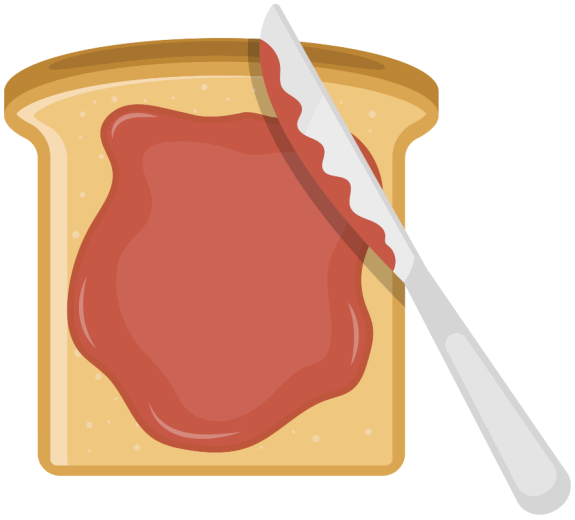
Cut into the JAMSTACK



Build and deploy a full-stack application using React and Next.js

MIKE CAVALIERE / echo/bind

About this Beta

Fellow developer:

I can't thank you enough for purchasing my book. It's the result of a lot of effort over the past few months (as well as a fair bit of coffee).

This is a beta release, which means a few things:

- It's over 100 pages, but it's gonna get bigger.
- There might be imperfections here and there. They will all get corrected eventually.
- YOU can play a part in shaping the book.

As you read and code, feel free to reach out about anything that is unclear, looks broken, or seems flat-out wrong. Or if there are things that feel like they're missing, I'll make a note and try to add them in.

In general, email me (at cutintothejamstack@echobind.com) with any suggestions or questions at any time.

This book is made for you, the engineers who want to build SaaS apps with the Jamstack. So your feedback will make this book better and better as it grows.

Thanks again, and enjoy!

—Mike Cavaliere cutintothejamstack@echobind.com

The What and Why

What you'll learn in this book

Cut Into The Jamstack (CITJS for short; we'll be referring to it as such from here on in) is designed to immerse you into a number of tools and concepts that power the Jamstack.

The Jamstack is a term for **J**avaScript, **A**PIs, and **m**arkup. It refers to the way many people are building websites nowadays—using JavaScript frameworks like React, Vue, and Angular to build a website, compiling the pages down to static markup (HTML files), and using APIs to populate those pages with data and dynamic functionality.

This book will introduce you to the process of building a full-stack Jamstack app from the ground up using some popular libraries like Next.js and React.

By the end of this book, you will: - have built a complete SaaS (software-as-a-service) application with these technologies working in tandem. - have gotten hands-on practice with some of the major concepts of each library we use (React, Next, and about 4 others); - learn some essential programming patterns for making these technologies work together in a Jamstack environment; - have a large, curated library of links to help you go deeper on each topic; - perhaps most importantly, be able to apply this knowledge and build your own Jamstack websites and applications.

Real-world programming

The app you'll be building in this book is a real app that will be shipping, and that (hopefully) at least some people will be using by the time the book is finished. I built it from a real need that I had, and I suspect others have as well.

Since people will be using it, I'll build it with professional software concerns in mind: stability, performance, maintainability, security, and so on. But also, practicality; since I'm one person shipping an app (and a book), I will be making conscious decisions on what *not* to do so that I can actually get it shipped.

Best practices are important, and I'll apply them where feasible in the app. *Shipping* is important as well, and those two can sometimes be at odds with each other (ever heard of the term, "premature optimization?"). So I'll make those decisions with practicality in mind.

My hope is that both the real-world programming practices I use in the book, as well as the decisions I make on how and what to build are also valuable for you, the reader, to apply in your own software projects.

Touching on popular technologies

We'll be using several libraries in this book. Each one of them is like an iceberg, and we'll just be seeing the tip of it or just below the surface.

I don't think there's any single book or online course that will teach you everything Next.js and React and Prisma (and so on) have to offer in one shot. In a single book, you can either go deep on one thing or shallow on many things. In CITJS, I chose to do the latter since the main thing I want you to take away from this is **how to make these technologies work together.**

In addition to that, I hope to impart some of the thinking behind decision making on projects like this, and give you a way to find out the deeper details for yourself.

Who this book is for

Cut Into The Jamstack was written with one goal in mind: to give engineers that are interested in full-stack JavaScript apps with Next.js and React a way to see a fully-built app in action, and have it reverse-engineered for them to learn from.

It's a fitting book for anyone that intends to build full-stack apps with Next.js and React. From junior engineers that have a fair understanding of JavaScript all the way up to Senior JavaScript engineers who haven't worked with the tools in this book yet (but want to save all that time Googling every detail, when they can get it all here in one place), anyone who wants a crash course in this stack will benefit.

This book might move a little quickly for anyone that hasn't a written a line of JavaScript in their lives yet. If this describes you, I highly recommend you start getting your hands dirty with JavaScript itself first. Using something like Code Academy or any good online JavaScript course will get you comfortable with some basics. Then, by all means, come back to this book, and it will start to make a lot more sense and help level you up way more quickly.

What you should know before reading this book

While CITJS is broken down in plain English as best as possible, programming novices may have a tough time initially. There are things we don't cover in the book (such as basic JavaScript syntax and concepts). That said, it's written to be very readable by people of all skill levels.

Here are the things you should have a fair understanding of beforehand:

JavaScript Syntax

You should be comfortable reading modern JavaScript, the type that Babel can work with. Basics such as the following should be at least somewhat familiar to you:

- Variable declaration and assignment
- Arrays and objects
- Defining and using functions
- Async / await syntax and promises
- Object destructuring (at least in concept)

HTTP Basics

You should understand a bit about how an HTTP request works and how API requests work in JavaScript (using `fetch` with `async/await` or `.then()` syntax); how a client and server communicate; HTTP codes; request and response objects.

That's really it. The more experience you have with full-stack development, the quicker you'll pick things up. But all skill levels can learn a ton from this book.

Tools and libraries we chose and why**

Here are some of the main JavaScript packages our app will be built upon, as well as some of the services we'll be leveraging. You probably know many or all of these already. Here are the reasons we chose each.

React

React is Facebook's increasingly popular front-end JavaScript library. Its tagline is "A library for building user interfaces," which is true at its core. But React has become so, so much more than that.

React is the foundation of almost all front-end work we do here at Echobind. We chose it as our core UI framework many years back for many reasons, but here's a brief list of the highlights:

- It can make your apps fast as hell.
- It's extremely enjoyable to write code in.
- It's got a huge, passionate open source community.
- It's got a very clean way of modularizing presentation and functionality.
- It's incredibly flexible and unobtrusive.
- It's got the backing of a major company, and as such isn't going anywhere.

Next.js

Next.js is a web application framework built by the people at Vercel. It includes React within it and adds a lot more to make building Jamstack (and full-stack) apps a lot easier.

Whereas React is an elegant way to build the front-end of an application, Next.js augments React by handling a lot of challenging-but-common full-stack tasks for you automatically. Namely:

- It's got a project starter which makes it very easy to get a basic app running.
- It's got its own routing system which handles both client-side and server-side routing seamlessly.
- It's got a simple, file-based convention for creating React-based website pages.
- Similarly, it lets you create custom API endpoints and serverless functions just by creating files in the right directory.
- It's got image optimization built-in, which is a large speed component of all webpages.
- By default all pages in Next.js render as static HTML, which makes everything fast.

Other stuff that is great about Next.js:

- It's also got the backing of a company with lots of funding and community (Vercel)
- It makes multi-language support really easy if you need it.
- It works seamlessly with Vercel's hosting platform, which makes deployment a cinch.
- Vercel's platform gives a ton of other benefits; see below.

TypeScript(-ish)

TypeScript is a superset of JavaScript; it's JavaScript with a few important things sprinkled on top of it. Namely, strong typing guard rails that JavaScript doesn't have on its own.

The values of this area as follows:

- You'll catch more errors while developing your app instead of when they're live for users to find.
- You'll reduce these types of issues while developing and save time since they can be confusing to debug (when your variable expects the number 1 but receives the string "1" for example).
- It'll be a bit easier to write complex code since it'll help you plan out how different pieces of code communicate with each other more clearly.

TypeScript can take some more effort to learn than plain old JavaScript, but the good news is you don't have to learn a lot of it to take advantage of the above benefits. So in this book, we'll be enabling TypeScript (since Next.js supports it by default) and using a minimal amount of TypeScript features to get you the most value from it with a small amount of new learning.

PostgresSQL

Postgres is possibly the most sturdy and robust of open-source relational database solutions. It's reliable, it's fast, it's powerful, it's supported by a lot of database hosting services (Amazon, Heroku, DigitalOcean, tons of others).

It's also worthy of note that we also chose a relational database over a NoSQL database like MongoDB, DynamoDB, or others. Whereas NoSQL databases are extremely powerful, relational DBs are versatile, scalable, and their data is easier to structure and read. We tend to choose them first as general-purpose solutions since they can work well for a great many situations.

Prisma

Prisma is a newer open source ORM (Object-Relational Mapping) for node.js. It's the tool we've chosen to interact with our database on this project for a number of reasons:

- It's schema-centric; you define how you want your database to look in one file, and it handles all the changes necessary to make that happen.
- It's got a clean, intuitive API. It's easy to understand and use.
- It's got excellent documentation.
- It's got a very active community.

I also think that because of the momentum Prisma is generating, it's wise to invest in learning it now in case it really takes off.

Vercel

Vercel is the company that created Next.js. They also have their hosting platform (named Vercel as well), which works very well with Next.js applications, as you'd imagine.

Benefits of hosting a Next.js app on Vercel:

- **Speed.** Their serverless functions run fast.
- **Convenience.** Deploying is a cinch, and each deploy gets its own url.

- **Cost.** They have a generous free plan.
- **Features.** Their analytics tools, for example, are awesome for Next.js applications.

Railway

There are plenty of options for hosting Postgres databases out there, but many cost money even for the basic plans. Railway is a newer hosting provider with an easy-to-use dashboard interface and a free plan that supports more than one database environment, which is unheard of in database hosting.

Since database costs are among the most expensive, I want to teach you how to get your app up and running with the smallest amount of money possible, so the free database plan from Railway is on point. Railway's interface is also excellent, and their service integrates beautifully with Vercel.

Cloudinary

You always need a storage service of some sort to handle the uploading, storing, and displaying of images in a web app. Amazon's S3, ImgIx, and Cloudinary are all great options with free or cheap plans and great features.

In the end, we chose Cloudinary since its free plan is generous, it has several great features like image transformations should we need that sort of thing, and it integrates well with Next.js's `<Image />` component.

react-hook-form

There are a number of form management libraries out there for React. react-hook-form is lightweight, easy to use, unobtrusive, and boasts far faster average performance than others like Formik and Redux-form.

next-auth

Authentication with Next.js can be challenging. Keeping track of the user's login status on both the front-end and back-end can be tricky, as well as lots of work. Integrating with different providers for one-click login/signup can also be very time-consuming.

next-auth simplifies all of that, giving you a standard API for doing all of the above and more. And it integrates with every provider imaginable, making one-click signup (which we'll be implementing in this

book) simple with Google, Facebook, or any other provider you can think of.

Conventions used in this book

Package management

We'll be using `yarn` as the package manager in this book, which is our default one at Echobind (and I use it by default in my own personal projects). So all of our package management commands and scripts will use it.

`yarn` is pretty interchangeable with `npm` / `npx`. So generally you can substitute one for the other: `yarn add` can be replaced with `npm install`; `yarn my-script-name` can be replaced with `npm run my-script-name`; `yarn external-package` can be replaced with `npx external-package`.

File naming

By default our source code files will be named with a .ts or .tsx extension. We'll largely be writing JavaScript with a sprinkle of TypeScript, so using the TypeScript extensions exclusively will keep it simple.

We'll use the `.tsx` extension for React components and pages, or any file containing JSX. `.ts` will be used for any files without JSX in them.

The only exception will be certain special files like next.config.js for which TypeScript isn't natively supported. Those will keep their default .js extension.

We'll also use `.env` instead of `.env.local` for environment variables. Even though Next.js supports `.env.local`, Prisma doesn't just yet.

Operating system

I do all of my development on a MacBook Pro, as does almost everyone at Echobind. As a result, some of the software I'll mention in the book is Mac-only software, but many apps like Postman for example exist on Windows and Linux as well.

The examples you see will also be Mac-based and Mac-tested, but should work easily on a Linux machine. They may not translate perfectly to a Windows machine, but altering them to work shouldn't be too much of a problem.

Iterative development

We'll be building the app in small pieces, and sometimes refactoring code as we go. There are multiple reasons for this.

One is for effective *teaching*: the smaller and more atomic our lessons are, the better I will be at explaining them concisely, and the more likely you, the reader will be to absorb them quickly.

Small and iterative is a valuable method of *learning* as well; getting something small working teaches you one thing, whereas refactoring it teaches you another, and layers upon existing knowledge. Each bit of typing and problem-solving is a drill, and contributes to memory consolidation and muscle memory.

The small, chunked lessons will also make this book more efficient to reference, since you'll less often have to dig through a long article to find a small piece you wanted to refresh your memory on.

And the final reason this book uses iterative development chunks is because it's an effective *development* paradigm. Especially when you're creating a small product with a small team (a team of one, in this case) and learning as you go, you want to try a small thing, ship it, and repeat, refactoring as needed.

The app we'll be building

JamShots is a simple app that allows people to create a photo gallery collaboratively, then share it with people.

After signing up, users will be able to create photo galleries and upload photos to them easily using their mobile devices or desktop browser. With a few clicks, they can invite collaborators to any gallery who can in turn upload photos. Gallery creators and collaborators can easily share a gallery link with anyone to view the photos.

The idea for this app was a personal one (see "Inspiration" below), but it also stood out to me as a great app to teach people on. Not only does it touch all aspects of the stack, but it's a lot more novel and fun than some full-stack learning apps. After all, who really wants to build another TODO list application?

JamShots features

Key features of JamShots will be:

- Easy gallery creation
- Easy photo uploads / photo deletion
- Easy sharing

- Beautiful photo galleries to view
- One-click signup with Google and Facebook
- Easy monthly subscription with Stripe Checkout

The complete app will contain the following screens:

- The homepage
- The dashboard (where we create and manage galleries)
- The gallery editor (where we upload and delete photos, share galleries, etc)
- A series of screens for signing up / logging in / payment, which will largely be handled by Google/Facebook/Stripe
- Static pages (Privacy, Terms and Conditions)

Inspiration for the app

It came out of a personal need. My wife and I are taking pictures of our son Leo often, and sharing them with family in text message threads. This wasn't ideal, for a number of reasons.

First, we had to do it multiple times for different groups of family and friends. Once for each group text thread. We'd select a bunch of photos and manually add them to each group text.

Most our relatives use iOS devices; iCloud sets an expiration date when you share a handful of files, which means that if I want to go find a set of photos I sent a while back, it may not be there. It's also not easy to find - ever try digging for a handful of photos you sent to your cousin last year?

The Google and Apple photo apps also just had too many features. They probably could all of do this stuff, but I wanted something that can do *only* the photo gallery sharing, without all the other bells and whistles. So JamShots was born.

Regardless of whether JamShots is the next great photo sharing SaaS app (probably not, but who knows!), it's going to be a great teaching tool for you to learn full-stack Jamstack development techniques on.

With that said, let's get building!

Laying Our Foundation

System Prerequisites

Your system will need the following:

- A recent version of nodeJS (12 or higher is good)

- A package manager (`yarn` is the book's default and all examples are written using it; `npm` can work as well, but you'll have to alter the examples slightly).
- A local installation of PostgreSQL (Postgres.app on the Mac is great; any will do)
- A code editor (I use VSCode)
- A way to interact with your PostgreSQL databases (the book uses Postico on Mac, plain old `psql` works also)
- A way to simulate HTTP requests (the book uses Postman)

Setting up our environment

Generating our app skeleton

`cd` to whatever directory you want your project to live, and run:

`yarn create --typescript next-app citjs-photo-app` Feel free to use a different app name if you like.

This will generate a Next.js directory structure, install its dependencies, and create a local Git repository for you. It will also include build scripts that we'll use to run our dev environment, and build our application for production. The `--typescript` flag makes it so that our project is automatically configured for TypeScript.

Configuring Next.js

Root URL path & other TypeScript helpers Edit your `tsconfig.json` and under `compilerOptions` add:

```
"baseUrl": ".",
```

This will make it so that instead of imports that look like this:

```
import ComponentName1 from '../../components/ComponentName1
import ComponentName2 from '../../components/ComponentName2
import ComponentName3 from '../../components/ComponentName3
import ComponentName4 from '../../components/ComponentName4
```

We can write ones that look like this:

```
import ComponentName1 from 'components/ComponentName1
import ComponentName2 from 'components/ComponentName2
import ComponentName3 from 'components/ComponentName3
import ComponentName4 from 'components/ComponentName4
```

...which is a little more readable. You can still do it the long way if you want and it will work.

There are some other TypeScript options that may make life easier, like setting `noImplicitAny` to `false`. I won't go into detail on all of them at this time, but here's the final `tsconfig.json` I'm using:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "sourceMap": true,
    "outDir": "dist",
    "strict": true,
    "lib": [
      "esnext"
    ],
    "esModuleInterop": true,
    "target": "es5",
    "allowJs": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "noEmit": true,
    "module": "esnext",
    "moduleResolution": "node",
    "noImplicitAny": false,
    "resolveJsonModule": true,
    "isolatedModules": true,
    "jsx": "preserve"
  },
  "include": [
    "next-env.d.ts",
    "**/*.ts",
    "**/*.tsx"
  ],
  "exclude": [
    "node_modules"
  ]
}
```

Optional: Linting and code formatting

ESLint will keep your code lint-free and automatically prevent certain types of bugs. Prettier will format your code and make it consistently more readable with no effort.

Next.js now supports ESLint automatically (as of version 11.0.0), so this part is done for us.

You can add Prettier

Run: `yarn add --dev prettier eslint-config-prettier`

You can run through the installation guides for ESLint and Prettier. Below are the configs I'm using at the time of writing this.

```
// .prettierrc.json
{
  "trailingComma": "es5",
  "bracketSpacing": true,
  "printWidth": 80,
  "tabWidth": 2,
  "singleQuote": true,
  "arrowParens": "always"
}

// .eslintrc.js
module.exports = {
  env: {
    browser: true,
    es2021: true,
    node: true,
  },
  extends: [
    'eslint:recommended',
    'plugin:react/recommended',
    'plugin:@typescript-eslint/recommended',
    'plugin:prettier/recommended',
  ],
  parser: '@typescript-eslint/parser',
  parserOptions: {
    ecmaFeatures: {
      jsx: true,
    },
    ecmaVersion: 12,
    sourceType: 'module',
  },
  plugins: ['react', '@typescript-eslint'],
  rules: {
    'react/react-in-jsx-scope': 'off',
    'react-hooks/exhaustive-deps': 'off',
    '@typescript-eslint/camelcase': 'off',
    '@typescript-eslint/no-empty-function': 'off',
    '@typescript-eslint/no-explicit-any': 'off',
    '@typescript-eslint/ban-ts-ignore': 'off',
    '@typescript-eslint/explicit-function-return-type': 'off',
    '@typescript-eslint/7006': 'off',
    '@typescript-eslint/explicit-module-boundary-types': 'off',
  },
```

```
};
```

Folder Structure

The Next.js app scaffold generates the following folders:

```
/node_modules - yarn / npm will install dependencies here

/pages - any tsx files in here represent website pages.

/pages/api - any ts files here become API endpoints automatically.

/public - static files (html, images, etc) placed here will automatically be served up by th

/styles - Next.js has built-in CSS support; this is where the .css files would normally go.

/.next - Next.js generates this hidden folder. You won't have to do much in here.
```

Throughout the course of the project, we'll also be adding the following folders. No need to create them now, since they'll be created during relevant steps in building the app.

```
/components - React components will go here, in subfolders.

/components/ComponentName - each component will have a CamelCased name, and will be stored i

/layouts - reusable page container templates will live here.

/lib - this is where we'll put reusable classes, functions, objects, and all general-purpose

/lib/client - front-end-only reusable code

/lib/server - back-end-only reusable code

/lib/shared - shared code that can be used either front-end or back-end
```

Setting up Chakra-UI

Chakra-UI Getting Started docs are here. Check there for the latest installation details.

If we haven't already in the dependencies section, we can add Chakra-UI and all its helpers:

```
yarn add @chakra-ui/react @emotion/react@^11 @emotion/styled@^11 framer-motion@^4
```

Then in `pages/_app.tsx` we add the ChakraProvider, a React Context API component that will make the Chakra theming system available everywhere in our app.

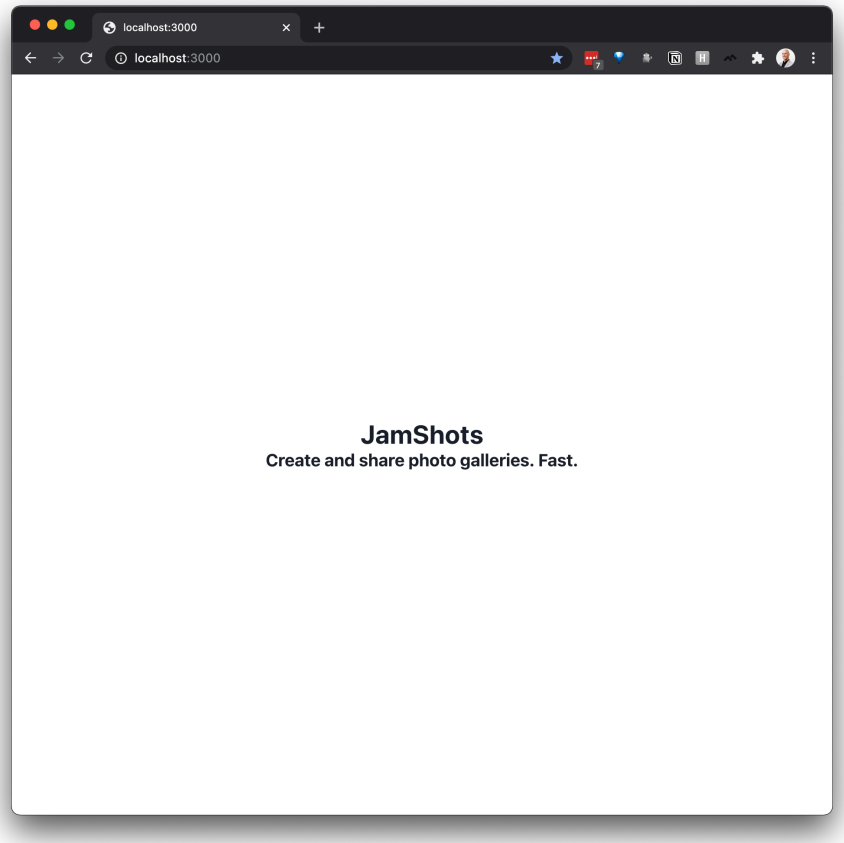
```
import { ChakraProvider } from "@chakra-ui/react"
import type { AppProps } from "next/app";

function MyApp({ Component, pageProps }: AppProps) {
  return (
            <ChakraProvider>
                <Component {...pageProps} />
            </ChakraProvider>
        );
}

export default MyApp;
```

Remove default landing page content

We can remove the homepage content that create-next-app generates for us, since we'll want our own custom content on the landing page. In `pages/index.tsx` we can remove the rendered contents and put some basic Chakra-UI components:

```
import { Heading, Flex } from '@chakra-ui/react';

export default function Home() {
  return (
    <Flex
      direction='column'
      minHeight='100vh'
      alignItems='center'
      justifyContent='center'
    >
      <Heading as='h1' size='lg'>
        JamShots
      </Heading>
      <Heading as='h2' size='md'>
        Create and share photo galleries. Fast.
      </Heading>
    </Flex>
  );
}
```

Now we've got a simpler landing page using Chakra:



Configuring Prisma

We're just going to install Prisma and add configuration. Their full tutorials are here and here, but we don't need all of those steps.

Install Prisma

To install the Prisma CLI tools and Prisma Client run:

```
yarn add --dev prisma
yarn add @prisma/client
```

Then to generate the default schema run:

```
yarn prisma init
```

This will create a `/prisma` directory in your project, along with a `schema.prisma` which is the database definition file.

Connect to the database

Add this to the `.env` at the root of your project:

```
DATABASE_URL="postgresql://myusername:mypassword@localhost:5432/cutintothejamstack"
```

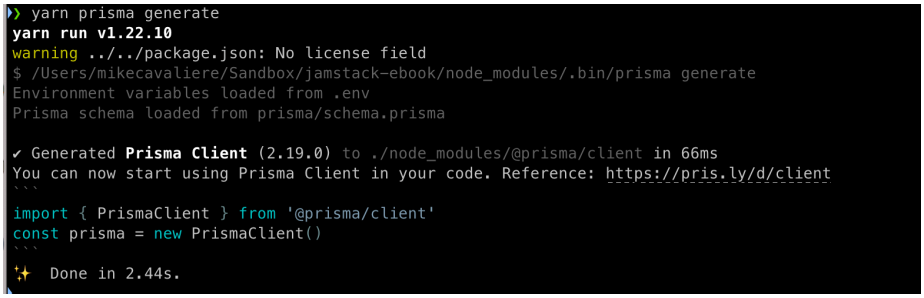
The username and password above should match the root Postgres user on your system; it may differ a bit from OS to OS. On the Mac using Postgres.app, it's the user you were logged in as when installing the app.

Generate prisma client

This step won't do much now, but it'll be necessary every time you update your database schema. Run this to generate an updated Prisma client:

```
yarn prisma generate
```

You'll see an OK message like the following:



Now we should be ready to start connecting to the database.

Building a Full-Stack Screen

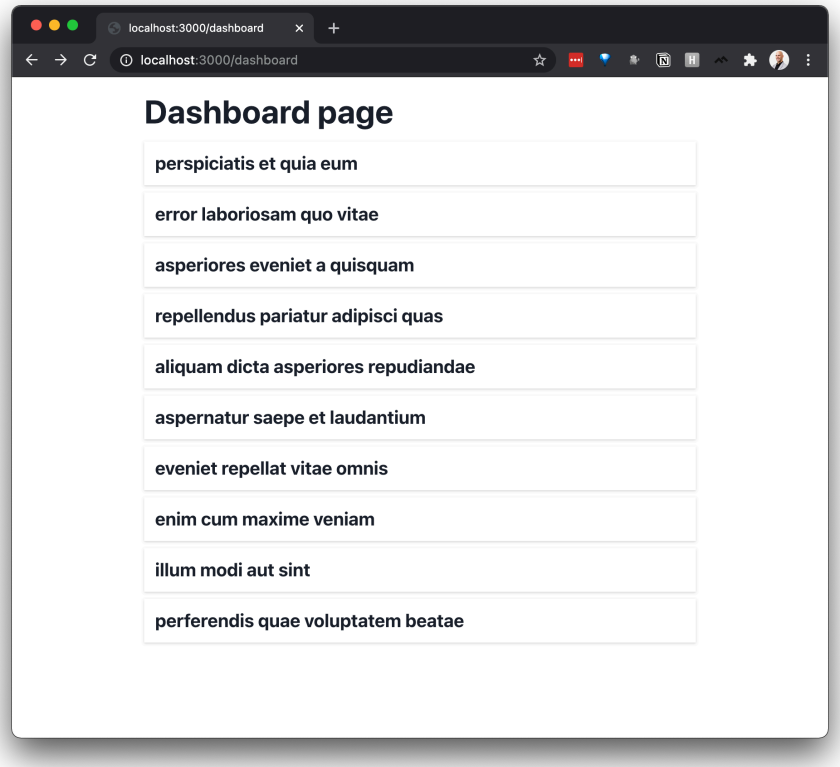
How we'll build our first screen: steps

Starting with one full screen will give you an understanding of some of the basic full-stack concepts we'll use in this app over and over.

In building one screen from top to bottom, we'll touch on a lot of important basic concepts for the full stack we're working with. You'll use a little bit of Next.js, React, Prisma and Chakra-UI.

The page we'll be building

The user dashboard is going to be the page we build. Specifically, the first version of the user dashboard.



By the end of this book, the dashboard will be the screen users see immediately after logging in. It will show the list of photo galleries the user has created or been added as an editor of.

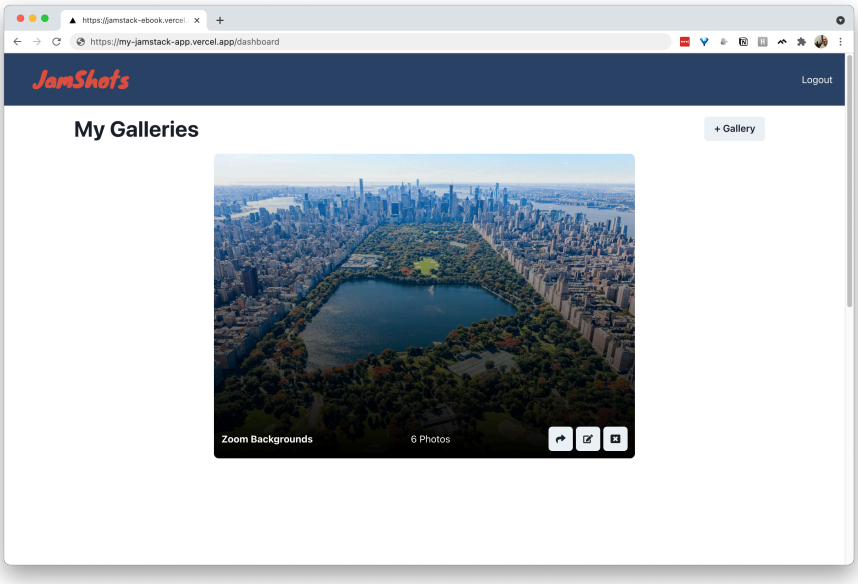
Here's the general plan:

1. We'll create a database, and add some data to it.
2. We'll create an API route, and use it to pull data from the database and send it to the front-end.
3. We'll create a page, fetch data from the API route, and show it on the page.

That's it for round one. Then we'll build on top of it.

Round 2 will involve full CRUD functionality for our app, so we can manipulate galleries easily.

Eventually the dashboard will evolve into the source of much of our app's functionality: the ability to create, manage, delete or share photo galleries.



Creating our first database table

Prisma has a great process for creating and altering database table structure.

Typically the database migration systems in most ORMs work *imperatively*; you tell them what steps to take in order, and they execute those steps. E.g., "create a Gallery table with these columns," and later on "add this column to the Gallery table," then "create the Photos table with these columns."

Prisma is the first node-based ORM I've known that operates *declaratively*; you tell it what you want the database to look like, and it automagically comes up with the steps to make it happen. This is the process for creating the initial database structure, and for changing it at any time.

We'll start by creating the simplest version of our Gallery table.

Initialize Prisma

If you haven't done this already, run `yarn prisma init`. This will create the `/prisma` folder and within it the `schema.prisma` file that will always contain our database structure.

Edit the schema

After the step above our `schema.prisma` file will look like this:

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

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

generator client {
  provider = "prisma-client-js"
}
```

Right now it just has basic connection information. Let's add our Gallery table:

```
model Gallery {
  id          Int      @id @default(autoincrement())
  name        String
  description String
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt
}
```

General Prisma conventions to note here:

1. We define our table with the `model` keyword. Table names are `PascalCased` as a Prisma convention.
2. The syntax of each field definition is:
   fieldName dataType attributes
3. Prisma field attributes are prefixed with an @ symbol.
4. Some field attributes take parameters, which can utilize Prisma functions like `now()` and `autoincrement()`

The purpose of most of the fields is self-explanatory if you've worked relational databases, but here's a breakdown:

- `id`: a unique identifier for the row. The `autoincrement()` function will cause this number to start at 1 and go up by one for each new row.
- `name` and `description`. These will contain descriptive information about this gallery.
- `createdAt` and `updatedAt`. These are timestamp fields, and will reflect the date and time at which the row was created or updated, respectively. The Prisma `now()` function and `@updatedAt` attribute are designed for this.

Generate and run migrations

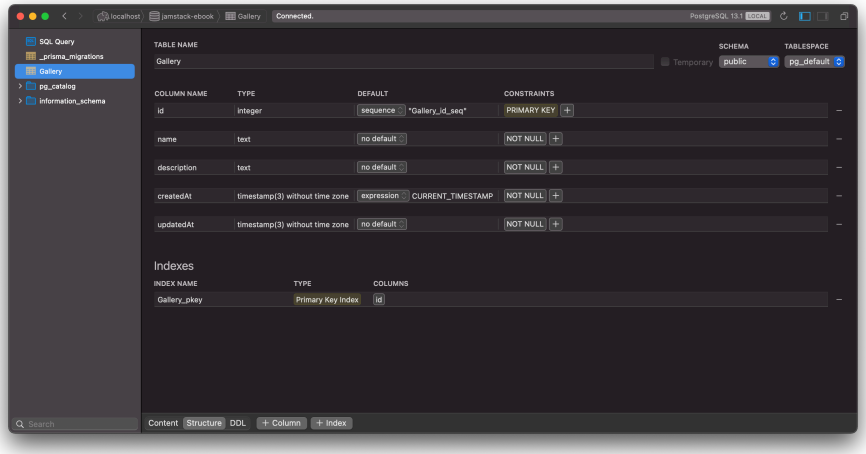
Now that the schema is defined, let's apply it.

```
yarn prisma migrate dev
```

You'll be prompted to give it a name, which can be anything useful. `CreateGalleries` will work.

```
❯ yarn prisma migrate dev --preview-feature
yarn run v1.22.10
warning ../../package.json: No license field
$ /Users/mikecavaliere/Sandbox/jamstack-ebook/node_modules/.bin/prisma migrate dev --previ
Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": PostgreSQL database "jamstack-ebook", schema "public" at "localhost:5432"

✔ Name of migration … CreateGalleries
The following migration(s) have been created and applied from new schema changes:

migrations/
  └─ 20210301013242_create_galleries/
    └─ migration.sql

✔ Generated Prisma Client (2.17.0) to ./node_modules/@prisma/client in 55ms

Everything is now in sync.
✨  Done in 105.00s.

~/Sandbox/jamstack-ebook  lesson-getting-started *6 !3 ?3
```

After running this command, you'll have a Gallery table in your database. If you connect with Postico or another GUI interface for Postgres, you'll see something like this:



Your Galleries table is now ready to get some data added to it.

Seeding the database

Our first screen in the app will pull a list of Galleries from the database and show it on the screen. As a prerequisite we'll need some dummy data in the database, since we haven't built any functionality to create Galleries yet.

So to facilitate this we'll use Prisma to *seed* the database; that is, we'll use it to create some dummy data that we can work with.

For reference, here's the documentation for seeding Prisma. Later on we'll adjust the seeder configuration to make it do advanced things and work around some Prisma configuration challenges; for now we only need simple data, so this'll be fairly straightforward.

Dependencies

If you haven't already, make sure Prisma Client is installed.

```
yarn add @prisma/client
```

Adding a dummy data lib

The Faker.js library is a great library for generating fake data of any kind. Let's add it:

```
yarn add --dev faker
```

Refresh Prisma Client

Prisma Client always needs the latest information about your schema to run queries against it. Make sure this is so by running:

```
yarn prisma generate
```

Edit package.json

To prevent some TypeScript issues when running the seed script, add the following to your `package.json` under the `scripts` block:

```
"ts-node": "ts-node --compiler-options '{\"module\":\"CommonJS\"}'"
```

Your `scripts` block will now look something like this:

```
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "ts-node": "ts-node --compiler-options '{\"module\":\"CommonJS\"}'"
  },
```

For background on this see Set up Seeding in TypeScript in the Prisma docs.

Structuring the seed file

The Prisma seed file is just a command line node script written in TypeScript.

By default it doesn't support the most recent JavaScript features, such as `import` statements, which we'll work around later - for now this just means we'll adhere to their rules and use `require` statements instead.

From the docs we import and instantiate a prisma client. Then we add a main() function that will run when we run the prisma CLI command for seeding. We'll also import Faker.

```
const { PrismaClient } = require('@prisma/client');
const faker = require('faker');

const prisma = new PrismaClient();


async function main() {

}

main()
  .catch(e => {
    console.error(e);
    process.exit(1);
  })
  .finally(async () => {
    await prisma.$disconnect();
  });
```

Creating galleries

Right now our Gallery table is really simple; all we need to give it is names and descriptions.

In the seed file we'll add a Prisma statement for creating 10 galleries:

```
const { PrismaClient } = require('@prisma/client');
const faker = require('faker');

const prisma = new PrismaClient();
```

```
const galleries = Array.from(Array(10).keys()).map(i => ({
  name: faker.lorem.words(4),
  description: faker.lorem.sentence()
}));

async function main() {
  await prisma.gallery.createMany({
    data: galleries,
    skipDuplicates: true,
  });
}

main()
  .catch(e => {
    console.error(e);
    process.exit(1);
  })
  .finally(async () => {
    await prisma.$disconnect();
  });

export {}
```

Breaking this down:

1. We import the libs we need up top (PrismaClient and Faker)
2. We create an instance of the PrismaClient, which we need to do anytime we want to run database queries.
3. We create a JSON object with dummy names and descriptions for 10 galleries, using some Faker methods designed for this.
    1. `Array(10)` generates a 10-element array, then `.keys()` gets an array of indexes for those (i.e., an array of the numbers 0 to 9).
    2. `.map()` returns a new array filled with 10 dummy gallery objects, each containing a name and a description.
    3. (Here's a link to this JS pattern for generating an N-length array.)
4. The rest is boilerplate right out of the Prisma docs for seeding. We run a prisma call to `createMany()`, giving it a list of galleries we want to create.
5. On the last line we add an `export {}` to prevent a specific TypeScript error like the one below.

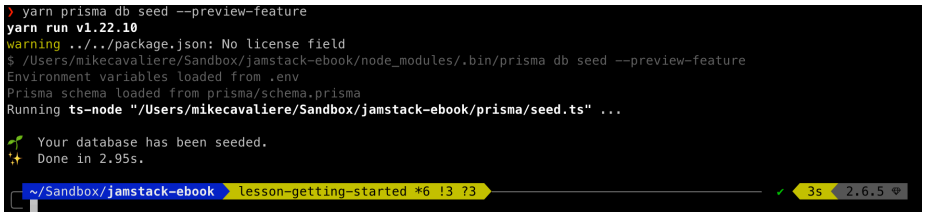
```
./prisma/seed.ts:1:1
Type error: 'seed.ts' cannot be compiled under '--isolatedModules' because i
t is considered a global script file. Add an import, export, or an empty 'ex
port {}' statement to make it a module.

> 1 | const { PrismaClient } = require('@prisma/client');
    | ^
  2 | const faker = require('faker');
  3 |
  4 | const prisma = new PrismaClient();
error Command failed with exit code 1.
info Visit https://yarnpkg.com/en/docs/cli/run for documentation about this
command.
```

Running the seed script

To run the seeder script we just created we will use the Prisma CLI again:

```
yarn prisma db seed --preview-feature
```

Your output will look something like this:



Adding a seed wrapper script

Sometimes you may get TypeScript issues when running the `prisma db seed` script directly. For this reason, and also for streamlining, I typically wrap Prisma commands with node scripts.

Add this to the `scripts` object in your `package.json` file:

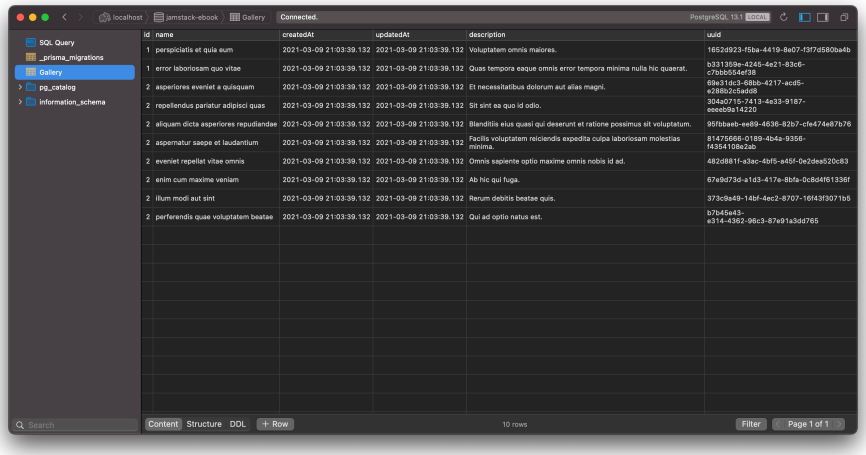
```
"db:seed": "ts-node --compiler-options '{\"module\":\"CommonJS\"}' -r tsconfig-paths/registe
```

This will register a seed command that you can run with `yarn db:seed`. It resolves at least one other TS-related error you might run into while running the seed script, and also makes it so that you can use `import` and other JavaScript and TypeScript features that might not be already supported.

Results

You can confirm that the Gallery table has some seed data in it by opening up the database in your favorite editor. You should see some-

thing like the screenshot below from Postico:



Creating our dashboard API endpoint

To serve up our gallery listing we'll create an API route that pulls a list of galleries from our Postgres database via a Prisma call, and renders them out on the page. That's it. Small iterations, remember?

Create API route file

`pages/api/galleries/index.ts` will be the file we're working in.

Structure of an API route Create-next-app generates a simple API route with the app. You can open up `pages/api/hello.js` (which was generated when you ran `yarn create next-app`) and see that it's very simple:

```
// Next.js API route support: https://nextjs.org/docs/api-routes/introduction

export default (req, res) => {
  res.statusCode = 200
  res.json({ name: 'John Doe' })
}
```

It's a function that takes in a request and a response object, then tells the response object how to respond. That's all that is required.

You can do a lot more with the request and response objects naturally. And you will. Check out the docs on API routes for a head start.

Fetch info with Prisma Inside `pages/api/galleries/index.ts` let's instantiate the Prisma client which we'll use to pull data, and add the boilerplate API route structure:

```
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export default (req, res) => {

}
```

Prisma recommends defining an async function that we run with a `catch` block (to intercept and handle / pass along any errors) and a `finally` block (that will always close the Prisma connection). This structure may change later (with explanations as to why), but let's use it for now.

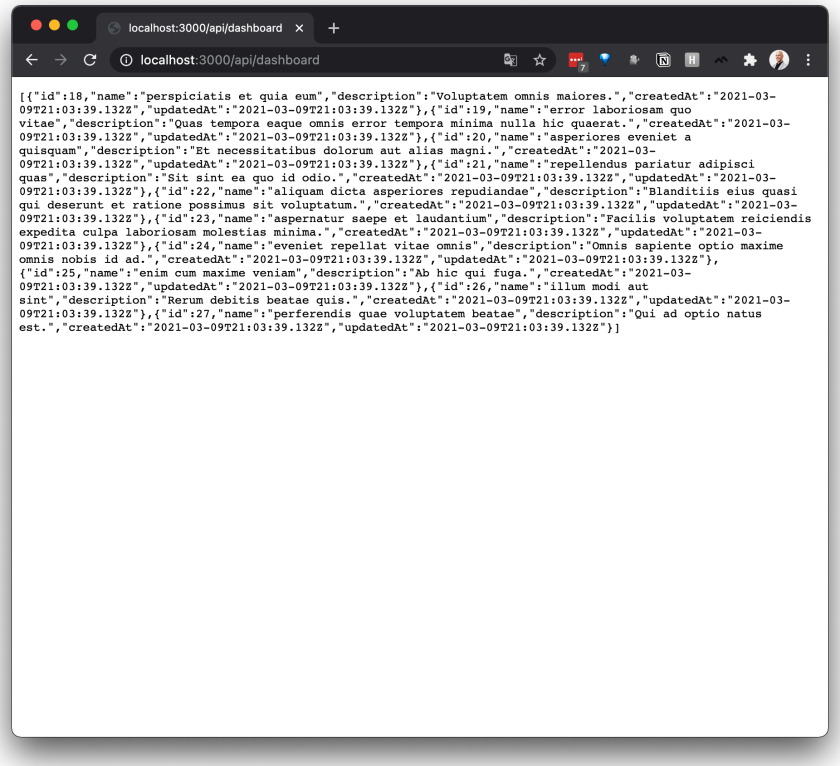
```
async function main() {
// ...
}

main()
  .catch(e => {
    throw e
  })
  .finally(async () => {
    await prisma.$disconnect()
  })
```

Then let's add a simple Prisma call to get all the galleries: `prisma.gallery.findMany()`. All together we have this:

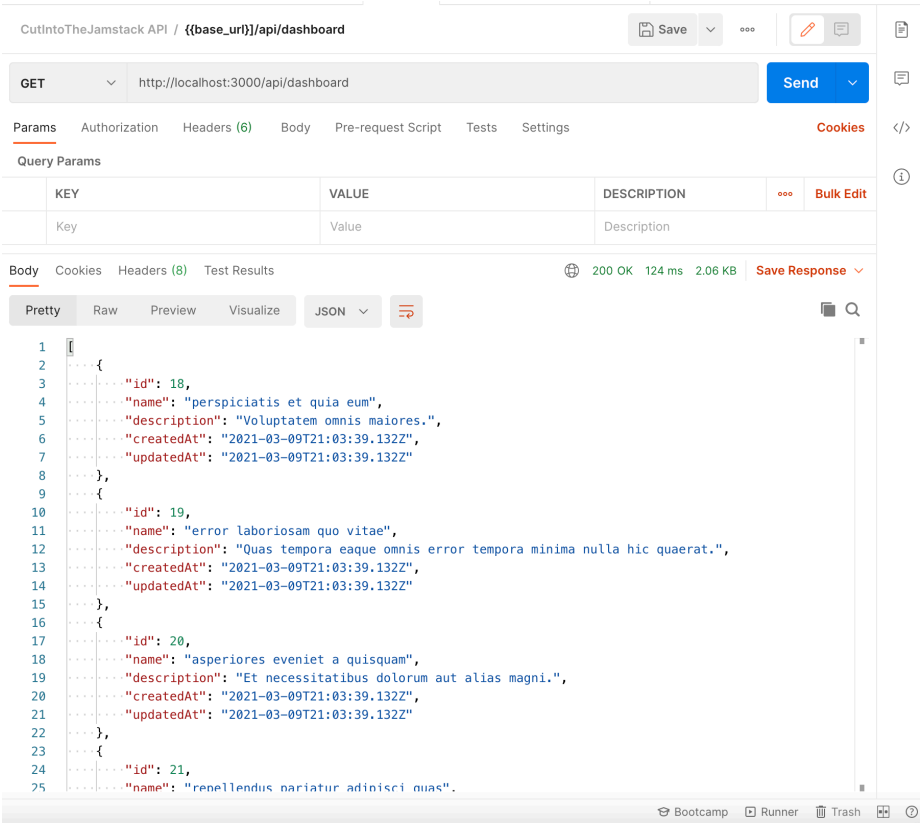
```
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export default (req, res) => {
  async function main() {
    const galleries = await prisma.gallery.findMany({
      orderBy: [{ 'name': 'asc' }]
    });

    return res.status(200).json(galleries);
  }

  main()
```

```
    .catch(e => {
      throw e;
    })
    .finally(async () => {
      await prisma.$disconnect();
    });
};
```

If we now hit http://localhost:3000/api/galleries in the browser, we'll see a nice JSON object with gallery data:



The same response in Postman:



Starting our dashboard page

Create a Next.js page

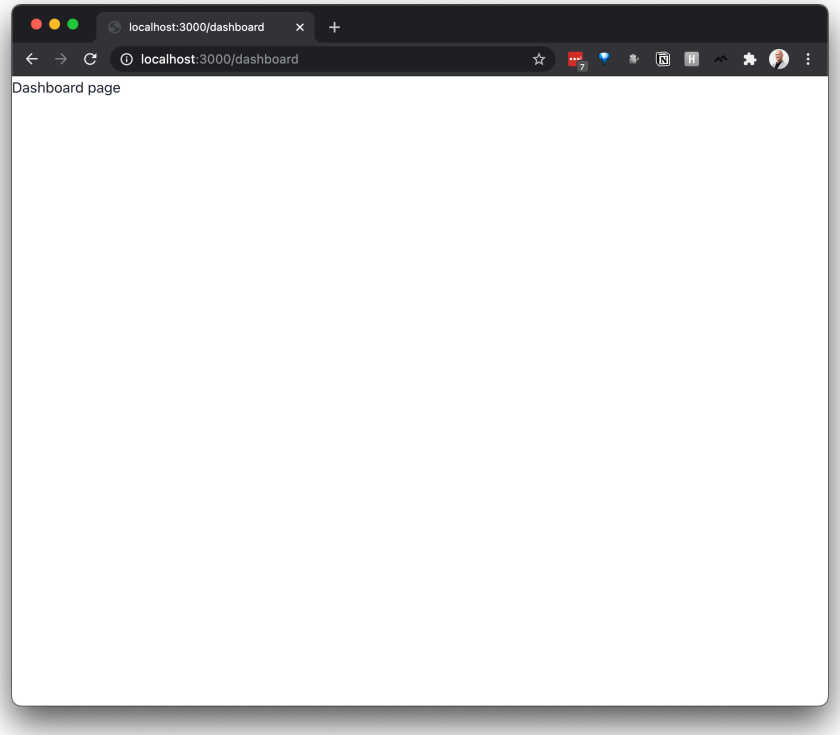
Adding pages in Next.js is very straightforward.

1. Create a file named `myPageName.tsx` file in the `/pages` directory. (Next.js supports other file extensions, but we're using this as a convention in this book).
2. Export a render function.
3. Visit the page in the browser. You're done.

We'll create `pages/dashboard.tsx` with the following shell:

```
export default function DashboardPage() {
  return (
    <h1>Dashboard page</h1>
  );
}
```

Now if we visit http://localhost:3000/dashboard in the browser we'll see our heading:



Pretty easy right? This fast page creation is one of the parts of Next.js I love the most.

Fetching data

For data fetching in our app, we'll use the SWR library created by the Vercel people. It's a simple, stateful data fetching library that is fast. It's also easier to use than something like react-query which is also phenomenal, but has so many features that we don't need right now.

Add SWR to the project like so:

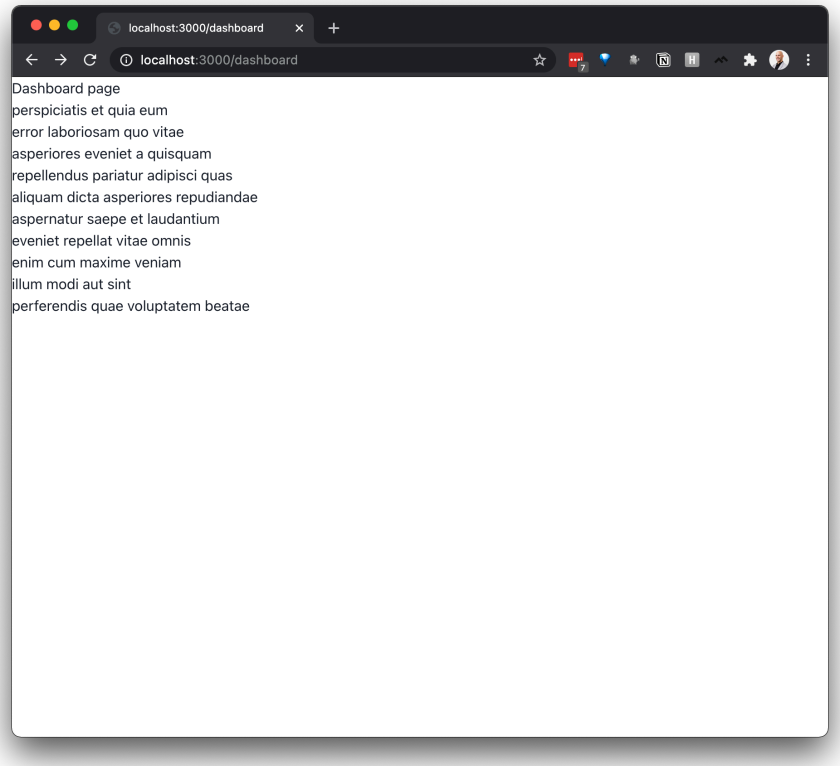
```
yarn add swr
```

Then we pull in the useSWR hook and use it to fetch our gallery data:

```
import useSWR from 'swr';

export default function DashboardPage() {
```

```
  const { data: galleries } = useSWR(`/api/dashboard`);

  return (
    <>
      <h1>Dashboard page</h1>
      <ul>
        {galleries?.map((g) => (
          <li key={g.id}>
            <h2>{g.name}</h2>
          </li>
        ))}
      </ul>
    </>
  );
}
```

Now the dashboard page fetches the JSON from the API and renders it.



Broken down, here's what we're doing here:

- We're telling the `useSWR` hook to intelligently fetch data from the `/api/dashboard`.
- We're assigning the `data` object that `useSWR` returns to a `galleries` variable.
- We check to see that the `galleries` variable exists before rendering, with TypeScript's optional chaining operator: by adding a question mark after the variable (`galleries?.map(...)`), we return `undefined` if `galleries` doesn't exist or is falsy, otherwise we return he result of calling `.map()` on the `galleries` array.
- When it exists, we iterate through the array using `.map()` and render out the name of each gallery in a `<li>` tag.
- `key={g.id}` ensures that each list item has a unique `key` prop, which is a React best practice (docs here). You'll see a console warning if you don't have this.

Additional things to note: * We enclosed the block in the tags <>...</> which is shorthand for a React Fragment. React functions always need to return a single container around everything, otherwise they'll throw an error. A fragment is a way to specify a container for everything without adding another actual html element.

`useSWR` gives us a lot more than the data though. Let's check the loading state and catch any errors that may show up:

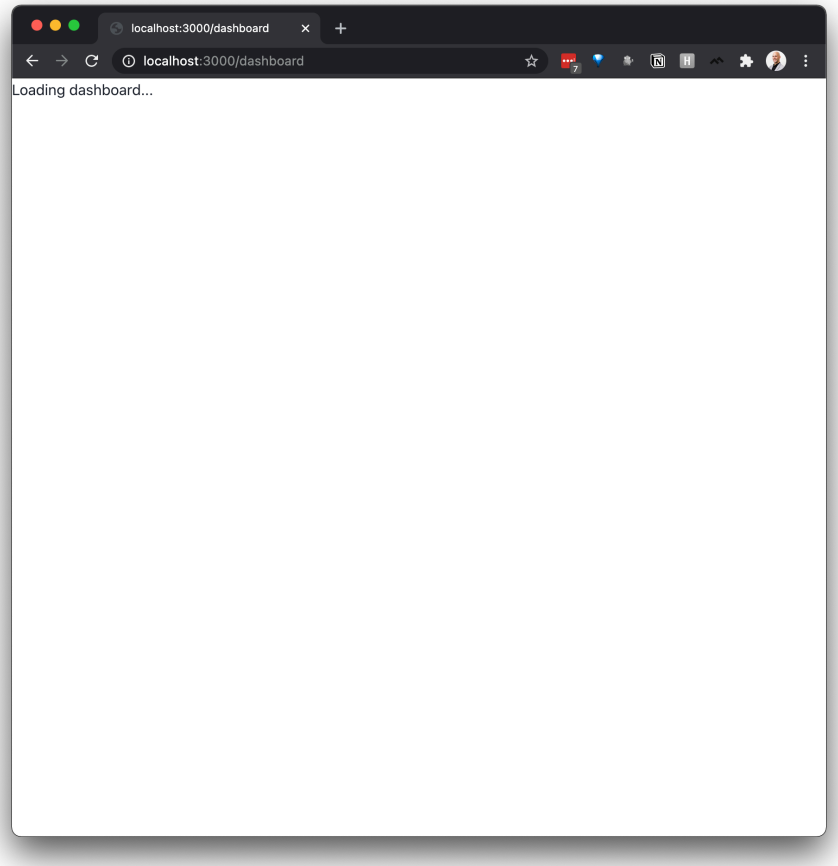
```
import useSWR from 'swr';

export default function DashboardPage() {
  const {
    data: galleries,
    isValidating: dashboardIsLoading,
    error: dashboardFetchError,
  } = useSWR(`/api/dashboard`);

  if (dashboardIsLoading) {
    return <h1>Loading dashboard...</h1>;
  }

  if (dashboardFetchError) {
    return <h1>Error loading the dashboard.</h1>;
  }

  return (
    <>
      <h1>Dashboard page</h1>
      <ul>
        {galleries?.map((g) => (
          <li key={g.id}>
```

```
            <h2>{g.name}</h2>
          </li>
        ))}
      </ul>
    </>
  );
}
```

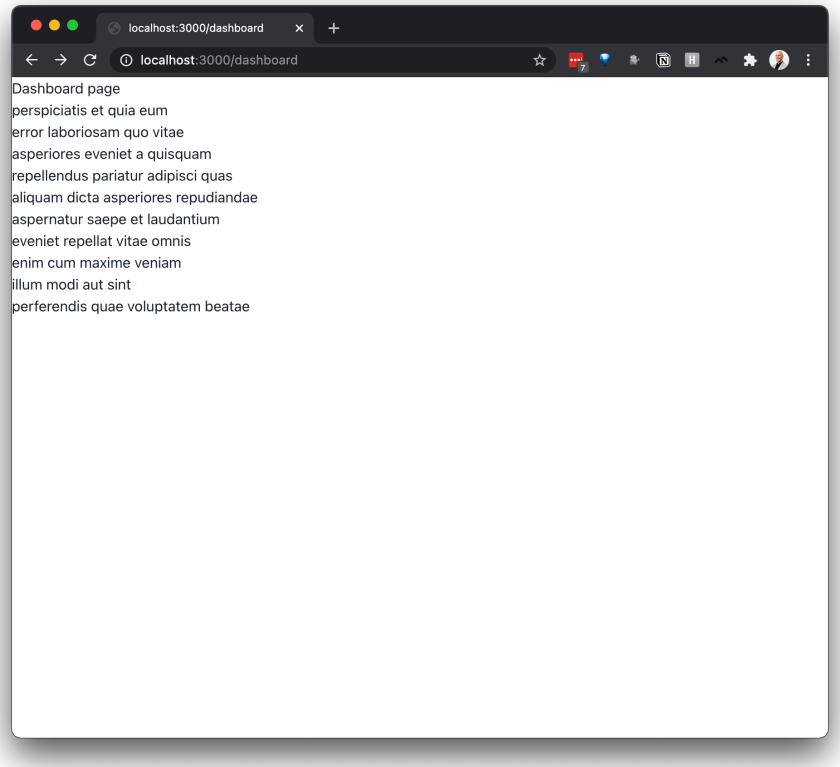
Breaking this down:

- We return 3 state objects from SWR (docs):
  - `data`: the gallery data if it is done loading.
  - `isValidating`: false if the data is fetching or revalidating against the cache.
  - `error`: a fetch error if one exists.
- I've renamed all of the variables that get returned for conciseness, since we'll have other state variables like this in the future that check on different things.
- If the dashboard is currently loading, or there is an error, we return an appropriate message. Otherwise if both of those checks pass, we render the gallery content.

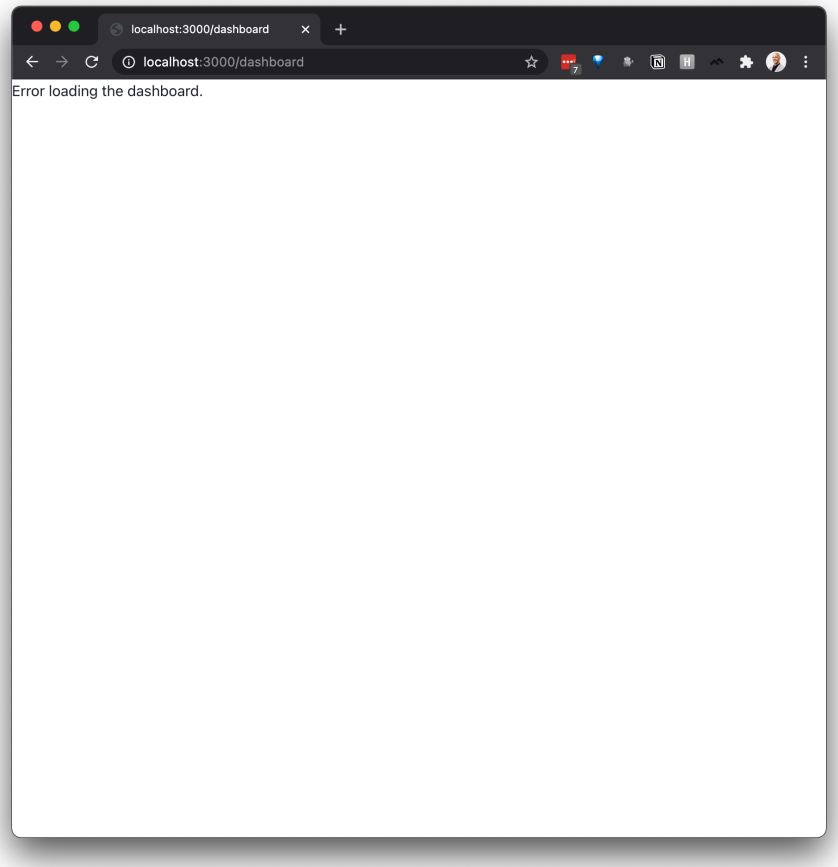
Going back to the dashboard in the browser, you'll briefly see the loading content:



Then when the loading is done, you'll see the main screen as before.



We can test the error state by going into `/pages/api/dashboard.ts` and adding a `throw "error!"` statement at the top of the file.



Adding some visual elements with Chakra-UI

Let's make this page presentable.

A Chakra-UI primer

Some light reading on Chakra to get you familiar with some concepts we'll use in the book: - Their "Getting Started" page. Scroll down on the left and you'll see examples of every component they have. Some will be immediately obvious as to their usefulness (Accordion, Tabs, etc), and some may not. - Their Style Props documentation. This talks in-depth about how the components take in styles as properties. They have various shorthands which are helpful. - Responsive Styles will explain you the methods we'll use to make a component's style properties change at different screen widths.

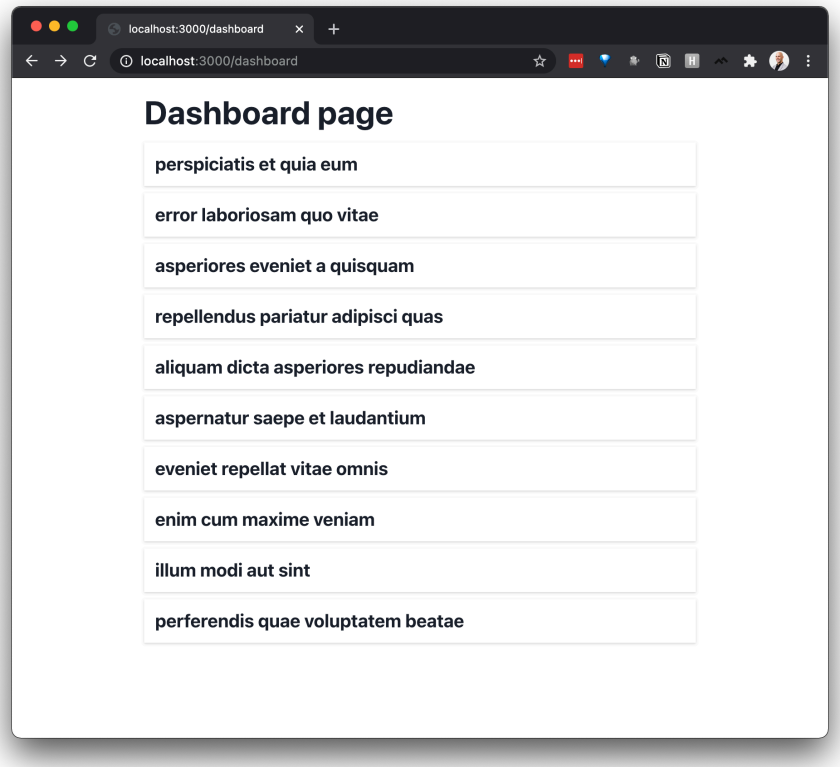
The main things to know are that: - their components are super modular; each is dedicated to a specific purpose. - they're also very composable; they're meant to be used inside each other easily - styles are passed in through props as mentioned above. There's no need to mess with React's `className` prop, ever.

Adding some simple styling with Chakra

```
import useSWR from 'swr';
import { Flex, Heading, Box, VStack } from '@chakra-ui/react';

export default function DashboardPage() {
  const {
    data: galleries,
    isValidating: dashboardIsLoading,
    error: dashboardFetchError,
  } = useSWR(`/api/galleries`);

  if (dashboardIsLoading) {
    return <h1>Loading dashboard...</h1>;
  }

  if (dashboardFetchError) {
    return <h1>Error loading the dashboard.</h1>;
  }

  return (
    <Box
      m='0 auto'
      p={5}
      maxWidth={{
        sm: '100%',
        md: '100%',
        lg: '40em',
        xl: '50em',
        '2xl': '74em',
      }}
    >
      <Heading size='xl' mb={3}>
        Dashboard page
      </Heading>
      <VStack spacing={5}>
        {galleries?.map((g) => (
          <Flex
            width='100%'
            direction='row'
```

```
            key={g.id}
            radius={10}
            boxShadow='base'
            p={3}
          >
            <Heading size='md'>{g.name}</Heading>
          </Flex>
        ))}
      </VStack>
    </Box>
  );
}
```

This will make the dashboard screen look a bit neater:



Here's the code breakdown:

- We wrap the page in a `<Box>` component (docs) which is an abstraction of a `<div>`.
  - We apply some margins and padding using the `m` and `p` short-

hand props. The values passed in are predefined units detailed in the spacing docs.
    - We set a max width that changes based on the breakpoint. (i.e. when the screen is at the small or medium breakpoint, max-width is 100%, at the large breakpoint it's 40em, and so on). These breakpoints have default values defined by Chakra, but can be customized if needed.
- The `<Heading>` component is an abstraction of `<h1>`...`<h6>` tags. We make it large and give it bottom margin (`mb`).
- The `<VStack>` component is the vertical version of the `Stack` component (docs). It's for a vertical list of things (of any type), with a defined space between them. In our case, these things will be the titles of galleries.
- The `<Flex>` component is a `<Box>` with some defaults made for applying CSS flexbox. We set these to have a `flex-direction="row"` so that they expand the full width of the container . Then we make it pretty with padding, a shadow and a border radius.

Deploying to Production

Ship, Ship, Ship - Going to Production

The whole purpose of building an app, other than doing it just for learning purposes, is so that someone, somewhere can see it and use it.

This is why we'll be shipping our app to a public url early and often. The reasons for this are manyfold:

- To resolve production-only configuration issues sooner rather than later.
- So we can regularly see if our development changes broke anything on a real server.
- To reduce the fear of going to production that some have initially.
- If we want to, to share our progress publicly.
- To learn techniques surrounding our hosting providers as we go.

Our hosting providers

We've cherry-picked two providers out of the many many options out there based on 2 important factors when shipping a SaaS application as a solo developer: ease of use and cost.

Both providers, Vercel and Railway, have free plans which allow more than enough flexibility to deploy and scale this app to a point of viability; that is, we can operate for free long enough to see whether

someone will pay for this thing we'll building.

At that point, if we need to grow past the free plans and start paying for database hosting or serverless function hosting, chances are we'll be making at least a little money. That's the strategy behind the approach we're teaching in this book.

How deployment will work

The process for deploying is relatively simple once it's set up.

Once it's set up, any commits to the `main` branch will do the following:

1. Deploy our code to Vercel;
2. Run any database migrations on Railway;
3. Give us a preview url (or production url) to view the app with.

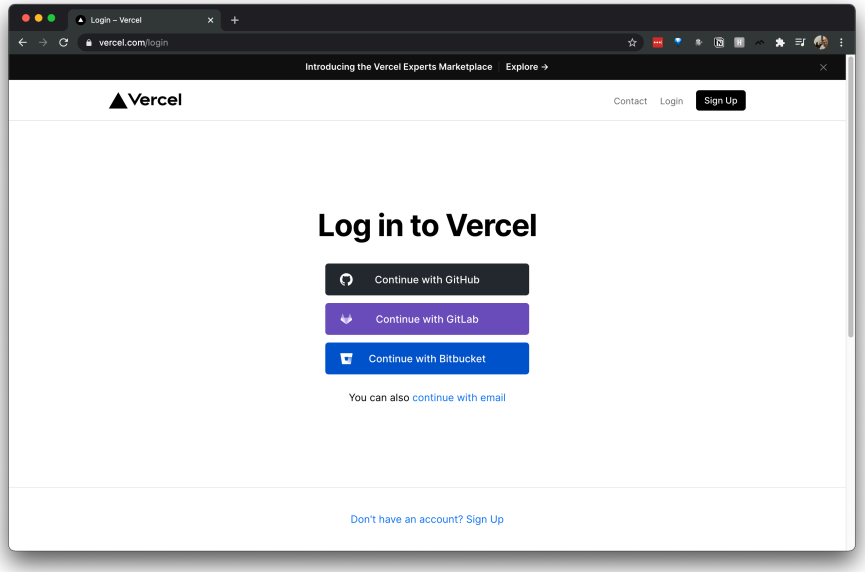
Configuring our setup for deployment

Vercel will host our code, and Railway will be our database provider. What's great about these two providers is that they interact very nicely with each other, so using them together makes setup a breeze.

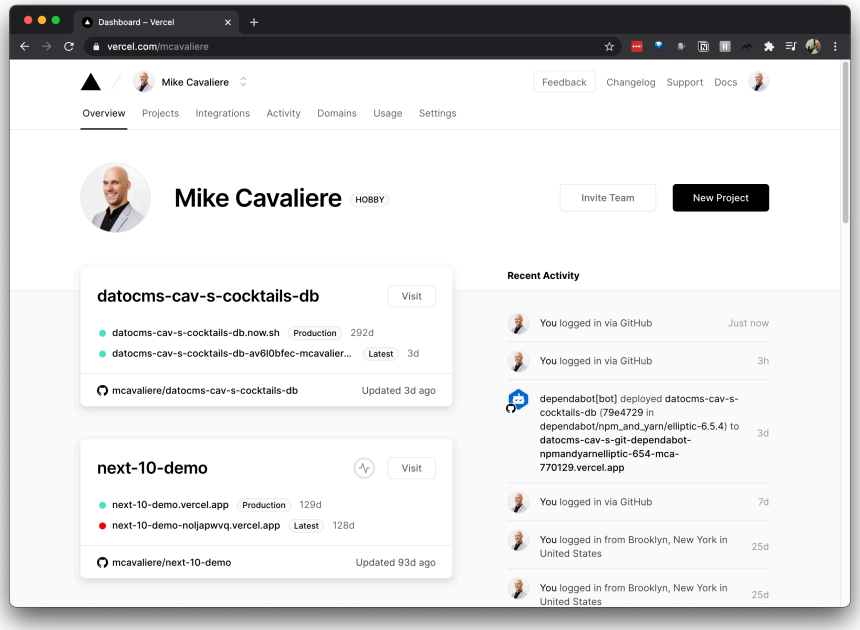
Configuring Vercel

Configuring Vercel to run a Next.js app is quite easy.

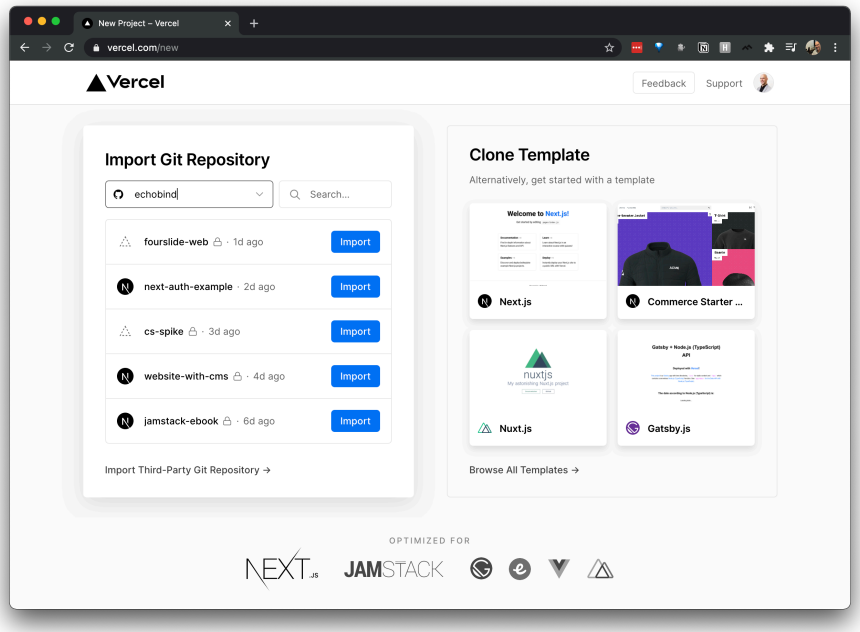
Visit vercel.com, and login with your GitHub account.



Click `New Project`.

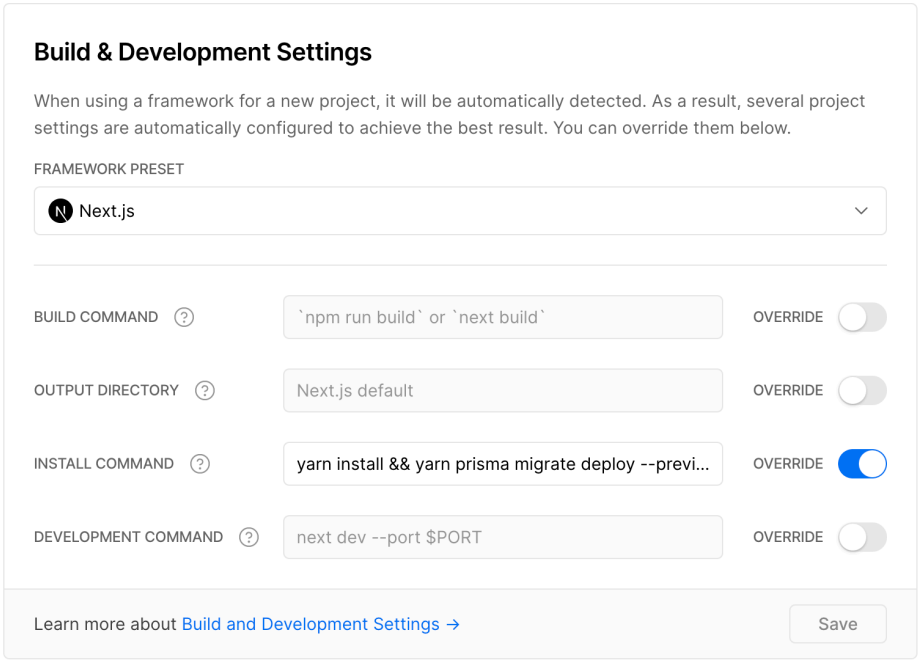


Import the repo you've created for this book.



You should get prompted for build settings. You can also change these settings at anytime in your Vercel dashboard.

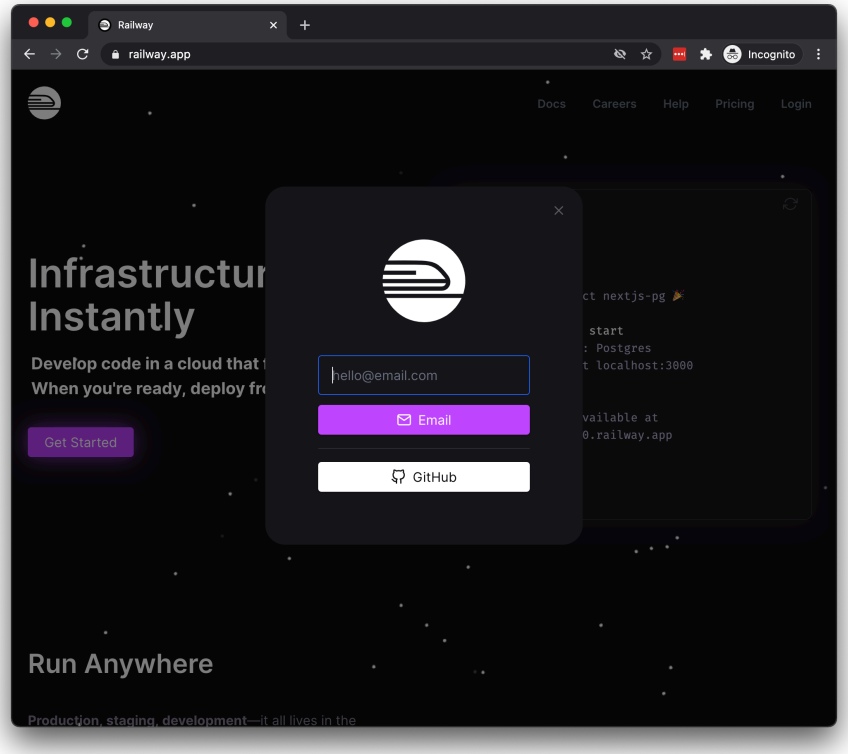
We'll use the defaults for everything except the install command. Toggle the "override" button and add this build command: `yarn install && yarn prisma migrate deploy`



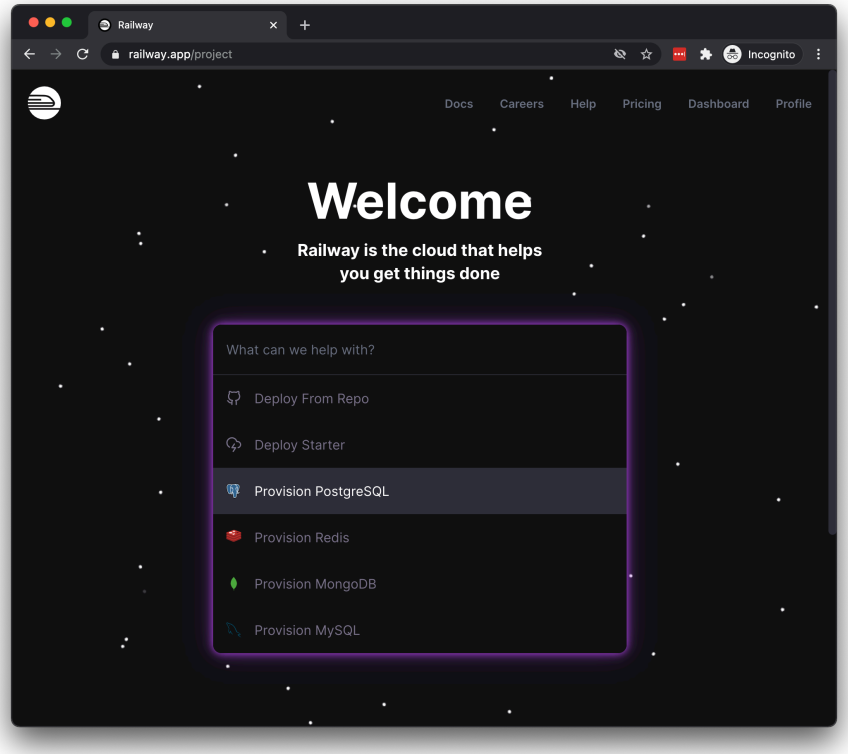
Give it a bit, and your project will be connected to Vercel.

Configuring Railway

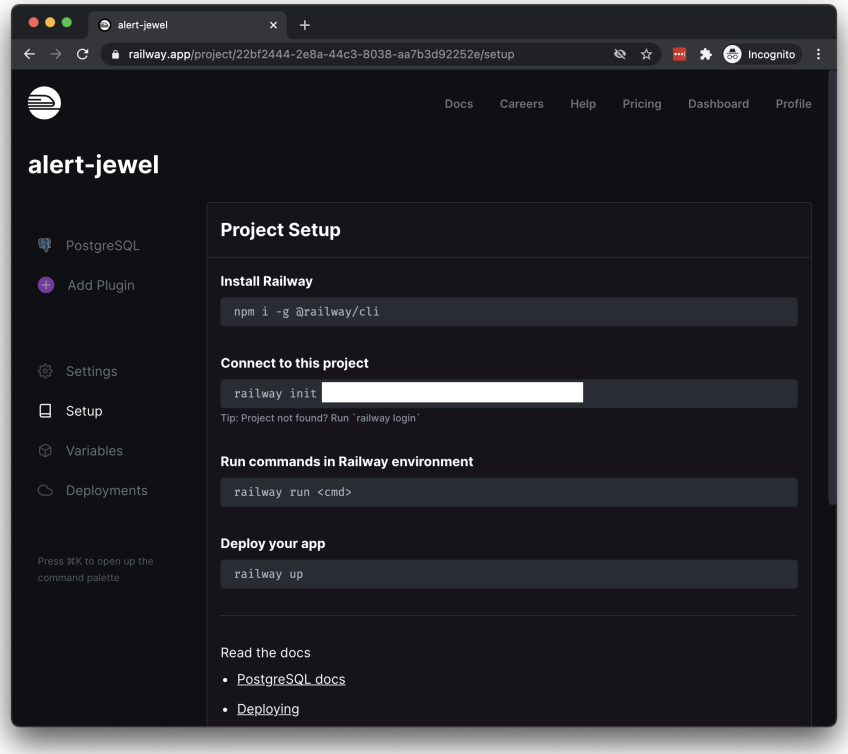
Configuring Railway is a cinch. Create an account by logging in with GitHub (you'll do this with Vercel as well).



Create a project - Provision PostgreSQL

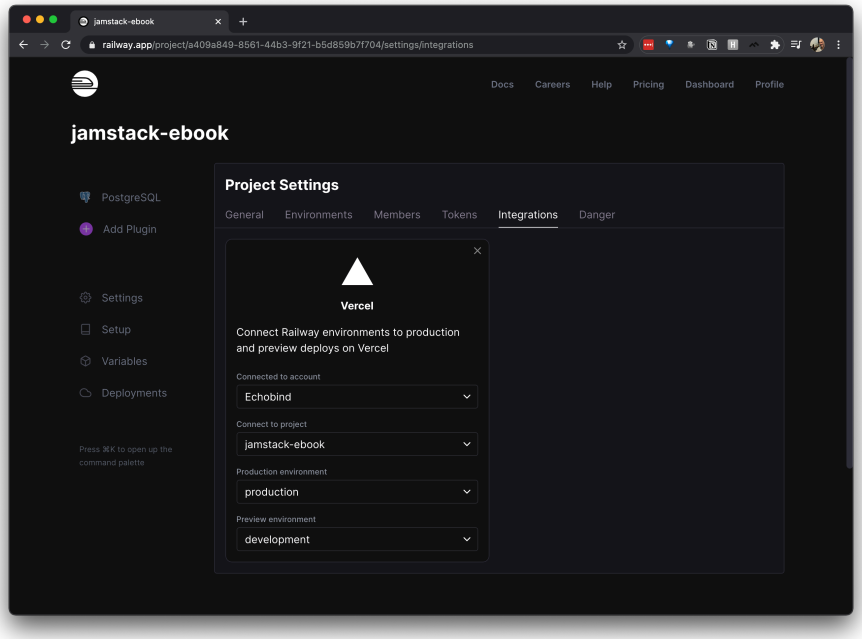


After a minute you'll get some instructions for next steps. The good news is you won't need these to deploy! But do them anyway. These tools will come in handy for doing any work on the database in the future.

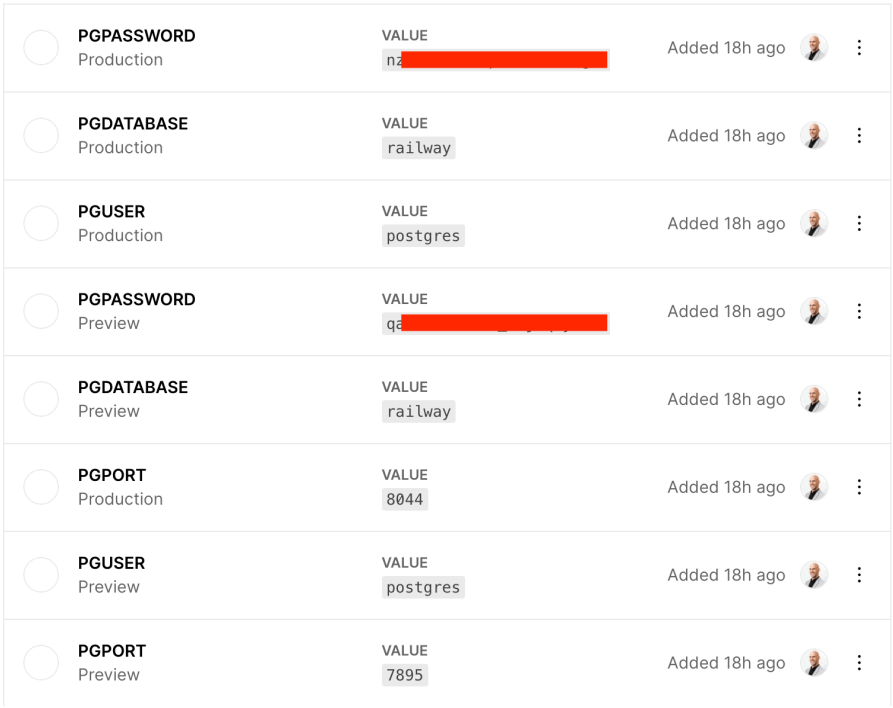


Now the fun part: we'll have Railway automagically configure our Vercel environment so that our production database is connected when we deploy.

Tap the `Integrations` tab and use the settings below:



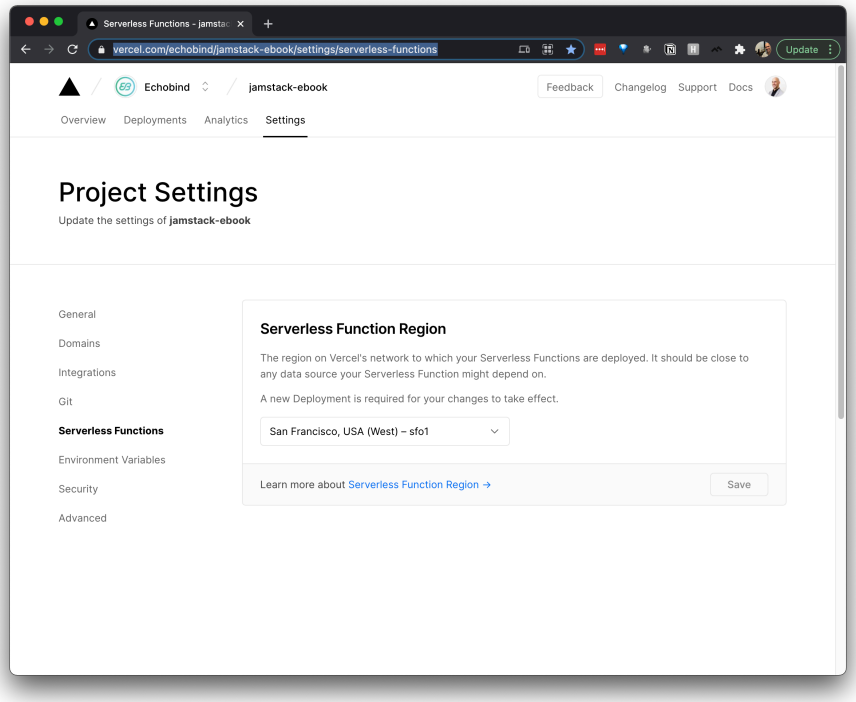
In moments you'll be connected. If you go back to your Vercel dashboard and look at `Settings` -> `Environment variables` again, you'll see that Railway has done the work of connecting database settings for you:



Updating deployment regions for speed

Railway servers are currently only in the western USA region, and by default Verbal deployments may not be. This means that database requests from the codebase (located in say, the eastern USA) may have to travel long distances to reach the database, slowing them down significantly. So it's easiest to have your deployments live in the same region.

Vercel lets us configure this easily. Under `Settings -> Serverless Functions`, set the Serverless Function Region to `San Francisco, USA (West)` to match Railway's available region.



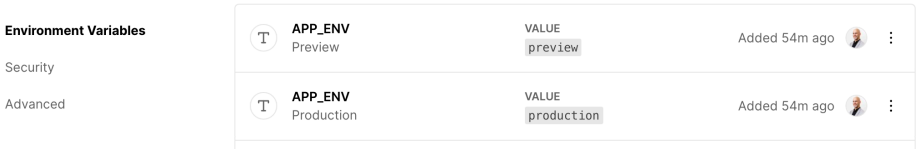
Railway will support multiple regions eventually, so you'll be able to set it to whichever region you want. Just remember to set your Vercel config to the same region for optimal speed.

Distinguishing preview and production deployments

Vercel gives us a fantastic feature called Preview deployments which let us deploy our feature branches and test them, without affecting our live app. It spins up a *new instance* of our app with every feature branch we deploy which is incredibly valuable.

To take full advantage of this, we'll want a way to distinguish preview and production environments so that we can do things like seed the database only in preview environments.

To achieve this we'll set an environment variable called `APP_ENV` in both environments, but set it to `production` in production in the production environment and `preview` in preview.



Adding a migration step to deploys

To make sure the right prisma command gets run when deploying, we'll edit our `package.json` and add the following `vercel-build` script right inside the `scripts` object:

```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start",
  "ts-node": "ts-node --compiler-options '{\"module\":\"CommonJS\"}'",
  "vercel-build": "prisma migrate deploy && next build"
},
```

Vercel exposes this custom build step (docs here) so that we can run a command only when deployed to Vercel.

In this case we want to run the `prisma migrate deploy` command (docs here). This is different from the `prisma migrate dev` command we run locally, in that it doesn't generate migrations by comparing the database to the schema; it just runs the ones we've already generated.

We have to also ensure we run `next build` afterwards, since using the `vercel-build` step overrides the default `build` step. If you don't supply that you'll likely see a "Routes Manifest Could Not Be Found" error.

Now to go a step beyond this, we'll add database seeding. But we want to only have this happen in our preview environment, not production—that would be bad. Good thing we set the `APP_ENV` environment variable above. Because we did that, we can now update our scripts to look like this:

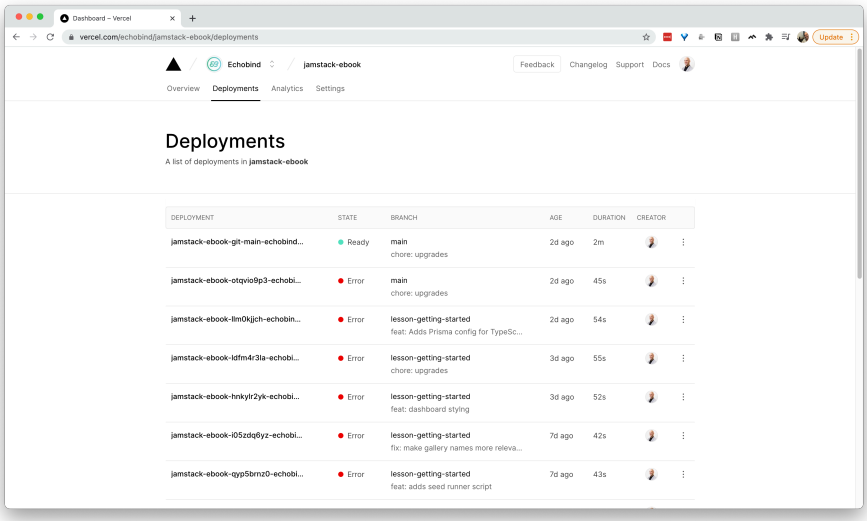
```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start",
  "ts-node": "ts-node --compiler-options '{\"module\":\"CommonJS\"}'",
  "db:reset": "prisma migrate reset --force --preview-feature",
  "db:migrate:deployment": "[[ \"$APP_ENV\" == \"preview\" ]] && yarn db:reset || yarn db
  "db:migrate:prod": "prisma migrate deploy",
  "vercel-build": "next build && yarn db:migrate:deployment"
},
```

Breaking this down: * We separate out a `db:migrate:prod` command to run the `prisma migrate deploy` command when called. * We also add a `db:reset` command to run `prisma migrate reset` which will, when called, erase the database, run migrations from scratch, and seed the database (docs) * We add a `db:migrate:deployment` task command which uses a little bit of shell scripting to check if our `APP_ENV` variable is set to `preview`; if we're in the preview environment, we'll run `db:reset` and start with a clean, seeded database; otherwise we're in production and we'll just run any outstanding migrations. * Lastly, we set the `vercel-build` command to run the `db:migrate:deployment` command, which does the rest.

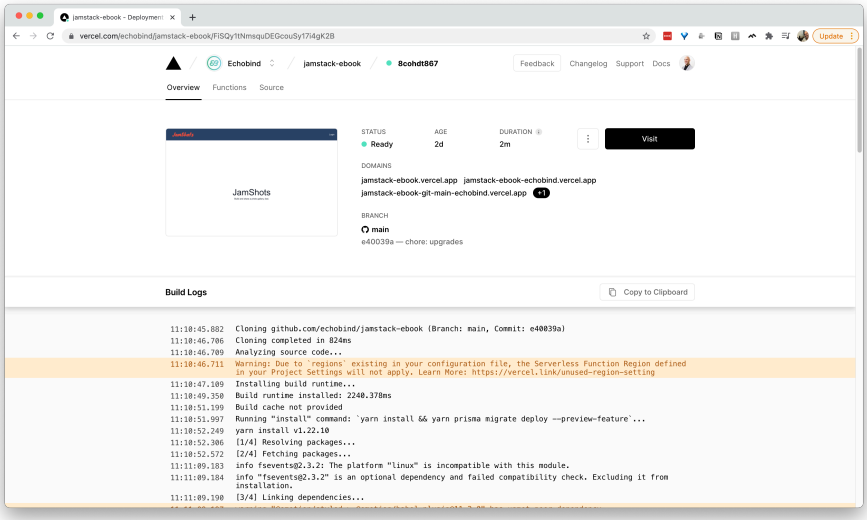
Push the red button: let's deploy!

This is the easiest part!

Whenever you push to `main`, Vercel will deploy to production. You can see a list of deployments on your Vercel dashboard under "Deployments":



When you click a single deployment, you can see the deployment's log output and troubleshoot any problems:



Deploying from the CLI is similarly easy/. Install the Vercel CLI:

```
yarn add --global vercel
```

Run `vercel init` to configure your project, then to deploy to production:

```
vercel --prod
```

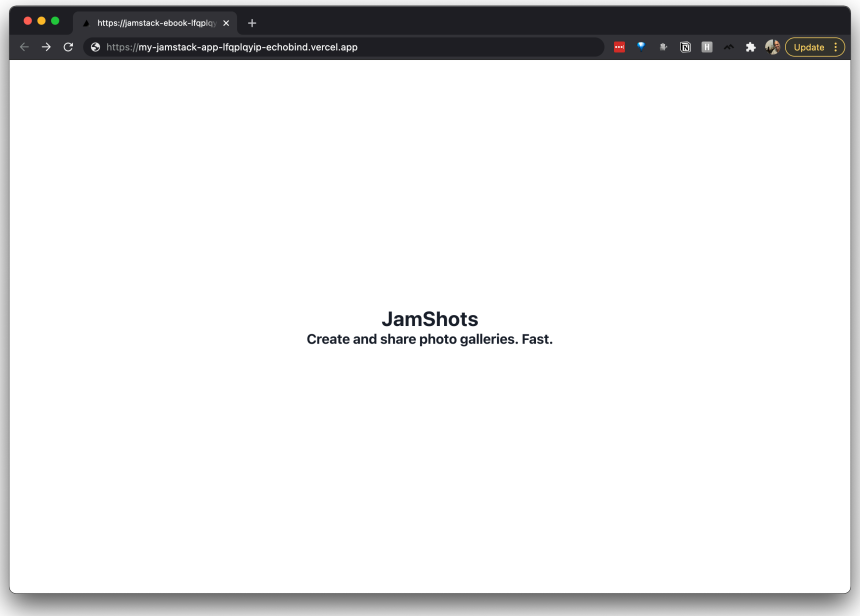
The CLI can be used for logging and a dozen other things. Here's the documentation.

Once the above deployment is done, your app should be live. Hit any of the domains listed in the deployment:

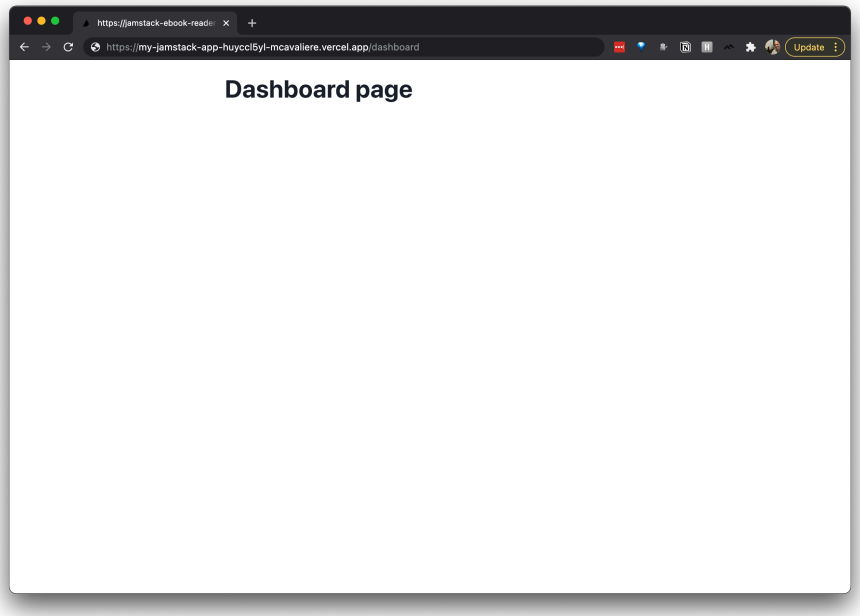
DOMAINS

my-jamstack-app.vercel.app **my-jamstack-app-echobind.vercel.app**
my-jamstack-app-git-main-echobind.vercel.app **+1**

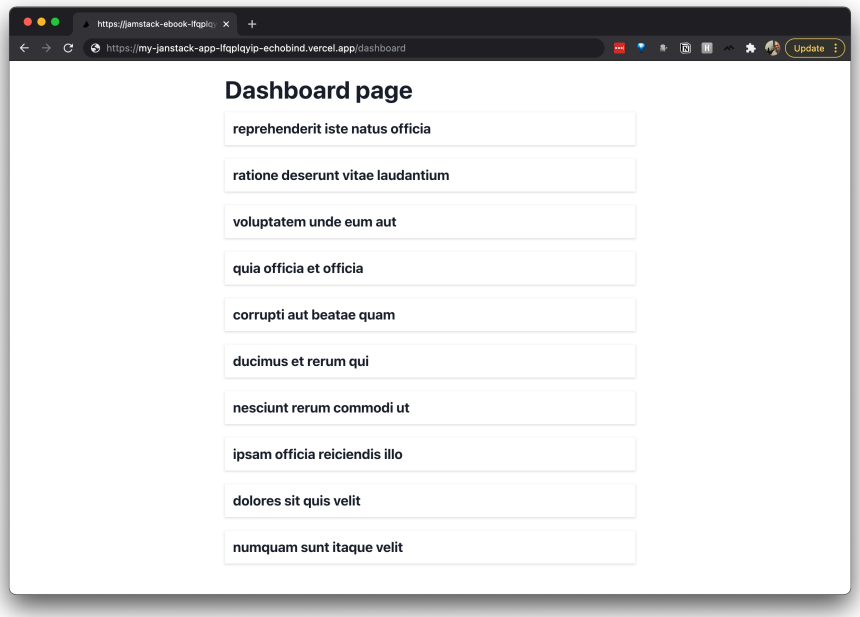
And you'll see our glorious, amazingly-designed homepage.



Then if you add `/dashboard` to the url you'll see the `loading...` text flicker briefly while it loads then an empty dashboard will show:



If you do a preview deploy (which you can do by creating a feature branch, then pushing an empty commit), *then* visit the dashboard at the preview url, you'll see some randomly-generated sample data:



Congrats!

You just built and shipped a (trivial, for now, but eventually amazing) app that you built from scratch with all of these tools. And you didn't spend a penny to get it done. Grab a quick beer to celebrate and let's keep moving.

Git branch strategy

From here on in, if you merge to `main` your code changes will deploy to production. This is fine while you're doing the exercises in this book or developing on your own for any project and aren't worried about breaking things.

If you're at all concerned with breaking things on production (or just want to build up a good habit), I encourage you to do your development in *feature branches*. All this means is that:

- You create a branch when you are doing anything even slightly risky (adding a feature, refactoring something, anything that might require multiple commits before it's done)

- You optionally create a pull request (which takes a few clicks on GitHub)
- When that branch is done and tested, you merge it to main.

This gives you the chance to develop freely and save your work as you go, then wait until you're ready to test everything before shipping it.

Full-Stack Gallery CRUD

Gallery CRUD

Since the Gallery is the central entity in our app, we're going to start by planning and building out API services and UI components for manipulating them - **c**reating, **r**eading, **u**pdating and **d**eleting them (commonly known as "CRUD"—an acronym you're probably familiar with already).

What we'll need for this version

To enable all these essential actions to work, we'll be building out a few key things on both the frontend and backend.

For starters, the API will need 4 main endpoints to do the work of creating, reading, updating and deleting galleries. These will be detailed in the next section.

Then we'll need UI elements for all 4 of these actions, and we'll need to show them on the dashboard screen as appropriate.

Lastly we'll need interactivity that makes the UI interact with the API. This will come in the form of a number of JS event handlers and React hooks that will respond to user interaction (indent to create, update, delete and so on), make the appropriate API request, then do something with that success dependent on whether it succeeded or not.

Future versions in this book

Later in the book we'll be taking these endpoints and enhancing them by adding a number of things:

1. authentication
2. access control
3. relationships between photos and galleries
4. Cloudinary API integration for photo upload, retrieval and deletion

The UI similarly will expand to contain a lot more than gallery CRUD. For example:

1. Photo uploading
2. Photo grids within the gallery
3. A modal for sharing a gallery URL
4. The gallery view that a user sees when the above url is shared with them
5. A delete confirmation dialog
6. ...etc

Planning Gallery API endpoints

Now that we have a data model, we'll take a first step toward building interactivity with that data. We'll build out some core pieces of functionality for interacting with the database known as CRUD (Create, Read, Update, Delete).

From the server side, we'll need the following API routes:

GET /api/galleries

- **File:** `/pages/api/galleries/index.ts`
- **Responsibility**: Return a list of galleries the logged-in user has access to. If you're working through this book from the beginning, you'll have this one already built.

GET /api/galleries/id

- **File:** `/pages/api/galleries/[id]/show.ts`
- **Responsibility**: Returns a single gallery by ID. This will be handy later as we add photos to a gallery; this endpoint will return the gallery and its photos.

POST /api/galleries/create

- **File:** `/pages/api/galleries/create.ts`
- **Responsibility**: Create a new gallery.

PATCH /api/galleries/:id/update

- **File:** `/pages/api/galleries/[id]/update.ts`
- **Responsibility**: Update the properties of a gallery the current user has access to.

DELETE /api/galleries/:id/delete

- **File:** `/pages/api/galleries/[id]/delete.ts`

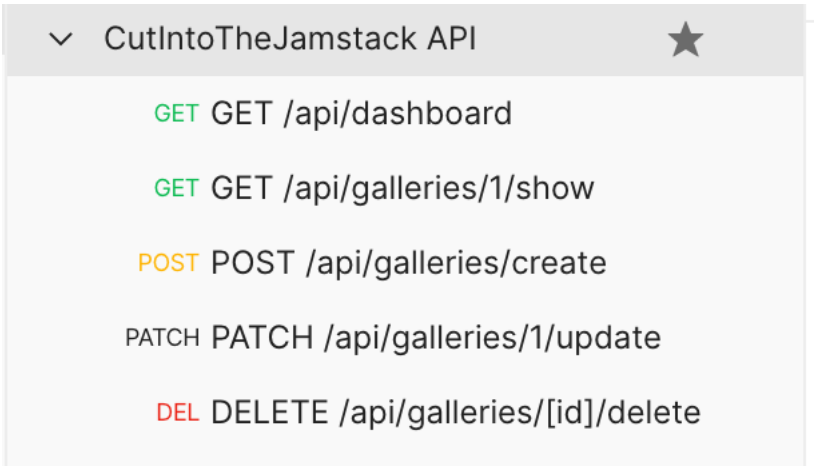
- **Responsibility**: Delete a gallery the logged-in user is the owner of.

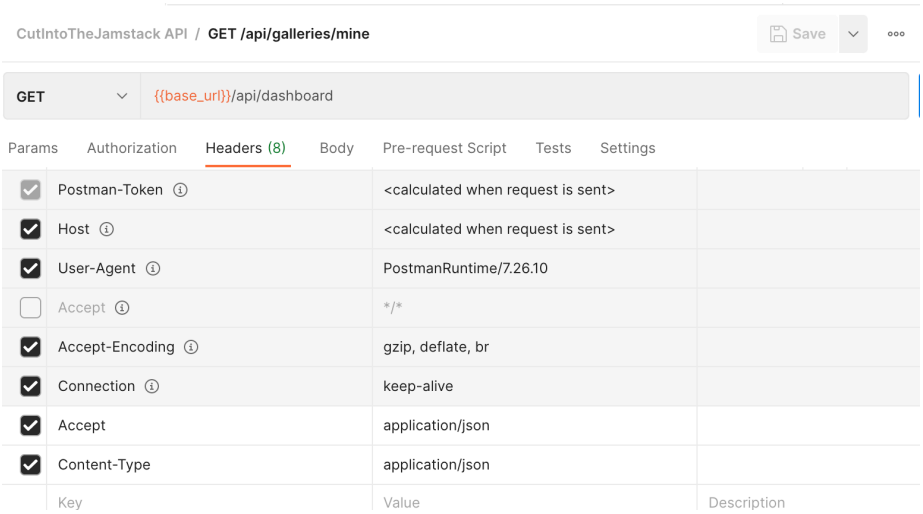
Each of these serverless functions will do a few things, initially: - Make sure the request type is correct for the given api route; - Execute the desired database operation; - Return some data related to the result; or - Return an error if any step above fails.

In the future we'll build on top of these routes to do things like photo manipulation, and restricting access to only eligible users. We'll also refactor some of the repetitive logic to simplify things. For now, these will lay some good groundwork for patterns we'll use elsewhere in the application.

Add failing requests to Postman

We'll stub out the requests we want in Postman, and test them as we create them. If you have another preferred way of testing them, go for it.





Note that we've set the `base_url` variable in Postman to be `http://localhost:3000` to reduce redundant typing. We do this also so that we can change the url in one place and affect all of the routes if we want to. This can come in handy for testing against a remote server.

All of these endpoints have the request headers `Accept` and `Content-Type` to `application/json` since we'll be sending and consuming json in all of our services.

All of them should currently fail since none of these routes exist yet. Let's create the first one.

Structure of Our CRUD Endpoints

The code will follow a similar pattern for each of the four endpoints we'll be creating. Here's the shell, with TODO comments for the parts that will vary between endpoints.

```
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export default (req, res) => {
  // TODO: [Optional] Check the type of request.

  // TODO: Get some data from the route and/or the request body.

  async function main() {
    // TODO: Manipulate database with Prisma.
```

```
    // TODO: Return a success response and data.
  }

  // This will stay the same in all endpoints, and later be extracted to middleware.
  return main()
    .catch((e) => {
      console.log(`---------------- error with request: `, e);
      return res.status(500).end();
    })
    .finally(async () => {
      await prisma.$disconnect();
    });
};
```

Gallery Creation API Endpoint

Here's what `/pages/api/galleries/create.ts` will look like:

```
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export default (req, res) => {
  // Fail if this isn't a POST request.
  if (req.method !== 'POST') {
    return res.status(400).end();
  }

  const { name, description } = req.body;

  async function main() {
    const gallery = await prisma.gallery.create({
      data: {
        name,
        description
      }
    });

    return res.status(200).json(gallery);
  }

  return main()
    .catch((e) => {
      console.log(`---------------- error creating gallery: `, e);
      return res.status(500).end();
```

```
    })
    .finally(async () => {
      await prisma.$disconnect();
    });
};
```

Let's break this down section by section.

As before, the shell of a Next.js API route is just a serverless function that takes in request and response parameters:

```
export default (req, res) => {
}
```

The serverless function must return success or error handler for all cases, or a promise that resolves to one.

The first thing we do is check to see if this is the right type of request:

```
  // Fail if this isn't a POST request.
  if (req.method !== 'POST') {
    return res.status(400).end();
  }
```

If this isn't a POST request, we have the response object send a status of 400 ("Bad Request"). The `.end()` ensures the request closes when this happens, otherwise this can result in requests that stall and don't complete for a long time.

```
const { name, description } = req.body;
```

Here we retrieve the name and description params from the body of the POST request. Later these will come from form fields on the frontend.

```
async function main() {
// ...
}
```

The async `main()` function will be the wrapper around our business logic. This is a pattern that Prisma defines in their docs.

Inside `main()` we have all of the real thinking for this API endpoint:

```
    const gallery = await prisma.gallery.create({
      data: {
        name,
        description
      }
    });

    return res.status(200).json(gallery);
```

We do the database INSERT by calling the Prisma `create()` function on the `gallery` model., using `await` since it's an asynchronous function. It will return the created gallery object so we save that to the `gallery` variable.

If this succeeds, we have the response object return an HTTP 200 ("Success") code, and the json for the created gallery object. But what if it fails? What if there's an error? This is why the last part the file is important:

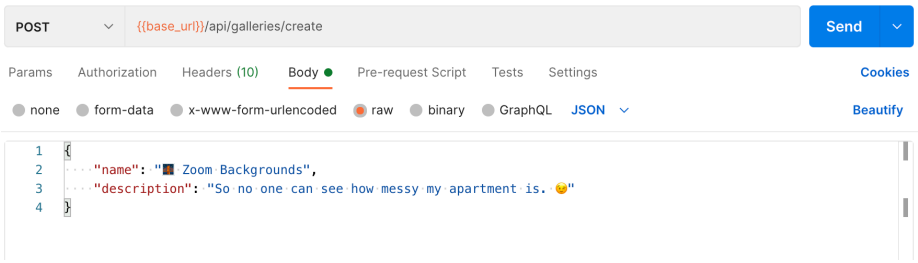
```
return main()
  .catch((e) => {
    // Later this will log to Sentry, or other error handling service.
    console.log(`---------------- error creating gallery: `, e);
    return res.status(500).end();
  })
  .finally(async () => {
    await prisma.$disconnect();
  });
```

Here we return the result of the `main()` function which is a Promise. Upon success, it'll do whatever `main()` is written to do, which in this case is return a status of 200 and the created gallery object. If it fails, it'll log the error to the server console and return a 500 error code. In any case it'll close the Prisma connection.

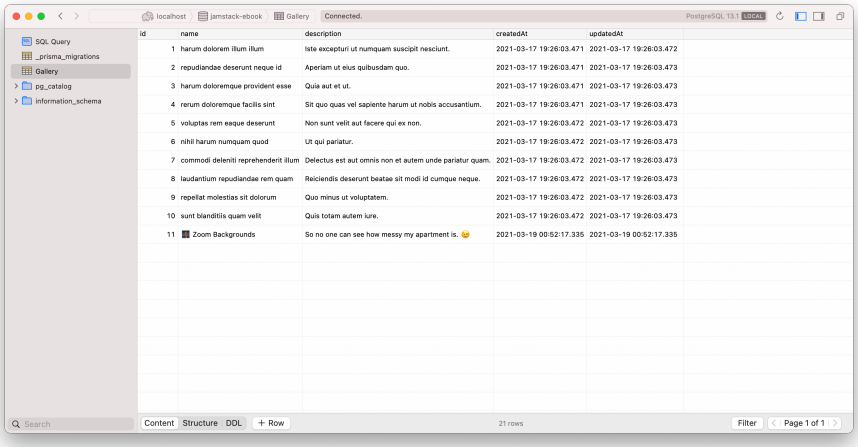
A few notes: * We don't resurface the details of server errors when caught, since they're more beneficial to us (the developers) than the users. When this launches to production, we'll use Sentry or another error handling service to report these errors to us silently. * The `finally` block will change later on, since we want to handle this a bit differently in our production environment. * We're going to be reusing this structure in almost every API call, so later on we'll be extracting it to a middleware.

Testing our API endpoint

Go back to our request in postman, and put a gallery object (with a name and description) into the body as raw JSON. Submit it and you should see the created object returned.



If you inspect the database with a GUI, you'll see the newly created gallery.



Gallery Retrieval API Endpoint

NOTE: The other endpoints from here on in will have the same structure, with different business logic in the middle. We'll DRY up this redundant code in the refactoring section at the end of this chapter.

For the `/api/galleries/123/show` endpoint, we're going to use a Next.js dynamic route (docs). The 123 in the url above is the ID of an existing gallery; when requests are made that match the above pattern, we'll extract the gallery ID from the URL and query the database with it.

Dynamic routes in Next.js are straightforward: since the folder path is what determines the route path, we use square brackets (`[]`) to denote the dynamic segments. In this case, we'll create the folder `/pages/api/galleries/[id]` and add a file to it called `show.ts`.

Now when you request the URL `/api/galleries/123/show`, it will match the 123 in the request to the dynamic route parameter `[id]`, and pass that ID into the API endpoint as a query variable.

Let's see this in action.

```
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export default (req, res) => {
  const id: number = parseInt(req.query.id);

  async function main() {
    // Get our data.
    const gallery = await prisma.gallery.findUnique({
      where: { id }
    });

    // Return the data.
    return res.status(200).json(gallery);
  }

  return main()
    .catch((e) => {
      // Later this will log to Sentry, or other error handling service.
      console.log(`---------------- error retrieving gallery: `, e);
      return res.status(500).end();
    })
    .finally(async () => {
      await prisma.$disconnect();
    });
};
```

When there are dynamic route parameters, they'll come through to the API endpoint inside the `req.query` object. In this case, since we are expecting a the `id` variable to be a number, we make sure the JavaScript variable reflects that by using `parseInt` (since the Next.js route is not type-aware). As an added measure we sprinkle some TypeScript here by defining the `id` variable as type `number`.

Then we do the database lookup by calling Prisma's `findUnique()` function (guide and API docs) which is the preferred way of looking up a single record.

Gallery Update API Endpoint

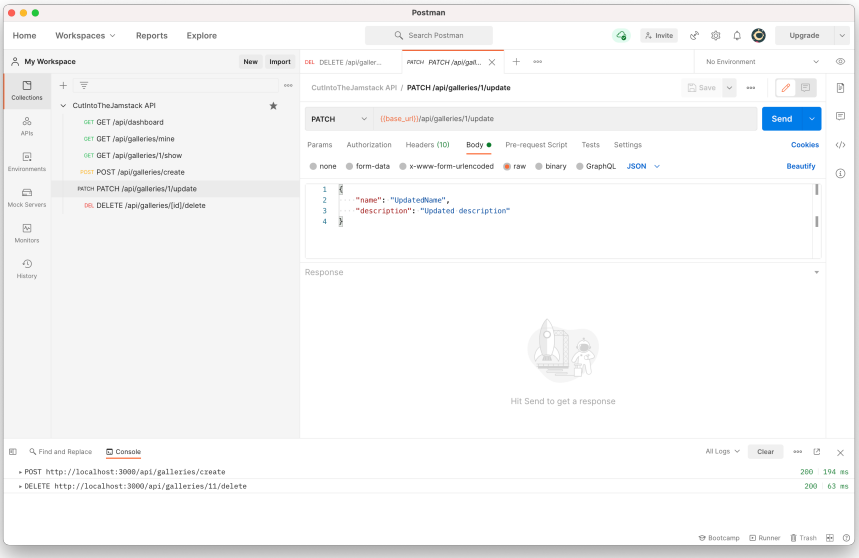
Updates will also use a dynamic route, combined with a similar check we have in the creation endpoint to make sure we're making a `PATCH` request.

Using a dynamic route again, here's what `/pages/api/galleries/[id]/update.ts` will look like. Note that we're pulling in the ID from the req.query object as in the `show` endpoint, and also the new `name` and `description` from the `body` like in our `create` endpoint.

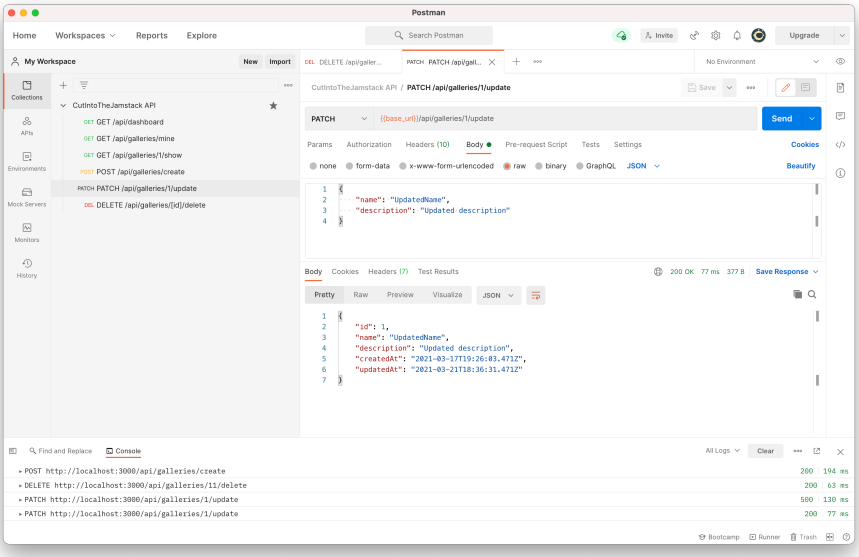
```
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export default async (req, res) => {
  if (req.method !== 'PATCH') {
    console.log(`---------------- not PATCH, returning 400 `);
    return res.status(400).end();
  }

  const id: number = parseInt(req.query.id);
  const { name, description } = req.body;

  // Get our data.
  const gallery = await prisma.gallery.update({
    where: { id },
    data: {
      name,
      description
    }
  });

  // Return the data.
  return res.status(200).json(gallery);
};
```

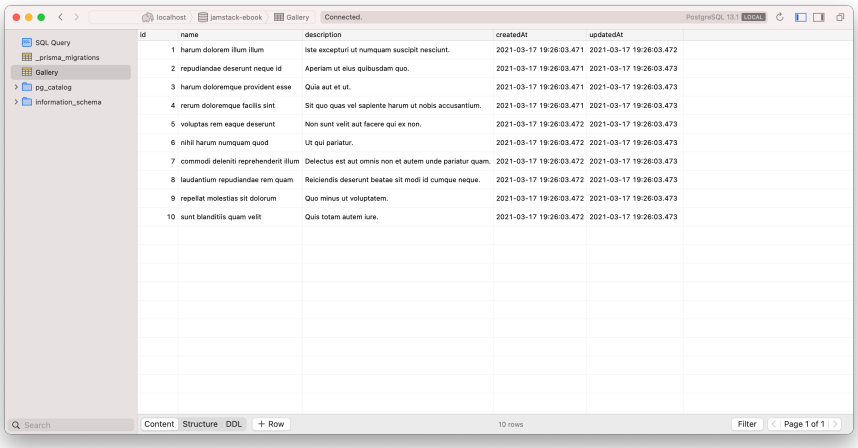
Testing

Pick a record you want to update and plug the ID into the PATCH endpoint, with a JSON body for the updated `name` and `description`:



After submitting you'll see the returned updated object and can verify in your DB interface:





Gallery Deletion API Endpoint

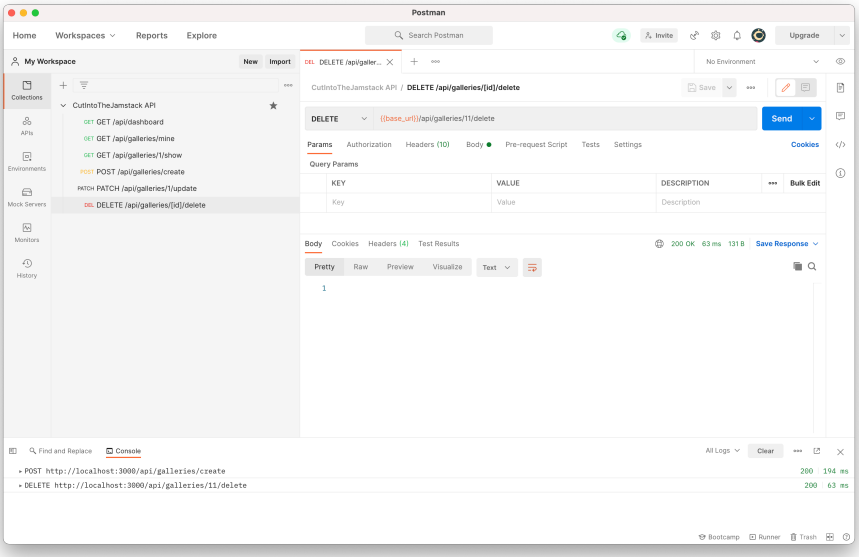
Deletion will also use a dynamic route, combined with a similar check we have in the creation endpoint to make sure we're making a `DELETE` request.

Using a dynamic route again, here's what `/pages/api/galleries/[id]/delete.ts` will look like:

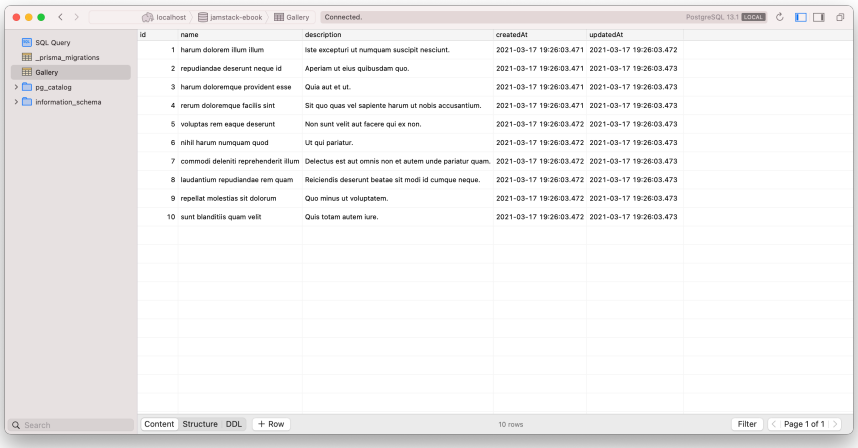
```
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export default (req, res) => {
  if (req.method !== 'DELETE') {
    console.log(`---------------- not DELETE, returning 400 `);
    return res.status(400).end();
  }

  const id: number = parseInt(req.query.id);

  async function main() {
    // Delete it.
    const json = await prisma.gallery.delete({
      where: {
        id,
      },
    });

    // Return nothing.
```

```
    return res.status(200).json(json).end();
  }

  return main()
    .catch((e) => {
      // Later this will log to Sentry, or other error handling service.
      console.log(`----------------- error deleting gallery: `, e);
      return res.status(500).end();
    })
    .finally(async () => {
      await prisma.$disconnect();
    });
};
```

Testing with Postman, we just put ID 11 in the path (or any other ID of a record you want to delete):



Verifying it's gone in the DB GUI:



Planning Gallery CRUD UI

Now that we have the API portion that does the gallery creation work, we'll add some UI components to interact with those endpoints.

The dashboard screen will be where all of our high-level gallery CRUD interactions will live. So we'll need a few things here:

- A UI component for listing the galleries I have access to;
- A modal form for creating a gallery with basic information;
- A modal form for editing basic gallery information; and
- A way to delete an existing gallery.

We'll introduce four components for the above bits of UI:

- `<GalleryCreateModal />`
- `<GalleryEditModal />`
- `<GalleryDeleteDialog />` (Confirm/cancel dialog for deleting a gallery)
- `<GalleryListItem />` (For a single gallery in a list; will contain delete and edit buttons)

We'll also need a new page for showing an individual gallery, which will later become the gallery editor page:

- `/pages/galleries/[id]/show.tsx`

The GalleryListItem Component

Our `<GalleryListItem />` component will represent a single gallery on the dashboard.

We currently have code which shows the name of each gallery on the dashboard, which we'll move into a component. We'll then make it so that clicking the link will navigate to another page. We'll also give it edit and delete buttons, and pass event handlers for them into the component.

We're going to make this a **"dumb component"** or **"presentational component,"** meaning that it's just for displaying something and doesn't contain any functionality. It will have event handlers for buttons, but those will be passed in as props.

A future version of this component will also: - link to the gallery's editor / photo uploader - show a photo count - show a photo preview

Moving to a component

We're already written the code for the basic gallery item, in `/pages/dashboard.tsx`:

```
<Flex width="100%" direction="row" key={g.id} radius={10} boxShadow='base' p={3}>
  <Heading size="md">
    {g.name}
  </Heading>
</Flex>
```

Let's transplant that to a new component, `/components/GalleryListItem/index.tsx`.

```
import { Flex, Heading } from '@chakra-ui/react';

export const GalleryListItem = ({ name, key }) => (
  <Flex width="100%" direction="row" key={key} radius={10} boxShadow='base' p={3}>
    <Heading size="md">
      {name}
    </Heading>
  </Flex >
);
```

Then we'll reference it in `dashboard.tsx` as:

```
// ...other imports...
import { GalleryListItem } from 'components/GalleryListItem';

// ...then in our return statement...

 {galleries?.map(({ name, id }) => (
   <GalleryListItem name={name} key={id} />
 ))}
```

Our dashboard page should look exactly the same after this refactor.

Adding buttons

For our edit and delete buttons, we'll grab the `react-icons` library (docs) since it has icons we can use for just about anything. We'll be using in plenty of other components we develop as well.

```
yarn add react-icons
```

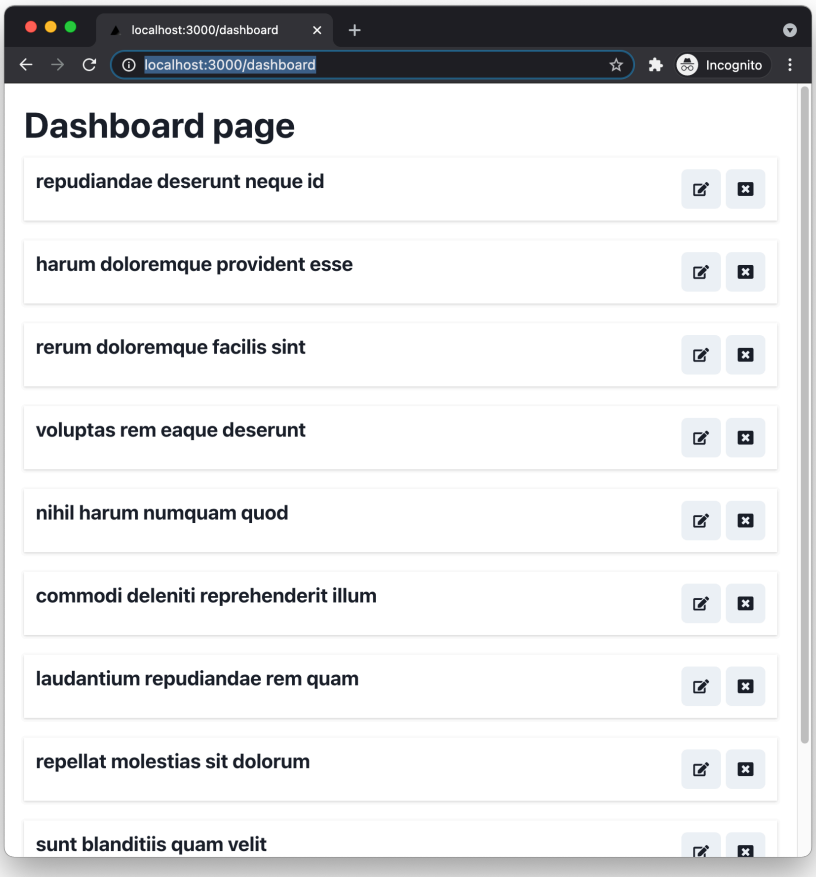
Chakra-UI has an `IconButton` component (docs) which works very nicely with `react-icons`, so we'll add an edit and delete button:

```
import { Flex, Heading, HStack, IconButton } from '@chakra-ui/react';
import { FaEdit, FaWindowClose } from 'react-icons/fa';

export const GalleryListItem = ({ name, key }) => (
  <Flex
    width="100%"
    justifyContent="space-between"
    direction="row"
    key={key}
    radius={10}
    boxShadow='base'
    p={3}
  >
    <Heading size="md">
      {name}
    </Heading>

    <HStack spacing="5px">
      <IconButton
        aria-label="Edit this gallery&#39;s name and description"
        icon={<FaEdit />}
        onClick={() => {}}
      />
      <IconButton
        aria-label="Delete this gallery"
        icon={<FaWindowClose />}
        onClick={() => {}}
      />
    </HStack>
  </Flex>
);
```

Key takeaways of this code: - We add two `<IconButton />` components using the Edit and WindowClose icons from FontAwesome that we imported from `react-icons`. For now we give them empty `onClick` handlers. - We wrap the buttons in an `HStack` component (docs) which just

keeps 5 pixels of horizontal space between its children. We'll add other icons here later, so they'll be automatically spaced out when that time comes. - We give the `<Flex>` a `justifyContent="space-between"` prop so that the text and the button container stay spaced apart.

The result looks like this:



Adding props

We want the link URL and event handlers to be props for two reasons: 1. To keep this component presentational-only, which makes it easy to maintain; and 2. So that we can change functionality if we decide to, without affecting the presentation.

So we'll expand the full list of props to

- name: the text name of the gallery
- href: the link that the gallery name will redirect to
- onDeleteClick: the delete button event handler
- onEditClick: the edit button link handler

Defining these with TypeScript will be extra handy too, since it will make sure that we see an error if we don't specify a required prop, or if we pass in the wrong format. So we'll declare a TypeScript interface for the props:

```
export interface Props {
  name: string;
  href: string;
  onDeleteClick: (e: any) => any;
  onEditClick: (e: any) => any;
}

// Component definition changes to this:

export const GalleryListItem = ({ name, key, href, onDeleteClick, onEditClick }: Props) => (

// ...
```

A TypeScript `interface` (docs) defines the type of each element in an object. In this case, our interface is defining the type of each prop in the `GalleryListItem` component.

Notes: - if we wanted to make any attribute of the interface optional, we'd add a question mark at the end of the attribute's name (e.g., `name?: string;`). In this case we want them all to be required. - We need to tell the component to use the interface by specifying the type of the props object with `: Props` at the end of the list of props.

Using the next/link component

To connect the `href` prop and make the gallery name link to another page, we use the Next.js `Link` component (docs).

```
//...other imports..
import Link from 'next/link';
// ...

export const GalleryListItem = ({ name, href, onDeleteClick, onEditClick }: Props) => (

    // ...

    <Heading size="md">
      <Link href={href}>
```

```
        {name}
      </Link>
    </Heading>

    // ...

);
```

Connecting event handlers

Now we just replace the empty arrow functions with the event handler props:

```
//...
      <IconButton
        aria-label="Edit this gallery&#39;s name and description"
        icon={<FaEdit />}
        onClick={onEditClick}
      />
      <IconButton
        aria-label="Delete this gallery"
        icon={<FaWindowClose />}
        onClick={onDeleteClick}
      />
//...
```

Testing the component

Now we just need to instantiate the component in our dashboard. Inside `dashboard.tsx`, before the `return` statement we'll declare some event handlers:

```
  const handleGalleryEdit = (e, id) => {
    e.preventDefault();
    console.log(`Editing gallery: ${id}`);
  };

  const handleGalleryDelete = (e, id) => {
    e.preventDefault();
    console.log(`Delete gallery: ${id}`);
  };
```

Then connect them to the related props in the `GalleryListItem` component:

```
        {galleries?.map(({ name, id }) => (
          <GalleryListItem
```

```
            href={`/galleries/${id}`}
            name={name}
            key={id}
            onDeleteClick={(e) => handleGalleryDelete(e, id)}
            onEditClick={(e) => handleGalleryEdit(e, id)}
          />
        ))}
```

This leaves the dashboard code looking like this:

```
import useSWR from 'swr';
import { Heading, Box, VStack } from '@chakra-ui/react';
import { GalleryListItem } from 'components/GalleryListItem';
export default function DashboardPage() {
  const { data: galleries, isValidating: dashboardIsLoading, error: dashboardFetchError } =

  if (dashboardIsLoading) {
    return (
      <h1>Loading dashboard...</h1>
    );
  }

  if (dashboardFetchError) {
    return (
      <h1>Error loading the dashboard.</h1>
    );
  }

  const handleGalleryEdit = (e, id) => {
    e.preventDefault();
    console.log(`Editing gallery: ${id}`);
  };

  const handleGalleryDelete = (e, id) => {
    e.preventDefault();
    console.log(`Delete gallery: ${id}`);
  };

  return (
    <Box m="0 auto" p={5} maxWidth={{ sm: '100%', md: '100%', lg: '40em', xl: '50em', '2xl'
      <Heading size="xl" mb={3}>Dashboard page</Heading>
      <VStack spacing={5}>
        {galleries?.map(({ name, id }) => (
          <GalleryListItem
            href={`/galleries/${id}`}
            name={name}
            key={id}
```

```
            onDeleteClick={(e) => handleGalleryDelete(e, id)}
            onEditClick={(e) => handleGalleryEdit(e, id)}
          />
        ))}
      </VStack>
    </Box >
  );
}
```

Now if you click the edit and delete buttons in your browser, you should see console output like this:

```
Editing gallery: 2
Delete gallery: 2
```

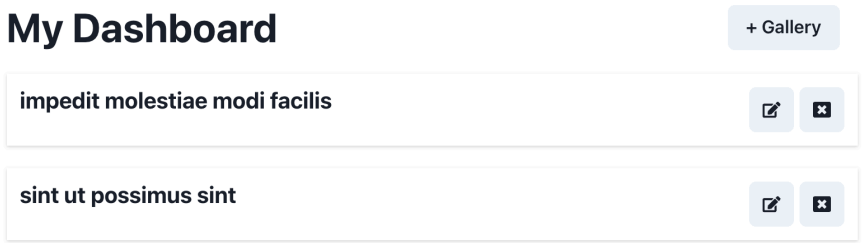
The GalleryCreateModal Component

On the dashboard we'll have a button for adding a new gallery, which will open a modal with a form for you to add a name and description. The `GalleryCreateModal` will contain the modal component. Like our `GalleryListItem` component, we'll keep the `GalleryCreateModal` relatively "dumb" and the logic and state will exist outside of it.

Adding the "add gallery" button

Inside `/pages/dashboard.tsx` let's replace the `Heading` with a row that has heading text and a button we can launch the "add gallery" modal with:

```
import { Flex, Heading, Box, Button, VStack } from '@chakra-ui/react';

      //...

      <Flex direction="row" alignItems="center" justifyContent="space-between" mb={5}>
        <Heading>My Galleries</Heading>
        <Button as="a" href="/galleries/new" mr={4}>
          + Gallery
        </Button>
      </Flex>

      //...
```



Result:

Creating the GalleryCreateModal component

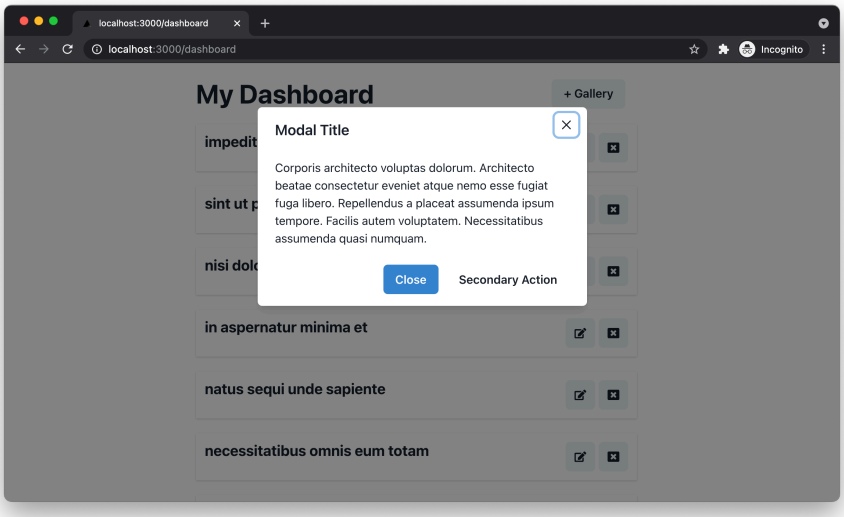
Inside a new file, `/components/GallerCreateModal.tsx` we'll add a Chakra-UI Modal (docs). Inside we'll post the contents of the usage example in the Chakra docs, but with some alterations to make it into a presentational component.

```
import {
  Button,
  Modal,
  ModalOverlay,
  ModalContent,
  ModalHeader,
  ModalFooter,
  ModalBody,
  ModalCloseButton,
} from "@chakra-ui/react";

export const GalleryCreateModal = ({ isOpen = false, onOpen = () => {}, onClose = () => {} }
  return (
    <Modal isOpen={isOpen} onOpen={onOpen} onClose={onClose}>
      <ModalOverlay />
      <ModalContent>
        <ModalHeader>Modal Title</ModalHeader>
        <ModalCloseButton />
        <ModalBody>
          Corporis architecto voluptas dolorum. Architecto beatae consectetur eveniet atque
        </ModalBody>

        <ModalFooter>
          <Button colorScheme="blue" mr={3} onClick={onClose}>
            Close
          </Button>
          <Button variant="ghost">Secondary Action</Button>
        </ModalFooter>
      </ModalContent>
```

```
    </Modal>
  );
};
```

Takeaways: * We've removed the `useDisclosure` since we'll add that outside of this presentational component. We want the component to be simple and flexible. * We're using placeholder text for now. This will be replaced with a form using react-hook-form. * We're specifying minimal props right now, and giving them default values.

Let's add our component to `/pages/dashboard.tsx` by adding `<GalleryCreateModal isOpen />` right before the closing `</Box>` tag. The result will look like this:



Adding modal toggling with `useDisclosure`

The Chakra-UI `useDisclosure` hook is helpful for extracting the state handling for toggling anything. In this case, we want to toggle modal show and hide.

We'll add it into the dashboard up top:

```
import { Flex, Heading, Box, Button, useDisclosure, VStack } from '@chakra-ui/react';
// ...

export default function DashboardPage() {
  const { data: galleries, isValidating: dashboardIsLoading, error: dashboardFetchError } =
  const { isOpen: isGalleryCreateOpen, onClose: onGalleryCreateClose, onOpen: onGalleryCreat
```

```
// ...

      <GalleryCreateModal
        isOpen={isGalleryCreateOpen}
        onOpen={onGalleryCreateOpen}
        onClose={onGalleryCreateClose}
      />

// ...
```

Notes: * We're aliasing `isOpen`, `onClose` and `onOpen` since we'll be using `useDisclosure` for multiple modals.

Now our 'add gallery' button will open the modal, and the modal's close buttons will close it.

Adding the gallery creation form

First make sure`react-hook-form` is installed:

```
yarn add react-hook-form
```

Here's what the component will look like with the form added:

```
import {
  Button,
  FormControl,
  FormErrorMessage,
  FormLabel,
  Input,
  Modal,
  ModalOverlay,
  ModalContent,
  ModalHeader,
  ModalFooter,
  ModalBody,
  ModalCloseButton,
  Textarea,
} from '@chakra-ui/react';
import { useForm } from 'react-hook-form';
import { Gallery } from '@prisma/client';

export interface Props {
  isOpen: boolean;
  onClose: () => any;
  onSubmit: (gallery: Gallery) => void;
}
```

```
export const GalleryCreateModal = ({
  isOpen = false,
  onClose = () => {},
  onSubmit = () => {},
}: Props) => {
  const { register, handleSubmit, errors } = useForm();

  return (
    <Modal isOpen={isOpen} onClose={onClose}>
      <form onSubmit={handleSubmit(onSubmit)}>
        <ModalOverlay />
        <ModalContent>
          <ModalHeader>Create Gallery</ModalHeader>
          <ModalCloseButton />
          <ModalBody>
            <FormControl id="name" isInvalid={!!errors.name}>
              <FormLabel>Name</FormLabel>
              <Input
                type="text"
                name="name"
                ref={register({ required: true })}
              />
              <FormErrorMessage mb={2}>This field is required</FormErrorMessage>
            </FormControl>

            <FormControl id="description">
              <FormLabel>Description</FormLabel>
              <Textarea name="description" ref={register} />
            </FormControl>
          </ModalBody>

          <ModalFooter>
            <Button colorScheme="blue" mr={3} onClick={onClose}>
              Cancel
            </Button>
            <Button variant="ghost" type="submit">
              Save
            </Button>
          </ModalFooter>
        </ModalContent>
      </form>
    </Modal>
  );
};
```

Let's break down the important parts:

```
import { useForm } from 'react-hook-form';
```

We import the `useForm` hook from `react-hook-form` (docs) (tutorial).

```
export interface Props {
  isOpen: boolean;
  onClose: () => any;
  onSubmit: (gallery: Gallery) => void;
}
```

We use a little TypeScript to declare the props to this component. `isOpen` and `onClose` get passed right to the Chakra-UI `Modal`.

`onSubmit` will get passed to `react-hook-form`'s `handleSubmit` method. This way, whatever `onSubmit` function we pass in will be called when the form is submitted.

```
  const { register, handleSubmit, errors } = useForm();
```

We call the `useForm` hook at the top of the component. This hook returns some important variables: - the `register` function ties each form input to the form. It's also used for telling us how the input field will be validated. - the `handleSubmit` is a callback that `react-hook-form` requires us to call when the form is submitted. We give it a function as an argument (defined by the `onSubmit` prop, in this case) which will be called after the form is submitted. - `errors` will be an object containing a key for every form field that contains errors.

```
<form onSubmit={handleSubmit(onSubmit)}>
```

We make sure `react-hook-form`'s `handleSubmit` function gets called when we submit the form. Passing our `onSubmit` prop makes sure that `onSubmit` gets called thereafter.

```
<FormControl id="name" isInvalid={!!errors.name}>
  <FormLabel>Name</FormLabel>
  <Input type="text" name="name" ref={register({ required: true })} />
  <FormErrorMessage mb={2}>This field is required</FormErrorMessage>
</FormControl>
```

This block is a Chakra-UI `FormControl`, (docs) which groups a label with an input field, and other optional components. Broken down: - The `isInvalid` prop (docs) determines whether the `FormErrorMessage` down below will be shown. The form is invalid if `errors.name` exists, and `react-hook-form` will populate `errors.name` if the `name` field below has any problems. - The `name` input gets `registered` into the form, and set as a required field. (docs) - Since we only are validating that the `name` is required, there's only one type of error (for an empty `name`), so we have a static `FormErrorMessage`.

```
<FormControl id="description">
  <FormLabel>Description</FormLabel>
  <Textarea name="description" ref={register} />
</FormControl>
```

The `description` field is way simpler, since it's optional. We just define the `TextArea`, and set `ref={register}` to register the form field, without having to invoke `register` or pass any parameters to it.

That's it for our component.

Testing our component

We need to add a new prop to our `<GalleryCreateModal>` :

```
<GalleryCreateModal
  isOpen={isGalleryCreateOpen}
  onClose={onGalleryCreateClose}
  onSubmit={handleGalleryCreateSubmit}
/>
```

And `handleGalleryCreateSubmit` can look like this to test our component:

```
const handleGalleryCreateSubmit = (gallery: Gallery): void => {
  console.log(`handleGalleryCreateSubmit -> `, gallery);
  onGalleryCreateClose()
};
```

If we open the modal and submit some data, the console will show this:

The GalleryEditModal Component

The `GalleryEditModal` is going to be very similar to the `GalleryCreateModal`, with a few notable differences. It has to accept new data every time it opens, and make sure the form repopulates with that data.

```
import { useEffect } from 'react';
import {
  Button,
  FormControl,
  FormErrorMessage,
  FormLabel,
  Input,
  Modal,
```

```
  ModalOverlay,
  ModalContent,
  ModalHeader,
  ModalFooter,
  ModalBody,
  ModalCloseButton,
  Textarea,
} from '@chakra-ui/react';
import { useForm } from 'react-hook-form';
import { Gallery } from '@prisma/client';

export interface Props {
  isOpen: boolean;
  onClose: () => any;
  onSubmit: (galleryId, gallery: Gallery) => void;
  defaultValues: Gallery;
  galleryId: number;
}

export const GalleryEditModal = ({
  isOpen = false,
  onClose = () => {},
  onSubmit,
  defaultValues,
  galleryId,
}: Props) => {
  const { register, reset, handleSubmit, errors } = useForm({ defaultValues });
  const composedOnSubmit = (gallery) => {
    onSubmit(galleryId, gallery);
  };

  useEffect(() => {
    reset(defaultValues);
  }, [defaultValues]);

  return (
    <Modal isOpen={isOpen} onClose={onClose}>
      <form onSubmit={handleSubmit(composedOnSubmit)}>
        <ModalOverlay />
        <ModalContent>
          <ModalHeader>Create Gallery</ModalHeader>
          <ModalCloseButton />
          <ModalBody>
            <FormControl id="name" isInvalid={!!errors.name}>
              <FormLabel>Name</FormLabel>
              <Input
```

```
              type="text"
              name="name"
              ref={register({ required: true })}
            />
            <FormErrorMessage mb={2}>This field is required</FormErrorMessage>
          </FormControl>

          <FormControl id="description">
            <FormLabel>Description</FormLabel>
            <Textarea name="description" ref={register} />
          </FormControl>
        </ModalBody>

        <ModalFooter>
          <Button colorScheme="blue" mr={3} onClick={onClose}>
            Cancel
          </Button>
          <Button variant="ghost" type="submit">
            Save
          </Button>
        </ModalFooter>
      </ModalContent>
    </form>
  </Modal>
);
};
```

Let's break this down.

```
export interface Props {
  isOpen: boolean;
  onClose: () => any;
  onSubmit: (galleryId, gallery: Gallery) => void;
  defaultValues: Gallery;
  galleryId: number;
}

export const GalleryEditModal = ({
  isOpen = false,
  onClose = () => {},
  onSubmit,
  defaultValues,
  galleryId,
}: Props) => {
```

Our props are very similar to those for `GalleryCreateModal`, with a few new ones:

- `defaultValues` will contain the existing gallery properties to populate the form fields with.
- `galleryId` is the unique ID of the gallery. We include this for sending to `onSubmit` below.
- `onSubmit` also has a new `galleryId` parameter. This will be sent back out so that the API call will know which gallery to update.

```
const { register, reset, handleSubmit, errors } = useForm({ defaultValues });
```

The `useForm` hook is only different for the edit modal in that it takes a set of `defaultValues`.

```
  const composedOnSubmit = (gallery) => {
    onSubmit(galleryId, gallery);
  };
```

`composedOnSubmit` will be the new callback passed to the `react-hook-form` `handleSubmit` function in our form.

```
  useEffect(() => {
    reset(defaultValues);
  }, [defaultValues]);
```

The React `useEffect` hook runs code whenever something in the component changes; specifically, we're telling it to run anytime `defaultValues` changes. When `defaultValues` does change (i.e., when a new gallery is selected for editing in the dashboard), the `reset` function of the `useForm` hook is called to reset the form the new gallery name and description.

```
      <form onSubmit={handleSubmit(composedOnSubmit)}>
```

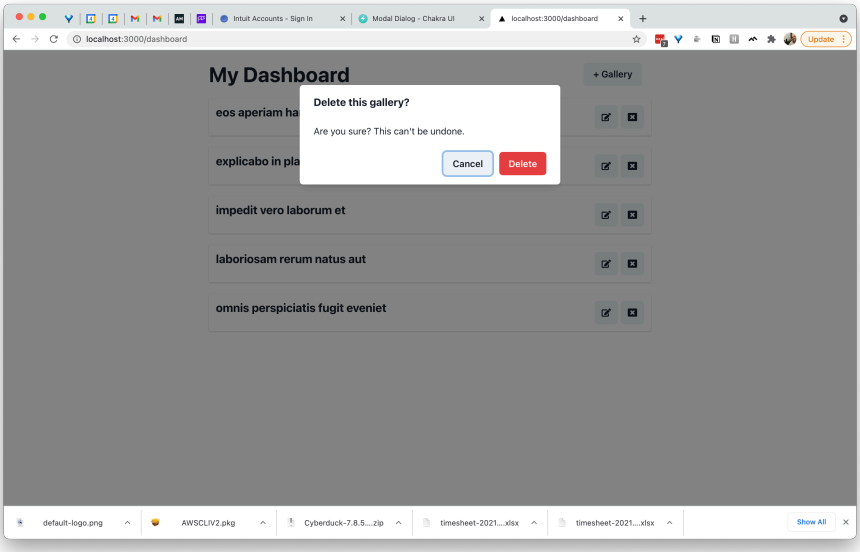
This is the same as in the create modal, except for the new `composedOnSubmit` handler.

That's it.

You'll notice that there's a lot of very similar code between the create and edit modals. That's ok! We're moving fast and shipping functionality here. We'll have time to DRY this code up later.

Now that our modal components exist, let's take the next steps to put them to use.

The GalleryDeleteDialog Component



The popup that shows when we try to delete a gallery will be a tad different.

We're going to use a Chakra `AlertDialog` in this case instead of a `Modal`, since we just want to show confirm/cancel buttons and a message ("Are you sure?"), instead of a modal with arbitrary content or a form in it.

```
import { useRef } from 'react';
import {
  AlertDialog,
  AlertDialogOverlay,
  AlertDialogContent,
  AlertDialogHeader,
  AlertDialogBody,
  AlertDialogFooter,
  Button,
} from '@chakra-ui/react';

export interface Props {
  isOpen: boolean;
  onCloseClick: () => unknown;
  onConfirmClick: (e, id) => unknown;
  galleryId: number;
}
```

```
export const GalleryDeleteDialog = ({
  isOpen,
  onCloseClick,
  onConfirmClick,
  galleryId,
}: Props) => {
  const cancelRef = useRef();
  const composedOnConfirmClick = (e) => {
    onConfirmClick(e, galleryId);
  };

  return (
    <AlertDialog
      isOpen={isOpen}
      leastDestructiveRef={cancelRef}
      onClose={onCloseClick}
    >
      <AlertDialogOverlay>
        <AlertDialogContent>
          <AlertDialogHeader fontSize="lg" fontWeight="bold">
            Delete this gallery?
          </AlertDialogHeader>

          <AlertDialogBody>Are you sure? This can't be undone.</AlertDialogBody>

          <AlertDialogFooter>
            <Button ref={cancelRef} onClick={onCloseClick}>
              Cancel
            </Button>
            <Button colorScheme="red" onClick={composedOnConfirmClick} ml={3}>
              Delete
            </Button>
          </AlertDialogFooter>
        </AlertDialogContent>
      </AlertDialogOverlay>
    </AlertDialog>
  );
};
```

The component looks a lot like the Chakra AlertDialog example with a few exceptions:

- There's no button for toggling the modal, that'll exist outside the component.
- We're passing in similar props to the other modals we've created. `galleryId` is all we need here since we're just passing it back out of the component in the `onConfirmClick` handler so that the `delete`

API function can be called with it.
- `composedOnConfirmClick` uses a pattern we saw in `GalleryEditModal`; since our `onConfirmClick` prop takes two params (an event and a galleryID), and the `Button` component's `onClick` only receives an event, we create the `composedOnConfirmClick` function when the component renders to merge the event parameter and the `galleryId` prop and send them to the `onConfirmClick` callback.

Creating an API Layer

Now that all of our API endpoints exist, the client side of the app needs a way to communicate with the server side. We'll create a series of functions for our pages to use to request the API routes.

API function planning

To recap, these are the API requests we'll need to access from the front end:

- GET /api/galleries
- GET /api/galleries/:id
- POST /api/galleries/create
- PATCH /api/galleries/:id/update
- DELETE /api/galleries/:id/delete

Scaffolding the API functions

We'll create functions in `/lib/client/api/Galleries.ts` to make all of these api requests. Here are the function definitions with comments to reiterate the purpose of each:

```
/**
 * GET /api/galleries
 *
 * Retrieve all galleries.
 */
export async function all(): Promise<Gallery[]> {

    // ...
}

/**
 * GET /api/galleries/:id
 *
 * Retrieve a single gallery.
 */
```

```
export async function get(id: number): Promise<Gallery> {

    // ...
}

/**
 * POST /api/galleries/create
 *
 * Create a gallery.
 */
export async function create(
  gallery: Prisma.GalleryCreateInput
): Promise<Gallery> {

    // ...
}

/**
 * PATCH /api/galleries/:id/update
 *
 * Update a gallery.
 */
export async function update(
  id: number,
  gallery: Prisma.GalleryUpdateInput
): Promise<Gallery> {

    // ...
}

/**
 * DELETE /api/galleries/:id/delete
 *
 * Delete a gallery.
 */
export async function destroy(id: number): Promise<Gallery> {
    // ...
}
```

Notes: * Since they're all `async` functions, they'll all return a `Promise`. The Promise type in TypeScript uses a TypeScript generic. Which means that it takes a second variable type as a type as a parameter inside the <>brackets. In the case of a `Promise`, the generic type we pass inside the <> tells TS what type the `Promise` will resolve to when it's done. * Our Promises return objects in the shape of the type `Gallery` , which is one of Prisma's generated types. The `create` and `update` calls

accept `GalleryCreateInput` and `GalleryUpdateInput` types respectively, which are similar to `Gallery`but all fields are not required. * Generated types are dynamically-generated types that are custom to our own schema, and get created when we run `prisma generate`. You can see all the generated types in your project at any time by opening `node_modules/.prisma/index.d.ts`.

API function structure

The API functions will use the global JavaScript `fetch` to make the requests. All will have the following structure:

```
// This is pseudocode.
export async function myfunction(params): Promise<ReturnValue> {
  const response = await fetch(`/path/to/api/endpoint`, {method: 'GET|POST|PATCH|DELETE', he
  const json = await response.json();

  if (!response.ok) {
    throw json;
  }

  return json;
}
```

Breakdown: - We do a `fetch` to make a request to the specified endpoint, passing parameters that will vary for each function. - `headers` will tell each request that we're sending and expecting JSON. - We convert the response data to JSON, which is another async call. - `fetch` responses won't automatically throw errors if the server returns HTTP 4xx codes and such, we use `response.ok` to check whether it's a 2xx code (success) or not. - If there's an error, we `throw` it as a JS error (formatted in JSON) and expect that error to be handled wherever we're calling this function. Otherwise we just return the successful object.

Putting it all together

Here's the full code for all of the API functions inside `/lib/client/api/Galleries.ts`:

```
import { Prisma, Gallery } from '@prisma/client';

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

/**
 * GET /api/galleries
```

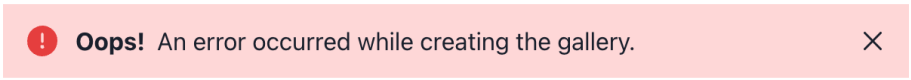
```
 *
 * Retrieve all galleries.
 */
export async function all(): Promise<Gallery[]> {
  const response = await fetch('/api/galleries/');
  const json = await response.json();

  if (!response.ok) {
    throw json;
  }

  return json;
}

/**
 * GET /api/galleries/:id
 *
 * Retrieve a single gallery.
 */
export async function get(id: number): Promise<Gallery> {
  const response = await fetch(`/api/galleries/${id}`);
  const json = await response.json();

  if (!response.ok) {
    throw json;
  }

  return json;
}

/**
 * POST /api/galleries/create
 *
 * Create a gallery.
 */
export async function create(
  gallery: Prisma.GalleryCreateInput
): Promise<Gallery> {
  const response = await fetch('/api/galleries/create', {
    method: 'POST',
    body: JSON.stringify(gallery),
    headers,
  });

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

```
  if (!response.ok) {
    throw json;
  }

  return json;
}

/**
 * PATCH /api/galleries/:id/update
 *
 * Update a gallery.
 */
export async function update(
  id: number,
  gallery: Prisma.GalleryUpdateInput
): Promise<Gallery> {
  const response = await fetch(`/api/galleries/${id}/update`, {
    method: 'PATCH',
    headers,
    body: JSON.stringify(gallery),
  });

  const json = response.json();

  if (!response.ok) {
    throw json;
  }

  return json;
}

/**
 * DELETE /api/galleries/:id/delete
 *
 * Delete a gallery.
 */
export async function destroy(id: number): Promise<Gallery> {
  const response = await fetch(`/api/galleries/${id}/delete`, {
    method: 'DELETE',
    headers,
  });

  const json = await response.json();

  if (!response.ok) {
    throw json;
```

```
  }

  return json;
}
```

Creating a Reusable Error Component

We need a way to show users when errors show up. Chakra-UI has a few components for things like this such as alert, alert dialog and toast. We'll use the alert component to create a reusable UI for showing errors inside the dashboard (and eventually other pages).



The ErrorAlert component

We'll add this code inside `/components/ErrorAlert/index.tsx`:

```
import {
  Alert,
  AlertIcon,
  AlertTitle,
  AlertDescription,
  CloseButton,
} from '@chakra-ui/react';

export interface Props {
  children: React.ReactElement;
  onCloseClick: (event) => any;
}

export const ErrorAlert = ({
  children,
  onCloseClick = defaultOnCloseClick,
}: Props) => (
  <Alert status="error" mb={3}>
    <AlertIcon />
    <AlertTitle mr={2}>Oops!</AlertTitle>
    <AlertDescription>{children}</AlertDescription>
    <CloseButton
      position="absolute"
      right="8px"
```

```
      top="8px"
      onClick={onCloseClick}
    />
  </Alert>
);
```

The breakdown:

```
export interface Props {
  children: React.ReactElement;
  onCloseClick: (event) => any;
}
```

This component will take 2 properties:

- `onCloseClick`: a function that will be called when we click the close button in the alert. We're exposing this as a prop since we want to control state outside of the component, and because it may behave a little differently on different pages.
- `children`: The text for the error message. This is a special prop name in React which I'll touch on more below. It's of type `React.ReactElement` which means we could pass another React component here as well if we wanted to.

The component definition looks like this:

```
export const ErrorAlert = ({ children, onCloseClick }: Props) => (
  <Alert status="error" mb={3}>
    <AlertIcon />
    <AlertTitle mr={2}>Oops!</AlertTitle>
    <AlertDescription>{children}</AlertDescription>
    <CloseButton
      position="absolute"
      right="8px"
      top="8px"
      onClick={onCloseClick}
    />
  </Alert>
);
```

It's largely taken straight out of the Chakra usage example. `<Alert>`wraps the whole thing. There's an icon on the left, followed by a title that doesn't change.

Inside the description component we pass in our React `children` prop. The `children` prop of any component will contain whatever element is inside the beginning and end tags of that component. For example:

`<Foo>These are the children</Foo>` <- Here the `children` prop will contain the string `These are the children`.

Similarly, we intend our `<ErrorAlert>` component to be used as such:

```
<ErrorAlert onClick={someFunction}>Error description goes here.</ErrorAlert>
```

The Connected Dashboard

As a reference for the upcoming sections where we connect the remaining pieces of the dashboard's core functionality, below is the complete source code of the dashboard as it will look at the end of this chapter. This version of the code is also in the source provided with this lesson of the book.

In the coming sections we'll highlight different portions as they pertain to each of the remaining UI components.

```
import { useEffect, useState } from 'react';
import useSWR from 'swr';
import {
  Flex,
  Heading,
  Box,
  Button,
  useDisclosure,
  VStack,
} from '@chakra-ui/react';

import { ErrorAlert } from 'components/ErrorAlert';
import { GalleryListItem } from 'components/GalleryListItem';
import { GalleryCreateModal } from 'components/GalleryCreateModal';
import { GalleryEditModal } from 'components/GalleryEditModal';
import { GalleryDeleteDialog } from 'components/GalleryDeleteDialog';
import { create, update, destroy } from 'lib/client/api/Galleries';

import { Prisma } from '@prisma/client';

export default function DashboardPage() {
  const {
    data: galleries,
    isValidating: dashboardIsLoading,
    error: dashboardFetchError,
    mutate: mutateGalleries,
  } = useSWR(`/api/galleries`);
  const {
    isOpen: isGalleryCreateOpen,
    onClose: onGalleryCreateClose,
    onOpen: onGalleryCreateOpen,
  } = useDisclosure();
```

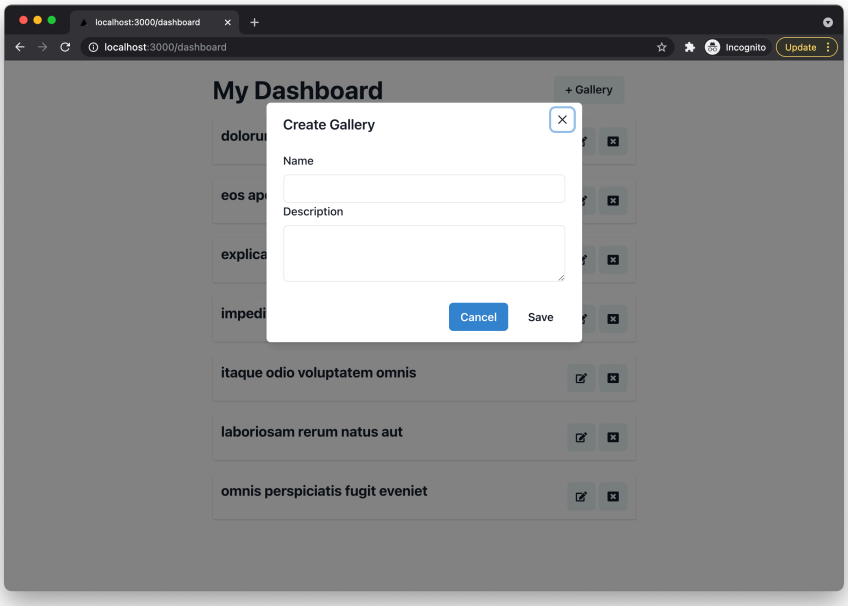
```
const {
  isOpen: isGalleryEditOpen,
  onClose: onGalleryEditClose,
  onOpen: onGalleryEditOpen,
} = useDisclosure();
const {
  isOpen: isGalleryDeleteOpen,
  onClose: onGalleryDeleteClose,
  onOpen: onGalleryDeleteOpen,
} = useDisclosure();
const [currentGalleryForEditing, setCurrentGalleryForEditing] = useState(
  null
);
const [currentGalleryForDeletion, setCurrentGalleryForDeletion] = useState(
  null
);
const [error, setError] = useState(null);

useEffect(() => {
  onGalleryEditOpen();
}, [currentGalleryForEditing]);

if (dashboardIsLoading) {
  return <h1>Loading dashboard...</h1>;
}

if (dashboardFetchError) {
  return <h1>Error loading the dashboard.</h1>;
}

const handleGalleryEdit = (e, gallery) => {
  e.preventDefault();

  setCurrentGalleryForEditing(gallery);
};

const handleGalleryEditSubmit = async (id, gallery) => {
  try {
    await update(id, gallery);
    mutateGalleries();
  } catch (error) {
    setError('An error occurred while creating the gallery.');
  } finally {
    onGalleryEditClose();
  }
};
```

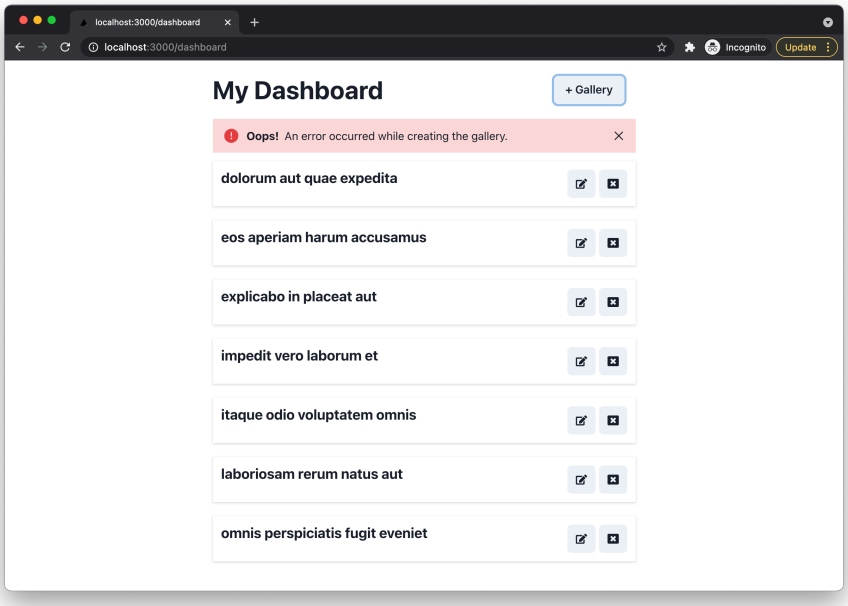
```
const handleGalleryDelete = (e, id) => {
  e.preventDefault();
  setCurrentGalleryForDeletion(id);
  onGalleryDeleteOpen();
};

const handleGalleryDeleteSubmit = async (e, id) => {
  e.preventDefault();

  try {
    await destroy(id);
    mutateGalleries();
  } catch (error) {
    setError('An error occurred while deleting the gallery.');
  } finally {
    onGalleryDeleteClose();
  }
};

const handleGalleryCreateSubmit = async (
  gallery: Prisma.GalleryCreateInput
): Promise<void> => {
  try {
    await create(gallery);
    mutateGalleries();
  } catch (error) {
    setError('An error occurred while creating the gallery.');
  } finally {
    onGalleryCreateClose();
  }
};

const handleErrorAlertClose = (event) => {
  event.preventDefault();
  setError(null);
};

return (
  <Box
    m="0 auto"
    p={5}
    maxWidth={{
      sm: '100%',
      md: '100%',
      lg: '40em',
```

```
    xl: '50em',
    '2xl': '74em',
  }}
>
  <Flex
    direction="row"
    alignItems="center"
    justifyContent="space-between"
    mb={5}
  >
    <Heading>My Dashboard</Heading>
    <Button mr={4} onClick={onGalleryCreateOpen}>
      + Gallery
    </Button>
  </Flex>

  {error && (
    <ErrorAlert onCloseClick={handleErrorAlertClose}>{error}</ErrorAlert>
  )}

  <VStack spacing={5}>
    {galleries?.map((gallery) => (
      <GalleryListItem
        href={`/galleries/${gallery.id}`}
        name={gallery.name}
        key={gallery.id}
        onDeleteClick={(e) => handleGalleryDelete(e, gallery)}
        onEditClick={(e) => handleGalleryEdit(e, gallery)}
      />
    ))}
  </VStack>

  <GalleryCreateModal
    isOpen={isGalleryCreateOpen}
    onClose={onGalleryCreateClose}
    onSubmit={handleGalleryCreateSubmit}
  />

  {currentGalleryForEditing && (
    <GalleryEditModal
      isOpen={isGalleryEditOpen}
      onClose={onGalleryEditClose}
      onSubmit={handleGalleryEditSubmit}
      galleryId={currentGalleryForEditing?.id}
      defaultValues={currentGalleryForEditing}
    />
```

```
      )}

      {currentGalleryForDeletion && (
        <GalleryDeleteDialog
          isOpen={isGalleryDeleteOpen}
          onCloseClick={onGalleryDeleteClose}
          onConfirmClick={handleGalleryDeleteSubmit}
          galleryId={currentGalleryForDeletion?.id}
        />
      )}
    </Box>
  );
}
```

Connecting the GalleryCreateModal and ErrorAlert





Let's tie the pieces together. We need to make the dashboard do the

following:

1. Respond to the `click` event for the "+ Gallery" button;
2. Open the gallery modal;
3. After the user fills in information and submits it, make a request to the API (using the `create` function in `/lib/client/api/Galleries.ts`;
4. If `create` succeeds, close the modal and refresh the list of galleries;
5. If there's an error, close the modal and show an error alert in the dashboard.

A little bit of the code is already in place, but we'll review everything to show how it works as a whole.

```
  const {
    data: galleries,
    isValidating: dashboardIsLoading,
    error: dashboardFetchError,
    mutate: mutateGalleries,
  } = useSWR(`/api/galleries`);
```

We've started returning a mutation function from `useSWR` (docs). Now we can`mutateGalleries` to explicitly have the list of galleries refreshed after a new one is created.

```
  const {
    isOpen: isGalleryCreateOpen,
    onClose: onGalleryCreateClose,
    onOpen: onGalleryCreateOpen,
  } = useDisclosure();
```

Here we use the `useDisclosure` hook from Chakra-UI to create functions for showing and hiding the modal, and a variable indicate whether it's currently opened or closed.

```
const [error, setError] = useState(null);
```

Here we use the React state hook (docs) to give us a way to record an error if there's one to show. We'll look for this variable later in the component and show the `<ErrorAlert />`if it exists.

```
  const handleGalleryCreateSubmit = async (
    gallery: Prisma.GalleryCreateInput
  ): Promise<void> => {
    try {
      await create(gallery);
      mutateGalleries();
    } catch (error) {
      setError('An error occurred while creating the gallery.');
    } finally {
```

```
      onGalleryCreateClose();
    }
  };
```

The submit handler we'll pass to the modal. Let's break this down: - The `gallery` object passed in is defined as the `GalleryCreateInput` type generated by `prisma generate`. It's like a `Gallery`, but specific to creation in that automatically-generated fields (like `createdAt`, `updatedAt`) are not required. The function is also going to return a `Promise`. - Our "happy path" code inside the `try` block does two things: - executes the `create`API request, passing in the `gallery` data. Remember that this will return the created gallery on success, or throw an error on failure. - calls `mutateGalleries`to refresh the gallery list. - If an error is thrown, we catch it and call `setError`. The text we pass it is an error message that will be displayed in an `<ErrorAlert />` later in the page. - In the `finally` block we make sure that the modal gets closed whether we succeeded in creating the gallery or not.

```
        <Button mr={4} onClick={onGalleryCreateOpen}>
          + Gallery
        </Button>
```

This button already existed; it triggers gallery opening.

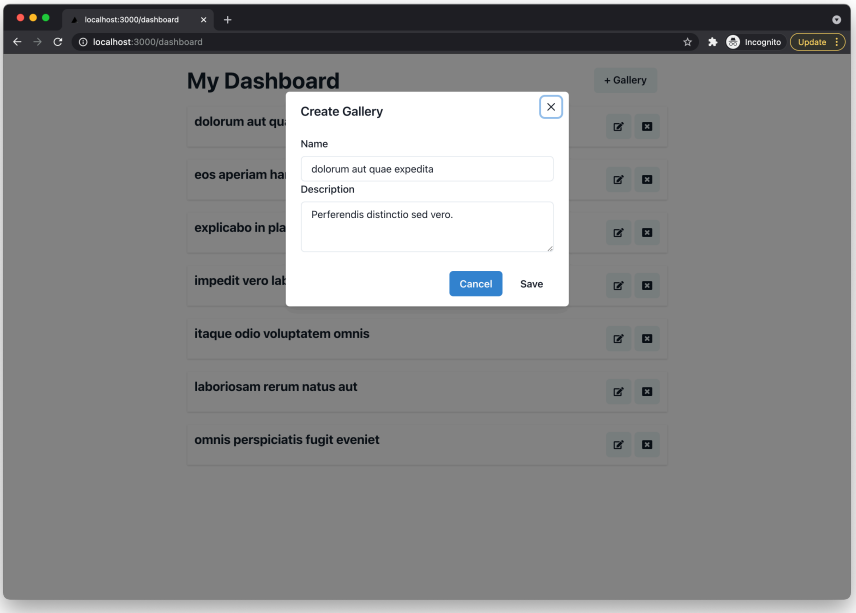
```
      {error && (
        <ErrorAlert onCloseClick={handleErrorAlertClose}>{error}</ErrorAlert>
      )}
```

Right above the gallery list, we add this statement. It looks to see if the `error` variable exists (which means that we called `setError` above from an API request failure), and shows the `<ErrorAlert />` with the error message.

```
      <GalleryCreateModal
        isOpen={isGalleryCreateOpen}
        onClose={onGalleryCreateClose}
        onSubmit={handleGalleryCreateSubmit}
      />
```

Later in the page, before the end of the `</Box>` container, we add the modal and connect it to the handlers and state variable we defined above.

Connecting the GalleryEditModal



We'll follow a similar pattern for the edit modal that we did for the create modal with some important differences.

We need to load the modal with existing information (the gallery's name and description). Since we'll be reusing the

- When the edit icon next to a gallery's name is clicked, we'll do two things:
  - we'll call a handler to show the modal. This time we'll be passing it the existing gallery's information to display in the edit form.
- When the user edits and submits the form, we'll call the `update` function from `/lib/client/api/Galleries.ts`

```
const {
  isOpen: isGalleryEditOpen,
  onClose: onGalleryEditClose,
  onOpen: onGalleryEditOpen,
} = useDisclosure();
```

As in the `GalleryCreateModal`, we call Chakra's `useDisclosure` to give us open/close functions, and a state variable to track whether it's currently open or not.

```
const [currentGalleryForEditing, setCurrentGalleryForEditing] = useState(
```

```
    null
  );
```

We call React's `useState` hook to create a state variable that will contain the gallery we're currently editing, so that we can pass its information into the `GalleryEditModal`.

```
  useEffect(() => {
    onGalleryEditOpen();
  }, [currentGalleryForEditing]);

  const handleGalleryEdit = (e, gallery) => {
    e.preventDefault();

    setCurrentGalleryForEditing(gallery);
  };
```

We'll talk about these two snippets together, since they work together and it's not necessarily obvious how or why.

The `handleGalleryEdit` function is called when we click the edit icon next to a gallery, kicking off this functionality. Inside it we call `setCurrentGalleryForEditing` to make sure the gallery we're editing in the modal is the one we just clicked.

At first it seems like this would be enough; below, we pass the id of the `currentGalleryForEditing` variable into the `GalleryEditModal`, so that seems pretty straight forward. Why can't we just do the following and call it a day?

```
// Wrong, don't do this.
const handleGalleryEdit = (e, gallery) => {
  e.preventDefault();

  setCurrentGalleryForEditing(gallery);
  onGalleryEditOpen();
};
```

The answer is that setting state in React is **asynchronous**. If we do the above, `setCurrentGalleryForEditing` may not complete before the modal is open, and it will throw an error since `currentGalleryForEditing` doesn't exist yet.

How, then, can we make sure that `currentGalleryForEditing` has a value (and the correct value at that) before we open the modal?

`useEffect` is the answer. Revisiting the correct snippet for opening the modal above:

```
  useEffect(() =>
```

```
    onGalleryEditOpen();
  }, [currentGalleryForEditing]);
```

If you recall, `useEffect` is called whenever something within the component changes. In this case we pass `[currentGalleryForEditing]` as a second parameter, telling `useEffect` to run `onGalleryEditOpen()` only when `currentGalleryForEditing` changes, and immediately after it changes. This is how we can ensure the new value is set correctly before opening the modal and populating its form data.

```
  const handleGalleryEditSubmit = async (id, gallery) => {
    try {
      await update(id, gallery);
      mutateGalleries();
    } catch (error) {
      setError('An error occurred while creating the gallery.');
    } finally {
      onGalleryEditClose();
    }
  };
```

This is the submit handler we pass to the modal below. It follows the same pattern as `handleGalleryCreateSubmit`: - call the `update` api call with the right data (the `id` of the gallery and the `gallery` object containing name and description); - call `mutateGalleries` so that `useSWR` knows to refresh the list of `galleries`; - if there's an error call `setError` so the `ErrorAlert` will show; - in either case, always close the gallery modal.

```
      <VStack spacing={5}>
        {galleries?.map((gallery) => (
          <GalleryListItem
            href={`/galleries/${gallery.id}`}
            name={gallery.name}
            key={gallery.id}
            onDeleteClick={(e) => handleGalleryDelete(e, gallery)}
            onEditClick={(e) => handleGalleryEdit(e, gallery)}
          />
        ))}
      </VStack>
```

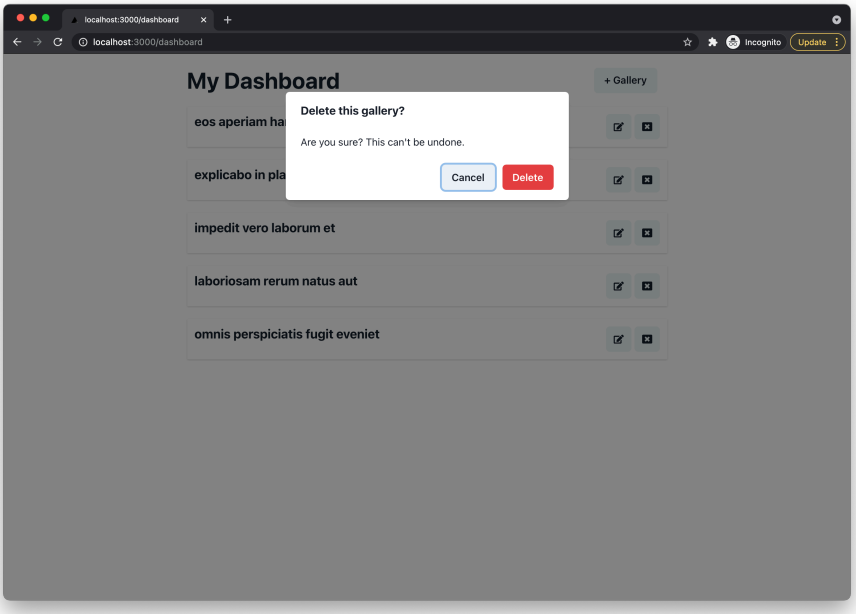
Above we connect the `handleGalleryEdit` handler, passing it the event and the current `gallery` object.

```
      {currentGalleryForEditing && (
        <GalleryEditModal
          isOpen={isGalleryEditOpen}
          onClose={onGalleryEditClose}
          onSubmit={handleGalleryEditSubmit}
```

```
        galleryId={currentGalleryForEditing?.id}
        defaultValues={currentGalleryForEditing}
      />
    )}
```

Then lastly there's the component itself, taking all the props we've defined above. As a final check we're wrapping it with a check to confirm that `currentGalleryForEditing` exists and is not a falsy value, otherwise we'll get an error when the page initially renders (since it defaults to `null`).

Enabling Gallery Deletion



This will follow a similar pattern to the gallery edit/update process, but with a few differences.

- When the gallery's delete icon is clicked, we'll call `setCurrentGalleryForDeletion`.
- A `useEffect` hook will listen for the above state change and call `onGalleryDeleteOpen` to open a `GalleryDeleteDialog`.
- The major difference is that this component is a `Dialog` with cancel and confirm buttons, and not a `Modal`with a form in it. But it will still submit an API request when the confirm button is clicked.

```
const {
```

```
    isOpen: isGalleryDeleteOpen,
    onClose: onGalleryDeleteClose,
    onOpen: onGalleryDeleteOpen,
  } = useDisclosure();
```

Nothing new here this is the same pattern as in the other two modals for handling showing/hiding/state of the displaying the dialog.

```
  const [currentGalleryForDeletion, setCurrentGalleryForDeletion] = useState(
    null
  );
```

Similarly to the `GalleryEditModal`, we're using the React state hook to reference the gallery we want to delete.

```
  useEffect(() => {
    onGalleryDeleteOpen();
  }, [currentGalleryForDeletion]);
```

Also like `GalleryEditModal`, we'll use the effect hook to ensure we open the dialog only after `setCurrentGalleryForDeletion` is done running.

```
  const handleGalleryDelete = (e, gallery) => {
    e.preventDefault();
    setCurrentGalleryForDeletion(gallery);
  };
```

The event handler for the delete icon just sets the gallery we want to delete into state.

```
const handleGalleryDeleteSubmit = async (e, id) => {
    e.preventDefault();

    try {
      await destroy(id);
      mutateGalleries();
    } catch (error) {
      setError('An error occurred while deleting the gallery.');
    } finally {
      onGalleryDeleteClose();
    }
  };
```

Same pattern as in the create and edit modals: - Make the API request; - Call `mutateGalleries()` to refresh the list; - If error, call `setError` to show the `ErrorAlert`; - Always close the dialog.

```
      {currentGalleryForDeletion && (
        <GalleryDeleteDialog
          isOpen={isGalleryDeleteOpen}
          onCloseClick={onGalleryDeleteClose}
```

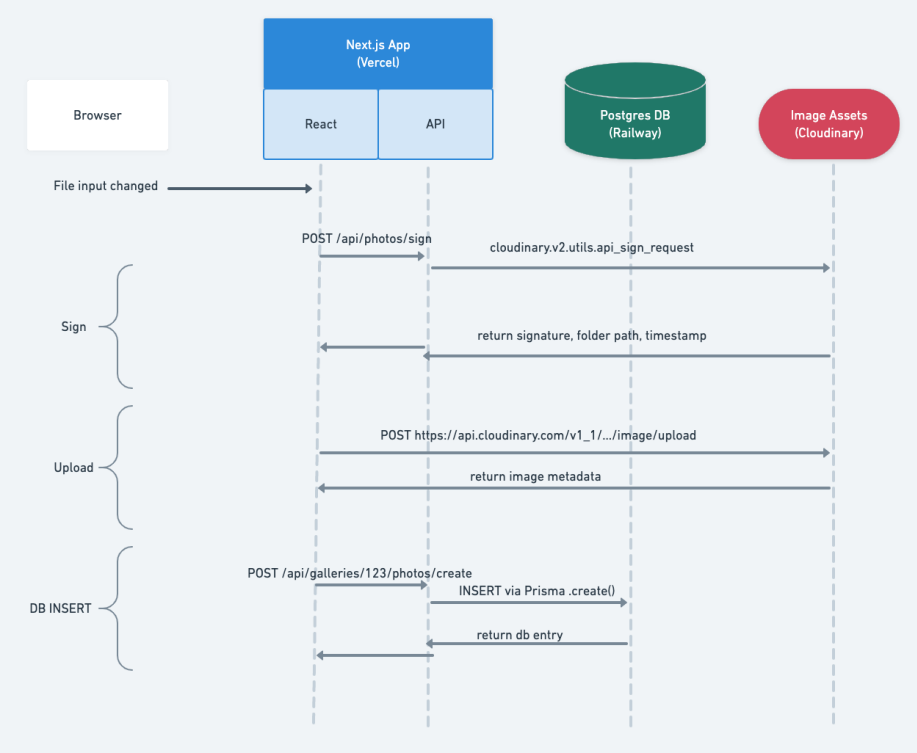
```
        onConfirmClick={handleGalleryDeleteSubmit}
        galleryId={currentGalleryForDeletion?.id}
      />
    )}
```

The actual component instantiation resembles `GalleryEditModal` the most, since it does a check to make sure `currentGalleryForDeletion` exists.

Photo Gallery and Image Uploads

Image Architecture Planning

Since where will be a few moving pieces in a single image upload request, let's walk through what an upload will look like from start to finish.



When the user taps the "add photos" button on a gallery page, selects a photo to upload, and presses the "Open" button, here are the things that will happen in terms of architecture:

1. Our API will talk to Cloudinary's API to sign the request. It'll basically say, "we want to upload something, give us a unique token."

2. We'll have the browser upload the image file directly to Cloudinary, which will return info about the uploaded image.
3. We'll tell our API to add photo metadata to our database, and return some metadata so we can update the UI.

Here are the pieces of functionality we'll ultimately create to achieve the above:

1. A hidden `<input type="file"/>` field on the gallery detail page with a change handler. The change handler will kick off the subsequent series of HTTP requests;
2. An API endpoint (`/pages/api/photos/sign.ts` that will call a Cloudinary API function to create a request signature ;
3. Some JavaScript code to upload the image file (with signature) to Cloudinary;
4. An API endpoint (`pages/api/galleries/[id]/photos/create.ts`) to add the photo to the database and return metadata ;
5. Some React code to update the UI when done uploading.

Important Cloudinary Links

Cloudinary is a robust media upload service with a number of APIs and SDKs. To avoid giving you a case of documentation overload, here are the parts of Cloudinary we'll need to know about, and links to their docs.

Cloudinary Node.js SDK

We'll use this to interact with Cloudinary in our API endpoints. Docs and GitHub repo.

REST API Uploads

An article on how to skip the SDK and upload directly via an HTTP POST directly to their REST API (link).

Upload Signing

An article on how to create an upload signature using the Node SDK (link).

Upload Presets

Info on how to configure upload settings in your Cloudinary account (link).

Configuring Cloudinary

Cloudinary registration is free and easy: https://cloudinary.com/users/register/free. Fill out the form.

When you're done signing up, you should be redirected to your dashboard. If not click the top nav Dashboard item:

There you'll find the credentials our app will need to interact with Cloudinary:

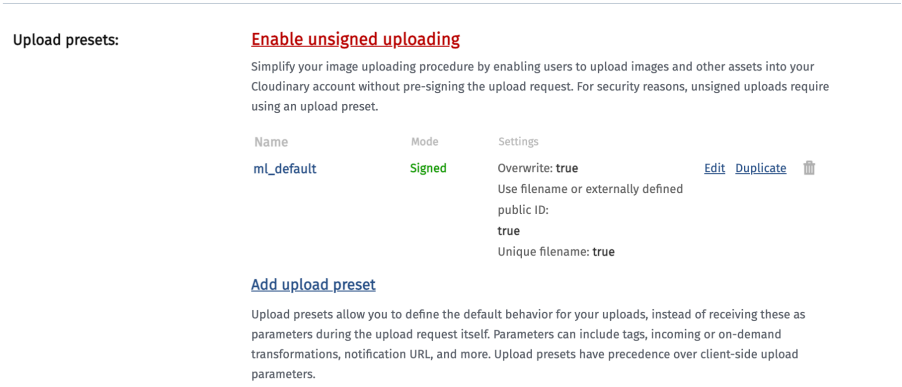
Save these into your `.env` as follows:

```
CLOUDINARY_API_KEY=81231321231328
CLOUDINARY_API_SECRET=fabcabcabcabcabcabc0
CLOUDINARY_ENV=cloudinary://81231321231328:fabcabcabcabcabcabc0@dabbbbbbg
CLOUDINARY_CLOUD_NAME=dabbbbbbg
CLOUDINARY_UPLOAD_PRESET=ml_default
```

All of the above environment variables will take their values from the above Account Details settings in your dashboard, with the exception of `CLOUDINARY_UPLOAD_PRESET`which can be set to `ml_default`.

The upload preset setting should already be `ml_default`in your Cloudinary account, but definitely verify it as follows:

1. First, skim the docs for managing upload presets (link), which are also provided earlier in this chapter.
2. In Cloudinary go to **Settings** -> **Upload**.
3. Make sure the Upload Presets section contains the following:



Note that we're using **signed** uploads which will be the most secure.

Adding Image Support to Our Database

In order to store or retrieve a list of photos within a gallery, we'll need two adjustments to our schema:

- A new `Photo` table to contain photo metadata, with a unique field that references the photo in Cloudinary;
- A way of associating the photo to the gallery that contains it. Galleries and next.config.js
- photos will have a **one-to-many relationship**; that is, a gallery will have many photos, but a photo will only be a part of a single gallery.

Updating Our DB Structure

Here's how we achieve this photo <-> gallery relationship in Prisma. We add the following to our `schema.prisma` file:

```
model Gallery {
  id          Int      @id @default(autoincrement())
  name        String
  description String?
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt
  photos      Photo[]
}

model Photo {
  id                  Int      @id @default(autoincrement())
```

```
  caption             String?
  url                 String
  createdAt           DateTime @default(now())
  updatedAt           DateTime @default(now())
  gallery             Gallery  @relation(fields: [galleryId], references: [id])
  galleryId           Int
  cloudinaryPublicId String
  width               Int
  height              Int
}
```

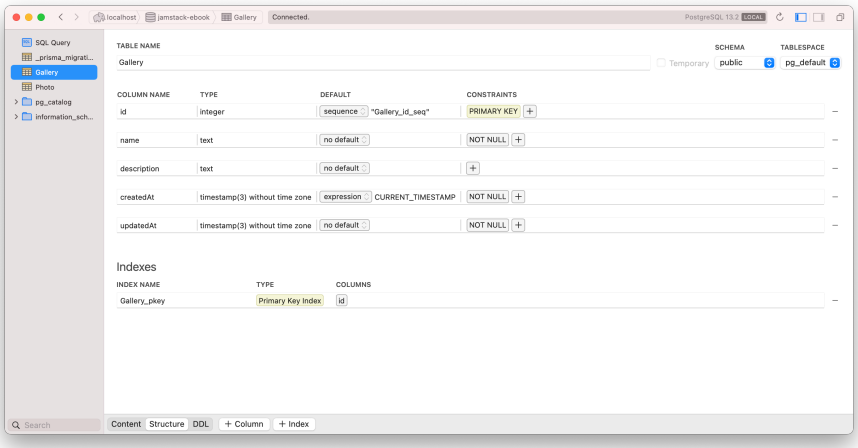
Breakdown:

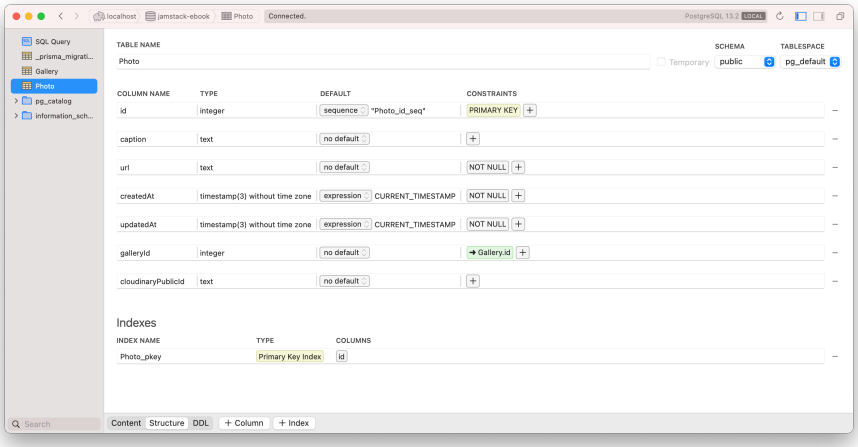
- `Gallery` already exists; we add the `photos` association to indicate that a single gallery will have an array of photos associated with it.
- We add the `Photo` table. It'll have a unique `id` and timestamps (`createdAt`, `updatedAt`) just like a gallery or any other model we'll use. Here are the other fields:
  - It will have an optional text `caption`
  - It will have a `url` to the photo on Cloudinary, which will be used to display the photo.
  - `cloudinaryAssetId` is the unique ID that refers back to the image on Cloudinary
  - the `gallery` line defines the relationship. `photo.gallery` is how we'll access it from a photo object. It's of type `Gallery` defined above.
  - The `@relation`declaration defines the relationship. It indicates that the field `galleryId` in this table (`Photo`) refers to the `id` field in the `Gallery` table.
  - `galleryId` is our foreign key association back to the `Gallery` table.
  - `width` and `height` will be used to store photo metadata in our database, so that we don't need to pull it from the Cloudinary API (and waste API requests, which can cost money)

Defining the relation this way gives us the ability to query and manipulate photos while looking up galleries and vice versa.

Reference links: - Prisma schema API (Reference) - Relations (Reference) - One-to-Many Relations

If we migrate by running `yarn prisma migrate dev`, we'll see in our Postgres database the new structure for each table:





Adding Photo Seed Data

Let's alter `prisma/seed.ts` to add 10 dummy photos for each dummy gallery we create. The code looks like this:

```
const { PrismaClient } = require('@prisma/client');
const faker = require('faker');

const prisma = new PrismaClient({
  log: ['query'],
});

// Generate random dummy gallery data.
const galleryData = Array.from(Array(10).keys()).map((i) => ({
```

```
  name: faker.lorem.words(4),
  description: faker.lorem.sentence(),
}));

const createGallery = async (g) => {
  // Generate random dummy photo data.
  const photoData = Array.from(Array(10).keys()).map((i) => {
    // Generate random dimensions, with a minimum width and height of 300.
    const width = (Math.floor(Math.random() * 12) + 3) * 100;
    const height = (Math.floor(Math.random() * 12) + 3) * 100;
    return {
      caption: faker.lorem.sentence(),

      // Get random image: https://stackoverflow.com/a/64094755/344391
      url: `https://source.unsplash.com/random/${width}x${height}?sig=incrementingIdentifier
      cloudinaryPublicId: faker.random.alphaNumeric(),
      width,
      height,
    };
  });

  const galleryDataWithPhotos = {
    // Insert the gallery.
    ...g,

    // Insert 10 photos.
    photos: {
      create: photoData,
    },
  };

  await prisma.gallery.create({ data: galleryDataWithPhotos });
};

async function main() {
  await Promise.all(galleryData.map((g) => createGallery(g)));
}

main()
  .catch((e) => {
    console.error(e);
    process.exit(1);
  })
  .finally(async () => {
    await prisma.$disconnect();
  });
```

```
export {};
```

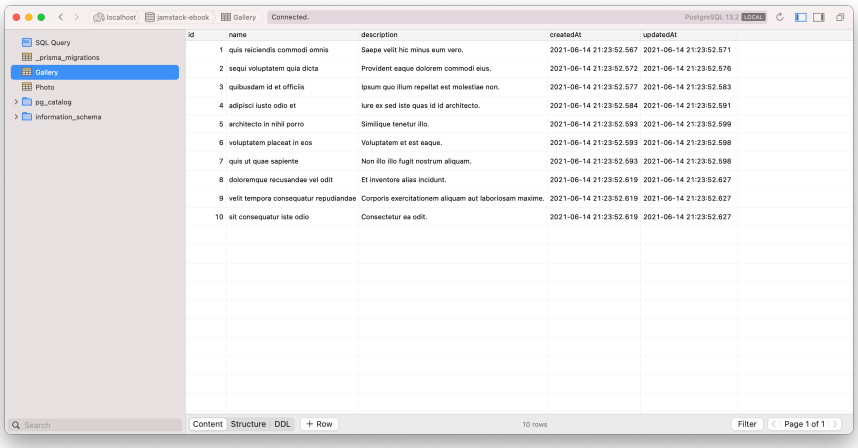
Breakdown:

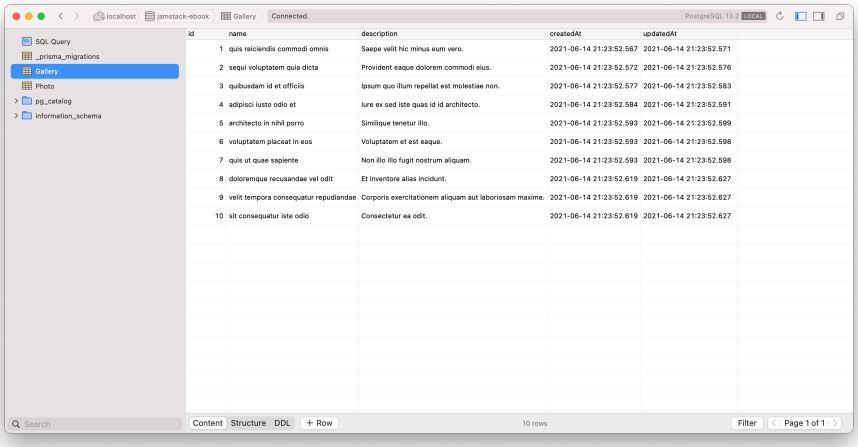
- The `galleryData` definition is carried over from our earlier work, to generate some dummy content we'll insert into the `Gallery` table .
- We define a `createGallery` function which we'll run for each dummy gallery.
  - `photoData` uses the same `Array` technique as before to generate an array of 10 dummy photos.
  - We'll make each photo have a `width` and `height` that each range from 300 to 1500, so that we have a variety of proportions to work with.
  - We're using a nice trick from unsplash.com that will use the same url to generate a random image each time it's loaded. (Link)
  - `caption` and `cloudinaryPublicId` are just more random dummy data.
- `galleryDataWithPhotos` is the structure we'll pass to `prisma.gallery.create` to create a single gallery.

Also note that I broke things into variables to make the code read a bit cleaner, but the important part is that we're creating galleries *with* their photos at the same time. The syntax, if the variables weren't there, would look like the below:

```
await prisma.gallery.create({
  data: {
    name: 'gallery name',
    description: 'gallery description',
    photos: {
      create: [
        {
          caption: 'photo caption',
          url: '...',
          cloudinaryPublicId: "abc123",
          width: 1000,
          height: 500
        },
      ],
    },
  },
});
```

After we run `prisma migrate reset --force --preview-feature`, we can look at our database and see that both tables now have data:





Building the Photo Gallery Page

We now have dummy photo data, so the next step is to take a look at it. While we're at it, we'll add a piece of UI that will be central to the project: the gallery interface.

Adding photo data to our API endpoint

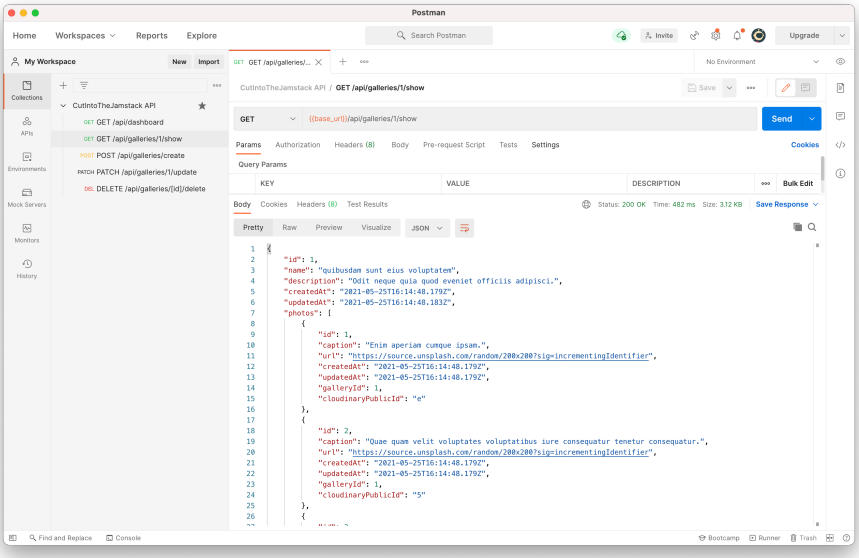
Our gallery data endpoint, `/pages/api/galleries/[id]/show.ts` Will currently return gallery info. Let's make it send over a list of photos as well.

This is as simple as adding an `include` option to the Prisma `findUnique`

call (docs). We'll also tell Prisma that we want the newest photos to come first so people will easily notice them (the `orderBy` clause).

```
const gallery = await prisma.gallery.findUnique({
  where: { id },
  include: {
    photos: {
      orderBy: { createdAt: 'desc' },
    },
  },
});
```

Testing the request in Postman, we see that the photos are now there:

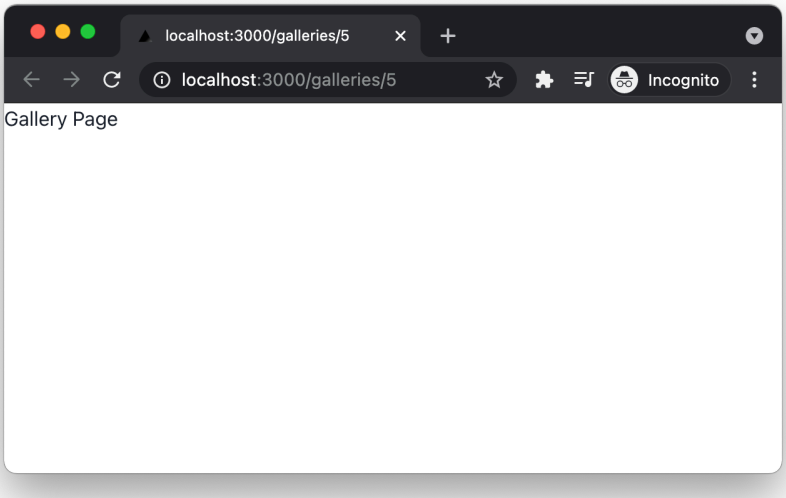


Adding a gallery page

Create a new file at `/pages/galleries/[id].tsx`. We'll be using Next.js's dynamic routing again here to render the page based on the gallery ID. Note that the dynamic element of the page's path is the filename in this case; both file names and directory names can be dynamic.

```
export default function GalleryShowPage() {
  return <h1>Gallery Page</h1>;
}
```

The stubbed page:



Magical. But not complicated yet.

Getting the gallery ID from the URL

We will fetch using `useSWR` as before, but first we need to get the dynamic gallery ID from the route so that we can pass it to the endpoint. We'll achieve this using the Next.js `useRouter` hook (dynamic routes docs, next/router docs).

```
import { useRouter } from 'next/router';

// ...

export default function GalleryShowPage() {
  const router = useRouter();
  const galleryId = Number(router.query.id);

// ..
```

Breakdown:

- `useRouter` gives us the router instance
- `router.query` represents the url parameters, including any dynamic route elements. In this case since our file path is `/pages/galleries/[id].tsx`, `id` will exist as a property of `router.query`.
- We force `id` to be a `Number` explicitly. This is because `router.query.id`

will be a `String` (as will any route variable), and it's safest to keep it to a `Number` in case since that's the type of data it represents.

Whitelisting image domains

Next we want to fetch the photo urls and display them, but first we'll need to add a `next.config.js` to allow those Unsplash images to load. Next.js requires you to do this to prevent abuse of external urls (docs).

Create a `next.config.js` at the root of the project if there isn't one already. And make sure the `images` property exists as follows:

```
module.exports = {
  images: {
    domains: ['source.unsplash.com'],
  },
};
```

This will allow the `Image` component to load and optimize images from the Unsplash URLs in the seeded database. Make sure to restart your local server for the changes to take effect.

Fetching and displaying images

Now let's get some data and display the images:

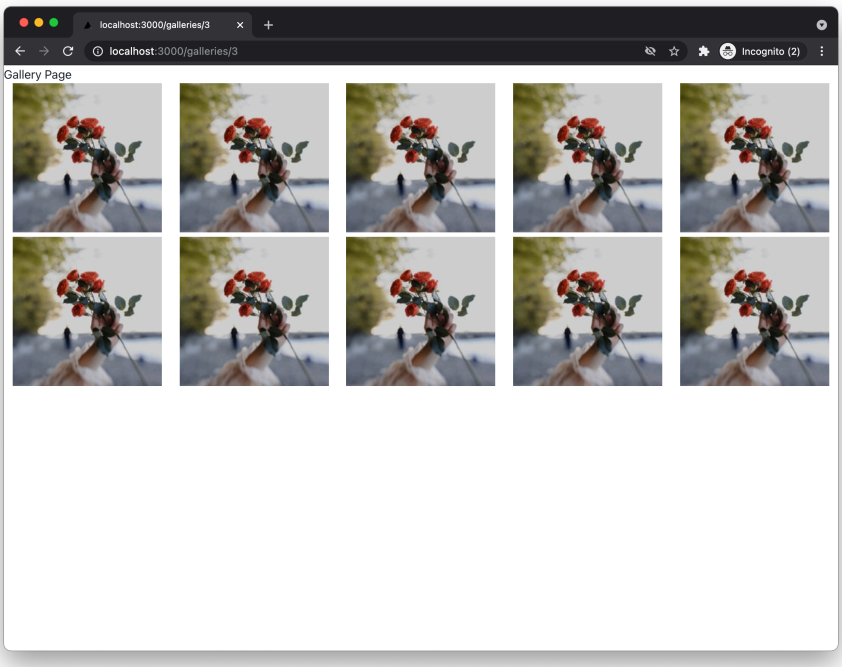
```
import useSWR from 'swr';
import { useRouter } from 'next/router';
import Image from 'next/image';
import { Box, Flex } from '@chakra-ui/react';

export default function GalleryShowPage() {
  const router = useRouter();
  const galleryId = Number(router.query.id);

  const { data: gallery } = useSWR(`/api/galleries/${galleryId}/show`);

  if (!gallery) {
    return null;
  }

  return (
    <>
      <h1>Gallery Page</h1>

      <Flex direction="row" wrap="wrap" justify="space-around">
        {gallery.photos.map((p) => {
```

```
          return (
            <Box key={p.id}>
              <Image src={p.url} width={200} height={200} />
            </Box>
          );
        })}
      </Flex>
    </>
  );
}
```

Breakdown:

- We call `useSWR` to fetch data from our API endpoint, `/api/galleries/${galleryId}/show`.
- `useSWR` operates asynchronously, so `gallery` might not always exist. So we return `null` if that is the case. When we touch up the UI, we'll remove this and show a loading animation in the body of the page for this case.
- We loop through `gallery.photos` and display each in a Next.js `Image` component (docs). This component is a fantastic Next.js feature since it handles optimization and responsive rendering for you.
- One thing to note is that by default the `Image` component will require fixed `width` and `height` attributes. This is easy now, since we know all of the Unsplash images are 200x200. Later we'll need to get data from Cloudinary to tell us what size the uploaded images are.
- We're temporarily using a Chakra-UI `Flex` component with `wrap="wrap"` to display a grid. This will be replaced with a real photo gallery UI later.

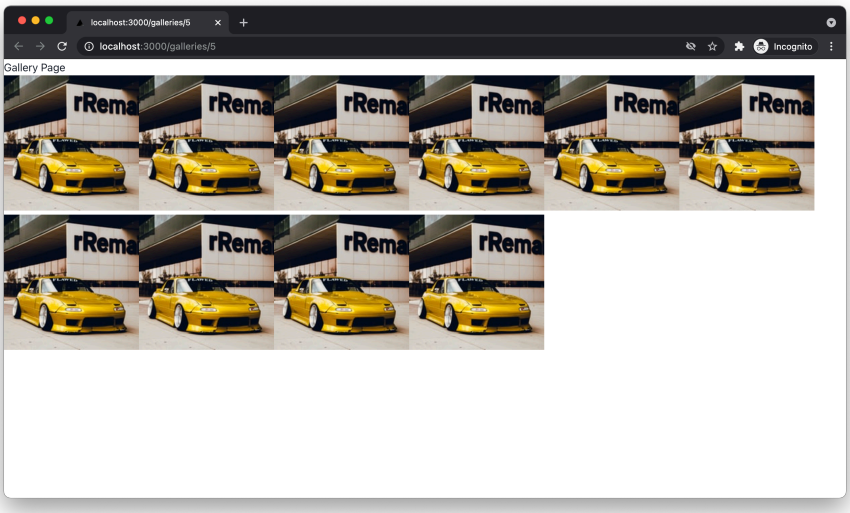
With the above you should now see a grid of a single random image:



Note that this will be the same image on each refresh because of the `Image` component optimizations. We can disable optimizations by adding the `unoptimized` attribute, so we can see different images after refreshes.

```
<Image src={photo.url} width={200} height={200} unoptimized />
```

Now each page refresh will show a new image.



Creating a photo grid

Let's display our photos in a more pleasant manner. We can use the Chakra-UI `Grid` component (docs) to display a responsive CSS grid for our photos. Final code for `/pages/galleries/[id].tsx`:

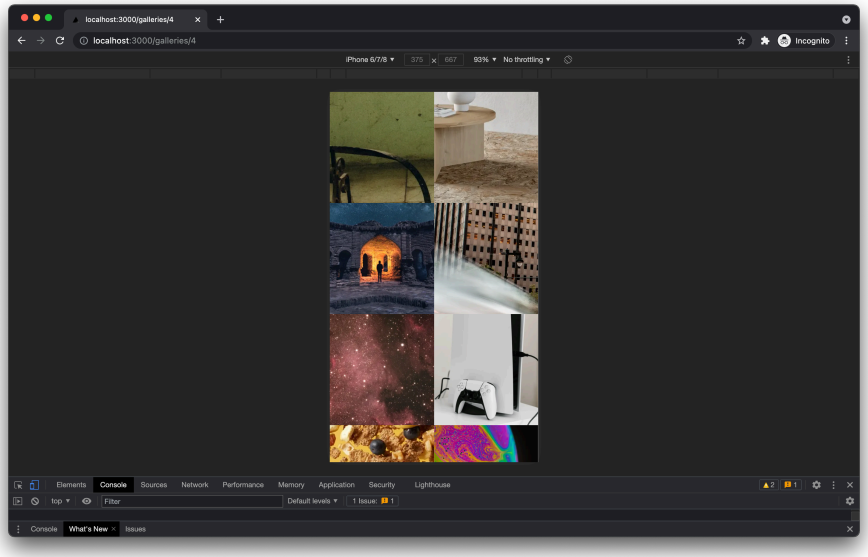
```
import useSWR from 'swr';
import { useRouter } from 'next/router';
import Image from 'next/image';
import { Box, Grid } from '@chakra-ui/react';

export default function GalleryShowPage() {
  const router = useRouter();
  const galleryId = Number(router.query.id);

  const { data: gallery } = useSWR(`/api/galleries/${galleryId}/show`);

  if (!gallery) {
    return null;
  }

  const rowHeight = { base: 200, md: 300 };

  return (
    <>
      <Grid
        templateColumns={{
```

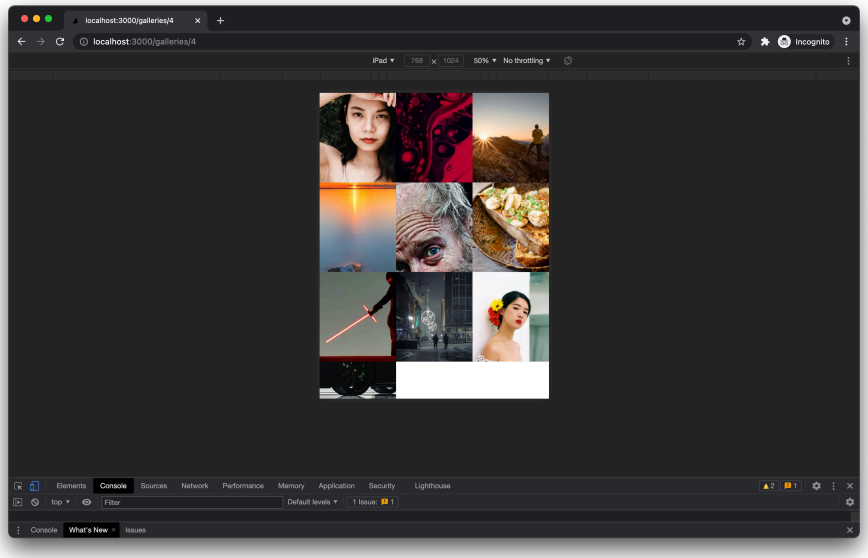
```
          base: '1fr 1fr',
          md: '1fr 1fr 1fr',
          lg: '1fr 1fr 1fr 1fr',
        }}
      >
        {gallery.photos.map(({ id, url }) => {
          return (
            <Box height={rowHeight} key={id} pos="relative">
              <Image key={id} src={url} layout="fill" objectFit="cover" />
            </Box>
          );
        })}
      </Grid>
    </>
  );
}
```

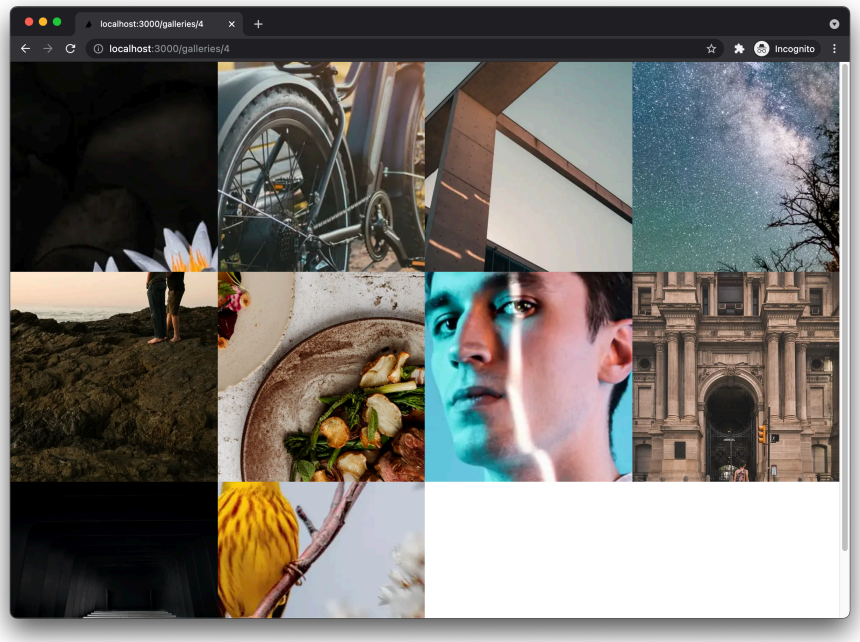
Breakdown:

- The `Grid` component defines a responsive number of columns. We'll have 2 columns on narrow mobile devices, then 3 medium-sized screens, and 4 on wide screens.
- Each row will be 200 pixels tall by default, then 300 pixels tall on medium-sized screens and up. This will help the images stay relatively square on all screen widths.
- The `1fr` measurement stands for one "**fr**actional" unit (docs). Declaring a series of columns with the same fractional value means they'll have the same width. (e.g., `'1fr 1fr 1fr` defines three columns that each take up the same percentage of the container's width—in this case, one-third.)

Now you'll see our gallery showing our images responsively, which will make it look nice on different device sizes:


localhost:3000/galleries/4
localhost:3000/galleries/4
Incognito
iPhone 6/7/8
375
667
93%
No throttling
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
top
Filter
Default levels
1 Issue:
Console
What's New
Issues


localhost:3000/galleries/4
localhost:3000/galleries/4
Incognito
iPad
768
1024
50%
No throttling
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
top
Filter
Default levels
1 Issue:
Console
What's New
Issues



DRYing up API Routes With Custom Middleware

Before we add a new set of API routes for photo manipulation, let's clean up the ones we've got. If you skim our existing endpoints, you'll see some code that is repetitive:

```
// Checking for the correct request method.
if (req.method !== 'DELETE') {
  console.log(`---------------- not DELETE, returning 400 `);
  return res.status(400).end();
}

// ...

// Importing and initializing Prisma.
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();
```

We'll use a useful middleware pattern I found on Hoang Vo's blog (link: Middleware in Next.js: The approaches without a custom server · Hoang Vo. See "Making a middleware as a wrapper.") to make both of these bits of code reusable. The general pattern is:

```
const withSomething = handler => {
  return async (req, res) => {
    // Do whatever this middleware needs to do.
    doSomething()

    // Attach anything to the request object, so that the
    //  API route has access to it.
    req.something = somethingToBeUsedByAPIRoute

    // Call the API route and return the result.
    return handler(req, res);
  };
}

export default withSomething;
```

This will let our API endpoints start to look like this:

```
export default withJamshotsApi(
  withPrismaClient(
    withSessionRequired(async (req, res) => {

    // Route-specific code here.

    })
  )
);
```

Which is way more compact! And You can have easy access to any or all middleware functions you need.

Extracting Prisma Client Instance

Some background reading:

- Deploying projects using Prisma to the cloud | Prisma Docs
- How to deploy a Prisma app to Vercel | Prisma Docs
- Connection management (Guide) | Prisma Docs
- prisma/deployment-example-vercel: Prisma deployment to Vercel example
  - Relevant code snippet

First we'll centralize the code that instantiates Prisma. This code can be centralized for two reasons: 1. To DRY up our code. 2. To add a bit of code that will prevent too many database connections in development.

Inside the new file `/lib/server/prismaClientInstance.ts`we'll add this code:

```
// /lib/server/prismaClientInstance.ts

import { PrismaClient } from '@prisma/client';

let prisma;

const options = {};

// Lets us set DEBUG=true in our .env for more logging.
if (process.env.DEBUG && process.env.DEBUG === 'true') {
  options['log'] = ['query', 'error', 'info', 'warn'];
}

if (process.env.NODE_ENV === 'production') {
  // Initialize the Prisma client normally in production.
  prisma = new PrismaClient(options);
} else {
  // If we're in development
  if (typeof global['prisma'] === 'undefined') {
    global['prisma'] = new PrismaClient(options);
  }

  prisma = global['prisma'];
}

export default prisma;
```

Breakdown:

- We added a little extra tool here for debugging in dev or production: if you set the flag `DEBUG=true` in your .env file, you'll see more detail in in the logs, down to the database queries.
- In production, we instantiate the prisma client as usual. Note that since we're exporting the instance, we'll no longer create and destroy the instance on every request; it'll exist in memory for all API endpoints to share and reuse.
- In development (aka, not on Vercel), we'll create a global variable for the prisma instance. This way it will also get reused locally, and we won't run into errors where Prisma can't get a database connection.

Now lets create a middleware to use this Prisma instance in our API routes:

```
// /middleware/withPrismaClient.ts
import { prisma } from 'lib/server/prismaClientInstance';

export const withPrismaClient = (handler) => {
```

```
  return async (req, res) => {
    try {
      req.prisma = prisma;
      return handler(req, res);
    } catch (e) {
      // TODO: Later this will log to Sentry, or other error handling service.
      console.log(`---------------- Prisma error: `, e);
      return res.status(500).end();
    }
  };
};

export default withPrismaClient;
```

Breakdown:

- `withPrismaClient` takes a function (our API route handler) its only param.
- It returns *another* `async` function that will eventually *call* the original function (`handler`)
- Inside the `try` block we attach the imported prisma client instance (the variable `prisma`) to the request object (`req`)
- If there's any error, we'll handle it in the `catch` block and set the HTTP response code to `500` (internal server error). Right now we just log the error, but in production we'll want to log this error with a service like Sentry.
- Now we call the prisma client instance at `req.prisma`and use it the same way we did before.

Now we can use this middleware in any API route by importing it and wrapping it around the main `async` function in the API route. For example, our gallery list endpoint will now look like this:

```
// /pages/api/galleries/index.ts
import { withPrismaClient } from 'middleware/withPrismaClient';

export default withPrismaClient(async (req, res) => {
  try {
    const galleries = await req.prisma.gallery.findMany({
      orderBy: [{ name: 'asc' }],
    });

    return res.status(200).json(galleries);
  } catch (error) {
    // Later this will log to Sentry, or other error handling service.
    console.log(`---------------- Prisma error: `, error);
    return res.status(500).end();
  }
```

```
});
```

Extracting HTTP Method Checking

For the bits of API code that check the HTTP request method, we'll add another middleware called `withHTTPMethodRequired`. This middleware function will take a parameter for the HTTP verb we want to check for.

This is a good case for using TypeScript. There are a small, finite amount of HTTP verbs we'll need for our app, so let's make them immutable so there's less room for error when using them.

```
// /lib/shared/types/HTTPMethod.ts
export enum HTTPMethod {
  GET = 'GET',
  POST = 'POST',
  PUT = 'PUT',
  PATCH = 'PATCH',
  DELETE = 'DELETE',
}
```

```
// /middleware/withHTTPMethodRequired
import { HTTPMethod } from 'lib/shared/types/HTTPMethod';

export const withHTTPMethodRequired = (method: HTTPMethod) => (handler) => {
  return async (req, res) => {
    if (req.method !== method) {
      console.log(
        `HTTP method ${method} is required for this endpoint, received ${req.method} instead
      );
      return res.status(400).end();
    }
    return handler(req, res);
  };
};

export default withHTTPMethodRequired;
```

Here's an example of usage in our gallery creation route:

```
// /pages/api/galleries/create.ts
import { HTTPMethod } from 'lib/shared/types/HTTPMethod';
import { withPrismaClient } from 'middleware/withPrismaClient';
import { withHTTPMethodRequired } from 'middleware/withHTTPMethodRequired';

export default withHTTPMethodRequired(HTTPMethod.POST)(
  withPrismaClient(async (req, res) => {
```

```
    try {
      const { name, description } = req.body;

      // Get our data.
      const gallery = await req.prisma.gallery.create({
        data: {
          name,
          description,
        },
      });

      // Return the data.
      return res.status(200).json(gallery);
    } catch (error) {
      // Later this will log to Sentry, or other error handling service.
      console.log(`---------------- error creating gallery: `, error);
      return res.status(500).end();
    }
  })
);
```

Breakdown:

- Whereas `withPrismaClient` is a function that just wraps a function, `withHTTPMethodRequired` does the following:
  - Takes an `HTTPMethod` as parameter;
  - Returns a function
  - That returned function in turn takes the `handler` as a parameter.

- `withHTTPMethodRequired` doesn't do anything with the request object (`req`); its job is to check the HTTP method. From there it will throw an error if the HTTP method is wrong, or continue on to running the `handler`.

We'll use these middlewares for all API endpoints going forward, and the code for this chapter will reflect the use of them everywhere. Furthermore, if we implement any API code that looks reusable, we'll implement it directly in a middleware like the above.

Building Photo API Endpoints

Let's create the serverless functions that will facilitate the manipulation of photo data.

Adding dependencies

The only package we need is the Cloudinary Node client.

```
yarn add cloudinary
```

Configuration

Make sure you've read the Cloudinary Configuration chapter beforehand, and set up your account.

If you haven't already, you should set the following environment variables in your `.env`:

```
# .env
CLOUDINARY_CLOUD_NAME=abcdef123
CLOUDINARY_API_KEY=ghijkl456
CLOUDINARY_API_SECRET=mnopqr789
```

Building Hollow API Routes

Let's create files for our endpoints that simply take requests and return responses.

Here are the photo-related endpoints we'll need for our app:

- Sign a photo upload request with Cloudinary
  - POST /api/photos/sign
  - (/pages/api/photos/sign.ts)
- Create a photo entry in our database, once it exists at Cloudinary
  - POST /api/galleries/456/photos/create
  - /pages/api/galleries/id/photos/create.ts
- Delete a photo from our database and from Cloudinary
  - DELETE /api/photos/123/delete
  - (/pages/api/photos/id/delete.ts)

Building Client-Side API Functions

Inside `/lib/client/api/Photos.ts`:

- sign: async (galleryId: number)
- create: async (galleryId, publicId, assetUrl, width, height)
- upload: async (file, folder, timestamp, signature)
- delete: async (id)

A Few Files We'll Need

For the sake of keeping our code organized, we'll have a few files for grouping bits of code with different purposes.

/lib/server/Cloudinary.ts

Here we'll import the Cloudinary Node library and configure it. Doing it in a separate file (then importing it from this file) means we won't have to configure it more than once.

Here's the code:

```
// /lib/server/Cloudinary.ts
import cloudinary from 'cloudinary';

const {
  CLOUDINARY_CLOUD_NAME,
  CLOUDINARY_API_KEY,
  CLOUDINARY_API_SECRET,
} = process.env;

cloudinary.v2.config({
  cloud_name: CLOUDINARY_CLOUD_NAME,
  api_key: CLOUDINARY_API_KEY,
  api_secret: CLOUDINARY_API_SECRET,
  secure: true,
});

export default cloudinary;
```

Breakdown:

- We import the Cloudinary node library. (Make sure you've run `yarn add cloudinary` already.)
- We import relevant Cloudinary information from our process.env (which comes from our `.env` file).
- We check that each one of these values exists; otherwise we throw an error and remind ourselves to add them.
- We call the Cloudinary config function to set global configuration settings (docs)
- We export the Cloudinary object now that the configuration has been applied.

Now in our other files, we'll import the above Cloudinary.ts instead of importing the node module directly.

/lib/shared/types/UploadSignatureMetadata.ts

This type will define the structure of the return object from our request signing code.

```
// /lib/shared/types/UploadSignatureMetadata
export type UploadSignatureMetadata = {
  timestamp: number;
```

```
  folder: string;
  signature: string;
};
```

/lib/server/ImageService.ts

Here we'll put functions that interact directly with Cloudinary. Keeping all of these in one location will make it slightly easier if we ever switch to another image hosting service.

```
// /lib/server/ImageService.ts
import cloudinary from 'lib/server/Cloudinary';
import { UploadSignatureMetadata } from 'lib/shared/types/UploadSignatureMetadata';

const {
  CLOUDINARY_UPLOAD_PRESET,
  CLOUDINARY_API_SECRET,
} = process.env;

export function signUploadRequest(
  timestamp: number,
  folder: string
): UploadSignatureMetadata {
  const signature = cloudinary.v2.utils.api_sign_request(
    {
      timestamp,
      upload_preset: CLOUDINARY_UPLOAD_PRESET,
      folder,
    },
    CLOUDINARY_API_SECRET as string
  );

  return { signature, folder, timestamp };
}
```

Breakdown:

- We import the cloudinary package from our own, configured version at `lib/server/Cloudinary`.
- We grab environment variables we'll need from `process.env`.
- The `signUploadRequest` function accepts a timestamp and a folder path.
  - These params are passed, along with our API secret to Cloudinary's `api_sign_request` function (source) which returns a request signature string.
  - Our function returns a `UploadSignatureMetadata` payload which includes the signature, the folder and the timestamp.

/lib/server/Cloudinary.ts

Here we'll have utility functions related specifically to galleries.

Right now, we'll add a single function to it, `galleryFolderPath`. This will calculate a unique url for each image that gets uploaded.

```
const { CLOUDINARY_BASE_PATH, APP_ENV } = process.env;

export const galleryFolderPath = (galleryId: number) =>
  `${CLOUDINARY_BASE_PATH}/${APP_ENV}/galleries/${galleryId}/photos`;
```

Breakdown:

- The folder structure we maintain on Cloudinary is very important. Primarily we want to keep everything in specific folders that won't overwrite each other.
- The first dimension we want to separate is the base path. `CLOUDINARY_BASE_PATH` in your environment should be set to `"citjs-photo-app"`, which means all of this app's uploaded photos will be stored under this path. This allows us to create another folder at this level to use for any other application we might use this Cloudinary account for.
- TODO - The second part of the path is `APP_ENV` which will be development, production or preview. This will allow us to upload and manage our images separately across our localhost, preview builds and production.
- Within each environment, we'll have a `galleries` folder, which will have numeric subfolders that represent a gallery's unique ID. Within there we'll have the photos for each gallery respectively.

/lib/server/PhotoUtils.ts

Here we'll have utility functions related specifically to photos. We don't need any functions just yet, so we'll export a blank object.

```
// /lib/server/PhotoUtils.ts
export default {};
```

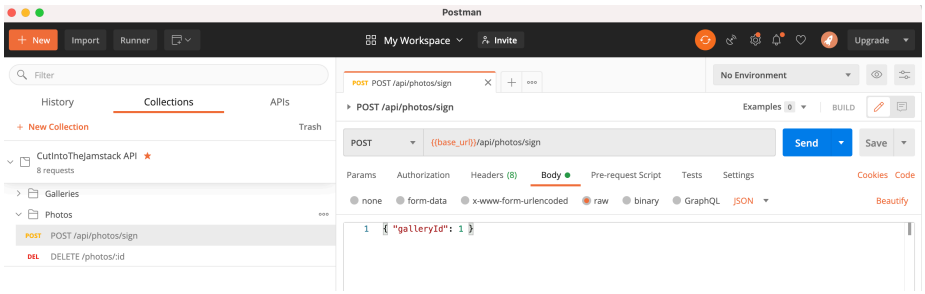
Building the Image Signing Endpoint

Image signing will work as follows:

- Our app will make a POST request to `/api/photos/sign`. The BODY of the request will have a JSON payload that looks like `{ "galleryId": 123 }`.
- Our API endpoint located at `/pages/api/photos/sign.ts` will call a function we create called `signUploadRequest`. '-signUploadRequest`will take as parameters the current time, and the path to where the image should be stored in our Cloudinary media library.` -signUploadRequest`calls Cloudinary's API function api_sign_request`

which will return a unique signature for this request and return it. –signUploadRequest‘ will then return the signature, folder and timestamp of the request for the client to use and do the actual file upload with.

In Postman, here's what the stubbed request will look like:



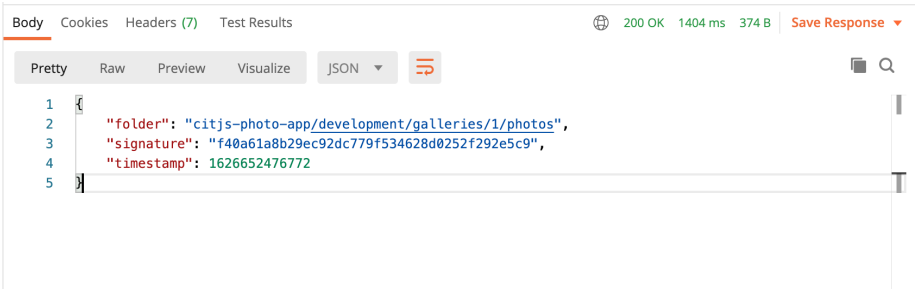
And here's the API endpoint:

```
// /pages/api/photos/sign.ts
import { withHTTPMethodRequired } from 'middleware/withHTTPMethodRequired';
import { HTTPMethod } from 'lib/shared/types/HTTPMethod';
import { galleryFolderPath } from 'lib/server/GalleryUtils';
import { signUploadRequest } from 'lib/server/ImageService';

export default withHTTPMethodRequired(HTTPMethod.POST)(
  async (req, res) => {
    try {
      const { galleryId } = req.body;
      const timeNow = new Date().getTime();
      const targetFolder = galleryFolderPath(galleryId);
      const {
        signature,
        folder,
        timestamp,
      } = signUploadRequest(timeNow, targetFolder);
      res.statusCode = 200;
      res.json({ folder, signature, timestamp });
    } catch (error) {
      res.statusCode = 500;
      // TODO: add Sentry.captureException() here.
      res.end();
    }
  }
);
```

Breakdown:

- We wrap the route function with the withHTTPMethodRequired middleware helper, so that we can enforce a POST request.
- Inside the main function, we wrap our main code inside a try...catch block to catch any errors and send a 500 error if so.
- We grab the galleryId from the request body, calculate the current time, and get the folder path for uploading the file using the `galleryFolderPath` function we created before.
- Using the `signUploadRequest` function we created, we generate the request signature, and return it to the client along with the folder and timestamp.

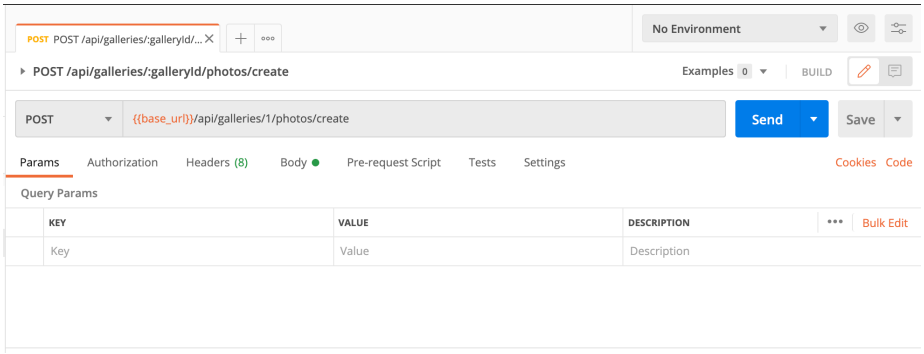
Now if you rerun the same Postman request for `POST /api/photos/sign`, you should now see a response like this:



Building the Photo Creation Endpoint

This API route will take photo information and insert it into the database.

Here's what the Postman request will look like:



Here's the code:

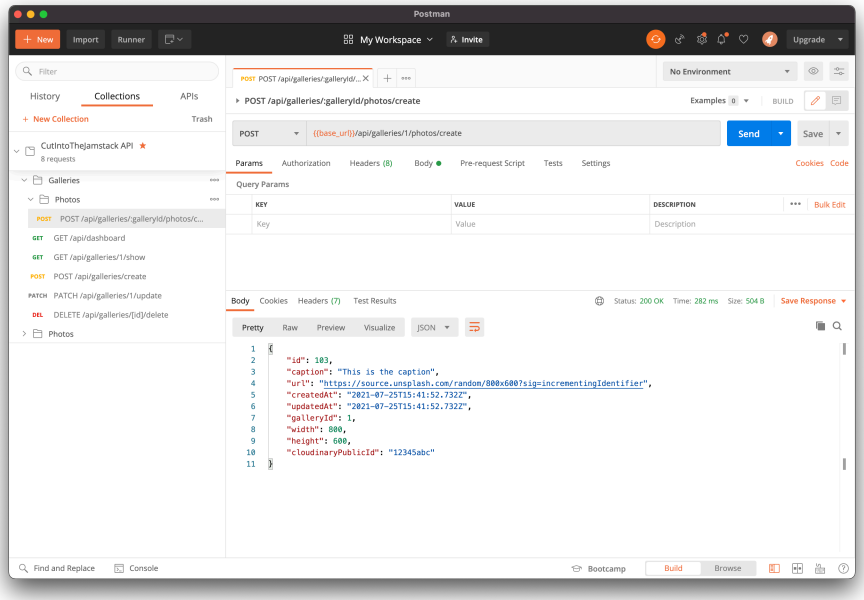
```
// /pages/api/galleries/[id]/photos/create.ts

import { withHTTPMethodRequired } from 'middleware/withHTTPMethodRequired';
```

```
import { withPrismaClient } from 'middleware/withPrismaClient';
import { HTTPMethod } from 'lib/shared/types/HTTPMethod';

export default withHTTPMethodRequired(HTTPMethod.POST)(
  withPrismaClient(async (req, res) => {
    const id: number = parseInt(req.query.id);

    if (!id) {
      return res.status(400);
    }

    try {
      const {
        caption,
        url,
        width,
        height,
        cloudinaryPublicId,
      } = req.body;

      const image = await req.prisma.photo.create({
        data: {
          caption,
          url,
          width,
          height,
          cloudinaryPublicId,
          gallery: {
            connect: { id },
          },
        },
      });

      res.statusCode = 200;
      res.json(image);
    } catch (error) {
      // TODO: add Sentry.captureException() here.
      console.log(
        `---------------- error creating photo: `,
        error
      );
      res.statusCode = 500;
      res.end();
    }
  })
);
```

Breakdown:

- With our middleware, we require that the request be a POST, and also bring in the Prisma client.
- We check that the gallery's `id` is defined in the route. If it's not, we return a HTTP `400` (Bad Request) code. If it's set, we set it into the `id` variable.
- We grab relevant attributes for the gallery from the request body.
- We use `prisma.photo.create()`(docs) to insert the photo meta-data into the database. Using the Photo model's `gallery` association (defined in your `schema.prisma` file), we associate the photo to the gallery via the `connect` parameter (docs).
- If the photo is inserted correctly, we return the photo object as JSON.
- If anything goes wrong, we return a HTTP`500` error code, and log the error.

Example of the successful POST in Postman:



Connecting Image Deletion

The API route for deleting an image must do three things:

1. Find the image in our database;
2. Delete the image from Cloudinary (using the `cloudinaryPublicId` from step 1; and

3. Delete the image from our database.

First we'll need another Cloudinary helper added to our `ImageService`:

```
// /lib/server/ImageService.ts

// .. imports ...

export async function deleteImage(
  id: string
): Promise<any> {
  const response = await cloudinary.v2.uploader.destroy(
    id,
    {
      resource_type: 'image',
    }
  );

  // If delete wasn't successful in cloudinary, stop here
  //  and pass along the error message.
  if (!response.result || response.result !== 'ok') {
    throw response;
  }

  return response;
}
// ... other functions ...
```

This one is pretty straightforward; it takes the unique id of the Cloudinary asset and calls the `destroy` method in the API (docs). It also checks the `result` property of the response object to see that it not only exists but is set to the string `'ok'`, which is how the Cloudinary APIs indicate a successful call.

Here's the photo deletion API endpoint:

```
// /pages/api/photos/[id]/delete.ts

import { deleteImage } from 'lib/server/ImageService';
// ... other imports ...

export default withHTTPMethodRequired(HTTPMethod.DELETE)(
  withPrismaClient(async (req, res) => {
    const { prisma } = req;

    const {
      query: { id: photoIdFromRoute },
    } = req;
```

```
    const photoId = parseInt(photoIdFromRoute);

    const photo = await req.prisma.photo.findUnique({
      where: { id: photoId },
    });

    try {
      // Delete the image from Cloudinary.
      await deleteImage(photo.cloudinaryPublicId);

      // If it was, delete the reference in our database.
      await req.prisma.photo.delete({
        where: { id: photoId },
      });

      // Return HTTP 204 (successful delete), and no content.
      return res.status(204).end();
    } catch (error) {
      // TODO: Later this will log to Sentry, or other error handling service.
      console.log(
        `---------------- error deleting photo: `,
        error
      );
      return res.status(500).end();
    }
  })
);
```

Breakdown:

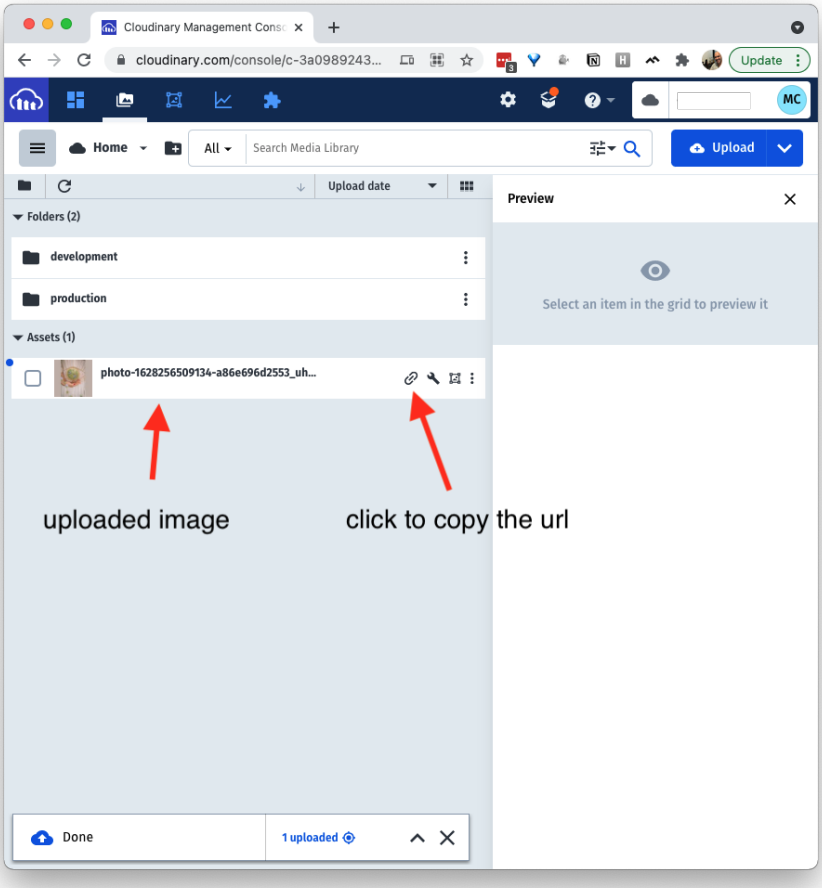
- We wrap the main async function with the same middlewares we've been using.
- We get the photoId from the dynamic route (the folder named `[id]` enables this).
- We call `deleteImage` from our ImageService, which we know calls the cloudinary API to delete the image from their service.
- If that succeeds, we call `prisma.photo.delete`with the `photoId` passed in from the route.
- If the `deleteImage` call fails, we'll return a `500` error and the database entry won't be touched. We'll only delete the entry from our database if it's gone from Cloudinary.

Testing Photo Deletion

Testing deletion requires three steps: 1. Upload an image manually to Cloudinary via the Media Console; 2. Add that image's public ID

to a Photo entry in your local database; 3. Run your API request in Postman.

Upload Image to Media Library In the Cloudinary **Media Library** (tab up top), you can drag and drop an image into the main area for testing, then click the link icon to get its direct url.

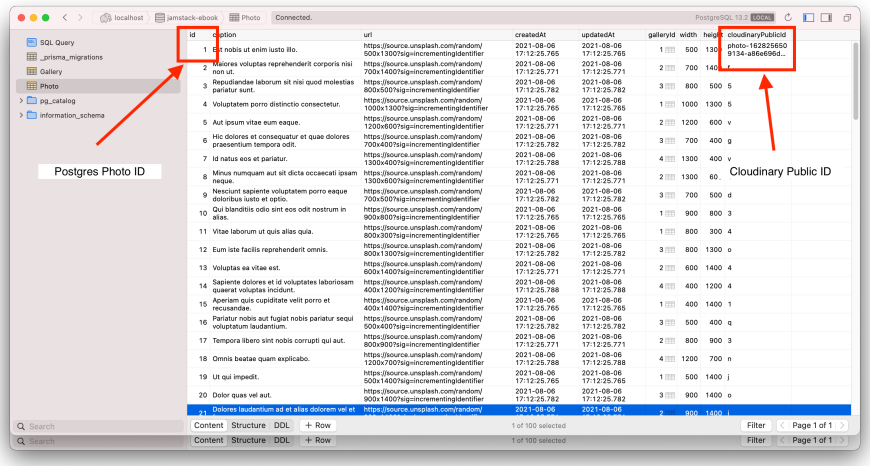


The url will look something like this:

https://res.cloudinary.com/dmaydrrcg/image/upload/v1628272685/**photo-1628256509134-a86e696d2553_uheocq**.jpg

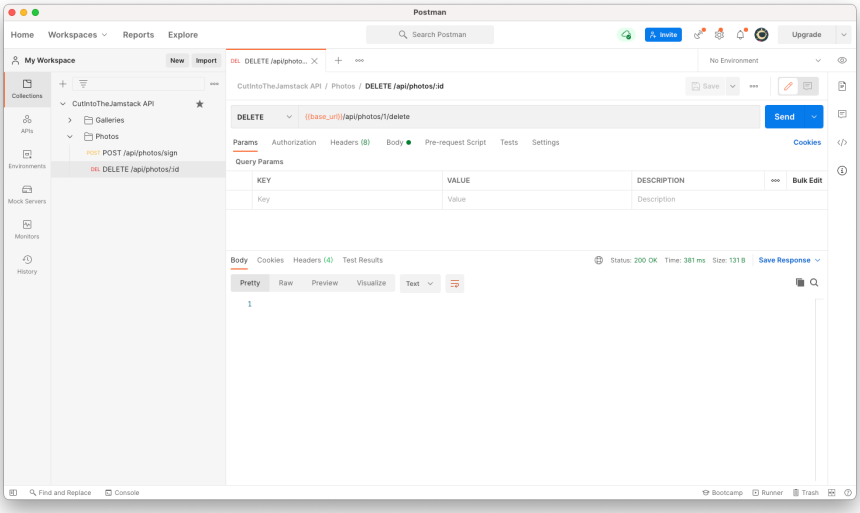
The bolded part is the Cloudinary `public_id`, which we'll use to delete it through the Cloudinary API.

Add `public_id` to Postgres In Postico, edit any of the seeded entires in the `Photo` table, and paste the public ID you got in the last step into any photo's `cloudinaryPublicId` field, and hit save.



Now this photo in our database will be associated with the photo in cloudinary. Grab the Postgres Photo ID for testing the API endpoint in the next step.

Test deletion in Postman Using the Photo ID, create the Postman endpoint for photo deletion as below. When you submit the request, you should see the photo gone from your Cloudinary media library, and the row gone from the Photos table in Postgres.



Adding the Front-End API Layer

Now that we've got all our photo-related API routes defined, we need to write some client-side functions to talk to them, as well as one to do the actual uploading to Cloudinary.

Prerequisite: Expose Cloudinary Environment Variables

The `upload()` function we'll write below will require three Cloudinary values that our .env should have already: `CLOUDINARY_API_KEY` , `CLOUDINARY_CLOUD_NAME` and `CLOUDINARY_UPLOAD_PRESET`.

Next.js won't expose these to the front-end by default (docs), so we'll need to make them available by adding them to `next.config.js` (docs), below. We'll also add cloudinary's domain to the list of allowed domains for images, so that our images will show up.

```
module.exports = {
  env: {
    CLOUDINARY_CLOUD_NAME:
      process.env.CLOUDINARY_CLOUD_NAME,
    CLOUDINARY_API_KEY: process.env.CLOUDINARY_API_KEY,
    CLOUDINARY_UPLOAD_PRESET:
      process.env.CLOUDINARY_UPLOAD_PRESET,
  },
  reactStrictMode: true,
  images: {
    domains: ['source.unsplash.com', 'res.cloudinary.com'],
```

```
  },
};
```

Restart your local server for the changes to take effect.

Building the API functions

We'll need 4 functions: - `sign()`: (`POST /api/photos/sign`) - `create()`: (`POST /api/galleries/[id]/photos/create`) - `upload()`: This will accept a request signature and file upload data, and execute the upload. - `delete()`: (`DELETE /api/photos/[id]/delete`)

Create `/lib/client/api/Photos.ts` and add the following code:

```
//
// /lib/client/api/Photos.ts
//
const defaultJSONHeaders = {
  Accept: 'application/json',
  'Content-Type': 'application/json',
};

export const sign = async (
  galleryId: number
): Promise<Record<string, any>> => {
  const response = await fetch('/api/photos/sign', {
    method: 'POST',
    body: JSON.stringify({ galleryId }),
    headers: defaultJSONHeaders,
  });

  const json = await response.json();

  if (!response.ok) {
    throw json;
  }

  return json;
};

export const create = async (
  galleryId,
  publicId,
  assetUrl,
  width,
  height
): Promise<Record<string, any>> => {
```

```
  const response = await fetch(
    `/api/galleries/${galleryId}/photos/create`,
    {
      method: 'POST',
      headers: defaultJSONHeaders,
      body: JSON.stringify({
        cloudinaryPublicId: publicId,
        url: assetUrl,
        width,
        height,
      }),
    }
  );

  const json = await response.json();

  if (!response.ok) {
    throw json;
  }

  return json;
};

export const upload = async (
  file: File,
  folder: string,
  timestamp: string | Blob,
  signature: string
): Promise<Record<string, any>> => {
  const formData = new FormData();

  formData.append('file', file);
  formData.append('folder', folder);
  formData.append('signature', signature);
  formData.append('timestamp', timestamp);
  formData.append(
    'api_key',
    process.env.CLOUDINARY_API_KEY as string
  );
  formData.append(
    'upload_preset',
    process.env.CLOUDINARY_UPLOAD_PRESET as string
  );

  const response = await fetch(url, {
    method: 'POST',
```

```
    body: formData,
  });

  const json = await response.json();

  if (!response.ok) {
    throw json;
  }

  return json;
};

export const destroy = async (id): Promise<boolean> => {
  const response = await fetch(`/api/photos/${id}/delete`, {
    method: 'DELETE',
  });

  if (!response.ok) {
    throw response;
  }

  return true;
};
```

This looks like a lot of code, but it's not that complicated. Here's the breakdown:

- Every function (except for `upload()` and `destroy()`) has a similar pattern:
  - The function has a return type of `Promise<<Record<string, any>>`. This means they're async functions (since they return a `Promise`), and the `Promise` resolves to a `Record<string, any>`, which is just a TypeScript way of saying a json object that has strings as keys and any type of value.
  - The function makes a `fetch` call to an API endpoint. The parameters will differ for each function. All will have the same JSON headers (since they'll all receive and send JSON data where appropriate).
  - If the `fetch` is successful (indicated by the `response.ok` flag), return the json object. Otherwise throw the returned object as an error.
- The `destroy()` function works similarly to the others described above, but since there's no object to be returned, we just return a `true` to indicate success.
- The `upload()` function is the larger outlier. Broken down:
  - We build the `url` for uploading, which comes from the Cloudinary REST API (docs)

- Since this is a file upload, we'll put our data into a JavaScript `FormData` object (docs).
- We put the required params into the `FormData` object using the `append` method (docs)
- Two of the params come from our environment config (`CLOUDINARY_API_KEY` and `CLOUDINARY_UPLOAD_PRESET`); the others will come from the `sign()` call we'll make beforehand.
- The rest of the `upload()` function is just submitting the `FormData`via `POST` to the cloudinary URL we built. Then we check and return the response as in all of the other functions in this file.

We don't have an easy way to test these API functions without adding a UI, but initially you can check for syntax errors by running `yarn tsc` to have TypeScript check for any problems.

Transforming iOS-Specific File Formats

iOS devices will upload their own .heic file format. We can save these into our database and into Cloudinary. But when we try to view them in a web browser this will fail, since browsers don't support them.

For this we'll add a simple workaround that leverages a great Cloudinary feature that allows us to convert an image to any format we want: we just change the file extension (docs).

To do this when trying to render photos, we'll add a simple transformation function to our gallery show endpoint:

```
import { withPrismaClient } from 'middleware/withPrismaClient';
import { GalleryWithPhotos } from 'lib/shared/types/GalleryWithPhotos';

export const transform = (gallery: GalleryWithPhotos) => ({
  ...gallery,
  photos: gallery.photos.map((photo) => ({
    ...photo,
    url: photo.url.replace(/\.heic/, '.jpg'),
  })),
});


export default withPrismaClient(async (req, res) => {
  const id: number = parseInt(req.query.id);
  if (!id) {
    return res.status(403);
  }
```

```
  try {
    // Get our data.
    const galleryFromDb =
      await req.prisma.gallery.findUnique({
        where: { id },
        include: {
          photos: true,
        },
      });

    const gallery = transform(galleryFromDb);

    // Return the data.
    return res.status(200).json(gallery);
  } catch (error) {
    // Later this will log to Sentry, or other error handling service.
    console.log(
      `---------------- error retrieving gallery: `,
      error
    );
    return res.status(500).end();
  }
});
```

We're defining a new type here as well. The reason for it will become apparent in the breakdown below. Here's there code:

```
// /lib/shared/types/GalleryWithPhotos.ts

import { Prisma } from '@prisma/client';

const galleryWithPhotos =
  Prisma.validator<Prisma.GalleryArgs>()({
    include: {
      photos: true,
    },
  });

export type GalleryWithPhotos = Prisma.GalleryGetPayload<
  typeof galleryWithPhotos
>;
```

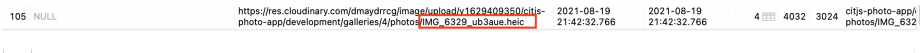
Breakdown:

- We add an `import` for the `GalleryWithPhotos` type that is returned from the `prisma.gallery.findUnique()` call.
  - We need this type since we're using the `include` option in `findUnique()`. The `Gallery` type Prisma defines for us doesn't

have a `photos` parameter, so TypeScript will give us an error if we use a `Gallery` type. Prisma's docs give us the methodology for making a type that includes the `photos` array (docs).

- We create a `transform()` function that will take the payload from `findUnique()`, loop through the `photos` in the returned gallery, and replace all `.heic` images with a `.jpg` extension that all browsers can view.
- In the main route function, we call the `transform` function on the `findUnique()` result, and return it.

If you re-run your Postman request for `GET /api/galleries/[id]/show` and compare the url for an uploaded image with the same image in the database, you'll see the difference in file extensions. Examples from Postman and Postico are below.




```
{
    "id": 105,
    "caption": null,
    "url": "https://res.cloudinary.com/dmaydrrcg/image/upload/v1629409350/citjs-photo-app/development/galleries/4/photos/
        IMG_6329_ub3aue.jpg",
    "createdAt": "2021-08-19T21:42:32.766Z",
    "updatedAt": "2021-08-19T21:42:32.766Z",
    "galleryId": 4,
    "width": 4032,
    "height": 3024,
    "cloudinaryPublicId": "citjs-photo-app/development/galleries/4/photos/IMG_6329_ub3aue"
},
```


Adding Single-File Image Uploads to the Gallery Editor

How it will work

The user will trigger a system file selection modal when clicking an "Add Photos" button we'll create. That button will simply trigger the `change` event of an HTML file input.

Once the user has selected an image, an event handler on our file input will run, making the necessary API route calls to sign, upload and so on.

When the upload is complete, we'll tell the gallery to update using the `mutate()` function from the `useSWR` hook.

Adding a header, button and hidden file input field

To allow users to start the image upload process, we'll add an "Add Photo" button. First make sure you have the `react-icons` package installed:

```
yarn add react-icons
```

Then make the following additions to `pages/galleries/[id].tsx`:

```
// /pages/galleries/[id].tsx

// ... other imports ...
import {
  Box,
  Button,
  Flex,
  Grid,
  Heading,
  VisuallyHidden,
} from '@chakra-ui/react';
import { FaPlus } from 'react-icons/fa';

export default function GalleryShowPage() {
  const router = useRouter();
  const galleryId = Number(router.query.id);
  const fileInput = useRef<HTMLInputElement>(null);

  const { data: gallery, mutate } = useSWR(
    `/api/galleries/${galleryId}/show`
  );

  // ... more existing code ...

  // Stub event handlers right before the return() statement.
  const handleFileChange = (event) => {
    console.log(`File changed.`, event.target.files);
  };

  const handleAddPhotoClick = (event) => {
    event.preventDefault();

    // Simulate click on the hidden input field.
    fileInput?.current?.click();
  };

  return (
    <>
      {/* Start of new code */}
      <Flex
        justify="space-between"
        align="center"
        my={5}
        px={4}
      >
        <Heading mb={{ base: 2, md: 0 }}>
```

```
        {gallery?.name}
      </Heading>

      <VisuallyHidden>
        <input
          type="file"
          accept="image/*,.heic"
          onChange={handleFileChange}
          ref={fileInput}
        />
      </VisuallyHidden>

      <Button
        leftIcon={<FaPlus />}
        onClick={handleAddPhotoClick}
      >
        Add Photo
      </Button>
    </Flex>
    {/* End of new code */}
    <Grid
      templateColumns={{
        base: '1fr 1fr',
        md: '1fr 1fr 1fr',
        lg: '1fr 1fr 1fr 1fr',
      }}
    >


      {/* ... the rest of the file. */}
```

Breakdown:

- We create the `fileInput` ref using the React `useRef` hook (docs), since we'll need to directly access the html file input field we'll be using to upload the file.
  - A ref in React is like a direct link to the HTML element (docs).
  - We tell TypeScript that the `useRef` hook will return a value of type `HTMLInputElement`(docs).
- In the response from our `useSWR` hook call, we extract the `mutate` function (docs). We'll use this later to explicitly tell the list of photos to refresh after we've uploaded a new one.
- The Chakra-UI `Flex` component (docs) is a convenient way of using CSS FlexBox. With the `justify` and `align` props set as they are, the gallery name and the button will be vertically centered and horizontally spaced apart.
- We add the gallery title on the left side of the page, using a

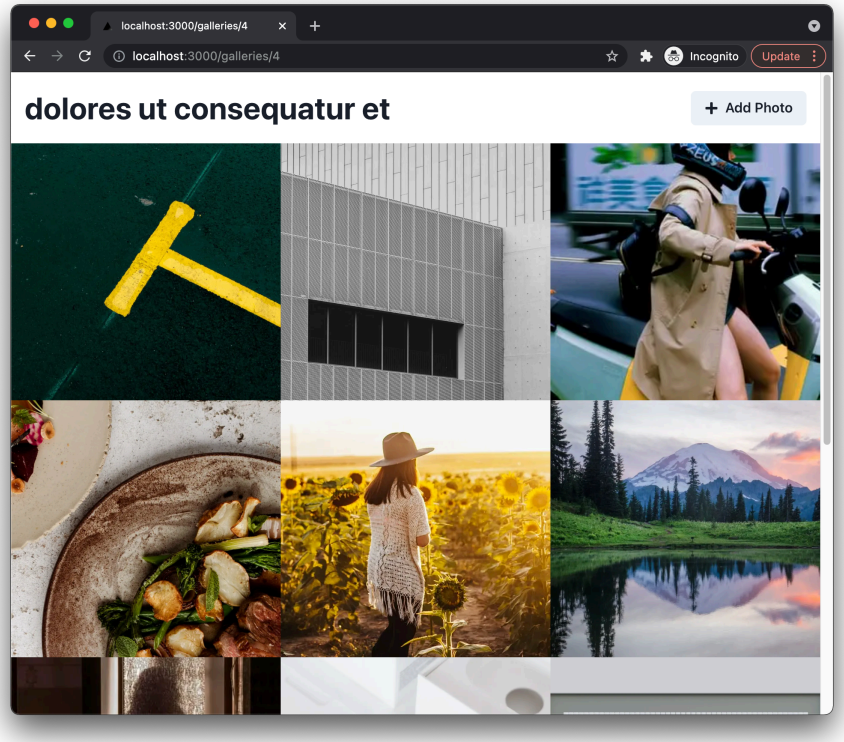
Chakra `Heading`(docs). Inside it we output the gallery's name if it exists (and we check this using the TypeScript optional chaining operator (the question mark; docs)

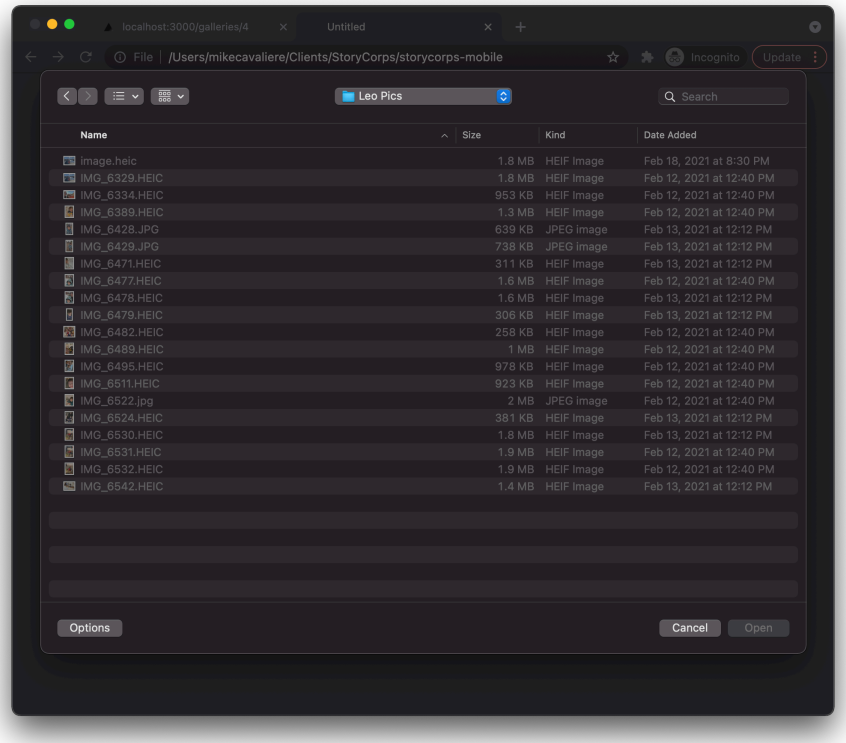
- We put the html file input field next, wrapped in the Chakra `VisuallyHidden` element (docs) which makes it invisible in browsers.
  - `accept="image/*,.heic"` tells the browser to accept any files whose contents are of a MIME type starting with `image/`. I.e., any type of image files. We also specifically add the `.heic` extension here, which is a file type created by the iOS camera, and browsers may not recognize it as image content.
  - `onChange={handleFileInputChange}`just connects the event handler function, so that when the input changes we'll log the selected files to the console.
  - `ref={fileInput}`causes the `fileInput` variable to be set as soon as the DOM element exists.
- We add a Chakra-UI `Button` with a plus icon (docs), and connect the `click` event to the `handleAddPhotoClick` function.
  - `event.preventDefault()` prevents the default button behavior when clicked; this will prevent the browser from making the page jump.
  - In the line`fileInput?.current?.click()`we use the TypeScript optional chaining operator again. Since `fileInput` is `null` by default, and `useRef`will set `fileInput.current` to reference the DOM element, both `fileInput` and `fileInput.current`have to exist before we can call the `click()` function, so we check for both.
  - Calling the `click()` function causes the file input to behave as if it were actually clicked; it causes the browser to open the system file upload dialog.

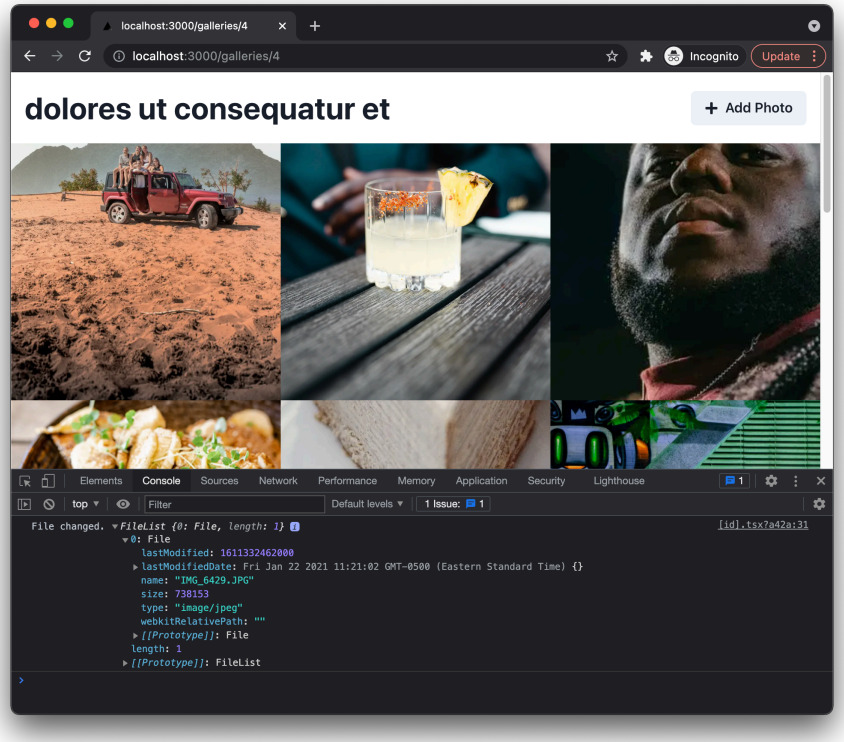
If you open the debugger console in your browser, you'll see that clicking the add button will open a file upload dialog on a desktop browser, and the iOS or Android photo picker on a mobile device. When you select an image, it will log the corresponding JavaScript `FileList` object containing your image's metadata. Cool!

Try this on a few different browsers, including a mobile device (for the latter, you'll have to commit and push to deploy to your Vercel preview environment, or view your local machine via IP address if you know how to do that).

The screenshots below show the behavior in Google Chrome on a Mac.

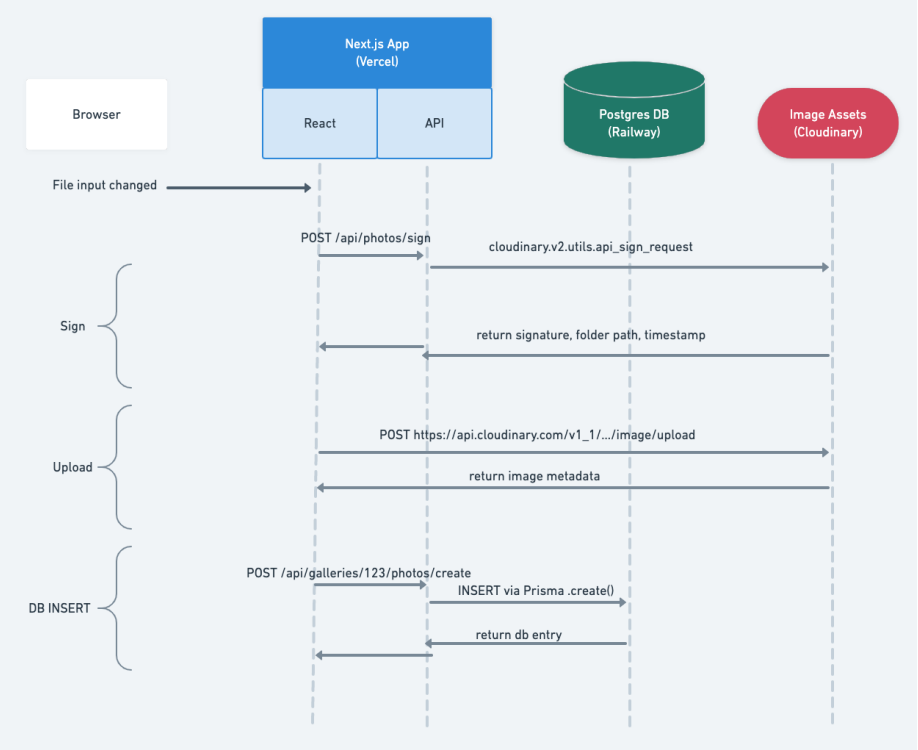

localhost:3000/galleries/4
localhost:3000/galleries/4
Incognito
Update
dolores ut consequatur et
+ Add Photo


localhost:3000/galleries/4
Untitled
File | /Users/mikecavaliere/Clients/StoryCorps/storycorps-mobile
Incognito
Update
Leo Pics
Search
Name
Size
Kind
Date Added
image.heic 1.8 MB HEIF Image Feb 18, 2021 at 8:30 PM
IMG_6329.HEIC 1.8 MB HEIF Image Feb 12, 2021 at 12:40 PM
IMG_6334.HEIC 953 KB HEIF Image Feb 12, 2021 at 12:40 PM
IMG_6389.HEIC 1.3 MB HEIF Image Feb 12, 2021 at 12:40 PM
IMG_6428.JPG 639 KB JPEG image Feb 13, 2021 at 12:12 PM
IMG_6429.JPG 738 KB JPEG image Feb 13, 2021 at 12:12 PM
IMG_6471.HEIC 311 KB HEIF Image Feb 13, 2021 at 12:12 PM
IMG_6477.HEIC 1.6 MB HEIF Image Feb 12, 2021 at 12:40 PM
IMG_6478.HEIC 1.6 MB HEIF Image Feb 13, 2021 at 12:12 PM
IMG_6479.HEIC 306 KB HEIF Image Feb 13, 2021 at 12:12 PM
IMG_6482.HEIC 258 KB HEIF Image Feb 12, 2021 at 12:40 PM
IMG_6489.HEIC 1 MB HEIF Image Feb 12, 2021 at 12:40 PM
IMG_6495.HEIC 978 KB HEIF Image Feb 12, 2021 at 12:40 PM
IMG_6511.HEIC 923 KB HEIF Image Feb 12, 2021 at 12:40 PM
IMG_6522.jpg 2 MB JPEG image Feb 12, 2021 at 12:40 PM
IMG_6524.HEIC 381 KB HEIF Image Feb 13, 2021 at 12:12 PM
IMG_6530.HEIC 1.8 MB HEIF Image Feb 13, 2021 at 12:12 PM
IMG_6531.HEIC 1.9 MB HEIF Image Feb 12, 2021 at 12:40 PM
IMG_6532.HEIC 1.9 MB HEIF Image Feb 12, 2021 at 12:40 PM
IMG_6542.HEIC 1.4 MB HEIF Image Feb 13, 2021 at 12:12 PM
Options
Cancel
Open



Connecting the API requests

Here's where we make the magic happens. If you remember from our diagram at the start of this chapter, the client side (React) part of the application has to sign the request, then upload, then create the database record.



Here's what the full file looks like with all the additions:

```
import { useRef } from 'react';
import useSWR from 'swr';
import { useRouter } from 'next/router';
import Image from 'next/image';
import {
  Box,
  Button,
  Flex,
  Grid,
  Heading,
  VisuallyHidden,
} from '@chakra-ui/react';
import { FaPlus } from 'react-icons/fa';

import {
  create,
  sign,
  upload,
} from 'lib/client/api/Photos';
```

```
const uploadFileToGallery = async (
  galleryId: number,
  file: File
) => {
  try {
    const { folder, timestamp, signature } = await sign(
      galleryId
    );

    const { secure_url, public_id, width, height } =
      await upload(file, folder, timestamp, signature);

    await create(
      galleryId,
      public_id,
      secure_url,
      width,
      height
    );
  } catch (error) {
    // TODO: show some error messaging.
    console.error(error);
  }
};

export default function GalleryShowPage() {
  const router = useRouter();
  const galleryId = Number(router.query.id);
  const fileInput = useRef<HTMLInputElement>(null);

  const { data: gallery, mutate } = useSWR(
    `/api/galleries/${galleryId}/show`
  );

  if (!gallery) {
    return null;
  }

  const rowHeight = { base: 200, md: 300 };

  const handleFileChange = async (event) => {
    const file = event.target.files[0];

    await uploadFileToGallery(galleryId, file);
    mutate();
  };
```

```
const handleAddPhotoClick = (event) => {
  event.preventDefault();

  // Simulate click on the hidden input field.
  fileInput?.current?.click();
};

return (
  <>
    <Flex
      justify="space-between"
      align="center"
      my={5}
      px={4}
    >
      <Heading mb={{ base: 2, md: 0 }}>
        {gallery?.name}
      </Heading>

      <VisuallyHidden>
        <input
          type="file"
          multiple
          accept="image/*,.heic"
          onChange={handleFileChange}
          ref={fileInput}
        />
      </VisuallyHidden>

      <Button
        leftIcon={<FaPlus />}
        onClick={handleAddPhotoClick}
      >
        Add Photo
      </Button>
    </Flex>

    <Grid
      templateColumns={{
        base: '1fr 1fr',
        md: '1fr 1fr 1fr',
        lg: '1fr 1fr 1fr 1fr',
      }}
    >
      {gallery.photos.map(({ id, url, caption }) => {
```

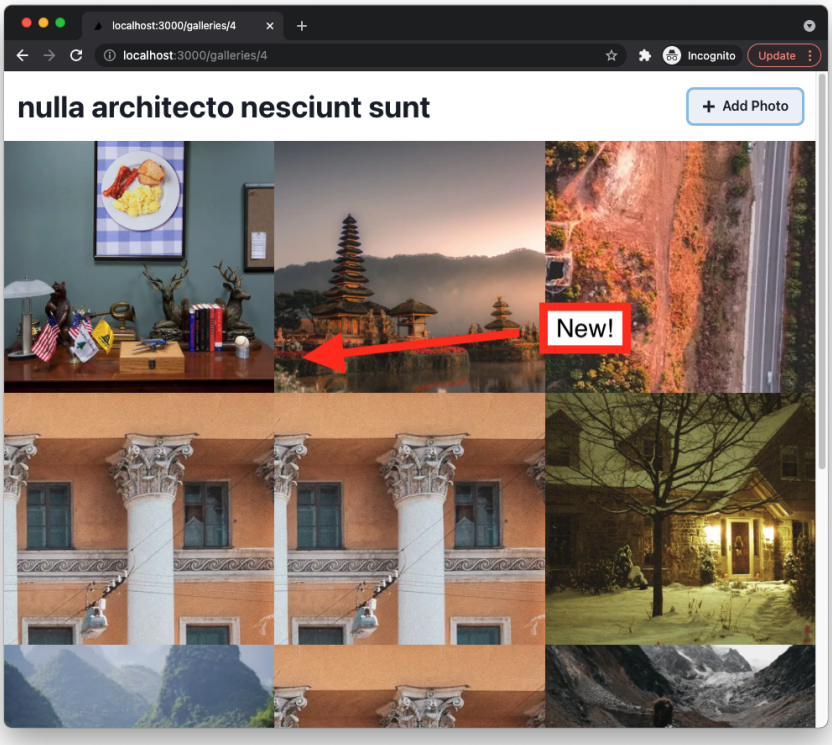
```
          return (
            <Box height={rowHeight} key={id} pos="relative">
              <Image
                key={id}
                src={url}
                layout="fill"
                objectFit="cover"
                alt={
                  caption || 'An image from this gallery.'
                }
              />
            </Box>
          );
        })}
      </Grid>
    </>
  );
}
```

Breakdown:

- Up top, we add an import for the `create`, `sign` and `upload` functions from the photo client API lib we created.
- We define the `uploadFileToGallery` function, which will do all of the heavy lifting.
  - It accepts a `galleryId` and a JavaScript `File` object (docs) as parameters.
  - First it calls the `sign()` function, which gets a request signature from Cloudinary and returns the `folder` path, `timestamp` and unique encrypted `signature`.
  - We pass these values along with the `file` object to the `upload()` function, which POSTs the file and other relevant into to the Cloudinary upload url. Upon success, it returns file metadata.
  - We take the file metadata we care about (the image's `secure_url`, unique cloudinary `public_id` and dimensions) and send them to our `create()` function to add it to our database.
- Now that we've got a function that does all the work, we've got to call it. Back down in the `handleFileChange` function:
  - We get the first `file` object from the event's `target` property (aka, the DOM element for the `input type="file"`).
  - We call `uploadFileToGallery()` to do the work.
  - After the upload is done, we call the `mutate()` function from the `useSWR` hook. This tells SWR that we're invalidating the cache and need to re-fetch the list of images. This triggers a refetch of the API call, and in turn a refresh of the photo

gallery.

We did it! If you got all the above working, you should be able to upload an image, and see it added at the top left.



Adding Photo Deletion to the Gallery Editor

Moving code into the PhotoTile component

The `Box` that renders a photo is pretty important to our app; it's a piece of UI that is central to the core use case of JamShots: viewing photos. We'll expand it over time, so it's definitely worth breaking out into its own component. It'll also make reusability easier, since later we'll have separate 'editor' and 'viewer' versions of the screen that will show the element a little differently, and have it behave a little differently.

Let's move the code that renders a single photo tile into a component file:

```
// /components/PhotoTile/index.tsx
```

```
import { Box, BoxProps } from '@chakra-ui/react';
import Image from 'next/image';

export type PhotoTileProps = {
  alt: string;
  height: BoxProps['height'];
  src: string;
};

export const PhotoTile = ({
  height,
  src,
  alt = '',
}: PhotoTileProps) => (
  <Box height={height} pos="relative">
    <Image
      src={src}
      layout="fill"
      objectFit="cover"
      alt={alt}
    />
  </Box>
);
```

The change to our gallery file is fairly straightforward:

```
// /pages/galleries/[id].tsx

// ...other imports...
// Note that we can remove Box from the chakra-ui
//  import, since we don't need it anymore.

import { PhotoTile } from 'components/PhotoTile';

// ... other code ...

        {gallery.photos.map(({ id, url, caption }) => (
          <PhotoTile
            alt={caption || ''}
            height={rowHeight}
            key={id}
            src={url}
          />
        ))}

// ... the rest of the file.
```

Breakdown:

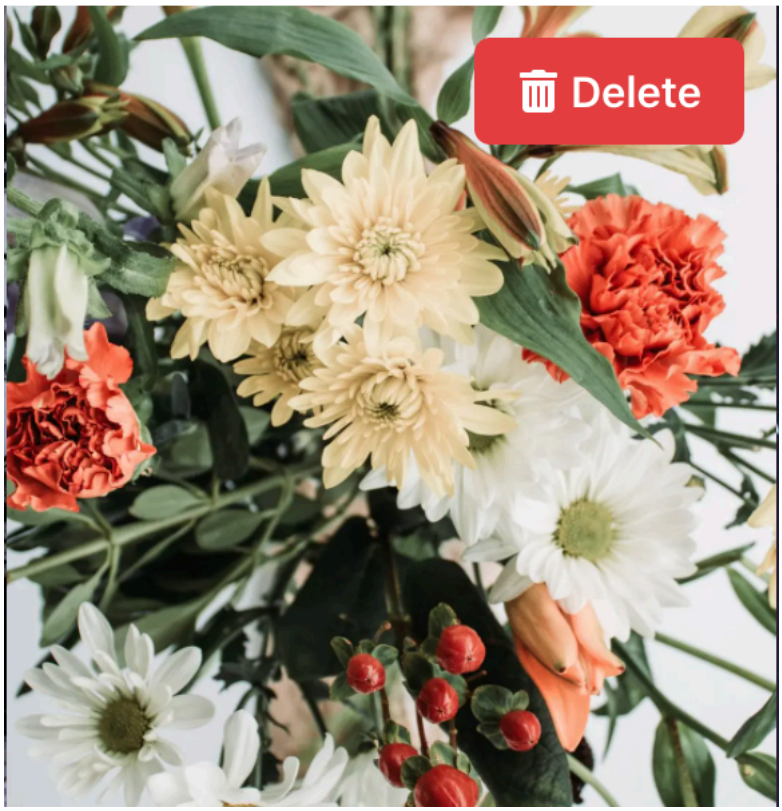
- The containing `Box` element constitutes one atomic photo tile element, so that's what we move into the new component file.
- Anything that might change becomes a prop. That includes image `alt` text, `src` and `height` for now.
- Defining the props is straightforward; it's notable that since `height` can accept any type of Chakra-IU props, it can be a single number, or an object as it is currently. So we pull this specific TypeScript type from Chakra-UI (`BoxProps['height']`).
- After that, we just import the new component in the gallery page, and replace the old code with it.

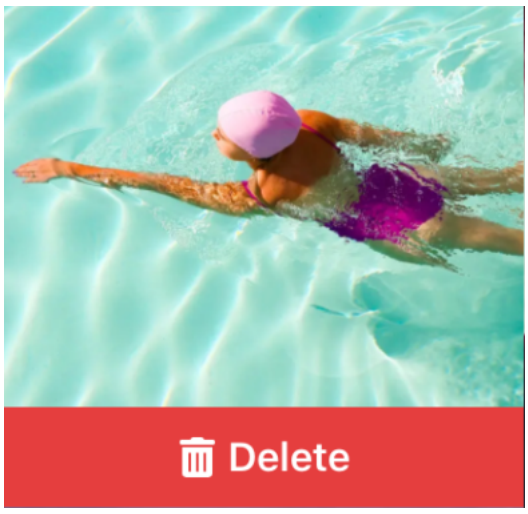
Adding UI for deleting on mobile and desktop

We're going to need a Delete button that will show or hide above an image either when a photo is hovered over (on a desktop browser) or when the image is tapped.

We also want to add a simple animation when showing the button. Since Chakra-UI comes with the Framer Motion library, we can utilize that (it's also super powerful for wild animations, so we can add more impressive ones if we choose to later on).

Here's what the button will look like on desktop and mobile:





Let's define the `PhotoTileDeleteButton` component that we will use inside our `PhotoTile`:

```
// /components/PhotoTileDeleteButton/index.tsx
```

```
import { SyntheticEvent } from 'react';
import { Button, Icon, Text } from '@chakra-ui/react';
import { Photo } from '@prisma/client';
import { FaTrashAlt } from 'react-icons/fa';

export type PhotoTileDeleteButtonProps = {
  photo: Photo;
  onDeleteButtonClick: (
    event: SyntheticEvent,
    photo: Photo
  ) => void;
};

export const PhotoTileDeleteButton = ({
  photo,
  onDeleteButtonClick,
}: PhotoTileDeleteButtonProps) => {
  const composedOnDeleteButtonClick = (event) => {
    onDeleteButtonClick(event, photo);
  };

  return (
    <Button
      aria-label="Delete this image."
      bg="red.500"
      borderRadius={{ base: 0, md: 'md' }}
      color="white"
      position="absolute"
      bottom={{ base: 0, md: 'unset' }}
      left={{ base: 0, md: 'unset' }}
      right={{ base: 0, md: 3 }}
      top={{ base: 'unset', md: 3 }}
      width={{ base: '100%', md: 'unset' }}
      _hover={{ bg: 'red.600' }}
      _active={{ bg: 'red.700' }}
      onClick={composedOnDeleteButtonClick}
    >
      <Icon as={FaTrashAlt} mr={1} />
      <Text>Delete</Text>
    </Button>
  );
};
```

Breakdown:

- In its simplest form, it's a Chakra `Button` component with a trash

`Icon` and some `Text` ("Delete").

- It's absolutely positioned (which will let us move it around easily inside the relatively positioned `PhotoTile`), but the position differs from narrow screens to wide ones.
  - On a narrow screen (`base:`), we'll remove the rounded corners (`borderRadius`), anchor it to the bottom of the image, and make it full width (`top`, `right`, `bottom` and `left`).
  - On a wide screen (`md:`), it'll have its natural width, and be near the top and right of the tile.
- We also make the button change to a deeper red when being hovered, and an even deeper one when pressed. (See Chakra default theme's color docs)
- For passing the `photo` along from the component prop and back out into the `onDeleteButtonClick` handler, we use a similar pattern from our `GalleryEditModal` earlier on in the book; we take the passed-in handler and wrap it in a `composedOnDeleteButtonClick`function which supplies the `photo`.

Detecting touch devices

We'll add a simple utility for detecting whether we're on a touch device, so that we can use the right type of event handler for toggling the button.

```
// /lib/client/isTouchDevice.ts

export const isTouchDevice = () =>
  (function () {
    return (
      'ontouchstart' in window ||
      navigator.maxTouchPoints > 0
    );
  })();
```

Breakdown:

- the `isTouchDevice` function returns the result of an IIFE (an immediately-invoked function expression). The pattern `(function () { ... })()` declares a function and executes it immediately, returning the result.
- We calculate whether this is a touch device by seeing whether the`window.ontouchstart` event property exists, or `navigator.maxTouchPoints`; both of these only exist on touch devices.
- The reason we use the IIFE is that we only need to perform this calculation once; it'll never change throughout the life of the application, so it's wasteful to calculate it each time.

Updating the `PhotoTile`to render the delete button

In summary, we'll use the following strategies: - We'll use `framer-motion` to wrap Chakra-UI `Box` components so they're animatable. - We'll use `isTouchDevice` to determine device type and define the right event handlers. - Mobile devices will toggle the delete button with touch events, and desktop browsers will use hover events. - The event handlers will use a simple `useState` hook to set `showButtons` to true or false. - We'll check `showButtons` in the renderer to determine whether the button should be visible or not.

Here's the updated code:

```
import { SyntheticEvent, useState } from 'react';
import { Box, BoxProps } from '@chakra-ui/react';
import Image from 'next/image';
import { AnimatePresence, motion } from 'framer-motion';
import { Photo } from '@prisma/client';

import { PhotoTileDeleteButton } from 'components/PhotoTileDeleteButton';
import { isTouchDevice } from 'lib/client/isTouchDevice';

export type PhotoTileProps = {
  alt: string;
  height: BoxProps['height'];
  src: string;
  photo: Photo;
  onDeleteButtonClick: (
    event: SyntheticEvent,
    photo: Photo
  ) => void;
};

const MotionBox = motion(Box);

const buttonVariants = {
  open: {
    opacity: 1,
    transition: {
      duration: 0.3,
    },
  },
  closed: {
    opacity: 0,
  },
};
```

```
export const PhotoTile = ({
  height,
  src,
  alt = '',
  photo,
  onDeleteButtonClick = () => {},
}: PhotoTileProps) => {
  const [showButtons, setShowButtons] = useState(false);

  // Event handlers.
  const onClick = () => {
    setShowButtons((show) => !show);
  };

  const onHoverStart = () => {
    setShowButtons(true);
  };

  const onHoverEnd = () => {
    setShowButtons(false);
  };

  const rest = {};

  if (isTouchDevice()) {
    rest['onClick'] = onClick;
  } else {
    rest['onHoverStart'] = onHoverStart;
    rest['onHoverEnd'] = onHoverEnd;
  }

  return (
    <MotionBox height={height} pos="relative" {...rest}>
      <Image
        src={src}
        layout="fill"
        objectFit="cover"
        alt={alt}
      />

      <AnimatePresence>
        {showButtons && (
          <MotionBox
            variants={buttonVariants}
            initial="closed"
            animate="open"
```

```
            exit="closed"
          >
            <PhotoTileDeleteButton
              photo={photo}
              onDeleteButtonClick={onDeleteButtonClick}
            />
          </MotionBox>
        )}
      </AnimatePresence>
    </MotionBox>
  );
};
```

Breakdown:

- We have a few new imports, from `react`, `framer-motion` and our `isTouchDevice` function.
- `SyntheticEvent` is the TypeScript definition of the `event` object React passes around on touch and hover events (docs). We use it to define the type of the `event` param in the `onDeleteButtonClick` prop.
- We declare a `MotionBox` react element, as per the Chakra-UI docs for using `framer-motion` (docs)
- We define `showButtons`, and `setShowButtons` using the `useState` hook (docs, tutorial)
- `buttonVariants` contains the styles for the `open` and `closed` states of our delete button. Framer-motion will animate the `opacity` value automatically when we set the delete button UI to be open or closed with `showButtons` (docs).
- We define the `onClick`, `onHoverStart` and `onHoverEnd` event handler functions. `onClick` toggles the `showButtons`state, whereas the other two set it explicitly to `true` or `false`
- `rest` is an object that will contain the appropriate event handlers for the device type, and pass them to our rendered `MotionBox` container.
- We pass the desired event handlers into the `rest` object dependent on the result of `isTouchDevice`. Browsers get the hover handlers, and touch devices get the click (tap) handler
- In the returned JSX object:
  - We change the outermost`Box` to a `MotionBox`. This allows us to optionally use `framer-motion` events like `onHoverStart` on the container.
    - We now add the event handlers by using the JS spread operator with the `rest` object. `{...rest}` translates to `{{onClick: onClick}}` or `{{onHoverStart: onHoverStart, onHoverEnd: onHoverEnd}}`.
  - Since we want the delete button to actually removed

from the DOM when it's not visible, we wrap it in an `AnimatePresence` element (docs).
    * Whenever the state variable `showButtons` is true, we render our delete button wrapped in a`MotionBox`. We wrap it in the `MotionBox` because we intend to potentially add more buttons here later.
        · We tell the MotionBox to use the variants contained in `buttonVariants`. Initially it uses the `closed`state, animates to `open` when it exists then the `end` prop ensures that it animates to the `closed` state again before being removed from the DOM.

Adding a confirmation modal component

We'll add a `PhotoDeleteDialog` component that will be a copy of our `GalleryDeleteDialog` from our gallery CRUD chapter. It's pretty redundant so I won't paste the full code here, but you can view it in the source. The only notable changes are:

- The `galleryId` prop will be changed to `photoId`;
- The component name changes to `PhotoDeleteDialog` and is saved at `/components/PhotoDeleteDialog/index.tsx`;
- The copy inside the `AlertDialogHeader` is changed to `Delete this photo?`.

Adding some UI for errors

We'll repurpose our `ErrorModal` component from our Gallery CRUD chapter, and connect it to the gallery page in the same way we did with the dashboard.

Relevant code additions:

```
// /pages/galleries/[id].tsx

// We'll add a few more imports
import {
  ReactElement,
  useEffect,
  useRef,
  useState,
} from 'react';
import { ErrorAlert } from 'components/ErrorAlert';

  // Add this near the top of the page function
  const [error, setError] = useState<
    ReactElement | string | null
```

```
>(null);

// Add this sometime thereafter, near the other
//  functiodefinitions.
const handleErrorAlertClose = (event) => {
  event.preventDefault();
  setError(null);
};

// We move this function back inside the page function,
//  since it has to access setError(), which didn't exist
//  until we created it above.
const uploadFileToGallery = async (
  galleryId: number,
  file: File
) => {
  try {
    const { folder, timestamp, signature } = await sign(
      galleryId
    );

    const { secure_url, public_id, width, height } =
      await upload(file, folder, timestamp, signature);

    await create(
      galleryId,
      public_id,
      secure_url,
      width,
      height
    );
  } catch (error) {
    // New addition here.
    setError(
      'An error occurred while uploading the photo.'
    );
  }
};

    // Inside our page's return() statement, below our <Flex>
    //  heading, we add this.
    {error && (
      <ErrorAlert onCloseClick={handleErrorAlertClose}>
        {error}
      </ErrorAlert>
    )}
```

Connecting the confirmation modal and our API endpoint

Inside the gallery page, we'll reuse some bits of code from the dashboard and tailor them to this situation. The final code is in the zip file that came with the book, but here are some notes on the relevant parts.

Notable code additions:

```
import {
  // ...other Chakra imports...
  useDisclosure,
} from '@chakra-ui/react';

import {
  // ...other imports...
  destroy,
} from 'lib/client/api/Photos';

import { PhotoDeleteDialog } from 'components/PhotoDeleteDialog';


export default function GalleryShowPage() {

  // ...

  const {
    isOpen: isPhotoDeleteOpen,
    onClose: onPhotoDeleteClose,
    onOpen: onPhotoDeleteOpen,
  } = useDisclosure();

  const [
    currentPhotoForDeletion,
    setCurrentPhotoForDeletion,
  ] = useState<Photo | null>(null);

  useEffect(() => {
    onPhotoDeleteOpen();
  }, [currentPhotoForDeletion]);

  const handleDeletePhotoClick = (event, photo) => {
    event.preventDefault();
    event.stopPropagation();
```

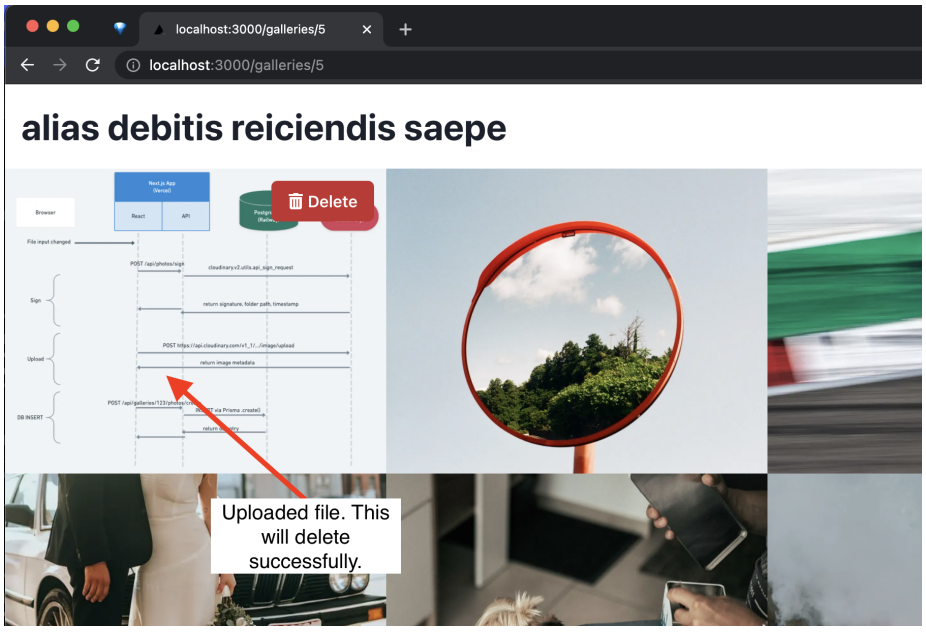
```
  setCurrentPhotoForDeletion(photo);
};

const handlePhotoDeleteSubmit = async (event, id) => {
  event.preventDefault();

  try {
    await destroy(id);
    mutate();
  } catch (error) {
    setError(
      'An error occurred while deleting the photo.'
    );
  } finally {
    onPhotoDeleteClose();
  }
};

// ...

return (

      // ...

      {gallery.photos.map((photo) => {
        const { id, url, caption } = photo;
        return (
          <PhotoTile
            alt={caption || ''}
            height={rowHeight}
            key={id}
            src={url}
            photo={photo}
            onDeleteButtonClick={handleDeletePhotoClick}
          />
        );
      })}


    // .. toward the bottom of the return()
    {currentPhotoForDeletion && (
      <PhotoDeleteDialog
        isOpen={isPhotoDeleteOpen}
        onCloseClick={onPhotoDeleteClose}
        onConfirmClick={handlePhotoDeleteSubmit}
        photoId={currentPhotoForDeletion?.id}
```

```
        />
      )}
    </>
  );
}
```

The overall approach for the `PhotoDeleteDialog` follows that of the `GalleryDeleteDialog`:

- We track the Photo that we intend to delete with React's `useState` hook, which sets the variable `currentPhotoForDeletion`.
- Inside our renderer, we only render the `PhotoDeleteDialog` if `currentPhotoForDeletion` exists.
- We pass `PhotoTile` a function, `handleDeletePhotoClick` to run when the delete button is clicked. It calls `setCurrentPhotoForDeletion` to set the photo into state.
- The above state change triggers the `useEffect` which opens the modal.
- When clicking the 'delete' button on the modal, we launch our `handlePhotoDeleteSubmit` handler. The remaining work is straightforward:
  - `event.preventDefault()` stops any potential screen-jumping.
  - We call our `destroy()` api function, which hits our `pages/api/photos/[id]/delete.ts` API function. This does the work of deleting the photo from Cloudinary and our database.
  - We call `mutate()` to refresh the gallery.
  - If there's an error we call `setError`, which shows our error UI with a message.
  - Whether the delete succeeds or fails, we call `onPhotoDeleteClose()`to close the modal.

You've now got working upload and delete functionality. Be sure to delete only files you've uploaded; the seeded images won't be associated with anything in Cloudinary, so they'll throw an error.


localhost:3000/galleries/5
localhost:3000/galleries/5
alias debitis reiciendis saepe
Delete
Uploaded file. This will delete successfully.