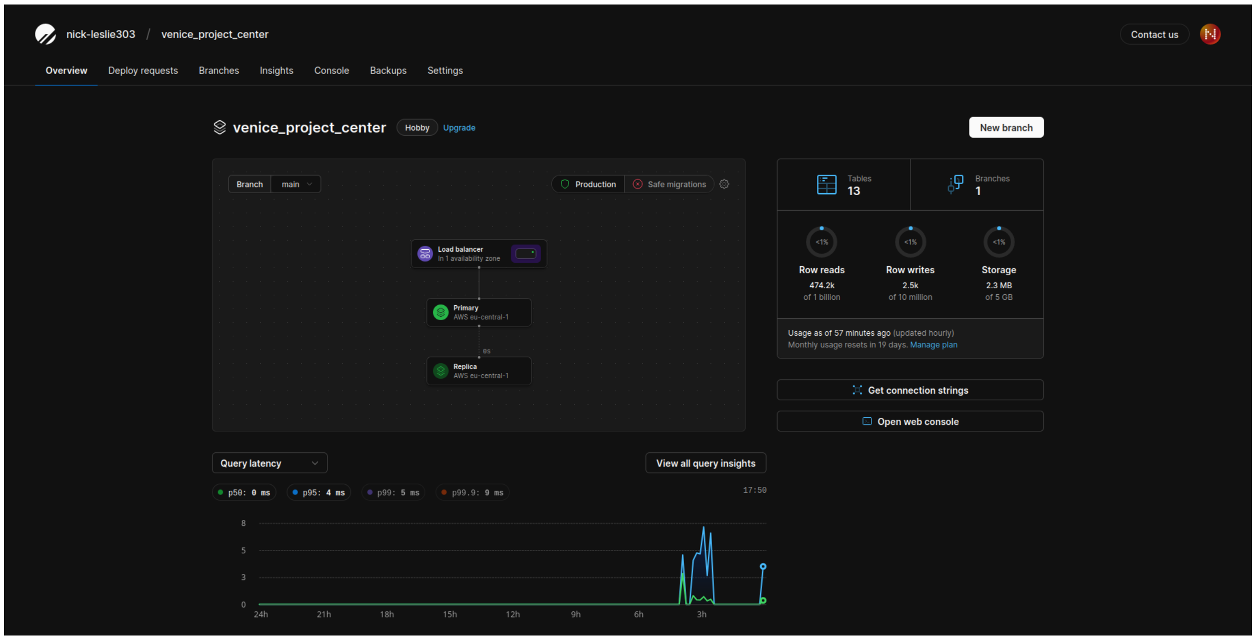
Here is an example of what a Vercel project looks like when setup through terraform. All you have to do to apply changes to the infrastructure is run the command **terraform apply -var-file=".tfvars"** this will generate a terraform state file which discribes the current state of your infrastructure. **The terraform state file will be in the V23E-Story drive**. Only one copy of that file should ever exist beacuse it could cause name conflicts. Additionally there will be a .tsvars file in the google drive. This file inclides all of the necessary tokens to interact with the services. If you are unsure about anything terraform or infrastructure related please email nick.leslie303@gmail.com. Terraform is very complicated but so is infrastructure, so make sure you do your research before updating the infrastructure.

Vercel hosting platform

We decided to use Vercel because it is free and incredibly easy to use. Continuous integration is supplied out of the box meaning that so long as your build passes, making updates to the website is as easy as pushing to the repo. Additionally Vercel is the company that developed NextJS meaning that our website is optimized for us.

Planet scale database

We picked planetscale because it was free and more advanced than most other databace platforms. It includes a load balencer to help distribute a trafic to multiple btween databases, and a replication database which means if our core database goes down we have a backup. You get 1 billon reads per month and 1 million writes with the free version of the database. Additionaly you are able to branch your database to have multiple versions of your data. This is especially important becuase it means you can seperate your production and development databases with branching. The database is described in the terraform files, however you will need to update the token in the planetscale database to include the newly created database if you ever decided to regenerate it.



Tables 13
Branches 1

Row reads 474.2k of 1 billion
Row writes 2.5k of 10 million
Storage 2.3 MB of 5 GB

Usage as of 57 minutes ago (updated hourly)
Monthly usage resets in 19 days. Manage plan

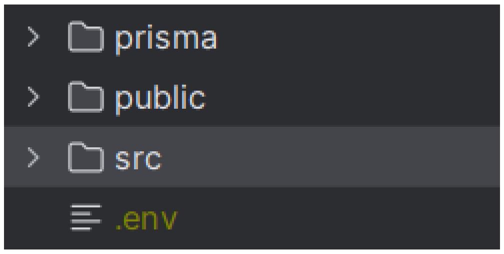
p50: 0 ms
p95: 4 ms
p99: 5 ms
p99.9: 9 ms

S3 buckets

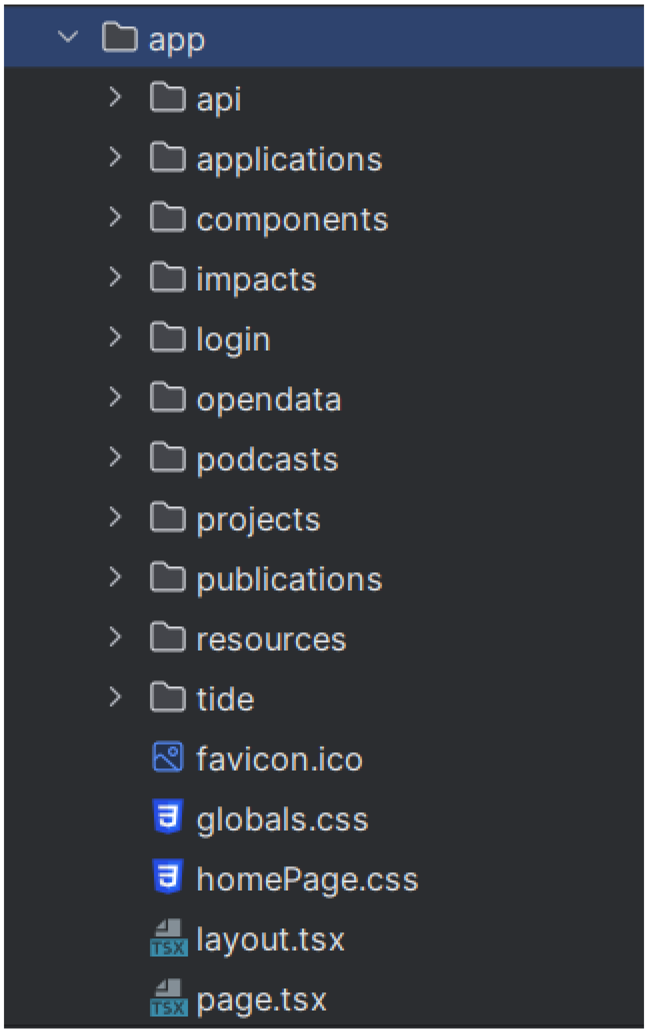
We use an aws s3 bucket for file storage. S3 stands for simple storage solution and it is the go to way for developers to store binary files like picures and pdfs. We set up our s3 bucket to be publicly accesable if you are trying to get files. But to write you need to first make a call to an api gateway that we setup. Once you call that api gateway you get a presighned url you can then make a post request to this url uploading the file. The reason we did it this was becuase it was the easyses to setup with terraform. We felt that due to the natral complexity that comes with setting up an application using aws it was very important that we set it up using terraform that way our arcatecture was documented as code.

Additional how to

File structure overview

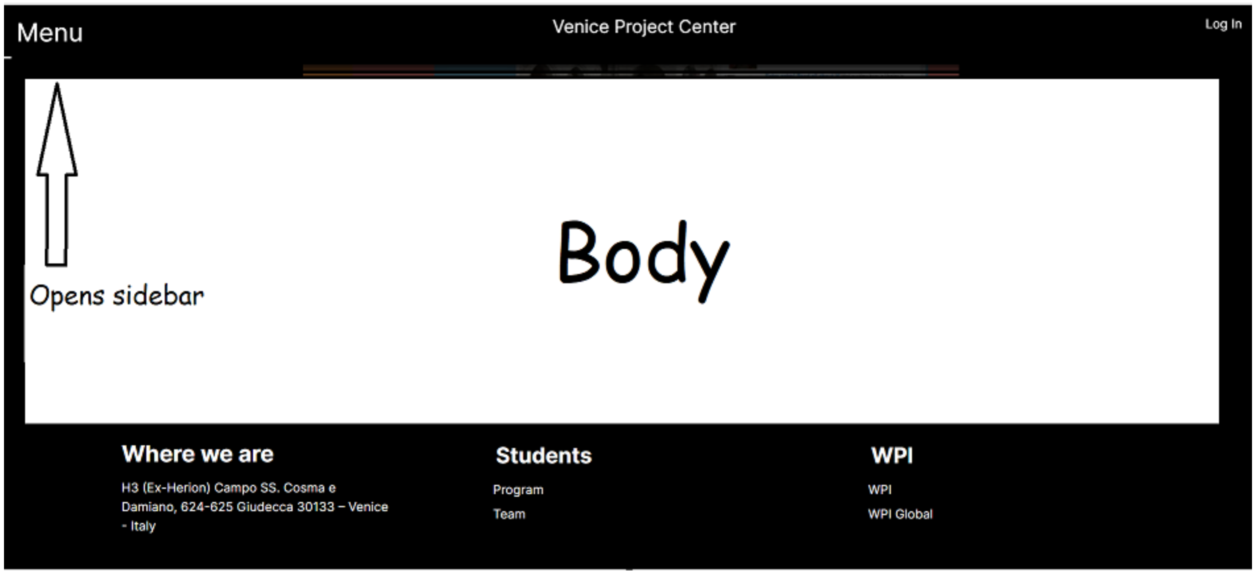


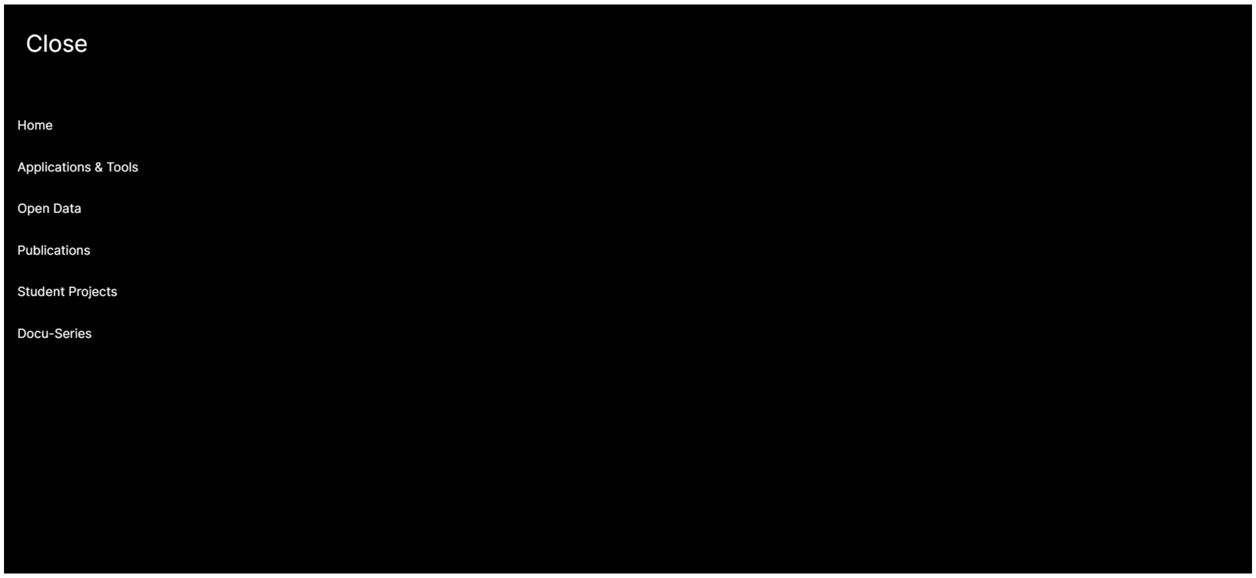
Prisma contains the models used throughout the api folder within src. .env contains the database URL, if for some reason the database has to be changed use this file. The structure of the front end of the application itself is in the app folder within src.



```
app
    api
    applications
    components
    impacts
    login
    opendata
    podcasts
    projects
    publications
    resources
    tide
    favicon.ico
    globals.css
    homePage.css
    layout.tsx
    page.tsx
```

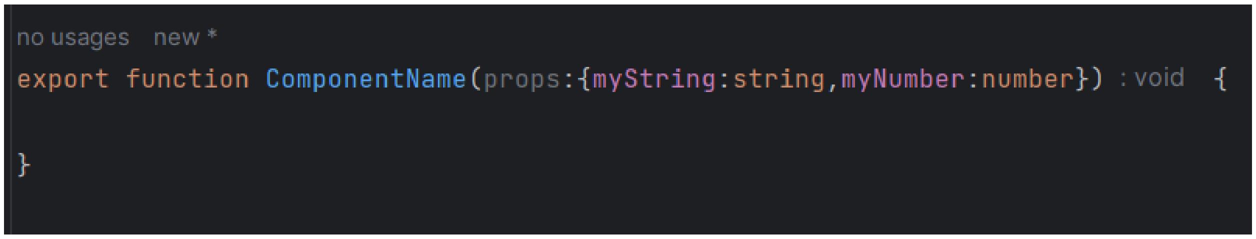
API is self explanatory. Components has various react components used throughout the application. The next 7 directories are all pages on the site. Edit(tide in the picture) is the unfinished main page edit page. Layout contains the layout model of every page on the site. The layout contains 4 components. Navbar, sidebar, body, and footer. Body is the current page the site is on. Navbar sits right above body and will remain there when you're scrolling through, and footer is right below body. Sidebar takes up the entire page when the menu button is pressed.





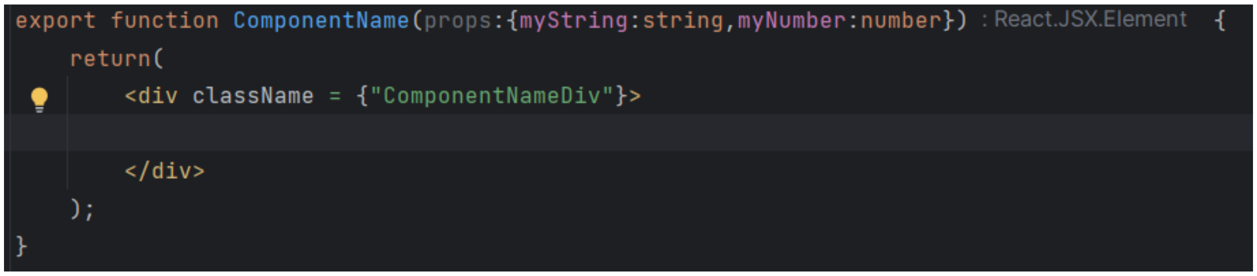
React Component creation guide

To make a react component, first make a .tsx file. Name it what you want the component to be called. Then type in "export function ComponentName() {}". For parameters, type in "props:{}" in the parentheses, and fill the braces in with whatever variables you want. Keep in mind to follow typescript syntax. The result should look something like this.



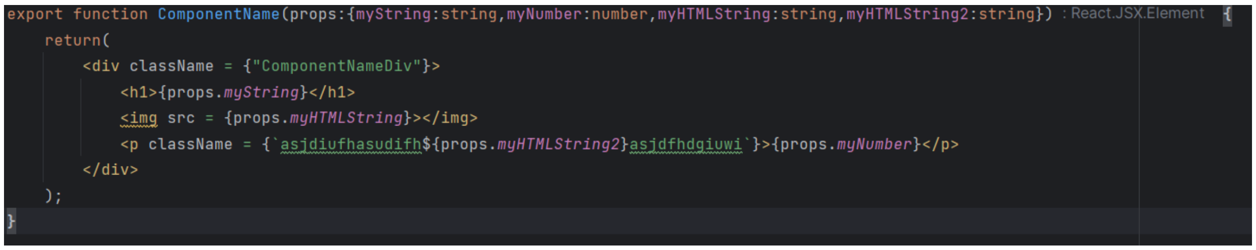
```
no usages   new *
export function ComponentName(props:{myString:string,myNumber:number}) : void  {

}
```

Within the braces of the function, type in return (); It is recommended you press enter after the first parenthesis to make writing the component easier. Within these parentheses, you can write your react component using HTML. You cannot have multiple HTML objects return, so it is recommended that you wrap everything in a div like this.



```
export function ComponentName(props:{myString:string,myNumber:number}) : React.JSX.Element  {
    return(
        <div className = {"ComponentNameDiv"}>

        </div>
    );
}
```

Now, you can fill the div with whatever you want in the component. In this example there's simply making a header and some text based on the two parameters, but theoretically you can fill it whatever you want. Look at the following image to see how to incorporate the parameters into the HTML. It's as simple as using {props.parameterName}. Although if you are putting a string *in the middle of* another string, you have to use $ before the braces.



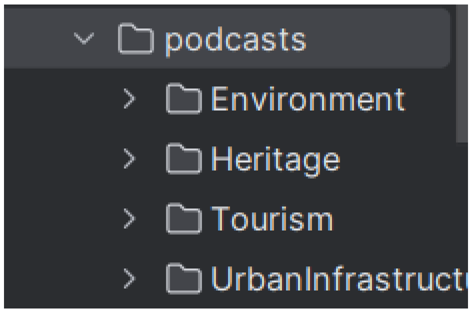
```
export function ComponentName(props:{myString:string,myNumber:number,myHTMLString:string,myHTMLString2:string}) : React.JSX.Element  {
    return(
        <div className = {"ComponentNameDiv"}>
            <h1>{props.myString}</h1>
            <img src = {props.myHTMLString}></img>
            <p className = {`asjdiufhasudifh${props.myHTMLString2}asjdfhdgiuwi`}>{props.myNumber}</p>
        </div>
    );
}
```

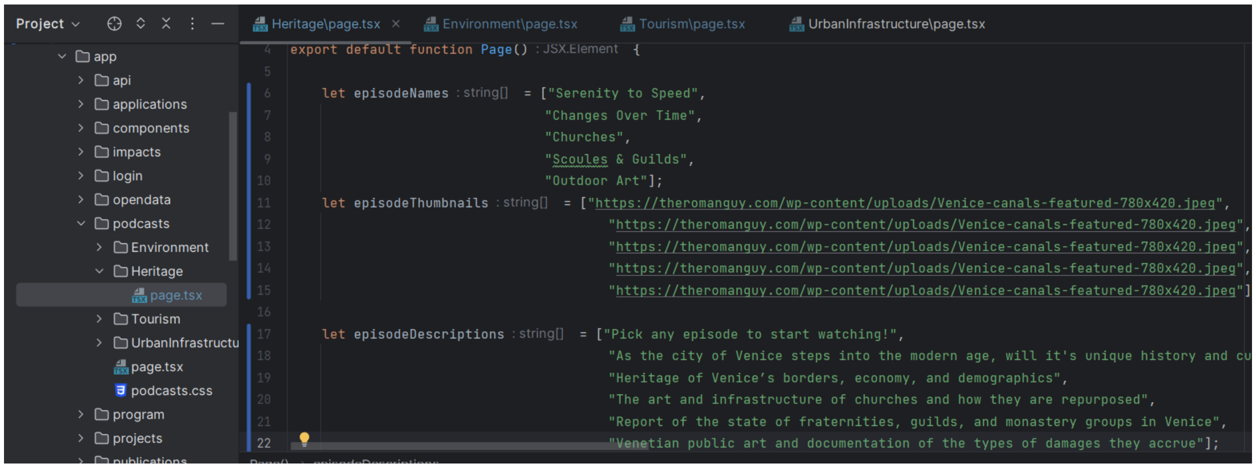
How to add/edit episodes

Until a way is developed to do it on the front end, we are leaving 2 ways to edit/add episodes.

1. Email sjdavid@wpi.edu(preferably), or samueljd881@gmail.com and send him the Episode Name, Episode Description, a Thumbnail, Youtube Link, and which of the 4 Themes it falls under. For the Subject of the email say something indicative that it's for Story IQP. When he receives it he will edit the source code, push to the github, and send an email in response saying he did.
2. Edit the source code manually, which is explained below

Refer above for how to get access to the github. Once obtained, go to src/app/podcasts. There should be 4 directories, each one is one of the 4 themes developed by the other members of the E23 Story Team. Enter the directory you want the episode to fall under, and open the page.tsx file.



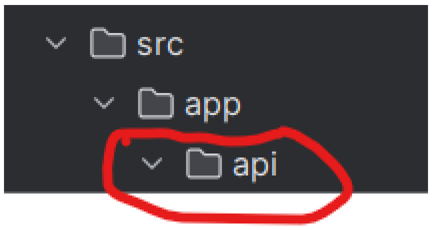


```
export default function Page() : JSX.Element  {

    let episodeNames : string[]  = ["Serenity to Speed",
                                    "Changes Over Time",
                                    "Churches",
                                    "Scoules & Guilds",
                                    "Outdoor Art"];
    let episodeThumbnails : string[]  = ["https://theromanguy.com/wp-content/uploads/Venice-canals-featured-780x420.jpeg",
                                          "https://theromanguy.com/wp-content/uploads/Venice-canals-featured-780x420.jpeg",
                                          "https://theromanguy.com/wp-content/uploads/Venice-canals-featured-780x420.jpeg",
                                          "https://theromanguy.com/wp-content/uploads/Venice-canals-featured-780x420.jpeg",
                                          "https://theromanguy.com/wp-content/uploads/Venice-canals-featured-780x420.jpeg"]

    let episodeDescriptions : string[]  = ["Pick any episode to start watching!",
                                          "As the city of Venice steps into the modern age, will it's unique history and cu
                                          "Heritage of Venice’s borders, economy, and demographics",
                                          "The art and infrastructure of churches and how they are repurposed",
                                          "Report of the state of fraternities, guilds, and monastery groups in Venice",
                                          "Venetian public art and documentation of the types of damages they accrue"];
```

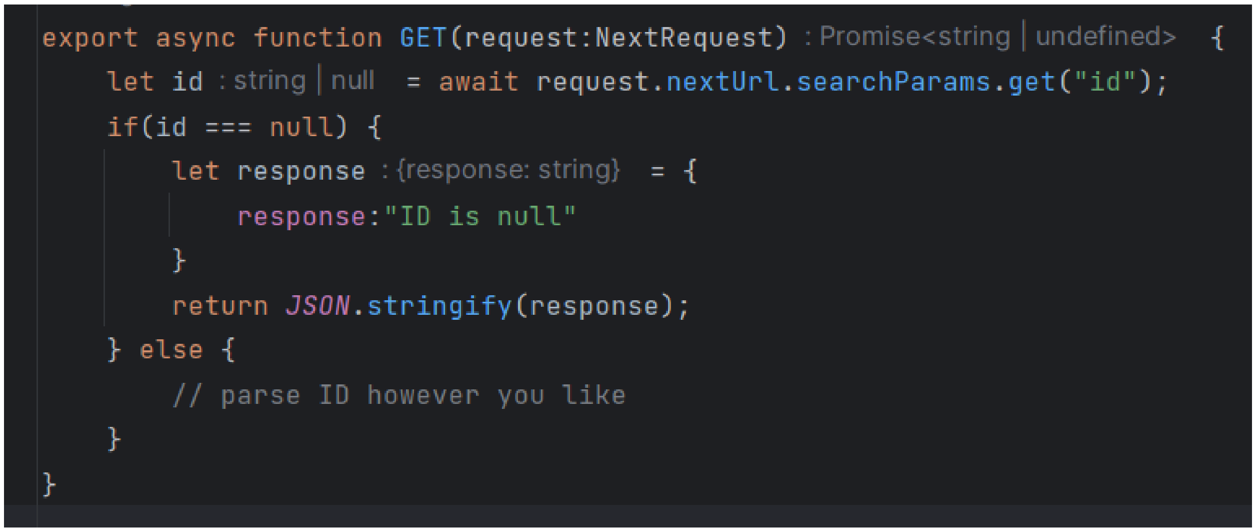
There are 5 variables, episodeNames (names of the episodes), episodeThumbnails (thumbail for it in the episode list), episodeDescriptions(description of the episode), episodeLinks(youtube link for the episode), and episodeTitles(title of the episode). Notice how episodeDescriptions, episodeLinks, and episodeTitles have 1 more string than episodeNames and episodeThumbails. The first string of the arrays in these 3 variables are used for when the page is first loaded. You can look at the react component at the bottom of this file to see how the page is structured using these variables, but if you just want to edit the episodes, then edit the variables accordingly to accommodate your needs. The episodes will be listed top to bottom in the order which they are in, in the arrays. Note that the number of episodes that will appear on the page is based on the number of episode names in episodeNames. However you edit the variables, make sure that episodeDescriptions, episodeLinks, and episodeTitles have 1 more string than episodeNames and episodeThumbails.

API route guide

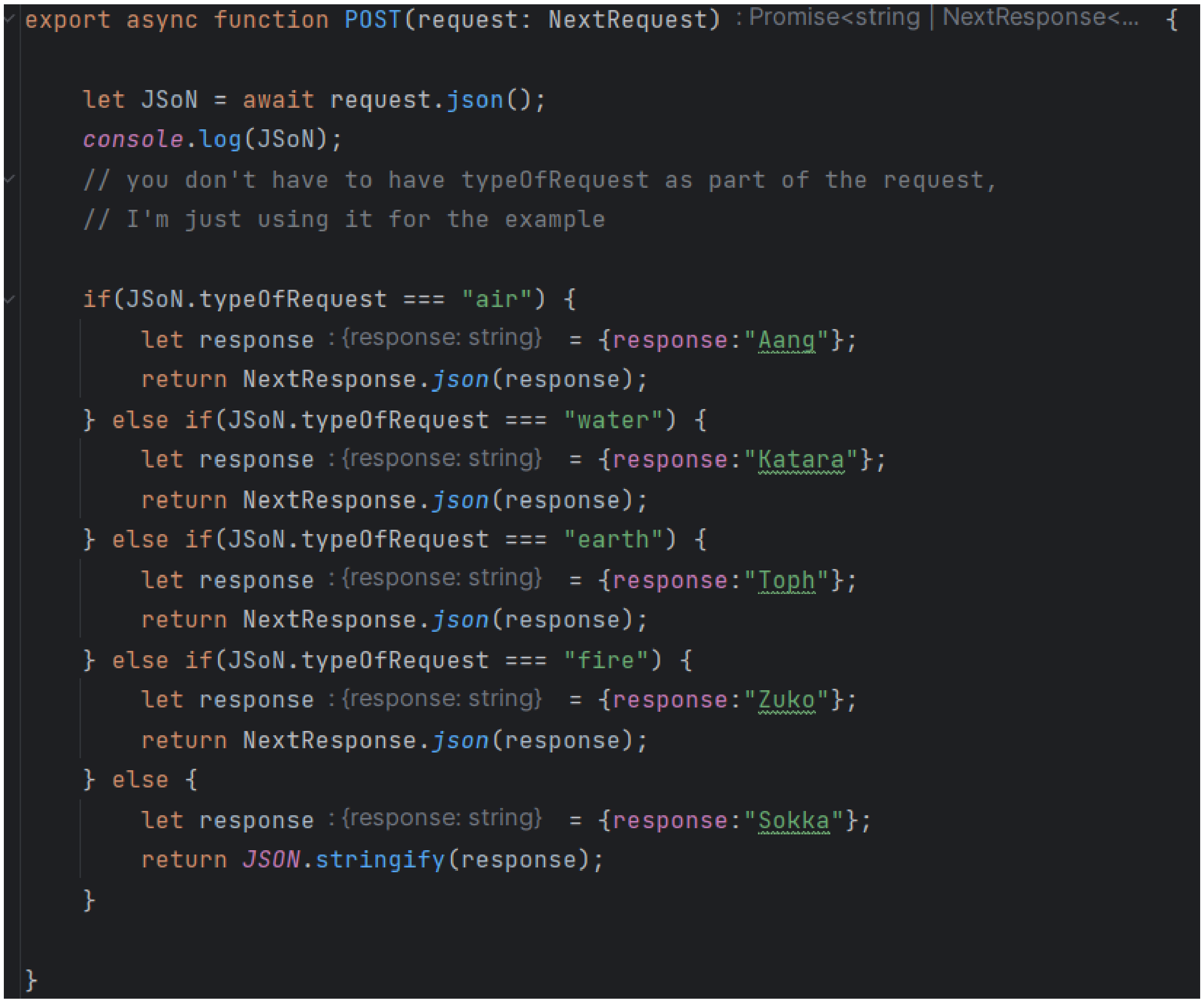
To create an API route, go to a subdirectory within the src/app/api directory.



Create a file called "route.ts" where you want the route to be. Within the route.ts file, create your HTTP request methods (GET,POST, PUT, DELETE, etc…). In the GET method I used ID, but you don't have to.

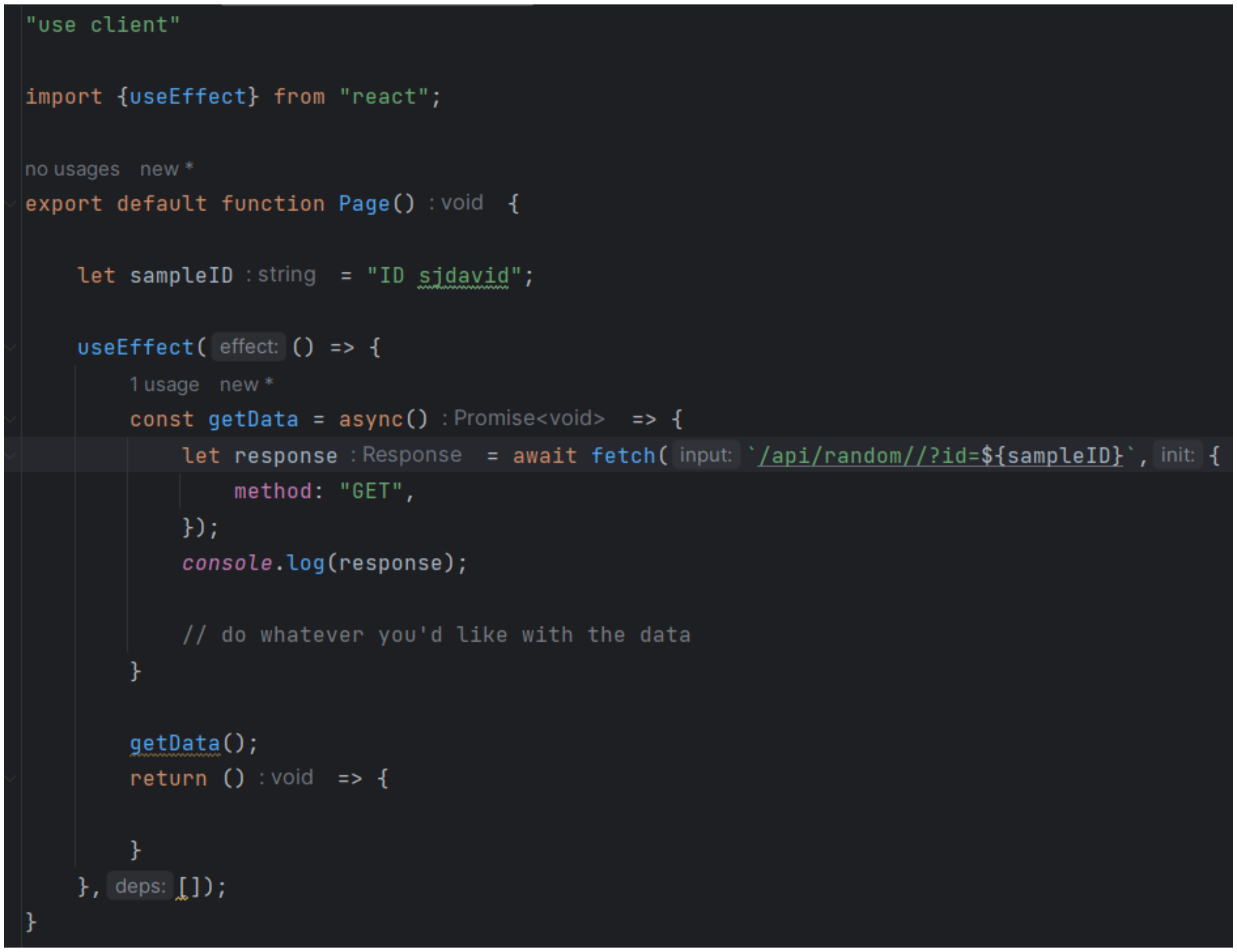


```
export async function GET(request:NextRequest) : Promise<string | undefined>  {
    let id : string | null  = await request.nextUrl.searchParams.get("id");
    if(id === null) {
        let response : {response: string}  = {
            response:"ID is null"
        }
        return JSON.stringify(response);
    } else {
        // parse ID however you like
    }
}
```



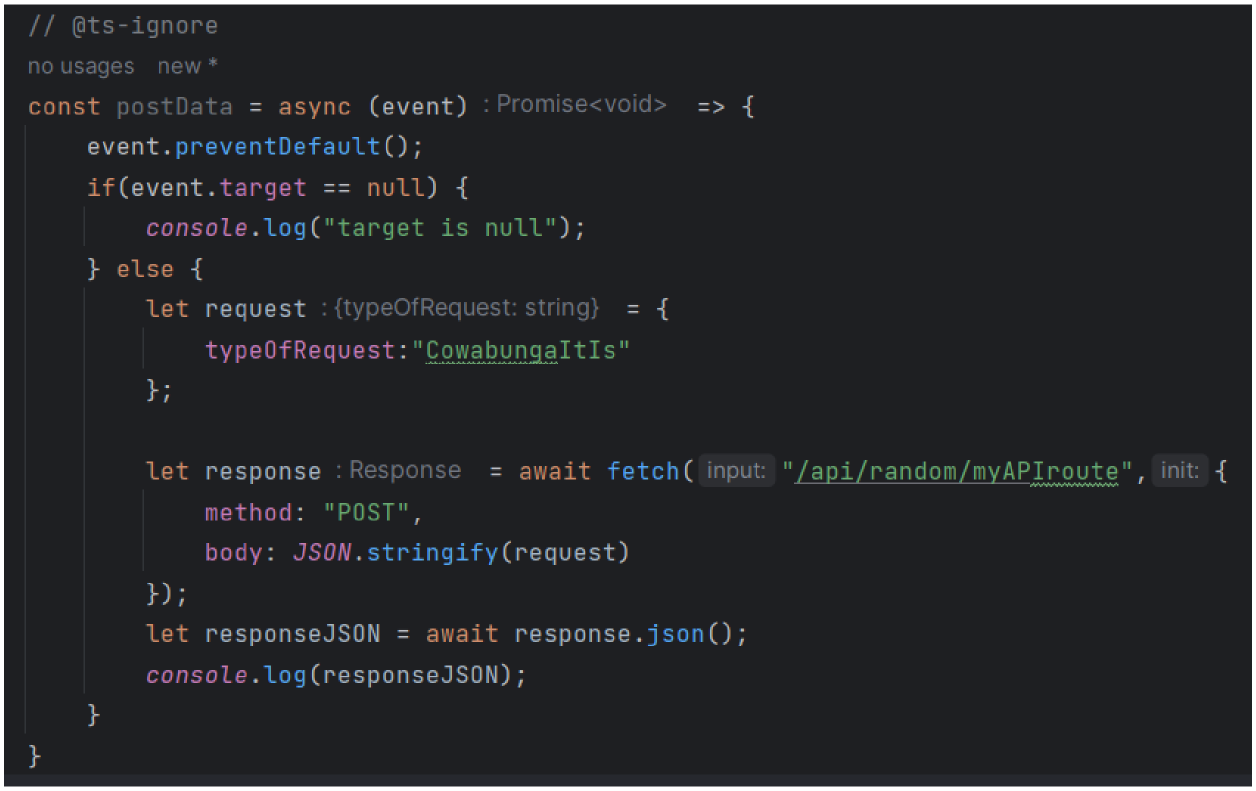
```
export async function POST(request: NextRequest) : Promise<string | NextResponse<...  {

    let JSoN = await request.json();
    console.log(JSoN);
    // you don't have to have typeOfRequest as part of the request,
    // I'm just using it for the example

    if(JSoN.typeOfRequest === "air") {
        let response : {response: string}  = {response:"Aang"};
        return NextResponse.json(response);
    } else if(JSoN.typeOfRequest === "water") {
        let response : {response: string}  = {response:"Katara"};
        return NextResponse.json(response);
    } else if(JSoN.typeOfRequest === "earth") {
        let response : {response: string}  = {response:"Toph"};
        return NextResponse.json(response);
    } else if(JSoN.typeOfRequest === "fire") {
        let response : {response: string}  = {response:"Zuko"};
        return NextResponse.json(response);
    } else {
        let response : {response: string}  = {response:"Sokka"};
        return JSON.stringify(response);
    }

}
```

If you wanted to implement GET in a tsx page, it would look something like this.



```
"use client"

import {useEffect} from "react";

no usages   new *
export default function Page() : void  {

    let sampleID : string  = "ID sjdavid";

    useEffect( effect: () => {
        1 usage   new *
        const getData = async() : Promise<void>   => {
            let response : Response  = await fetch( input: `/api/random//?id=${sampleID}`, init: {
                method: "GET",
            });
            console.log(response);

            // do whatever you'd like with the data
        }

        getData();
        return () : void  => {

        }
    }, deps: []);
}
```

GET is asynchronous, so you have to use useEffect in your page to handle it, since the page function is not, and you don’t know exactly when the data will be received.



```
// @ts-ignore
no usages   new *
const postData = async (event) : Promise<void>  => {
    event.preventDefault();
    if(event.target == null) {
        console.log("target is null");
    } else {
        let request : {typeOfRequest: string}  = {
            typeOfRequest:"CowabungaItIs"
        };

        let response : Response  = await fetch( input: "/api/random/myAPIroute", init: {
            method: "POST",
            body: JSON.stringify(request)
        });
        let responseJSON = await response.json();
        console.log(responseJSON);
    }
}
```

POST on the other hand does not have to be asynchronous, since you have to wait for the GET method to respond, but not for POST.