21.3.6 Set Up Full-Stack Structure with Client-Side React App

As you've learned by now, developing a MERN-stack application can be cumbersome, requiring separate servers for the front and back ends. We need the back-end server, of course, to serve the API data. And we need the front-end server mainly to enhance the development process using webpack, as the production-level codebase doesn't require a server to run. In fact, when we deploy this application, we'll serve up the production-ready React front end as a route for the Express server!

So far, no one has offered an industry-standard solution for this issue. But some developers have shared their processes with the development community for others to use. We'll use one such process now.

First, let's address an issue that may arise in the client-side codebase. In the `App.js` file, we use the absolute path to the server when we initialize the connection using the `ApolloClient()` constructor. This served our needs well, allowing us to retrieve and display thought data from the server—but will it still be `http://localhost:3001/graphql` in production?

The answer is no. So as it stands now, we'd have to remember to change the absolute path every time we push it to production. This could lead to frustrating errors, which is why the Create React App team created a fix for this exact issue.

In the `App.js` file of the `client` folder, change the connection URI value so the entire code block looks as follows:

```
const client = new ApolloClient({
  uri: '/graphql'
});
```

> **NERD NOTE**
>
> URI stands for "Uniform Resource Identifier."

Save `App.js` and then open the `package.json` file in the `client` directory. Once that's open, add one more key-value pair towards the top of the JSON object that looks like the following code:

```
"proxy": "http://localhost:3001",
```

With this `proxy` value in place, the Create React App team set up the development server to prefix all HTTP requests using relative paths (e.g., `/graphql` instead of `http://localhost:3001/graphql`) with whatever value is provided to it. Now the HTTP requests will work in both development and production environments!

To see this fix in action, feel free to stop and restart the development server in the `client` directory using `npm start`. Unfortunately, because `package.json` isn't part of the actual React application, the Hot Module Replacement aspect of the webpack development server doesn't listen for changes in that file. Thus, any change to the `package.json` file will always necessitate a manual restart of the server from the command line.

DEEP DIVE

DEEP DIVE

To learn more, read the **Create React App documentation for proxying API requests (https://create-react-app.dev/docs/proxying-api-requests-in-development/)**.

Now we'll set up some tools that will allow us to control both the `client` and `server` functionality with one command-line window!

Set Up Application Root Functionality

Take a moment and shut down any running servers right now. You should have two command-line windows open, running each folder's process, but you won't need that after these next additions.

Once you've quit both the `client` and `server` processes, use the command line to navigate to the root of the application, which is one directory outside of either the `client` or `server` folders.

HIDE HINT

To confirm you're in the right location, use `pwd` to see if you're at the root of the `deep-thoughts` repository folder.

Because we have two servers for two different purposes, we don't want to actually combine anything between the two—mainly because if we were to ever use this client or server codebase elsewhere, we could easily take the entire folder and move it to another location.

Instead, we'll create a third application using npm. This application will simply control the other two applications. This may sound a little confusing at first, but you'll soon understand how they'll all work in unison.

At the root directory of `deep-thoughts`, create a new application using `npm init` or `npm init -y`. Once that's complete, we need to install a couple of dependencies. One will be a regular dependency; the other will be a development dependency. Run the following commands:

```
# install if-env library
npm install if-env

#install concurrently library as a dependency for development environment on
npm install -D concurrently
```

The `if-env` tool is used not through JavaScript itself but rather in the `package.json` file. It'll check what environment we're in and execute a select npm script based on the response.

We can use the `concurrently` package to run multiple processes, or servers, from a single command line interface. This way we don't have to remember if we opened up a second command-line window to start the other server; we can simply do them both at the same time.

Notice the `-D` flag in the second command. That instructs that the dependency listed should be only installed in a development environment and not used in production. This will keep the overall `node_modules` folder size a bit smaller for production, which makes the application take up less space!

Lastly, we need to add some scripts to this root-level `package.json` file to put these tools to use. Let's open that file and use the following scripts:

```
"scripts": {
  "start": "if-env NODE_ENV=production && npm run start:prod || npm run star
  "start:prod": "cd server && npm start",
  "start:dev": "concurrently \"cd server && npm run watch\" \"cd client && n
  "install": "cd server && npm i && cd ../client && npm i",
  "seed": "cd server && npm run seed"
},
```

All these scripts may seem confusing at first glance, so let's review them. From the command line, when we run `npm start`, we employ `if-env` to check if the Node environment is production. If it is, we'll run the `npm run start:prod` command; otherwise, we'll run the `npm run start:dev` command.

The `start:prod` script will be used in production. It simply navigates to the `server` directory and runs the server's `npm start` command. This is good, because in production we can only run one server!

We'll use the `start:dev` script as we continue to build this application. We use `concurrently` to run two separate commands, each one wrapped in its own quotation marks. Note the escape characters around the scripts' quotes `\"`. JSON requires double quotes, and to use double quotes within a string, we need to use the backslash `\` escape character before it.

The script will run the `npm run watch` script in the `server` directory and the `npm start` script in the `client` directory, allowing us to use both servers at the same time from a single command-line window.

The `install` script is interesting, as it's already a built-in script that works well. But with this one, we can use `npm i` or `npm install` to first install all of the dependencies at this root level. Then it navigates to the `server` directory and installs those dependencies. Lastly, it navigates to the `client` directory to install those dependencies as well.

Last, we have the `seed` script, which will navigate to the `server` directory and seed the database using the `npm run seed` command. This will be useful if we ever want to seed the database in a production environment, which we'll touch on later.

IMPORTANT

> Your application will work regardless, but it's good practice to update this `package.json` file's `main` field to be your main server file, so set it to be `server/server.js`!

Now there's no need to keep navigating in and out of the `client` or `server` directories, as we can run all of the commands here! We'll only need to go directly into one of those directories if we have to install a new dependency to the respective location.

Last, we need to update the back-end server's code to serve up the React front-end code in production. Let's do that now.

In the `server.js` file inside of the `server` directory, let's import the `path` module by placing the following code at the top of the file:

```
const path = require('path');
```

Then, towards the bottom of the file but above the `db.once()` code, add the following code:

```
// Serve up static assets
if (process.env.NODE_ENV === 'production') {
  app.use(express.static(path.join(__dirname, '../client/build')));
}

app.get('*', (req, res) => {
```

```
  res.sendFile(path.join(__dirname, '../client/build/index.html'));
});
```

We just added two important pieces of code that will only come into effect when we go into production. First, we check to see if the Node environment is in production. If it is, we instruct the Express.js server to serve any files in the React application's `build` directory in the `client` folder. We don't have a `build` folder yet—because remember, that's for production only!

The next set of functionality we created was a wildcard GET route for the server. In other words, if we make a GET request to any location on the server that doesn't have an explicit route defined, respond with the production-ready React front-end code.

We can't necessarily see or test this at the moment, because we aren't in production, but it will set us up for later when we deploy to Heroku.

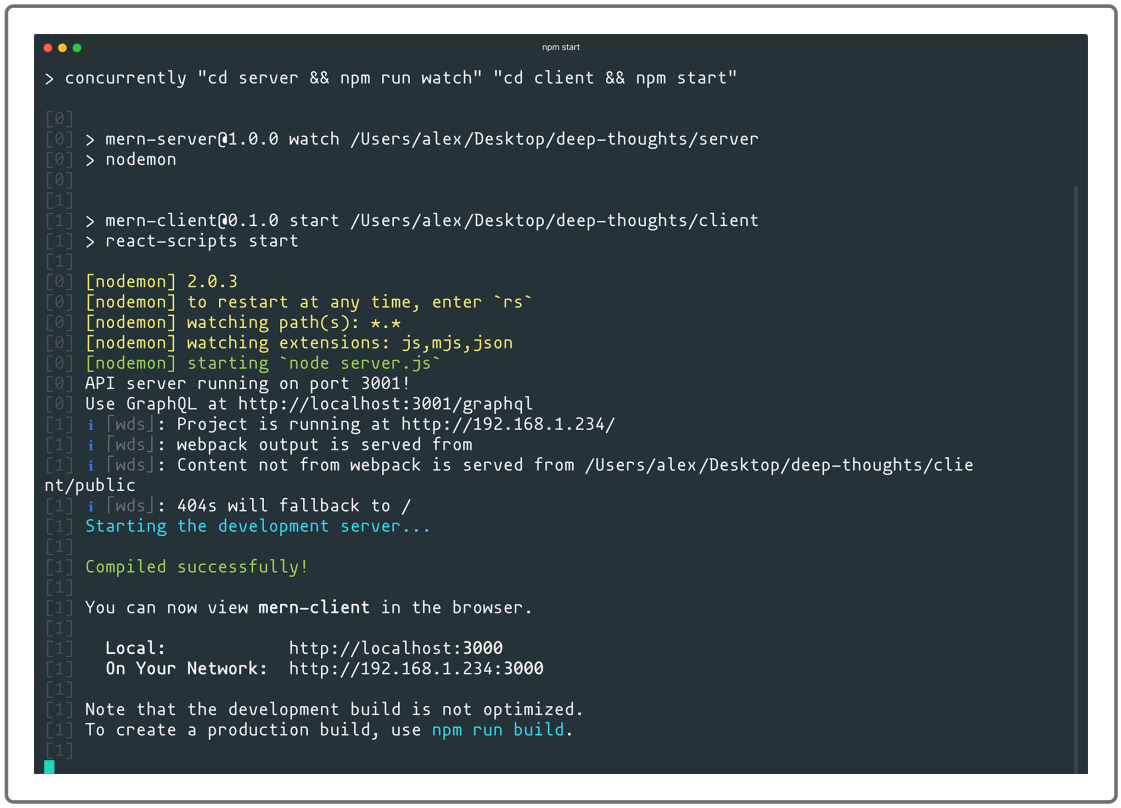
We can, however, test to see if those scripts we employed at the root of the application work. Let's do that now!

Run the Development Scripts

At the command prompt, navigate to the root of the `deep-thoughts` directory, and start up the application with the following command:

```
npm start
```

Once that's up and running, your command line should look a bit like the following image:



Notice the `0`'s and `1`'s being printed alongside the command-line output? That's concurrently's way of indicating which process is doing what. The process labeled with a `0` will be the `server` output, and the process labeled with a `1` will be the `client` output. Keep this in mind as you develop. It's easy to lose track when we have so much going on.

If everything starts up okay, you should see your application open in the browser and populate with thought data, much like before! If you have any issues or errors, navigate through the command line and see if either process failed to start up on load.

IMPORTANT

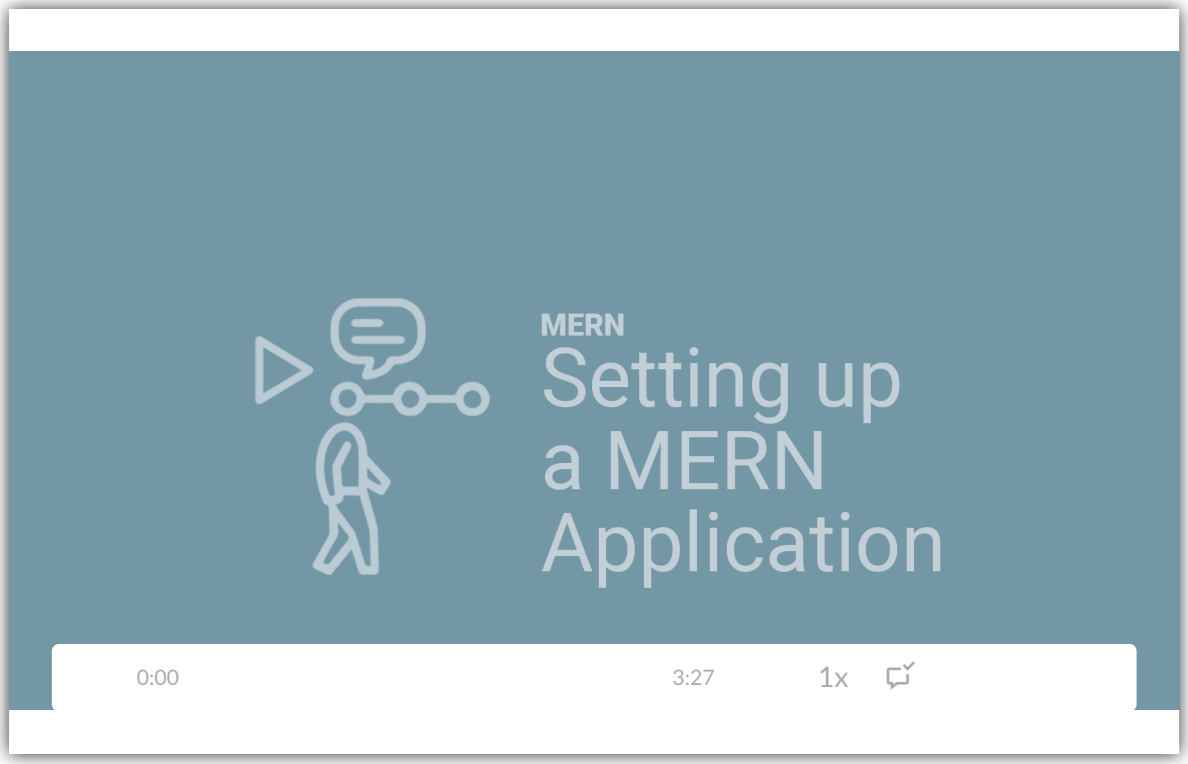
> It's easy to miss on startup that one process may have failed using concurrently. The client may still load and make you think everything is working, but not communicate with the server because it failed on startup.

> Also feel free to not use concurrently if you're running into too many issues with it. Computers are sometimes fickle, and you don't want to waste all of your time on the tooling aspect of it if you know it'll work in other ways!

Great job setting all of this up. The application can now communicate across both servers, all from one command! This will allow us to focus on actually writing the application's code instead of having to spend too much time on running separate processes.

The best part is, this root-level application set of tooling we just employed can be used for any MERN-stack application. So be sure to study exactly how this works and use it for any future MERN applications you build!

For a more visual explanation of this setup, watch the following walkthrough video that recaps exactly what we just did:



Let's take a moment to stop the application, save the feature branch, push it to GitHub, and merge to `develop`. Let's also close the GitHub issue associated with it.

We covered a lot of ground in this lesson, so take a moment to put the new knowledge to use with this assessment:

Which npm package is NOT required to set up Apollo and GraphQL in React?

- `@apollo/react-hooks`
- `apollo-boost`
- `graphql-tag`
- `apollo-server-express` ✔

Response-specific feedback
Yes, this is a back-end dependency and is not required to set up Apollo and GraphQL in React.

Which React Hook would you use to retrieve data from an Apollo server?

- `useEffect`
- `useQuery` ✔
- `useState`
- `useContext`

Response-specific feedback
Yes, the `useQuery` Hook is used to execute GraphQL queries.

Given the following `package.json` file, what would print at the command line if the environment were set to "staging" and the app were started with `npm start`?

```
"scripts": {
  "start": "if-env NODE_ENV=production && npm run start:prod || npm run start:dev",
  "start:prod": "echo \"Start production server\"",
  "start:dev": "echo \"Start development server\"",
  "start:stage": "echo \"Start staging server\"",
}
```

- Start development server ✔
- Start staging server
- Start production server

Response-specific feedback
Yes, if the environment is anything other than production, the command `npm run start:dev` will run.

Finish ▶