Create a full stack web application with Next.js, TypeScript and GraphQL

<https://www.udemy.com/course/strongly-typed-next-js/learn/lecture/23446596#overview>

Chapter 1: Next.js

https://michaelstromer.nyc/books/strongly-typed-next-js/next-js

https://nextjs.org/docs/routing/introduction

https://nextjs.org/docs/basic-features/data-fetching

https://nextjs.org/docs/basic-features/typescript

https://nextjs.org/

https://material-ui.com/

https://nextjs.org/docs

https://nextjs.org/docs/advanced-features/custom-app

https://nextjs.org/docs/advanced-features/custom-document#customizing-renderpage

https://material-ui.com/guides/server-rendering/

https://github.com/mui-org/material-ui/tree/master/examples/nextjs

Chapter 2: TypeGraphQL

https://michaelstromer.nyc/books/strongly-typed-next-js/typegraphql

https://typegraphql.com/

https://typegoose.github.io/typegoose/

https://www.npmjs.com/package/jsonwebtoken

https://typegraphql.com/docs/introduction.html

https://www.digitalocean.com/community/tutorials/how-to-work-with-typescript-in-visual-studio-code

https://docs.mongodb.com/manual/reference/database-references/

https://typegraphql.com/

https://expressjs.com/en/api.html#res.locals

https://www.typescriptlang.org/docs/handbook/decorators.html

https://docs.mongodb.com/manual/reference/database-references/

Chapter 3: Typegoose

https://michaelstromer.nyc/books/strongly-typed-next-js/typegoose

https://typegoose.github.io/typegoose/

https://github.com/MichalLytek/type-graphql/tree/v1.1.0/examples/typegoose

Chapter 4: Apollo Server

https://michaelstromer.nyc/books/strongly-typed-next-js/apollo-server-1

https://github.com/apollographql/apollo-server

https://expressjs.com/

https://github.com/motdotla/dotenv

https://www.npmjs.com/package/nodemon

https://www.mongodb.com/cloud/atlas/signup

https://docs.atlas.mongodb.com/getting-started/#b-create-an-service-free-tier-cluster

https://www.mongodb.com/cloud/atlas

https://account.mongodb.com/account/login

login with google sramos30@hotmail.com

https://www.apollographql.com/docs/apollo-server/v1/servers/express/

https://expressjs.com/

https://github.com/motdotla/dotenv

https://www.npmjs.com/package/nodemon

Chapter 5: Apollo Client

https://michaelstromer.nyc/books/strongly-typed-next-js/apollo-client-1

https://github.com/piglovesyou/graphql-let

https://graphql-code-generator.com/

https://github.com/piglovesyou/graphql-let

https://www.apollographql.com/docs/react/

https://medium.com/swlh/building-frontend-applications-by-mocking-your-entire-api-with-testing-tools-2f050359677f

https://github.com/Maelstroms38/stream-me/tree/mock-provider

Chapter 6: Authentication

https://michaelstromer.nyc/books/strongly-typed-next-js/authentication

https://nextjs.org/docs/api-reference/next/router

https://reactjs.org/docs/hooks-state.html

https://leerob.io/blog/nextjs-authentication

https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

https://www.joshwcomeau.com/react/persisting-react-state-in-localstorage/

https://www.joshwcomeau.com/react/the-perils-of-rehydration/

Chapter 7: Streaming

https://michaelstromer.nyc/books/strongly-typed-next-js/streaming

https://nextjs.org/docs/api-reference/next/router

https://reactjs.org/docs/hooks-state.html

http://twitch.tv/

http://youtube.com/

https://soundcloud.com/

http://giphy.com/

Chapter 8: Deployment

https://michaelstromer.nyc/books/strongly-typed-next-js/deployment-1

https://classic.yarnpkg.com/en/docs/install/

https://devcenter.heroku.com/articles/heroku-cli#download-and-install

https://heroku.com/

https://classic.yarnpkg.com/en/docs/workspaces/

https://nextjs.org/docs/advanced-features/custom-server

https://devcenter.heroku.com/articles/git

git init

git add .

git commit -m "First commit"

git remote add heroku https://git.heroku.com/sramos30-stream-me.git

git push heroku master

https://sramos30-stream-me.herokuapp.com/241226

# Chapter 1: Next.js

### By the end of this lesson, developers will be able to:

* Create a Next.js web application with TypeScript
* Include custom fonts and styles in your web app
* Link pages using Next.js routing

## Introduction

Next.js offers some serious performance improvements over the standard React web application. Without going into too much detail, we will be using Next.js for:

* [Routing Pages](https://nextjs.org/docs/routing/introduction)
* [Data fetching](https://nextjs.org/docs/basic-features/data-fetching)
* [Typescript Support](https://nextjs.org/docs/basic-features/typescript)

If you are new to Next.js, be sure to visit their [website](https://nextjs.org/) for an overview.

## Installation

Let's begin building the frontend Next.js app.

mkdir stream-me

cd stream-me

mkdir app

cd app

npm init -y

npm install next react react-dom

Open package.json and add the following scripts:

"scripts": {

"dev": "next dev",

"build": "next build",

"start": "next start"

}

We can specify rules on how TS should be compiled into JS. These rules are listed in a tsconfig.json file. Create a new file called app/tsconfig.json, then run npm run dev or yarn dev and Next.js will guide you through the installation of the required packages to finish the setup:

npm install --save-dev typescript

npm install --save-dev @types/react @types/react-dom @types/node

After installing TypeScript, run npm run dev. Next.js should automatically generate a tsconfig.json.

### TypeScript Config

Let's make one modification to the newly generated file app/tsconfig.json:

{

"compilerOptions": {

// ...

"baseUrl": "."

}

}

With this change, Next.js allows us to reference files from the root directory of our project. This will remove the need for relative file imports.

Instead of:

import { Component } from '../../components/example';

We can simply write:

import { Component } from 'components/example';

### Git Ignore

Before moving forward, let's ignore some files to prevent checking them into our source control repository.

At the root of your project insert the following inside a new .gitignore file.

.env

node\_modules

.DS\_Store

.next

## Pages

In this section, we create our first Next.js pages. Next.js serves pages from a unique folder called pages. Instead of routing with React Router, we use this folder to create routes on our website.

Let's start with three core components of Next.js:

1. index.tsx: homepage for our website.
2. \_app.tsx: handles global page props.
3. \_document.tsx: handles global styles.

### Index

Create a new directory and file, app/pages and app/pages/index.tsx and insert the following:

export default () => (

<div>

<p>Hello World!</p>

</div>

);

Fire up the server with npm run dev, and you will see "Hello World" at http://localhost:3000

### App Component

Let's discuss a core component of Next.js: App.

Next.js uses the App component to initialize pages. You can override it and control the page initialization. Which allows you to do amazing things like:

* Persisting layout between page changes
* Keeping state when navigating pages
* Custom error handling using componentDidCatch
* Inject additional data into pages
* Add global CSS

To override the default App, create the file app/pages/\_app.tsx as shown below:

import App from 'next/app';

import React from 'react';

class MyApp extends App {

public render() {

const { Component, pageProps } = this.props;

return <Component {...pageProps} />;

}

}

export default MyApp;

If your app is running and you just added a custom App, you'll need to restart the development server. Only required if pages/\_app.tsx didn't exist before.

### Document Component

A custom Document is commonly used to augment your application's <html> and <body> tags. This is necessary because Next.js injects some stylesheets into the DOM using the custom Document.

To override the default Document, create the file app/pages/\_document.tsx and extend the Document class as shown below:

import Document, { Head, Html, Main, NextScript } from 'next/document';

import React from 'react';

class MyDocument extends Document {

static async getInitialProps(ctx) {

const initialProps = await Document.getInitialProps(ctx)

return { ...initialProps }

}

render() {

return (

<Html>

<Head />

<body>

<Main />

<NextScript />

</body>

</Html>

);

}

}

export default MyDocument;

<Html>, <Head />, <Main /> and <NextScript /> are required for the page to be properly rendered.

Document is only rendered in the server, event handlers like onClick won't work.

### Custom Font

Let's try adding a custom font. Insert the following <link /> inside your custom Document's render method:

render() {

return (

<Html>

<Head>

<link

rel="stylesheet"

href="https://fonts.googleapis.com/css?family=Roboto:300,400:latin"

/>

</Head>

<body>

<Main />

<NextScript />

</body>

</Html>

);

}

Refresh your browser at http://localhost:3000 and you will see "Hello World" with a new Roboto font.

## Material UI

Getting started with Next.js, it is important to decide which components we would like to build with. Thankfully, there are many component libraries to choose from. In this course, we will implement components with [Material UI](https://material-ui.com/).

### Server Rendering

Material-UI was designed from the ground-up with the constraint of rendering on the server, but it's up to you to make sure it's correctly integrated. It's important to provide the page with the required CSS, otherwise the page will render with just the HTML then wait for the CSS to be injected by the client, causing it to flicker. To inject the style down to the client, we need to:

* Create a fresh, new ServerStyleSheets instance on every request.
* Render the React tree with the server-side collector.
* Pull the CSS out.
* Pass the CSS along to the client.
* On the client side, the CSS will be injected a second time before removing the server-side injected CSS.

### Installation

Let's begin with the installation of Material UI:

npm install @material-ui/core

## App Theme

Let's create a Material UI theme for our website.

Create a new directory and file, app/lib and app/lib/theme.ts and insert the following:

import grey from "@material-ui/core/colors/grey";

import { createMuiTheme } from "@material-ui/core/styles";

const themeDark = createMuiTheme({

palette: {

primary: { main: grey[200] },

secondary: { main: grey[400] },

type: "dark",

},

});

const themeLight = createMuiTheme({

palette: {

primary: { main: grey[800] },

secondary: { main: grey[900] },

type: "light",

},

});

export { themeDark, themeLight };

We just created two themes: themeDark and themeLight, which are used to toggle dark and light color palettes.

Let's integrate a ThemeProvider with our global styles. Insert the following into pages/\_app.tsx:

import CssBaseline from "@material-ui/core/CssBaseline";

import { ThemeProvider } from "@material-ui/core/styles";

import { themeDark, themeLight } from "lib/theme";

export default function MyApp(props) {

const { Component, pageProps } = props;

return (

<ThemeProvider theme={false ? themeDark : themeLight}>

<CssBaseline />

<Component {...pageProps} />

</ThemeProvider>

);

}

When rendering, we will wrap Component, the root component, inside a ThemeProvider to make the style configuration and the theme available to all components in the component tree.

import { useEffect } from "react";

export default function MyApp(props) {

const { Component, pageProps } = props;

useEffect(() => {

// Remove the server-side injected CSS.

const jssStyles = document.querySelector("#jss-server-side");

if (jssStyles && jssStyles.parentNode) {

jssStyles.parentNode.removeChild(jssStyles);

}

}, []);

{/\* ... \*/}

}

To prevent loading server-side injected CSS, we also added the useEffect block.

## Server Stylesheets

The final step on the server-side is to inject the initial component HTML and CSS into a template to be rendered on the client side.

Let's create ServerStyleSheets for each request in pages/\_document.tsx:

import { ServerStyleSheets } from "@material-ui/core/styles";

import Document, { Head, Html, Main, NextScript } from 'next/document';

import React from 'react';

class MyDocument extends Document {

static async getInitialProps(ctx) {

// Render app and page and get the context of the page with collected side effects.

const sheets = new ServerStyleSheets();

const originalRenderPage = ctx.renderPage;

ctx.renderPage = () =>

originalRenderPage({

enhanceApp: (App) => (props) => sheets.collect(<App {...props} />),

});

const initialProps = await Document.getInitialProps(ctx);

return {

...initialProps,

// Styles fragment is rendered after the app and page rendering finish.

styles: [

...React.Children.toArray(initialProps.styles),

sheets.getStyleElement(),

],

};

}

{/\* ... \*/}

}

With the setup finally completed, we can start building Material UI components inside our application.

## Linking Pages

Let's start modifying pages/index.tsx with some Material UI components:

import React from "react";

import { Container, Typography, Box, Button } from "@material-ui/core";

import Link from "next/link";

export default function Index() {

return (

<Container maxWidth="sm">

<Box my={4}>

<Typography variant="h4" component="h1" gutterBottom>

Next.js example

</Typography>

<Link href="/about">

<Button variant="contained" color="primary">

Go to the about page

</Button>

</Link>

</Box>

</Container>

);

}

### About Page

Copy and paste the contents of pages/index.tsx into a new file, pages/about.tsx:

import React from "react";

import { Container, Typography, Box, Button } from "@material-ui/core";

import Link from "next/link";

export default function About() {

return (

<Container maxWidth="sm">

<Box my={4}>

<Typography variant="h4" component="h1" gutterBottom>

Next.js example

</Typography>

<Link href="/">

<Button variant="contained" color="primary">

Go to the index page

</Button>

</Link>

</Box>

</Container>

);

}

When you refresh the page at http://localhost:3000/, you may notice that the styles on each page are loading without delay.

### **Congratulations!**

Today we created a Next.js application, with Material UI as our theme provider. In the next lesson, we will begin to design the backend models.

### References

* [Next.js Docs](https://nextjs.org/docs)
* [Next.js App Component](https://nextjs.org/docs/advanced-features/custom-app)
* [Next.js Custom Render](https://nextjs.org/docs/advanced-features/custom-document#customizing-renderpage)
* [Material UI - Server Rendering](https://material-ui.com/guides/server-rendering/)
* [Material UI - Next.js Example](https://github.com/mui-org/material-ui/tree/master/examples/nextjs)

# Chapter 2: TypeGraphQL

## TypeGraphQL

### By the end of this lesson, developers will be able to:

* Create a GraphQL API schema with TypeGraphQL
* Create a UserResolver for fetching user data

### Introduction

Authentication is one of the most challenging tasks for developers just starting with GraphQL. There are a lot of technical considerations, including what ORM would be easy to set up, how to generate secure tokens and hash passwords, and even what HTTP library to use and how to use it.

In this section, we will start building a GraphQL API server. With MongoDB as our database, we learn how to incorporate TypeGraphQL and Typegoose. Let's begin with an overview of these libraries and their dependencies:

* [TypeGraphQL](https://typegraphql.com/): We will use TypeGraphQL to declare our models and their properties.
* [Typegoose](https://typegoose.github.io/typegoose/): Once the models are declared, we will use Typegoose to manage how MongoDB works in the database.
* [jsonwebtoken](https://www.npmjs.com/package/jsonwebtoken): Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. jsonwebtoken will be used to generate a JWT which will be used to authenticate users.

### TypeGraphQL

Starting to incorporate Typescript and GraphQL, we can utilize existing libraries for creating strongly typed schemas. For a brief introduction to TypeGraphQL, be sure to visit the TypeGraphQL [documentation](https://typegraphql.com/docs/introduction.html).

TypeGraphQL solves many problems for us, like schema validation, authorization and dependency injection, which helps develop GraphQL APIs quickly and easily. TypeGraphQL also integrates with several third party libraries like Typegoose.

## Entities

In order to design a relational database, we need to understand the connections between our models. Let's define an Entity Relationship Diagram, or ERD for our models:

As seen above, each User can create many Stream entities. We will continue to define the backend models with TypeGraphQL and Typegoose.

### Installation

First, create a new directory at the root of your project and cd into it:

mkdir api

cd api

Let's begin by installing each library and their peer dependencies.

npm init -y

### TypeGraphQL

npm install typescript type-graphql graphql reflect-metadata

npm install --save-dev @types/node

### Typegoose

npm install @typegoose/typegoose mongoose connect-mongo

npm install --save-dev @types/mongoose

### Express and JWT

npm install express jsonwebtoken

npm install --save-dev @types/express @types/jsonwebtoken

### TypeScript Config

To create a TypeScript configuration file, you can run the following command (similar to npm init -y):

npx tsc --init

You will receive this output:

Output

message TS6071: Successfully created a tsconfig.json file.

Open your new api/tsconfig.json file and you will see lots of different options, most of which are commented out.

Replace your existing api/tsconfig.json file with the following contents:

{

"compilerOptions": {

"target": "es6",

"module": "commonjs",

"lib": ["dom", "es6", "es2017", "esnext.asynciterable"],

"sourceMap": true,

"outDir": "./dist",

"moduleResolution": "node",

"removeComments": true,

"noImplicitAny": true,

"strictNullChecks": true,

"strictFunctionTypes": true,

"noImplicitThis": true,

"noUnusedLocals": true,

"noUnusedParameters": true,

"noImplicitReturns": true,

"noFallthroughCasesInSwitch": true,

"allowSyntheticDefaultImports": true,

"esModuleInterop": true,

"emitDecoratorMetadata": true,

"experimentalDecorators": true,

"resolveJsonModule": true,

"baseUrl": "."

},

"exclude": ["node\_modules", "./generated"],

"include": ["./\*\*/\*.ts"]

}

TypeScript has gained popularity over the last couple of years. In modern front-end frameworks, TypeScript has become a first-class citizen.

For more information on TypeScript and Visual Studio Code, visit [digitalocean.com](https://www.digitalocean.com/community/tutorials/how-to-work-with-typescript-in-visual-studio-code)

## User Entity

Let's prepare our database schema with our first entity, called User.

Create a new directory and file, api/entity and api/entity/User.ts and insert the following:

import { prop as Property, getModelForClass } from "@typegoose/typegoose";

import { ObjectId } from "mongodb";

import { Field, ObjectType } from "type-graphql";

@ObjectType()

export class User {

@Field()

readonly \_id: ObjectId;

@Field()

@Property({ required: true })

email: string;

@Property({ required: true })

password: string;

}

export const UserModel = getModelForClass(User);

Each user has two accessible fields: \_id and email. Passwords are readonly for security purposes. We will demonstrate how to securely store passwords in an upcoming section.

### Decorators

We just wrote our first decorators!

A Decorator is a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter. Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about the decorated class.

For more information on decorators visit [typescriptlang.org](https://www.typescriptlang.org/docs/handbook/decorators.html)

### Ref Type

Before going on to the Stream entity, let's define a Ref object type for our database. A Ref is considered to be a manual reference. A manual references is where you save the ObjectId field of one document in another document as a reference. Then your application can run a second query to return the related data. These references are simple and sufficient for most use cases.

Create a new directory and file, api/types and api/types/Ref.ts and insert the following:

import { ObjectId } from "mongodb";

export type Ref<T> = T | ObjectId;

Using manual references is the practice of including one document's ObjectId field in another document. The application can then issue a second query to resolve the referenced fields as needed.

For nearly every case where you want to store a relationship between two documents, use manual references. The references are simple to create and your application can resolve references as needed. However, if you need to reference documents from multiple collections, consider using DBRefs.

For more information about Database References, visit [MongoDB.org](https://docs.mongodb.com/manual/reference/database-references/)

## Stream Entity

Streams are considered to be embedded posts. We reference the User entity with our Ref type, and assign them as the stream's author.

Create a new file, api/entity/Stream.ts and insert the following:

import { prop as Property, getModelForClass } from "@typegoose/typegoose";

import { ObjectId } from "mongodb";

import { Field, ObjectType } from "type-graphql";

import { User } from "./User";

import { Ref } from "../types/Ref";

@ObjectType()

export class Stream {

@Field()

readonly \_id: ObjectId;

@Field()

@Property({ required: true })

title: string;

@Field()

@Property({ required: true })

description: string;

@Field()

@Property({ required: true })

url: string;

@Field(() => User)

@Property({ ref: User, required: true })

author: Ref<User>;

}

export const StreamModel = getModelForClass(Stream);

With User and Stream entities defined we can go on to create a new ObjectID scalar.

## ObjectID

Before moving on to the resolvers, we will define an ObjectId scalar for our schema. This scalar is specific to MongoDB, because an ObjectId has a unique format, e.g. ObjectId("adaj130jfsdm10").

We will see a few examples of working with ObjectId scalars using Typegoose.

Create a new directory and file, api/schema/ and api/schema/object-id.scalar.ts and insert the following:

import { GraphQLScalarType, Kind } from "graphql";

import { ObjectId } from "mongodb";

export const ObjectIdScalar = new GraphQLScalarType({

name: "ObjectId",

description: "Mongo object id scalar type",

parseValue(value: string) {

return new ObjectId(value); // value from the client input variables

},

serialize(value: ObjectId) {

return value.toHexString(); // value sent to the client

},

parseLiteral(ast) {

if (ast.kind === Kind.STRING) {

return new ObjectId(ast.value); // value from the client query

}

return null;

},

});

The GraphQLScalarType code handles parsing objects as strings, and serializing them as hex strings. This is useful for converting ObjectId properties into string values.

In summary ObjectID("adaj130jfsdm10") becomes a text string: adaj130jfsdm10, and vice versa.

## Middleware

### MyContext

Before getting started with the resolvers, we will declare a new type: MyContext, which will be used to infer the current user's session.

Create a new file, api/types/MyContext.ts and insert the following:

import { Request, Response } from 'express';

export interface MyContext {

req: Request;

res: Response;

}

We will begin to modify MyContext with each session's userId, once we create the authentication resolver.

In order to help create the authentication resolver, we will create a middleware that handles checking for the current user's userId.

### isAuth

Let's declare our first middleware, isAuth (or "isAuthenticated") to make sure the current session contains a logged in user.

Create a new directory and file, api/middleware and api/middleware/isAuth.ts and insert the following:

import { MiddlewareFn } from 'type-graphql';

import { MyContext } from '../types/MyContext';

import jwt from 'jsonwebtoken';

const APP\_SECRET = process.env.SESSION\_SECRET || 'aslkdfjoiq12312';

export const isAuth: MiddlewareFn<MyContext> = async ({ context }, next) => {

const authorization = context.req.headers['authorization'];

try {

const token = authorization?.replace('Bearer ', '')!;

const user = jwt.verify(token, APP\_SECRET) as any;

context.res.locals.userId = user.id;

return next();

} catch (err) {

throw new Error(err.message);

}

};

In the above code, we throw an error if the current session contains no userId property. The isAuth middleware will be applied to a few resolvers, as we will see in a few moments.

Similar to Express middleware, TypegraphQL allows us to write custom middlewares for each request. So we can add this custom logic before each incoming request.

For more information on res.locals, visit the Express.js [documentation](https://expressjs.com/en/api.html#res.locals).

We have one more middleware for handling Typegoose documents, so let's go ahead and add it now.

### Typegoose Middleware

Create a new file, api/middleware/typegoose.ts and insert the following:

import { Model, Document } from 'mongoose';

import { getClassForDocument } from '@typegoose/typegoose';

import { MiddlewareFn } from 'type-graphql';

export const TypegooseMiddleware: MiddlewareFn = async (\_, next) => {

const result = await next();

if (Array.isArray(result)) {

return result.map((item) =>

item instanceof Model ? convertDocument(item) : item

);

}

if (result instanceof Model) {

return convertDocument(result);

}

return result;

};

function convertDocument(doc: Document) {

const convertedDocument = doc.toObject();

const DocumentClass = getClassForDocument(doc)!;

Object.setPrototypeOf(convertedDocument, DocumentClass.prototype);

return convertedDocument;

}

In the above code, we convert MongoDB Documents into readable objects. Without this middleware, our Ref types would not be able to reference other database objects.

## UserResolver

Without further ado, let's create our first resolver called UserResolver. This resolver will handle any queries related to fetching user data.

Create a new directory and file, api/resolvers/ and api/resolvers/UserResolver.ts and insert the following:

import { Resolver, Query, UseMiddleware, Arg, Ctx } from 'type-graphql';

import { ObjectId } from 'mongodb';

import { MyContext } from '../types/MyContext';

import { isAuth } from '../middleware/isAuth';

import { User, UserModel } from '../entity/User';

import { ObjectIdScalar } from '../schema/object-id.scalar';

@Resolver(() => User)

export class UserResolver {

@Query(() => User, { nullable: true })

async user(@Arg('userId', () => ObjectIdScalar) userId: ObjectId) {

return await UserModel.findById(userId);

}

@Query(() => User, { nullable: true })

@UseMiddleware(isAuth)

async currentUser(

@Ctx()

ctx: MyContext

): Promise<User | null> {

return await UserModel.findById(ctx.res.locals.userId);

}

}

You may notice the @UseMiddleware decorator, which is used to integrate the isAuth middleware. Using this resolver, we are able to fetch either individual users or the current logged in user.

### **Congratulations!**

Today we wrote our first resolver. In the next section, we implement the authentication and stream resolvers.

### References

* [TypeGraphQL](https://typegraphql.com/)
* [Express](https://expressjs.com/en/api.html#res.locals)
* [Decorators](https://www.typescriptlang.org/docs/handbook/decorators.html)
* [MongoDB](https://docs.mongodb.com/manual/reference/database-references/)

[Log in](https://michaelstromer.nyc/login?redirectUrl=%2Fbooks%2Fstrongly-typed-next-js%2Ftypegoose)

# Chapter 3: Typegoose

## Typegoose

### By the end of this lesson, developers will be able to:

* Create authentication and stream resolvers
* Create password manager for authentication

### Introduction

In choosing a database, developers should be aware of which frameworks are supported. In the following diagram, we can review the frameworks that led to using MongoDB and Mongoose.

Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment. Mongoose supports both promises and callbacks, and even Typescript.

Take Mongoose, add TypGraphQL and you get Typegoose. Typegoose is a wrapper library for easily writing MongoDB models with TypeScript. It allows us to easily apply Mongoose schemas and models in TypeScript.

Typegoose will create the correct schemas and model mappings for our database. In getting started, it is important to realize that the decision to use Typegoose is based on using MongoDB. Given another database driver, you may want to consider using an Object Relational Mapping like TypeORM.

## Authentication

Before creating the next resolver called AuthResolver, we will need to create two things:

1. api/types/AuthInput - allows users to enter their credentials
2. api/types/UserResponse - returns an authentication user response

### Auth Input

Create a new file, api/types/AuthInput.ts and insert the following:

import { InputType, Field } from 'type-graphql';

@InputType()

export class AuthInput {

@Field()

email: string;

@Field()

password: string;

}

When writing GraphQL mutations, we should create an input type to handle sending the data values. In this case, new users are asked to enter their email and password.

### UserResponse

Create a new file, api/types/UserResponse.ts and insert the following:

import { ObjectType, Field } from 'type-graphql';

import { User } from '../entity/User';

@ObjectType()

export class UserResponse {

@Field(() => User, { nullable: true })

user?: User;

@Field(() => String, { nullable: true })

token?: string;

}

The user response returns a User object and JWT string. Let's begin to integrate a password manager for JWT authentication.

### AuthResolver

Before creating the auth resolver, let's install our hashing library: bcryptjs, which relies on Web Crypto API's getRandomValues interface to obtain secure random numbers.

npm install bcryptjs

npm install -D @types/bcryptjs

Create a new file, api/resolvers/AuthResolver.ts and insert the following:

import { Arg, Mutation, Resolver } from 'type-graphql';

import jwt from 'jsonwebtoken';

import bcrypt from 'bcryptjs';

import { UserModel } from '../entity/User';

import { AuthInput } from '../types/AuthInput';

import { UserResponse } from '../types/UserResponse';

@Resolver()

export class AuthResolver {

@Mutation(() => UserResponse)

async register(

@Arg('input')

{ email, password }: AuthInput

): Promise<UserResponse> {

// 1. check for existing user email

const existingUser = await UserModel.findOne({ email });

if (existingUser) {

throw new Error('Email already in use');

}

// 2. create new user with hash password

const hashedPassword = await bcrypt.hash(password, 10);

const user = new UserModel({ email, password: hashedPassword });

await user.save();

// 3. store user id on the token payload

const payload = {

id: user.id,

};

const token = jwt.sign(

payload,

process.env.SESSION\_SECRET || 'aslkdfjoiq12312'

);

return { user, token };

}

@Mutation(() => UserResponse)

async login(

@Arg('input') { email, password }: AuthInput

): Promise<UserResponse> {

const user = await UserModel.findOne({ email });

if (!user) {

throw new Error('Invalid login');

}

const valid = await bcrypt.compare(password, user.password);

if (!valid) {

throw new Error('Invalid login');

}

// Store user id on the token payload

const payload = {

id: user.id,

};

const token = jwt.sign(

payload,

process.env.SESSION\_SECRET || 'aslkdfjoiq12312'

);

return { user, token };

}

}

The AuthResolver is primarily responsible for the following:

1. Given an email address, check if a user already exists
2. If not, create a new user with a hashed password value
3. Finally, assign and return the new user's JSON Web Token

During testing we will see how the token object is passed to the server. For now, let's continue with creating the StreamResolver.

## Stream Input

Before creating the third resolver StreamResolver, let's declare an input type for creating streams: StreamInput.

Create a new file, api/types/StreamInput.ts and insert the following:

import { InputType, Field } from 'type-graphql';

import { ObjectId } from 'mongodb';

import { Stream } from '../entity/Stream';

@InputType()

export class StreamInput implements Partial<Stream> {

@Field({ nullable: true })

id?: ObjectId;

@Field()

title: string;

@Field({ nullable: true })

description?: string;

@Field()

url: string;

}

Similar to UserInput, StreamInput will accept some parameters to create the new model object. In this case, it accepts title, description and url.

## StreamResolver

Create a new file, api/resolvers/StreamResolver.ts and insert the following:

import {

Resolver,

Query,

Mutation,

FieldResolver,

Ctx,

Arg,

Root,

UseMiddleware,

} from 'type-graphql';

import { ObjectId } from 'mongodb';

import { MyContext } from '../types/MyContext';

import { User, UserModel } from '../entity/User';

import { Stream, StreamModel } from '../entity/Stream';

import { ObjectIdScalar } from '../schema/object-id.scalar';

import { StreamInput } from '../types/StreamInput';

import { isAuth } from '../middleware/isAuth';

@Resolver(() => Stream)

export class StreamResolver {

@Query(() => Stream, { nullable: true })

stream(@Arg('streamId', () => ObjectIdScalar) streamId: ObjectId) {

// 1. find a single stream

return StreamModel.findById(streamId);

}

@Query(() => [Stream])

@UseMiddleware(isAuth)

streams(@Ctx() ctx: MyContext) {

// 2. display all streams for the current user

return StreamModel.find({ author: ctx.res.locals.userId });

}

@Mutation(() => Stream)

@UseMiddleware(isAuth)

async addStream(

@Arg('input') streamInput: StreamInput,

@Ctx() ctx: MyContext

): Promise<Stream> {

// 3. create a new user's stream

const stream = new StreamModel({

...streamInput,

author: ctx.res.locals.userId,

} as Stream);

await stream.save();

return stream;

}

@Mutation(() => Stream)

@UseMiddleware(isAuth)

async editStream(

@Arg('input') streamInput: StreamInput,

@Ctx() ctx: MyContext

): Promise<Stream> {

const { id, title, description, url } = streamInput;

const stream = await StreamModel.findOneAndUpdate(

{ \_id: id, author: ctx.res.locals.userId },

{

title,

description,

url,

},

{ runValidators: true, new: true }

);

if (!stream) {

throw new Error('Stream not found');

}

return stream;

}

@Mutation(() => Boolean)

@UseMiddleware(isAuth)

async deleteStream(

@Arg('streamId', () => ObjectIdScalar) streamId: ObjectId,

@Ctx() ctx: MyContext

): Promise<Boolean | undefined> {

const deleted = await StreamModel.findOneAndDelete({

\_id: streamId,

author: ctx.res.locals.userId,

});

if (!deleted) {

throw new Error('Stream not found');

}

return true;

}

@FieldResolver()

async author(@Root() stream: Stream): Promise<User | null> {

return await UserModel.findById(stream.author);

}

}

With the above code, we can query for single and multiple streams. We can create new streams, given a user is logged in and has a valid session userId.

The @FieldResolver decorator assign streams to their respective authors.

Note: Without TypegooseMiddleware, the @FieldResolver would not work, as it relies on converting Document objects to model objects.

With the resolvers ready, we are ready to complete the server implemetation.

**Congratulations!**

Today we wrote authentication and stream resolvers. In the next section, we implement the schema, session and server.

### References

* [Typegoose](https://typegoose.github.io/typegoose/)
* [TypeGraphQL](https://github.com/MichalLytek/type-graphql/tree/v1.1.0/examples/typegoose)

[Log in](https://michaelstromer.nyc/login?redirectUrl=%2Fbooks%2Fstrongly-typed-next-js%2Fapollo-server-1)

# Chapter 4: Apollo Server

## Apollo Server

### By the end of this lesson, developers will be able to:

* Create a GraphQL API server with Apollo Server Express
* Create a MongoDB database connection with Connect Mongo
* Create environment variables for your localhost server

### Introduction

In the previous section, we wrote a strongly typed schema for our GraphQL API server. In this section we will complete the implementation with Apollo Server and Express. Apollo Server Express allows us to build a GraphQL query interface on top of an existing Express server.

Let's begin with an overview of these libraries and their dependencies:

* [Apollo Server](https://github.com/apollographql/apollo-server): An open-source GraphQL server that is compatible with any kind of GraphQL client. We won’t be using Express for our server in this project. Instead, we will use the power of Apollo Server to expose our GraphQL API.
* [Express](https://expressjs.com/): Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.
* [dotenv](https://github.com/motdotla/dotenv): We will use dotenv to load environment variables from our .env file.
* [nodemon](https://www.npmjs.com/package/nodemon): A tool that helps develop Node-based applications by automatically restarting the node application when changes in the directory are detected. We don’t want to be closing and starting the server every time there’s a change in our code. Nodemon inspects changes every time in our app and automatically restarts the server.

### Installation

Heads Up! This lesson's code edits will only affect your API project.

First, change directory into your root level api folder:

cd api

Let's begin with the installation of Apollo Server Express:

npm install apollo-server-express cors ts-node

npm install --save-dev nodemon

### Server Scripts

Next, we will edit package.json with two new scripts:

/\* ... \*/

"scripts": {

"start": "ts-node --transpile-only server/index.ts",

"dev": "nodemon server/index.ts"

},

/\* ... \*/

The ts-node --transpile-only command will skip type checking and compile the server code.

The nodemon command will run a development server, and reload when any changes occur.

Based on the above scripts, we will also create directories with the following contents:

mkdir session

mkdir server

Each of these directories contains a unique purpose:

1. api/session: Mongoose creates the server sessions
2. api/server: Apollo Server Express runs the server

Before moving on to the session and server, we will define a new schema.

### Schema

Create a new file, api/schema/index.ts and insert the following:

import { GraphQLSchema } from 'graphql';

import { buildSchema } from 'type-graphql';

import { ObjectId } from 'mongodb';

import path from 'path';

import { UserResolver } from '../resolvers/UserResolver';

import { AuthResolver } from '../resolvers/AuthResolver';

import { StreamResolver } from '../resolvers/StreamResolver';

import { ObjectIdScalar } from './object-id.scalar';

import { TypegooseMiddleware } from '../middleware/typegoose';

// build TypeGraphQL executable schema

export default async function createSchema(): Promise<GraphQLSchema> {

const schema = await buildSchema({

// 1. add all typescript resolvers

resolvers: [UserResolver, AuthResolver, StreamResolver],

emitSchemaFile: path.resolve(\_\_dirname, 'schema.gql'),

// 2. use document converting middleware

globalMiddlewares: [TypegooseMiddleware],

// 3. use ObjectId scalar mapping

scalarsMap: [{ type: ObjectId, scalar: ObjectIdScalar }],

validate: false,

});

return schema;

}

In the above code, we do the following:

1. Add all typescript resolvers into the schema from the ../resolvers directory.
2. Apply typegoose middleware to allow GraphQL work with MongoDB Documents.
3. Apply the scalars map to parse ObjectId properties into string values.

Up next, we will create a free MongoDB Atlas account and cluster.

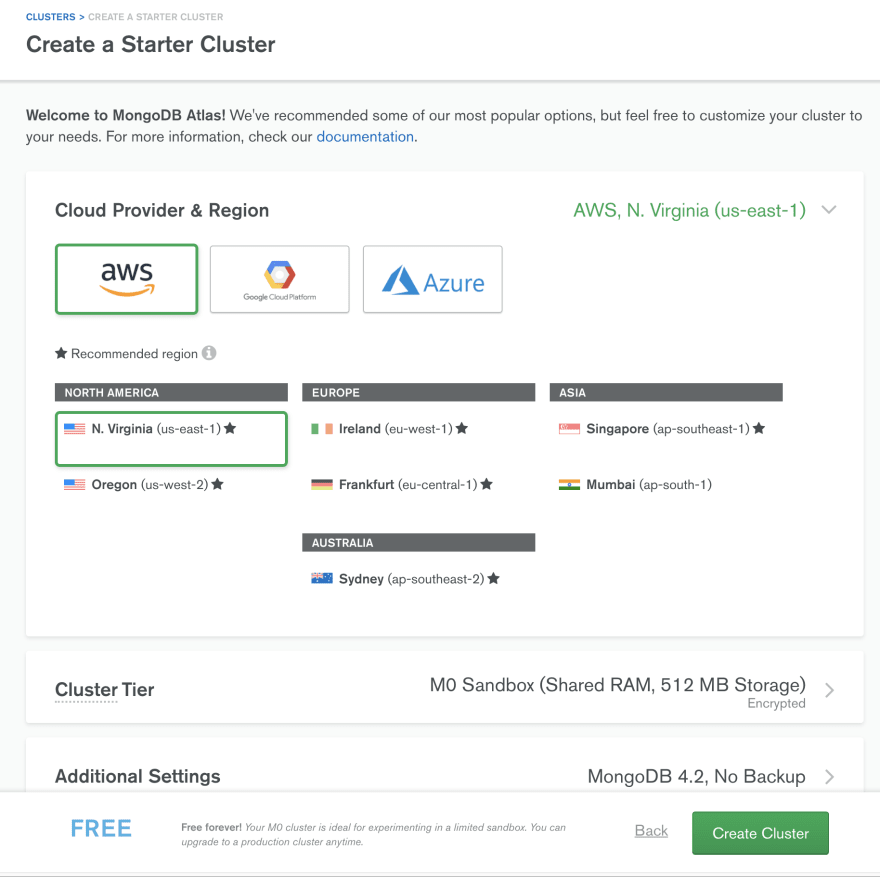
### MongoDB Atlas

We recommend creating a free tier database cluster at MongoDB Atlas. Sign up for [MongoDB Atlas](https://www.mongodb.com/cloud/atlas/signup) and follow this [simple tutorial](https://docs.atlas.mongodb.com/getting-started/#b-create-an-service-free-tier-cluster) to create a free cluster.

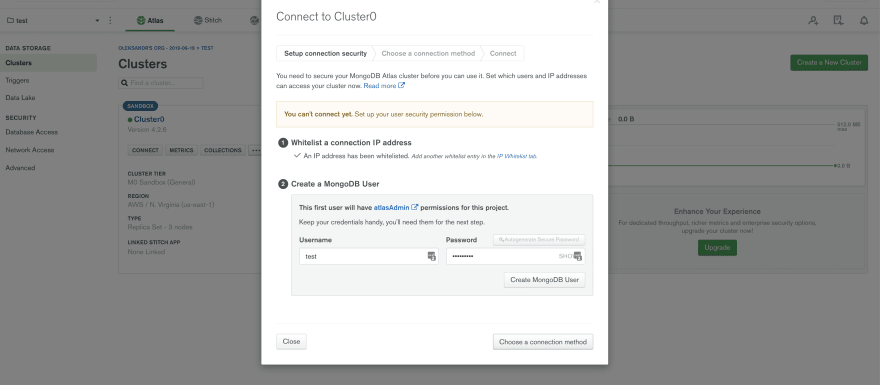
MongoDB Atlas is the global cloud database service for modern applications. Deploy fully managed MongoDB across AWS, Azure, or GCP. Best-in-class automation and proven practices guarantee availability, scalability, and compliance with the most demanding data security and privacy standards. Use MongoDB’s robust ecosystem of drivers, integrations, and tools to build faster and spend less time managing your database.

We are going to be using an instance of the MongoDB Atlas database.

1. Navigate to the [MongoDB Atlas](https://www.mongodb.com/cloud/atlas) page
2. Click "Start Free" and sign up for the MongoDB account
3. On the "Projects" page click on "New Project" give it a name and create
4. Add Members. You’re already a member -> hit continue
5. Build Cluster -> Select Free Tier
6. Select Cloud Provider & Region and Create Cluster



1. After the cluster was initialized click on "connect"



1. Choose a connection method -> Select Connect Your Application and select Node.js
2. Copy and save your connection string, which should look like the following:

mongodb+srv://<dbname>:<password>@cluster0-yvwjx.mongodb.net/<dbname>?retryWrites=true&w=majority

All set! Up next, we will store the connection string as a new environment variable called MONGO\_URL.

### Environment Setup

Before creating the session, we will install dotenv to initialize the MONGO\_URL environment value.

npm install dotenv

npm install --save-dev @types/dotenv

Create a new file, api/server/env.ts and insert the following:

import { config } from 'dotenv';

const result = config();

// Only override process.env if .env file is present and valid

if (!result.error) {

const parsed = result.parsed;

if (parsed) {

Object.keys(parsed).forEach((key) => {

const value = parsed[key];

if (value) {

process.env[key] = value;

console.log(`${key}=${value}`);

}

});

}

}

Using the dotenv library, we are able to determine if a valid .env file is present and log all it's respective values.

### Environment File

Create a new file, api/.env and insert the following:

MONGO\_URL=REPLACE\_WITH\_MONGO\_URL

URL\_APP=http://localhost:3000

Be sure to include your connection url (MONGO\_URL) from MongoDB Atlas.

Remember, this file is not meant to be committed with your git repository, as it contains sensitive information.

Up next, we will implement the mongo session.

### Session

Create a new file, api/session/index.ts and insert the following:

import { connect } from 'mongoose';

export default async function createSession() {

const MONGO\_URL = process.env.MONGO\_URL || '';

if (!MONGO\_URL) {

throw new Error('Missing MONGO\_URL');

}

const options = {

useNewUrlParser: true,

useCreateIndex: true,

useFindAndModify: false,

useUnifiedTopology: true,

};

await connect(MONGO\_URL, options);

}

In the above code, we initialize a MongoDB connection. If any errors are thrown, they will be handled by the server at runtime.

### Server

Create a new file, api/server/index.ts and insert the following:

import './env';

import 'reflect-metadata';

import { ApolloServer } from 'apollo-server-express';

import express from 'express';

import cors from 'cors';

import createSchema from '../schema';

import createSession from '../session';

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

async function createServer() {

try {

// 1. create mongoose connection

await createSession();

// 2. create express server

const app = express();

// allow CORS from client app

const corsOptions = {

origin: 'http://localhost:3000',

credentials: true,

};

app.use(cors(corsOptions));

// allow JSON requests

app.use(express.json());

const schema = await createSchema();

// 3. create GraphQL server

const apolloServer = new ApolloServer({

schema,

context: ({ req, res }) => ({ req, res }),

introspection: true,

// enable GraphQL Playground with credentials

playground: {

settings: {

'request.credentials': 'include',

},

},

});

apolloServer.applyMiddleware({ app, cors: corsOptions });

// start the server

app.listen({ port }, () => {

console.log(

`🚀 Server ready at http://localhost:${port}${apolloServer.graphqlPath}`

);

});

} catch (err) {

console.log(err);

}

}

createServer();

The ApolloServer is initialized with an Express server, MongoDB session middleware and GraphQL schema. Most of the logic is abstracted out to simplify this file, so here's an overview:

1. We initialize a MongoDB database connection.
2. We initialize an Express server and apply cors, json and session middleware.
3. We initialize an Apollo GraphQL server with the schema definition, context and playground options.

Note: Setting 'request.credentials': 'include', in the playground options is necessary for testing authenticated requests.

### Testing

It's time to fire up the server and test our queries.

npm run dev

Be sure to visit the playground and test your queries at <http://localhost:8000/graphql>

### **Congratulations!**

Today we completed the GraphQL API server. Up next, we will begin to integrate an Apollo Client with the frontend.

### References

* [Apollo Server Express](https://www.apollographql.com/docs/apollo-server/v1/servers/express/)
* [Express](https://expressjs.com/)
* [dotenv](https://github.com/motdotla/dotenv)
* [nodemon](https://www.npmjs.com/package/nodemon)

[Log in](https://michaelstromer.nyc/login?redirectUrl=%2Fbooks%2Fstrongly-typed-next-js%2Fapollo-client-1)

# Chapter 5: Apollo Client

## Apollo Client

### By the end of this lesson, developers will be able to:

* Create an Apollo Client instance on the frontend web
* Generate queries and mutations with GraphQL codegen

### Introduction

In the previous section, we completed the implementation for our GraphQL API server. In this section, we switch back to the frontend client and start generating queries and mutations.

### GraphQL Codegen

In this course, we are able to leverage Typescript in a variety of ways. The most efficient use of Typescript and GraphQL is generating frontend queries, using the GraphQL Codegen library.

Using GraphQL Codegen, we generate queries and mutations for the frontend client.

#### **Installation**

Heads Up! This lesson's code edits will only affect your APP project.

First, change directory into your root level app folder:

cd app

Let's begin with the installation of GraphQL Codegen and GraphQL Let:

npm install --save-dev graphql-let @graphql-codegen/cli @graphql-codegen/plugin-helpers @graphql-codegen/typescript @graphql-codegen/typescript-operations @graphql-codegen/typescript-react-apollo yaml-loader

npm install @apollo/client graphql

npx graphql-let init

#### GraphQL Let

Similar to GraphQL Codegen, GraphQL Let allows us to reference types that belong to our API, and generate queries for the frontend. The major difference with the GraphQL Let library is that it generates typed files (example.d.ts) for each query.

For an overview of GraphQL Let, [click here](https://github.com/piglovesyou/graphql-let).

Getting started with GraphqL Codegen, we will create a file to configure it's behavior.

Create a new file, app/.graphql-let.yml and insert the following:

schema: 'lib/schema.graphqls'

documents: '\*\*/\*.graphql'

plugins:

- typescript

- typescript-operations

- typescript-react-apollo

cacheDir: \_\_generated\_\_

In this file, we specify where the schema definition lives. In this case, we are pulling the schema definition from our local GraphQL API server.

#### Copy Schema

Grab a copy of the GraphQL schema by copying the entire file api/schema/schema.gql and pasting it's contents inside a new file called app/lib/schema.graphqls.

You may noticed we referenced this new file lib/schema.graphqls in the previous step. Be sure to give it the correct file extension: .graphqls, otherwise GraphQL Codegen will not work.

#### Git Ignore

Before moving forward, add the following to your root level .gitignore file:

\*.graphql.d.ts

\*.graphqls.d.ts

\_\_generated\_\_

#### Next.js Config

Add the following to app/next-env.d.ts:

/// <reference types="next" />

/// <reference types="next/types/global" />

declare module '\*.graphqls' {

import { DocumentNode } from 'graphql';

export default typeof DocumentNode;

}

declare module '\*.yml';

We made this change to allow Next.js to read .yml files.

#### Webpack Config

Let's add webpack support for GraphQL Codegen with the following changes in next.config.js:

module.exports = {

webpack(config, options) {

config.module.rules.push({

test: /\.graphql$/,

exclude: /node\_modules/,

use: [options.defaultLoaders.babel, { loader: 'graphql-let/loader' }],

});

config.module.rules.push({

test: /\.graphqls$/,

exclude: /node\_modules/,

use: ['graphql-let/schema/loader'],

});

config.module.rules.push({

test: /\.ya?ml$/,

type: 'json',

use: 'yaml-loader',

});

return config;

},

};

### Generate Queries

With GraphQL Codegen configured, we will create three new queries for the client:

1. currentUser
2. stream
3. streams

First, create a new directory and file, called app/lib/graphql and app/lib/graphql/currentUser.graphql:

query CurrentUser {

currentUser {

\_id

email

}

}

Every query needs a name, so we called this one CurrentUser.

Second, create a new file, called app/lib/graphql/stream.graphql:

query Stream($streamId: ObjectId!) {

stream(streamId: $streamId) {

\_id

title

description

url

author {

\_id

email

}

}

}

Note that this query requires an input called streamId. We will use it to fetch individual streams.

Third, create a new file, called app/lib/graphql/streams.graphql:

query Streams {

streams {

\_id

title

description

url

}

}

Note that this query fetches the current user's streams, so they will need to be logged in.

### Generate Mutations

We are going to repeat the above process for five new mutations:

1. createStream
2. editStream
3. deleteStream
4. signin
5. signup

### Create Stream

Create a new file called app/lib/graphql/createStream.graphql:

mutation CreateStream($input: StreamInput!) {

addStream(input: $input) {

\_id

title

description

url

}

}

Note that this mutation has an input value of StreamInput, which is required.

### Edit Stream

Create a new file called app/lib/graphql/editStream.graphql:

mutation EditStream($input: StreamInput!) {

editStream(input: $input) {

\_id

title

description

url

}

}

### Delete Stream

Create a new file called app/lib/graphql/deleteStream.graphql:

mutation DeleteStream($id: ObjectId!) {

deleteStream(streamId: $id)

}

### Sign In

Create a new file called app/lib/graphql/signin.graphql:

mutation SignIn($email: String!, $password: String!) {

login(input: { email: $email, password: $password }) {

user {

\_id

email

}

token

}

}

### Sign Up

Create a new file called app/lib/graphql/signup.graphql:

mutation SignUp($email: String!, $password: String!) {

register(input: { email: $email, password: $password }) {

user {

\_id

email

}

token

}

}

### Test Codegen

Let's take this opportunity to generate the queries and mutations with graphql-let.

Heads Up! Your GraphQL API server needs to be running in order to run graphql-let.

In a separate terminal window, run the API server using npm run dev, and then try the following command in your app directory:

npx graphql-let

You should see something like the following output:

✔ Parse configuration

[ graphql-let ] 8 .d.ts were generated.

If you encountered any issues with the above commands, verify your local GraphQL API server is running and try again.

### Init Client

With our queries and mutations generated, we can continue to initialize the Apollo Client.

Create a new file, app/lib/apollo.ts and insert the following:

import { useMemo } from 'react';

import {

ApolloClient,

InMemoryCache,

NormalizedCacheObject,

HttpLink,

} from '@apollo/client';

import { setContext } from '@apollo/client/link/context';

let apolloClient: ApolloClient<NormalizedCacheObject> | undefined;

function createApolloClient() {

const authLink = setContext((\_, { headers }) => {

// get the authentication token from local storage if it exists

const token = sessionStorage.getItem('token');

// return the headers to the context so httpLink can read them

return {

headers: {

...headers,

authorization: token ? `Bearer ${token}` : '',

},

};

});

const httpLink = new HttpLink({

uri: 'http://localhost:8000/graphql',

credentials: 'include',

});

return new ApolloClient({

link: authLink.concat(httpLink),

cache: new InMemoryCache(),

});

}

export function initializeApollo(initialState: any = null) {

const \_apolloClient = apolloClient ?? createApolloClient();

// If your page has Next.js data fetching methods that use Apollo Client, the initial state

// get hydrated here

if (initialState) {

\_apolloClient.cache.restore(initialState);

}

// For SSG and SSR always create a new Apollo Client

if (typeof window === 'undefined') return \_apolloClient;

// Create the Apollo Client once in the client

if (!apolloClient) apolloClient = \_apolloClient;

return \_apolloClient;

}

export function useApollo(initialState: any) {

const store = useMemo(() => initializeApollo(initialState), [initialState]);

return store;

}

In the above code, we connect to the Apollo Client with an HttpLink. During initialization, we also need to include credentials in order to support GraphQL authentication.

On any Next.js page that uses data fetching methods, we can re-hydrate the Apollo client cache and fetch data from the cache instead of the server. The useApollo hook will handle initializing and caching once its integrated with a root component, like \_app.tsx.

### Apollo Provider

Make the following changes to app/pages/\_app.tsx:

import { useEffect, useState } from 'react';

import CssBaseline from '@material-ui/core/CssBaseline';

import { ThemeProvider } from '@material-ui/core/styles';

import { ApolloProvider } from '@apollo/client';

import { useApollo } from 'lib/apollo';

import { themeDark, themeLight } from 'lib/theme';

export default function MyApp({ Component, pageProps }) {

const [darkState, setDarkState] = useState(false);

const handleThemeChange = () => {

setDarkState(!darkState);

};

const apolloClient = useApollo(pageProps.initialApolloState);

useEffect(() => {

// Remove the server-side injected CSS.

const jssStyles = document.querySelector('#jss-server-side');

if (jssStyles && jssStyles.parentNode) {

jssStyles.parentNode.removeChild(jssStyles);

}

}, []);

return (

<ApolloProvider client={apolloClient}>

<ThemeProvider theme={darkState ? themeDark : themeLight}>

<CssBaseline />

<Component {...pageProps} />

</ThemeProvider>

</ApolloProvider>

);

}

### Testing

It's time to fire up the Next.js app and test our new header at [http://localhost:3000](http://localhost:3000/)

npm run dev

If you are still able to see the original landing page, success! We have completed the installation of Apollo Client with Next.js.

### **Congratulations!**

Today we completed the frontend Apollo Client setup. Up next, we start building data-fetching queries and create the authentication flow.

## References

* [GraphQL Codegen](https://graphql-code-generator.com/)
* [GraphQL Let](https://github.com/piglovesyou/graphql-let)
* [Apollo Client](https://www.apollographql.com/docs/react/)

[Log in](https://michaelstromer.nyc/login?redirectUrl=%2Fbooks%2Fstrongly-typed-next-js%2Fauthentication)

# Chapter 6: Authentication

## Auth Flow

### By the end of this lesson, developers will be able to:

* Create components and pages to handle data fetching
* Create an authentication flow for login and registration

### Introduction

With the Apollo Client initialized, we can start writing functional components with hooks. We will start with our first context provider: AuthProvider.

After creating the AuthProvider, we will implement authetication flow screens and allow users to login, register and sign out.

Following authentication, we will continue with the streaming flow and allow users to create and view their streams.

### Auth Provider

The purpose of the AuthProvider is to create a global user object, and handle any authentication logic with a single hook.

Once we declare the AuthProvider, you may notice that authentication becomes much quicker to implement.

Let's begin with a new file, app/lib/useAuth.tsx:

import { useState, useContext, createContext, useEffect } from 'react';

import { useApolloClient } from '@apollo/client';

import { useSignInMutation } from 'lib/graphql/signin.graphql';

import { useSignUpMutation } from 'lib/graphql/signup.graphql';

import { useCurrentUserQuery } from 'lib/graphql/currentUser.graphql';

import { useRouter } from 'next/router';

type AuthProps = {

user: any;

error: string;

signIn: (email: any, password: any) => Promise<void>;

signUp: (email: any, password: any) => Promise<void>;

signOut: () => void;

}

const AuthContext = createContext<Partial<AuthProps>>({});

// You can wrap your \_app.js with this provider

export function AuthProvider({ children }) {

const auth = useProvideAuth();

return <AuthContext.Provider value={auth}>{children}</AuthContext.Provider>;

}

// Custom React hook to access the context

export const useAuth = () => {

return useContext(AuthContext);

};

function useProvideAuth() {

const client = useApolloClient();

const router = useRouter();

const [error, setError] = useState('');

const { data } = useCurrentUserQuery({

fetchPolicy: 'network-only',

errorPolicy: 'ignore',

});

const user = data && data.currentUser;

// Signing In

const [signInMutation] = useSignInMutation();

// Signing Up

const [signUpMutation] = useSignUpMutation();

const signIn = async (email, password) => {

try {

const { data } = await signInMutation({ variables: { email, password } });

if (data.login.token && data.login.user) {

sessionStorage.setItem('token', data.login.token);

client.resetStore().then(() => {

router.push('/');

});

} else {

setError("Invalid Login");

}

} catch (err) {

setError(err.message);

}

}

const signUp = async (email, password) => {

try {

const { data } = await signUpMutation({ variables: { email, password } });

if (data.register.token && data.register.user) {

sessionStorage.setItem('token', data.register.token);

client.resetStore().then(() => {

router.push('/');

});

} else {

setError("Invalid Login");

}

} catch (err) {

setError(err.message);

}

}

const signOut = () => {

sessionStorage.removeItem('token');

client.resetStore().then(() => {

router.push('/');

});

}

return {

user,

error,

signIn,

signUp,

signOut,

};

}

The above code snippet creates a global auth context called AuthContext, but to access it's props we need to wrap the root app/pages/\_app.tsx file:

import { AuthProvider } from 'lib/useAuth';

/\* ... \*/

return (

<ApolloProvider client={apolloClient}>

<ThemeProvider theme={darkState ? themeDark : themeLight}>

<CssBaseline />

<AuthProvider>

<Component {...pageProps} />

</AuthProvider>

</ThemeProvider>

</ApolloProvider>

);

Now we can access the global user object using the useAuth hook. Let's incorporate the useAuth hook with a new Header component.

### Header Component

First, create a new directory and file, called app/components and app/components/Header.tsx:

import React from 'react';

import { makeStyles, Theme } from '@material-ui/core/styles';

import {

AppBar,

Toolbar,

Typography,

Button,

Link as LinkText,

Switch,

} from '@material-ui/core';

import Link from 'next/link';

import { useAuth } from 'lib/useAuth';

export default function Header({ darkState, handleThemeChange }) {

const classes = useStyles();

const { user } = useAuth();

const links = [

!user && { label: 'Sign Up', href: '/auth/signup' },

!user && { label: 'Sign In', href: '/auth/signin' },

user && { label: 'Create', href: '/streams/new' },

user && { label: 'Sign Out', href: '/auth/signout' },

]

.filter((link) => link)

.map(({ label, href }) => {

return (

<Link href={href} key={href}>

<Button color="inherit">{label}</Button>

</Link>

);

});

return (

<div className={classes.root}>

<AppBar position="static">

<Toolbar>

<Typography variant="h6" className={classes.title}>

<Link href="/">

<LinkText href="" color="inherit">

Stream.me

</LinkText>

</Link>

</Typography>

<Switch checked={darkState} onChange={handleThemeChange} />

{links}

</Toolbar>

</AppBar>

</div>

);

}

const useStyles = makeStyles((theme: Theme) => ({

root: {

flexGrow: 1,

},

menuButton: {

marginRight: theme.spacing(2),

},

title: {

flexGrow: 1,

},

list: {

width: 250,

},

}));

The Header will be displayed globally, so we can include it in the app/pages/\_app.tsx component:

Import and render the Header component inside app/pages/\_app.tsx:

import Header from 'components/Header';

/\* ... \*/

return (

<ApolloProvider client={apolloClient}>

<ThemeProvider theme={darkState ? themeDark : themeLight}>

<CssBaseline />

<AuthProvider>

<Header darkState={darkState} handleThemeChange={handleThemeChange} />

<Component {...pageProps} />

</AuthProvider>

</ThemeProvider>

</ApolloProvider>

);

### Auth Screens

#### **Sign In**

Create a new directory and file, app/pages/auth and app/pages/auth/signin, and insert the following:

import { useState } from 'react';

import Typography from '@material-ui/core/Typography';

import Container from '@material-ui/core/Container';

import TextField from '@material-ui/core/TextField';

import Box from '@material-ui/core/Box';

import Button from '@material-ui/core/Button';

import { useAuth } from 'lib/useAuth';

export default function SignIn() {

const [email, setEmail] = useState('');

const [password, setPassword] = useState('');

const { error, signIn } = useAuth();

const onSubmit = async (event) => {

event.preventDefault();

signIn(email, password);

};

return (

<Container maxWidth="sm">

<Box my={4}>

<form onSubmit={onSubmit}>

{error && <p>{error}</p>}

<Typography variant="h4">Sign In</Typography>

<Box pb={2.5} />

<TextField

value={email}

onChange={(e) => setEmail(e.target.value)}

className="form-control"

label="Email"

required

/>

<Box pb={2.5} />

<TextField

value={password}

onChange={(e) => setPassword(e.target.value)}

type="password"

className="form-control"

label="Password"

required

/>

<Box pb={2.5} />

<Button

variant="contained"

color="primary"

size="large"

type="submit"

>

Sign In

</Button>

</form>

</Box>

</Container>

);

}

## Sign Up

Create a new file app/pages/auth/signup, and insert the following:

import { useState } from 'react';

import Typography from '@material-ui/core/Typography';

import Container from '@material-ui/core/Container';

import TextField from '@material-ui/core/TextField';

import Box from '@material-ui/core/Box';

import Button from '@material-ui/core/Button';

import { useAuth } from 'lib/useAuth';

export default function SignUp() {

const [email, setEmail] = useState('');

const [password, setPassword] = useState('');

const { error, signUp } = useAuth();

const onSubmit = async (event) => {

event.preventDefault();

signUp(email, password);

};

return (

<Container maxWidth="sm">

<Box my={4}>

<form onSubmit={onSubmit}>

{error && <p>{error}</p>}

<Typography variant="h4">Sign Up</Typography>

<Box pb={2.5} />

<TextField

value={email}

onChange={(e) => setEmail(e.target.value)}

className="form-control"

label="Email"

required

/>

<Box pb={2.5} />

<TextField

value={password}

onChange={(e) => setPassword(e.target.value)}

type="password"

className="form-control"

label="Password"

required

/>

<Box pb={2.5} />

<Button

variant="contained"

color="primary"

size="large"

type="submit"

>

Sign Up

</Button>

</form>

</Box>

</Container>

);

}

#### Sign Out

Create a new file app/pages/auth/signout.tsx, and insert the following:

import { useEffect } from 'react';

import { useAuth } from 'lib/useAuth';

export default function SignOut() {

const { signOut } = useAuth();

useEffect(() => {

signOut();

}, []);

return <div>Signout</div>;

}

#### **Test Auth**

It's time to fire up the Next.js app, GraphQL server and test our new authentication flow at [http://localhost:3000](http://localhost:3000/)

Heads Up! Your GraphQL API server needs to be running in order to test authentication.

npm run dev

Try reloading the Next.js application, and you will see a new header above every page. The header also displays links to Sign Up and Sign In. Let's visit and test their functionality as well.

Be sure you are able to:

* Log in an existing user
* Register a new user
* Log out of the session

### **Congratulations!**

Today we built client-side data-fetching queries and created an authentication flow. Up next, we will create the streaming flow.

### References

* [Next Router](https://nextjs.org/docs/api-reference/next/router)
* [useState Hook](https://reactjs.org/docs/hooks-state.html)
* [Lee Robinson](https://leerob.io/blog/nextjs-authentication)

[Log in](https://michaelstromer.nyc/login?redirectUrl=%2Fbooks%2Fstrongly-typed-next-js%2Fstreaming)

# Chapter 7: Streaming

## Stream Flow

### By the end of this lesson, developers will be able to:

* Create components and pages to handle data fetching
* Create a streaming flow for creating and viewing streams

### Introduction

With the authentication flow completed, we will move on to create a streaming flow. Bear in mind, we haven't implemented a streaming flow. First, we will scaffold the components to create and view streams.

### Posts Component

To display the users posted streams, we will create a list component called Posts.

Create a new file, app/components/Posts.tsx, and insert the following:

import Typography from '@material-ui/core/Typography';

import Grid from '@material-ui/core/Grid';

import Card from '@material-ui/core/Card';

import CardActionArea from '@material-ui/core/CardActionArea';

import CardContent from '@material-ui/core/CardContent';

import CardMedia from '@material-ui/core/CardMedia';

import Hidden from '@material-ui/core/Hidden';

import Link from 'next/link';

import { makeStyles, Theme } from '@material-ui/core/styles';

import { Stream } from '../lib/graphql/streams.graphql';

interface Props {

streams: Stream[];

}

export default function Posts(props: Props) {

const styles = useStyles();

const { streams } = props;

return (

<Grid container className={styles.container} spacing={4}>

{streams.map((post) => (

<Grid item key={post.\_id} xs={12} md={6}>

<Link href={`/streams/${post.\_id}`}>

<CardActionArea component="a" href="#">

<Card className={styles.card}>

<div className={styles.cardDetails}>

<CardContent>

<Typography

component="h2"

variant="h5"

className={styles.cardText}

>

{post.title}

</Typography>

<Typography

noWrap={true}

variant="subtitle1"

color="textSecondary"

className={styles.cardText}

>

{post.url}

</Typography>

<Typography

variant="subtitle1"

paragraph

className={styles.cardText}

>

{post.description}

</Typography>

</CardContent>

</div>

<Hidden xsDown>

<CardMedia

className={styles.cardMedia}

image="https://source.unsplash.com/random"

title="Image title"

/>

</Hidden>

</Card>

</CardActionArea>

</Link>

</Grid>

))}

</Grid>

);

}

const useStyles = makeStyles((theme: Theme) => ({

container: {

marginTop: theme.spacing(4),

},

card: {

display: 'flex',

},

cardDetails: {

flex: 1,

},

cardText: {

maxWidth: '26rem',

},

cardMedia: {

width: 160,

},

}));

### Hero Component

On each stream's detail page, we will display a banner or "hero" component called Hero.

Create a new file, app/components/Hero.tsx, and insert the following:

import Typography from '@material-ui/core/Typography';

import Link from 'next/link';

import Button from '@material-ui/core/Button';

import Box from '@material-ui/core/Box';

import Grid from '@material-ui/core/Grid';

import Paper from '@material-ui/core/Paper';

import { makeStyles } from '@material-ui/core/styles';

import { Stream } from '../lib/graphql/streams.graphql';

import { useAuth } from 'lib/useAuth';

interface Props {

stream: Stream;

}

export default function Hero({ stream }: Props) {

const styles = useStyles();

const { user } = useAuth();

const showEdit =

user &&

user.\_id === stream.author.\_id;

return (

<Paper className={styles.mainFeaturedPost}>

<div className={styles.overlay} />

<Grid container>

<Grid item md={6}>

<div className={styles.mainFeaturedPostContent}>

<Typography

component="h1"

variant="h3"

color="inherit"

gutterBottom

>

{stream.title}

</Typography>

<Typography variant="h5" color="inherit" paragraph>

{stream.description}

</Typography>

<Box pb={1} />

{showEdit && (

<Link href={`edit/${stream.\_id}`}>

<Button variant="outlined" color="inherit">

Edit Stream

</Button>

</Link>

)}

</div>

</Grid>

</Grid>

</Paper>

);

}

const useStyles = makeStyles((theme) => ({

toolbar: {

borderBottom: `1px solid ${theme.palette.divider}`,

},

toolbarTitle: {

flex: 1,

},

mainFeaturedPost: {

position: 'relative',

backgroundColor: theme.palette.grey[800],

color: theme.palette.common.white,

marginBottom: theme.spacing(4),

backgroundImage: 'url(https://source.unsplash.com/random)',

backgroundSize: 'cover',

backgroundRepeat: 'no-repeat',

backgroundPosition: 'center',

},

overlay: {

position: 'absolute',

top: 0,

bottom: 0,

right: 0,

left: 0,

backgroundColor: 'rgba(0,0,0,.7)',

},

mainFeaturedPostContent: {

position: 'relative',

padding: theme.spacing(3),

[theme.breakpoints.up('md')]: {

padding: theme.spacing(6),

paddingRight: 0,

},

},

}));

#### Content Component

On each stream's detail page, we will also display content based on the current stream's url, called Content.

Create a new file, app/components/Content.tsx, and insert the following:

import { makeStyles } from '@material-ui/core/styles';

type VideoProps = {

url: string;

};

export default function Video({ url }: VideoProps) {

const classes = useStyles();

return (

<div className={classes.container}>

<iframe

className={classes.iframe}

src={url}

frameBorder="0"

allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"

allowFullScreen

loading="lazy"

/>

</div>

);

}

const useStyles = makeStyles(() => ({

container: {

// overflow: 'hidden',

/\* 16:9 aspect ratio \*/

paddingTop: '56.25%',

position: 'relative',

},

iframe: {

border: '0',

height: '100%',

left: '0',

position: 'absolute',

top: '0',

width: '100%',

},

}));

With our components completed, we can move onto their supporting screens.

#### Streams Page

Create a new directory and file, app/pages/streams and app/pages/streams/index.tsx, and insert the following:

import { useEffect } from 'react';

import Container from '@material-ui/core/Container';

import Typography from '@material-ui/core/Typography';

import Box from '@material-ui/core/Box';

import Posts from 'components/Posts';

import { useStreamsQuery, Stream } from 'lib/graphql/streams.graphql';

export default function Streams() {

const { data, loading, refetch } = useStreamsQuery({ errorPolicy: 'ignore' });

useEffect(() => {

refetch();

}, []);

return (

<Container maxWidth="lg">

<Box my={4}>

<Typography variant="h4">Streams</Typography>

</Box>

{!loading && data && data.streams && (

<Posts streams={data.streams as Stream[]} />

)}

</Container>

);

}

#### Stream Detail Page

Create a new directory and file, app/pages/streams/[id] and app/pages/streams/[id]/index.tsx, and insert the following:

import Container from '@material-ui/core/Container';

import Hero from 'components/Hero';

import Content from 'components/Content';

import { useStreamQuery, Stream } from 'lib/graphql/stream.graphql';

export default function StreamDetail({ id }) {

const { data, loading } = useStreamQuery({

variables: { streamId: id },

});

if (!loading && data && data.stream) {

return (

<Container maxWidth="lg">

<Hero stream={data.stream as Stream} />

<Content url={data.stream.url} />

</Container>

);

}

return null;

}

StreamDetail.getInitialProps = ({ query: { id } }) => {

return { id };

};

#### Stream Create Page

Create a new file, app/pages/streams/new.tsx and insert the following:

import React, { useState } from 'react';

import { useRouter } from 'next/router';

import { useCreateStreamMutation } from 'lib/graphql/createStream.graphql';

import Container from '@material-ui/core/Container';

import TextField from '@material-ui/core/TextField';

import Typography from '@material-ui/core/Typography';

import Box from '@material-ui/core/Box';

import Button from '@material-ui/core/Button';

export default function CreateStream() {

const [title, setTitle] = useState('');

const [description, setDescription] = useState('');

const [url, setUrl] = useState('');

const router = useRouter();

// Signing In

const [createStream] = useCreateStreamMutation();

const onSubmit = async (event) => {

event.preventDefault();

try {

const { data } = await createStream({

variables: { input: { title, description, url } },

});

if (data.addStream.\_id) {

router.push('/streams');

}

} catch (err) {

console.log(err);

}

};

return (

<Container maxWidth="sm">

<Box my={4}>

<Typography variant="h4">Create Stream</Typography>

<form onSubmit={onSubmit}>

<Box pb={2.5} />

<TextField

autoFocus

label="Title"

value={title}

onChange={(e) => setTitle(e.target.value)}

required

/>

<Box pb={2.5} />

<TextField

label="Description"

value={description}

onChange={(e) => setDescription(e.target.value)}

required

/>

<Box pb={2.5} />

<TextField

label="URL"

value={url}

onChange={(e) => setUrl(e.target.value)}

required

/>

<Box pb={2.5} />

<Button type="submit" variant="contained" color="primary">

Create Stream

</Button>

</form>

</Box>

</Container>

);

}

#### Stream Edit Page

Create a new directory and file, app/pages/streams/edit/[id] and app/pages/streams/edit/[id]/index.tsx, and insert the following:

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

import { useRouter } from 'next/router';

import { initializeApollo } from 'lib/apollo';

import { useEditStreamMutation } from 'lib/graphql/editStream.graphql';

import { useDeleteStreamMutation } from 'lib/graphql/deleteStream.graphql';

import { StreamDocument } from 'lib/graphql/stream.graphql';

import Container from '@material-ui/core/Container';

import TextField from '@material-ui/core/TextField';

import Typography from '@material-ui/core/Typography';

import Box from '@material-ui/core/Box';

import Button from '@material-ui/core/Button';

export default function EditStream({ id }) {

const router = useRouter();

const [editStream] = useEditStreamMutation();

const [deleteStream] = useDeleteStreamMutation();

const [state, setState] = useState({

\_id: '',

title: '',

description: '',

url: '',

});

const { \_id, title, description, url } = state;

const fetchStream = async () => {

const apollo = initializeApollo();

const { data } = await apollo.query({

query: StreamDocument,

variables: { streamId: id },

});

setState(data.stream);

};

useEffect(() => {

fetchStream();

}, []);

const onSubmit = async (event) => {

event.preventDefault();

try {

const { data } = await editStream({

variables: { input: { id: \_id, title, description, url } },

});

if (data.editStream.\_id) {

router.push('/streams');

}

} catch (err) {

console.log(err);

}

};

const onDelete = async (event) => {

event.preventDefault();

try {

const { data } = await deleteStream({

variables: { id },

});

if (data.deleteStream) {

router.push('/streams');

}

} catch (err) {

console.log(err);

}

};

return (

<Container maxWidth="sm">

<Box my={4}>

<Typography variant="h4">Edit Stream</Typography>

<form onSubmit={onSubmit}>

<Box pb={2.5} />

<TextField

autoFocus

label="Title"

value={title}

onChange={(e) => setState({ ...state, title: e.target.value })}

required

/>

<Box pb={2.5} />

<TextField

label="Description"

value={description}

onChange={(e) =>

setState({ ...state, description: e.target.value })

}

required

/>

<Box pb={2.5} />

<TextField

label="URL"

value={url}

onChange={(e) => setState({ ...state, url: e.target.value })}

required

/>

<Box pb={2.5} />

<Button type="submit" variant="contained" color="primary">

Save

</Button>

<Box pb={2.5} />

<Button onClick={onDelete} variant="contained">

Delete

</Button>

</form>

</Box>

</Container>

);

}

EditStream.getInitialProps = ({ query: { id } }) => {

return { id };

};

#### Test Streaming

It's time to fire up the Next.js app, GraphQL server and test the new streams flow at <http://localhost:3000/streams>

Heads Up! Your GraphQL API server needs to be running in order to test the frontend.

npm run dev

Try reloading the Next.js application, and you will be able to navigate to a "Create Stream" page.

Be sure you are able to:

* Create a new stream
* View your list of streams
* View an individual stream

#### **Congratulations!**

Today we built data-fetching queries and created the streaming flow. Up next, we will prepare our application for deployment.

#### References

* [Next Router](https://nextjs.org/docs/api-reference/next/router)
* [useState Hook](https://reactjs.org/docs/hooks-state.html)

# Chapter 8: Deployment

## Deployment

### By the end of this lesson, developers will be able to:

* Create a monorepo for their project using yarn workspaces
* Deploy their project to Heroku using the Heroku CLI

### Introduction

So far, our project is using what's known as a BFF (Backend for Frontend) style of architecture. The app communicates with an api directly, and both are running separately.

In this chapter, we will introduce yarn workspaces to run both projects at the same time.

Afterwards, we will deploy the project to Heroku using the Heroku CLI.

### Installation

Visit the installation [guide](https://classic.yarnpkg.com/en/docs/install/) for your respective operating system. Once you are all set, be sure to test your local version with the following command:

yarn --version

Once you verify the local version appears, you may continue.

### Workspaces

Workspaces make it possible to work on multiple, interdependent libraries at once. For more information on workspaces, visit the [documentation](https://classic.yarnpkg.com/en/docs/workspaces/).

Create a new file in your root directory, package.json and insert the following:

{

"name": "stream-me",

"version": "1.0.0",

"description": "",

"main": "index.js",

"private": true,

"scripts": {

"build": "yarn workspace @stream-me/app run build",

"start": "yarn workspace @stream-me/api run start",

"dev": "yarn workspace @stream-me/api run dev"

},

"workspaces": [

"api",

"app"

],

"keywords": [],

"author": "",

"license": "ISC"

}

Our workspace references both api and app projects. We also include scripts to build and start each project, by referencing their namespaces.

### Packages

Before creating the monorepo, we will need to rename both api and app projects in their respective package.json files:

Return to app/package.json and insert the following:

{

"name": "@stream-me/app",

}

Return to api/package.json and insert the following:

{

"name": "@stream-me/api",

}

At this point, we simply run yarn install in the root directory to install both projects' dependencies:

yarn install

After the installation completes, you may notice @stream-me in your node\_modules directory, with some symbolic links to each project. Well done!

### App Updates

#### Build Command

Let's go ahead and add the graph-let command to our app project's package.json.

Visit the scripts section of the file named app/package.json, and insert the following:

"scripts": {

"dev": "next dev",

"build": "npx graphql-let && next build",

"start": "next start"

},

We are adding npx graphql-let to the build command, so that with each build we are able to generate new GraphQL queries.

### Apollo Client

Back in app/lib/apollo.ts, remove the absolute URL pointing to http://localhost:8000/graphl, and replace it with a relative URL: /graphql.

function createApolloClient() {

{/\* ... \*/}

const httpLink = new HttpLink({

uri: '/graphql',

credentials: 'include',

});

{/\* ... \*/}

}

### Next Project

Create a new file in your app directory, app/index.ts and insert the following:

import next from 'next';

const nextApp = next({

dev: process.env.NODE\_ENV !== 'production',

dir: \_\_dirname,

});

export default nextApp;

We are exporting the entire Next.js application as a module, which will be served by the Express server.

### Next Custom Server

In order to serve the frontend app on the backend, we will prepare a Next.js custom server.

Make the following changes inside api/server/index.ts:

import nextApp from '@stream-me/app';

const handle = nextApp.getRequestHandler();

{/\* ... \*/}

const corsOptions = {

credentials: true,

};

{/\* ... \*/}

apolloServer.applyMiddleware({ app, cors: corsOptions });

// create next app request handler

await nextApp.prepare();

app.get('\*', (req, res) => handle(req, res));

In the above code snippet, we serve the frontend Next.js application with a custom route handler. At this point, we are ready to test the application using the following commands:

Heads Up! These commands should be ran in your project's root directory.

npm run dev

npm run build

npm run start

Revisit your project at <http://localhost:8000/graphql> and verify your queries are working. Be sure to visit your homepage at [http://localhost:8000](http://localhost:8000/).

### Git Ignore

Before moving forward, add the following to your root level .gitignore file:

node\_modules

.DS\_Store

### Heroku

Make sure you download the [heroku cli](https://devcenter.heroku.com/articles/heroku-cli" \l "download-and-install) and run heroku login

heroku create your-heroku-app-name

git status

git add .

git commit -m "update project"

git remote add heroku <your-heroku-remote-url>

git push heroku master

Having issues? Debug with the Heroku command heroku logs --tail to see what's happening on the Heroku server.

Test the endpoints!

<https://your-heroku-app-name.herokuapp.com/graphql>

You may notice everything breaks unless we add the missing environment variables: MONGO\_URL. We will need to visit the new project's settings on [https://heroku.com](https://heroku.com/) to add these missing environment variables.

### **Congratulations!**

Today we built a monorepo using yarn workspaces and deployed the project to Heroku. Well done!

### References

* [Yarn Workspaces](https://classic.yarnpkg.com/en/docs/workspaces/)
* [Next.js Custom Server](https://nextjs.org/docs/advanced-features/custom-server)