React Jobly

Springboard

Download starter code.

In this sprint, you'll create a React front end for the Jobly backend.

Step Zero: Setup

- **The backend for this will be our solution to the *express-jobly* exercise.**

  You can find this in the starter code.

  Use this instead of the backend you built for jobly — ours is feature-complete with what the front-end will need.

- Re-create the ***jobly*** database from the backend solution using the ***jobly.sql*** file.

  Note: even if you have a jobly database from previously, you'll find it helpful to replace it with ours — we have lots of sample data, which will help you test the app.

- Create a new React project.
- It may help to take a few minutes to look over the backend to remind yourself of the most important routes.
- Start up the backend. We have it starting on port 3001, so you can run the React front end on 3000.

Step One: Design Component Hierarchy

It will help you first get a sense of how this app should work.

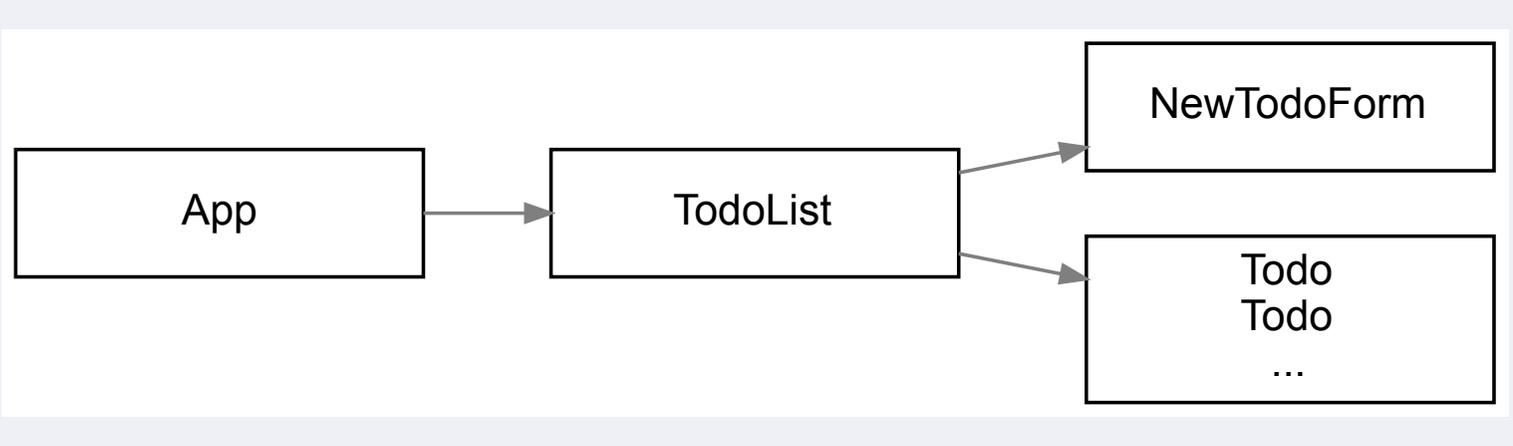
We have a demo running at http://joelburton-jobly.surge.sh. Take a tour and note the features.

Please register as a new user to explore the site.

A big skill in learning React is to learn to design component hierarchies.

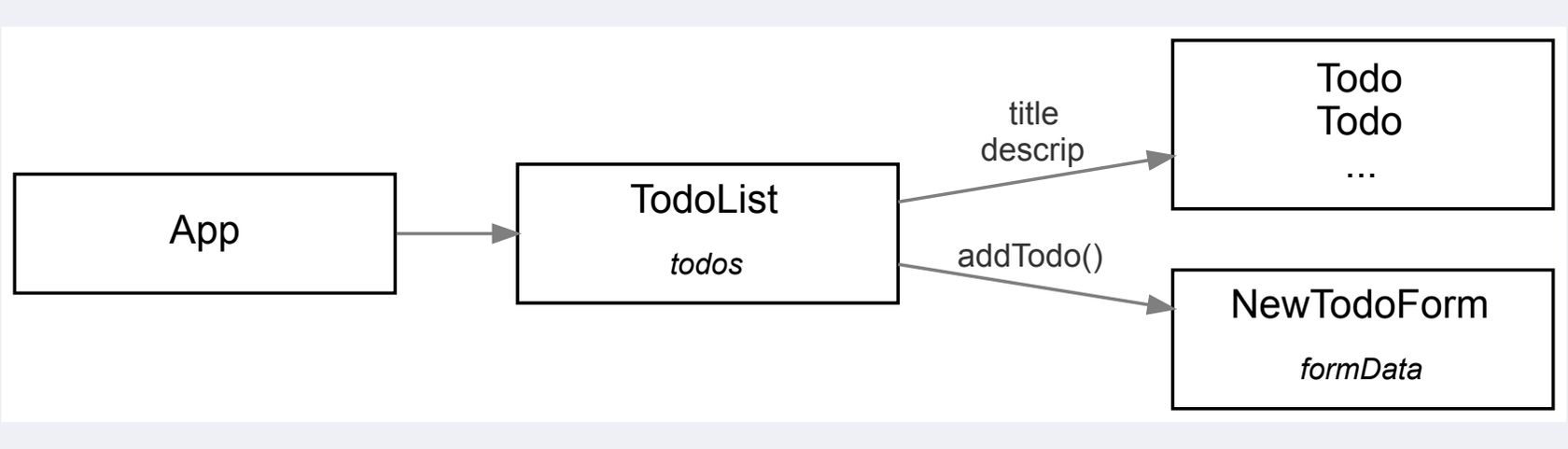
It can be very helpful to sketch out a hierarchy of components, especially for larger apps, like Jobly.

As an example of this kind of diagram, here's one for a sample todo list application:



Once you've done this, it's useful to think about the props and state each component will need. Deciding *where* individual state is needed is one of the most critical things to figure out.

Here's our simple todo list application, with component state and passed props:



We're showing these to you with diagrams, and it can be helpful to do this with pen and paper or using a whiteboard.

You can also write this out as an indented list:

```
App                           General page wrapper
  no props or stage

  TodoList                    Manages todos, shows form &
    state=todos                 list of todos

    NewTodoForm               Manages form data, submits
      state=formData            new todo to parent
      props=addTodo()

    Todo                      One rendered for each todo,
      props=title, descrip      pure presentational
```

Take time to diagram what components you think you'll need in this application, and what the most important parts of state are, and where they might live.

Notice how some things are common: the appearance of a job on the company detail page is the same as on the jobs page. You should be able to re-use that component.

Spend time here. This may be one of the most important parts of this exercise.

Step Two: Make an API Helper

Many of the components will need to talk to the backend (the company detail page will need to load data about the company, for example).

It will be messy and hard to debug if these components all had AJAX calls buried inside of them.

Instead, make a single ***JoblyAPI*** class, which will have helper methods for centralizing this information. This is conceptually similar to having a model class to interact with the database, instead of having SQL scattered all over your routes.

Here's a starting point for this file:

api.js

```
import axios from "axios";

const BASE_URL = process.env.REACT_APP_BASE_URL || "http://localhost:3001";

/** API Class.
 *
 * Static class tying together methods used to get/send to to the API.
 * There shouldn't be any frontend-specific stuff here, and there shouldn't
 * be any API-aware stuff elsewhere in the frontend.
 *
 */

class JoblyApi {
  // the token for interactive with the API will be stored here.
  static token;

  static async request(endpoint, data = {}, method = "get") {
    console.debug("API Call:", endpoint, data, method);

    //there are multiple ways to pass an authorization token, this is how you pass it in the header.
    //this has been provided to show you another way to pass the token. you are only expected to read this code for this project.
    const url = `${BASE_URL}/${endpoint}`;
    const headers = { Authorization: `Bearer ${JoblyApi.token}` };
    const params = (method === "get")
        ? data
        : {};

    try {
      return (await axios({ url, method, data, params, headers })).data;
    } catch (err) {
      console.error("API Error:", err.response);
      let message = err.response.data.error.message;
      throw Array.isArray(message) ? message : [message];
    }
  }

  // Individual API routes

  /** Get details on a company by handle. */

  static async getCompany(handle) {
    let res = await this.request(`companies/${handle}`);
    return res.company;
  }

  // obviously, you'll add a lot here ...
}

// for now, put token ("testuser" / "password" on class)
JoblyApi.token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZ" +
    "SI6InRlc3R1c2VyIiwiaXNBZG1pbiI6ZmFsc2UsImlhdCI6MTU5ODE1OTI1OX0." +
    "FtrMwBQwe6Ue-glIFgz_Nf8XxRT2YecFCiSpYL0fCXc";
```

You won't build authentication into the front end for a while—but the backend needs a token to make almost all API calls. Therefore, for now, we've hard-coded a token in here for the user "testuser", who is also in the sample data.

(Later, once you start working on the login form, you may find it useful to log in as "testuser". Their password is "password").

You can see a sample API call — to ***getCompany(handle)***. As you work on features in the front end that need to use backend APIs, add to this class.

Step Three: Make Your Routes File

Look at the working demo to see the routes you'll need:

/
Homepage — just a simple welcome message

/companies
List all companies

/companies/apple
View details of this company

/jobs
List all jobs

/login
Login/signup

/signup
Signup form

/profile
Edit profile page

Make your routes file that allows you to navigate a skeleton of your site. Make simple placeholder components for each of the feature areas.

Make a navigation component to be the top-of-window navigation bar, linking to these different sections.

When you work on authentication later, you need to add more things here. But for now, you should be able to browse around the site and see your placeholder components.

Step Four: Companies & Company Detail

Flesh out your components for showing detail on a company, showing the list of all companies, and showing simple info about a company on the list (we called these ***CompanyDetail***, ***CompanyList***, and ***CompanyCard***, respectively —but you might have used different names).

Make your companies list have a search box, which filters companies to those matching the search (remember: there's a backend endpoint for this!). Do this filtering in the backend — **not** by loading all companies and filtering in the front end!

Step Five: Jobs

Similarly, flesh out the page that lists all jobs, and the "job card", which shows info on a single job. You can use this component on both the list-all-jobs page as well as the show-detail-on-a-company page.

Don't worry about the "apply" button for now — you'll add that later, when there's authentication for the app.

Step Six: Current User

This step is tricky. Go slowly and test your work carefully.

Add features where users can log in, sign up, and log out. This should use the backend routes design for authentication and registration.

When the user logs in or registers, retrieve information about that user and keep track of it somewhere easily reached elsewhere in the application.

Things to do:

- Make forms for logging in and signing up
- In the navigation, show links to the login and signup forms if a user is not currently logged in.

  If someone is logged in, show their username in the navigation, along with a way to log out.

- Have the homepage show different messages if the user is logged in or out.
- When you get a token from the login and register processes, store that token on the ***JoblyApi*** class, instead of always using the hardcoded test one. You should also store the token in state high up in your hierarchy; this will let use use an effect to watch for changes to that token to kick off a process of loading the information about the new user.

Think carefully about where functionality should go, and keep your components as simple as you can. For example, in the ***LoginForm*** component, its better design that this doesn't handle directly all of the parts of logging in (authenticating via API, managing the current user state, etc). The logic should be more centrally organized, in the ***App*** component or a specialized component.

While writing this, your server will restart often, which will make it tedious to keep typing in on the login and signup forms. A good development tip is to hardcode suitable defaults onto these forms during development; you can remove those defaults later.

Hover to reveal

Step Seven: Using localStorage and Protecting Routes

If the user refreshes their page or closes the browser window, they'll lose their token. Find a way to add ***localStorage*** to your application so instead of keeping the token in simple state, it can be stored in localStorage. This way, when the page is loaded, it can first look for it there.

Be thoughtful about your design: it's not great design to have calls to reading and writing localStorage spread around your app. Try to centralize this concern somewhere.

As a bonus, you can write a generalized ***useLocalStorage*** hook, rather than writing this tied specifically to keeping track of the token.

Protecting Routes

Once React knows whether or not there's a current user, you can start protecting certain views! Next, make sure that on the front-end, you need to be logged in if you want to access the companies page, the jobs page, or a company details page.

Step Eight: Profile Page

Add a feature where the logged-in user can edit their profile. Make sure that when a user saves changes here, those are reflected elsewhere in the app.

Step Nine: Job Applications

A user should be able to apply for jobs (there's already a backend endpoint for this!).

On the job info (both on the jobs page, as well as the company detail page), add a button to apply for a job. This should change if this is a job the user has already applied to.

Step Ten: Deploy your Application

We're going to use Heroku to deploy our backend and Surge to deploy our frontend! Before you continue, make sure you have two folders, each with their own git repository (and make sure you do not have one inside of another!)

Your folder structure might look something like this

```
jobly-backend
jobly-frontend
```

It's important to have this structure because we need two different deployments, one for the front-end and one for the backend.

Backend

Make sure you are running the following commands in the ***jobly-backend*** folder — *do not copy and paste these commands!**

```
$ heroku login
$ heroku create NAME_OF_APP
$ echo "web: node server.js" > Procfile
$ heroku git:remote -a NAME_OF_APP
$ git add .
$ git commit -m "ready to deploy backend"
```

These commands will create a web application and the ***Procfile*** which tells Heroku what command to run to start the server.

Now that you have a remote named, run the following commands in the **jobly-backend** folder. We're going to push our code to Heroku and copy our local database (which we have named **jobly**) to the production one (so that we can have a bunch of seed data in production)

```
$ git push heroku master
$ heroku addons:create heroku-postgresql:hobby-dev -a NAME_OF_APP
$ heroku pg:push jobly DATABASE_URL -a NAME_OF_APP
$ heroku config:set PGSSLMODE=no-verify
$ heroku open
```

If you are getting any errors, make sure you run ***heroku logs -t -a NAME_OF_APP***

Frontend

Now let's deploy the frontend! To do that, we're going to be using a tool called Surge, which is a very easy way to deploy static websites!

Make sure that you have the ***surge*** command installed. You can run this command anywhere in the Terminal:

```
$ npm install --global surge
```

In your ***JoblyApi.js*** and **anywhere else you make requests to localhost:3001** make sure you have the following:

```
const BASE_URL = process.env.REACT_APP_BASE_URL || "http://localhost:3001";
```

Next, let's make sure we define the environment variable **for our frontend app**. YOUR_HEROKU_BACKEND_URL should be something like ***https://YOUR_BACKEND_APP_NAME.herokuapp.com***.

Make sure you are running the following commands in the ***jobly-frontend*** folder

```
$ REACT_APP_BASE_URL=YOUR_HEROKU_BACKEND_URL npm run build
$ cp build/index.html build/200.html
$ surge build
```

Further Study

There's already plenty here! But if you do finish early, or want to learn more by continuing to work on this, we have some suggestions for Further Study

Solution

View our solution