## Table of Contents

* Welcome
* Tech Stack
* Project Structure
* Setup
* Scripts
* Business Rules

## Welcome

Sup SDC team!

This documentation is intended to give you some insight into the design of the Parks Passport app. We hope to make it easier for you to get started adding on to the codebase and to help you understand the “why” behind some of the decisions we made.

## Tech Stack

### Overview

We’re going to talk about our tech stack and the reason we chose the things we did. But here’s a quick summary: - Next.js - Tanstack Query - Prisma - Valibot - Tailwind CSS - Typescript - Biome.js

### Next.js

Next.js is a powerful opinionated React framework. At NCDPR, we are currently migrating legacy PHP apps to Next.js, hence the odd choice of a full stack React framework for an offline first app.

The app itself is a Next.js app using the App Router. We aren’t using any server components since all data must be cached on the client. We chose to use route handlers to create explicit API endpoints which will come in handy for offline caching.

### Tanstack Query

Tanstack Query is a React library that simplifies data fetching and state management. It provides a set of hooks that make it easy to fetch, cache, and synchronize server state with the UI. Without it, you’d be stuck using useEffect and useState to manage data fetching, which can lead to a lot of boilerplate code and potential bugs. FYI, It used to be called ‘React Query’. [Their docs are very detailed.](https://tanstack.com/query/latest) but make sure you’re reading the docs for v5!

Tanstack Query also allows us to easily cache the state of API requests on the client, which is essential for offline functionality. It provides built-in support for caching, background updates, and stale data management, making it a great fit for our offline-first app.

### Prisma

Prisma is an ORM that simplifies database access and management. It provides a type-safe query builder and a schema definition language that makes it easy to define and manage database schemas. Migrations are a breeze with Prisma, and it generates types that we can use safely in code running on both the client and server. All things Prisma are imported from @prisma/client Note that while it’s called the client you should only use import from here on the server!!

The sole exception is where we export Prisma *types* from @/types.

### Valibot

Valibot is a runtime type validator. It lets you verify the shape of data at runtime and keeps the communication between the client and the server typesafe. You can create schemas and ‘infer’ typescript types from them.

### Tailwind CSS

Tailwind CSS is a styling framework that provides a set of utility classes for building responsive and customizable user interfaces. It allows us to quickly prototype and iterate on designs without writing custom CSS. Tailwind’s utility-first approach promotes consistency and reusability across the codebase. The styles are short and easy to read, and they have great IDE support.

### Typescript

Typescript is a superset of JavaScript that adds static typing to the language. It provides type checking at compile time, which helps catch errors early in the development process. Typescript also improves code readability and maintainability by providing clear type definitions and interfaces. It will save you from many mysterious undefined errors at runtime. It also is really useful if you change the name of things frequently, as you will get errors in all the places you need to update.

### Biome.js

Biome.js is a code formatter and linter that helps maintain code quality and consistency across the codebase. It provides a set of rules and guidelines for writing clean and maintainable code, and it can automatically format code to adhere to these standards. Biome.js also integrates with popular IDEs and text editors, making it easy to use in the development workflow. We chose Biome because it is faster than alternatives like ESLint and Prettier, and it combines both linting and formatting into a single tool. Also because ESLint can be a pain to configure properly and Biome works well out of the box with minimal configuration. We’ve added the Biome.js VSCode extension to the .vscode recommended extensions so it should prompt you to install it when you open the project.

## Project Structure

* public
  + fonts
  + icons
  + photos
  + stamps
    - parks
    - trails
* prisma
  + migrations
  + schema.prisma
* src
  + app
    - (auth)
    - (protected)
    - api
    - layout.tsx
    - page.tsx
    - globals.css
  + components
  + lib
    - validation
  + hooks
    - queries
  + types
  + middleware.ts
* package.json
* tsconfig.json
* next.config.ts
* biome.json

public contains all our static assets, such as icons, photos, and stamps prisma contains our database schema and migration files src is where the main application code lives: - app contains the Next.js app structure, including authentication and protected routes (pages) - components holds UI components used across multiple pages. if it’s only used on one page, it will be in that page’s directory’s components folder - lib contains utility functions and validation logic - hooks contains custom React hooks, particularly for data fetching with queries - types holds TypeScript type definitions, mostly joined fields for Prisma (such as Park & Address or Trail & TrailIcon) - middleware.ts is where we handle authentication and authorization logic for protected routes

Feel free to change or reorganize this however you want. You don’t have to use our project structure, but we hope it gives you a good starting point. The important thing is to keep your code organized and maintainable as you add new features. We’ve found this to be a good structure for our app, but you may find something that works better for you. Just remember to keep the public and prisma directories at the root level, as they are used by Next.js and Prisma respectively. We also recommend keeping a src directory at the root level to keep things organized, but you can remove it if you prefer. Just make sure to update the tsconfig.json and next.config.js files to reflect the new paths.

## Scripts

All project scripts are located in the package.json file under the scripts section. Here’s what they all do and what they are used for:

### General

* npm run dev
  + Starts the Next.js development server. This is the command you will use most often during development to see your changes in real-time.
* npm run build
  + Builds the Next.js app for production. This command compiles the app and prepares it for deployment.
* npm run start
  + Starts the Next.js app in production mode. This command is used to run the app after it has been built.
* npm run lint
  + Runs the Biome.js linter to check for code quality and style issues. This command is useful to ensure your code adheres to the project’s coding standards.
* npm run fix
  + This will run the Biome.js linter and automatically fix any issues it can. This is a great way to quickly clean up your code, but make sure you commit your changes before running this command, as it will modify your files.
* npm run typecheck
  + Runs the TypeScript type checker to ensure your code is type-safe. This command is useful to catch any type errors in your code before running it. Or if you are playing whack-a-mole after moving a file or renaming a field, this will help you find any type errors that may have been introduced.
* npm run clean
  + Cleans up the .next and node\_modules directories. You’ll only really use this if you’re having issues with your local environment and need to start fresh. It will remove all the built files and dependencies, so you’ll need to run npm run install again after running this command.
* npm run env:copy: Copies the .env.example file to .env.development and .env.local. This command is useful for setting up your local environment variables without having to do it manually. :)

### Database

* npm run db:migrate
  + Runs the Prisma migrations to update the database schema. This command is used to apply any changes made to the Prisma schema.
* npm run db:seed
  + Seeds the database with initial data. This command is useful for populating the database with sample data for development and testing purposes.
* npm run db:reset
  + Resets the database by dropping it and recreating it. This command is useful for starting fresh with a clean database. It will also run the migrations and seed the database again.
* npm run db:dump
  + Dumps the database schema to a file (dump.sql). This command is useful for backing up the database schema or sharing it with others. We use this to let us share any changes to the actual data within the database with other developers. It’s very clutch for keeping the database in sync across all your teams’ local environments.

## Setup

Okay now that you’ve seen a high-level overview of the tech stack and project structure, let’s get you set up to start developing.

First you must download nodejs. We recommend using a version manager like nvm to install Node.js. You can find instructions for installing nvm [here](https://github.com/nvm-sh/nvm).

nvm install --lts nvm use --lts

After you install node on your system, install the dependencies for the project using the Node Package Manager npm. Navigate to the root of the repo and run:

npm install

One you have the dependencies installed, let’s set up the database. We use MariaDB as our database, so you’ll need to have it installed on your machine for local development. You can find instructions for installing MariaDB [here](https://www.bytebase.com/reference/mariadb/how-to/how-to-install-mariadb-on-mac-ubuntu-centos-windows/).

Once you have MariaDB installed and running, you’ll need to create a database for the Parks Passport app. You can do this by running the following command in your terminal:

mysql -u root -p -e "CREATE DATABASE parks\_passport;"

Now let’s copy over the environment variables. In the root of the project, you’ll find a file called .env.example. We made a script to copy this file to .env for you, so you don’t have to do it manually. Run the following command in your terminal:

npm run env:copy

That will copy the example env files to .env.development and .env.local. You can now edit the .env.local file to set your local environment variables. Make sure to set the DATABASE\_URL variable to point to the database you just created. It should look something like this:

DATABASE\_URL="mysql://root:mariadb\_password\_here@localhost:3306/parks\_passport"

**Make sure you replace root:mariadb\_password\_here with the proper credentials you setup when installing mariadb!**

Next, you’ll need to ‘seed’ your database with some initial data. We have a script that will do this for you. Run the following command in your terminal:

npm run db:seed

After you seed, make sure you run the migrations to ensure your database schema is up to date. You can do this by running:

npm run db:migrate

If everything went well, it should run with no errors.

Now that you have your environment set up, you can start the development server by running:

npm run dev

This will start the Next.js development server and you should be able to access the app at http://localhost:3000.

You can find more infomation about all these scripts in the [scripts documentation](scripts).

## Business Rules

I hate the word “business rules” but I can’t find another term for this that fits better. Essentially, this describes all of the rules, relationships, and nuances we found out when building the application. These are mostly immutable since they describe things at NCDPR. Some of the names can be quite confusing, but we are stuck with these names that other people at the Division have been using long before even the paper passport was created. There are also passport specific rules that are more flexible.

The table names are the first monospaced bullet.

### Division Rules

#### State Park

* parks
* Each row represents a State Park
* These fields in the schema were chosen (and named) to be a superset of the information on the paper passport
  + park\_type
  + city
  + phone
  + email
  + established
  + landmark
  + you\_can\_find
  + accesses

#### Trail

* trails
* Each row represents a Park Trail or a State Trail
* State Trails will have a non-null stamp\_image and Park Trails will have this field set to null

### Passport Rules

The Prisma schema contains comments on the relationships between each of these, but for the most part they are one-to-many.

#### User

* users
* Logs in with an email and password
* We sign a JWT and persist it within the client’s cookies to keep track of their session

#### Note

* notes
* A user can create notes related to a park OR notes unrelated to a park (aka “general notes”)
* A user can only have a single note per park

#### Collected Stamp

* stamp\_collected
* A collectable associated with a park XOR a trail
* Only one stamp can be collected per feature (with one exception)
* Location verification
  + Verified stamp
    - User was within the bounds of a park/trail when stamp was collected
  + Unverified stamp
    - User was not withing the bounds of a park/trail when stamp was collected
* Collection types
  + Automatic collection
    - Users location was automatically submitted to the backend and they were within the bounds of a park/trail
    - Always verified
  + Manual collection
    - User manually initiated the collection from the park/trail’s page
    - If a manual collection is done within the bounds of a park/trail, then it is marked as verified
  + If a user has an unverified stamp and attempts to create a verified stamp, then the unverified one is soft-deleted and a new collection is created for them

#### Visit

* visits
* Essentially the same thing as a stamp except visits are only marked automatically and you can have multiple visits persisted in the database
* Used to track when & how often a user visits a park or a trail

### Bucket List Item

* bucket\_list\_items
* A small challenge that a user can complete
* Usually associated with a park but a few are not
* These are not verified and can be toggled freely at any time

### Collected Bucket List Item

* bucket\_list\_items\_collected
* Represents a user checking off a bucket list item as ‘done’

### Favorite Park/Trail

* favorites
* A user can mark a park or a trail as a favorite
* This can be used as a filter within the app

### Address

* addresses
* A street address and/or GPS location associated with a park/trail
* Parks will have an address\_id that points to the primary address associated with the park, usually a visitor’s center

### Some tidbits of info

* Read @types/index.ts and you can see what fields are joined
* You’ll notice that bucket\_list\_items has its own table, but there is only a stamp\_collection table. A stamp has no information on it’s own that is not directly associated with it’s respective park/trail
* For the most part, abbreviations within the division are the first two letters of each word in the park. So Lake Norman becomes LANO.
* Any tables where the rows have a relationship with a user have created, updated, and deleted fields.
* We only let users soft-delete things, that is, instead of removing a row from the database we set the deleted field to true.