React Jobly

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

Step Zero: Setup

- Download the starter code. Note: this is the backend. You'll start the front end yourself.

  Download <../react-jobly.zip>

- Create a database, ***jobly***, and use the seed data provided. You can run the seed file using `psql jobly < data.sql`

  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.

- We've provided a backend for this. **Use this instead of the backend you may already have** — you'll find it easier to work on this project with our tested and consistent backend.

- 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.

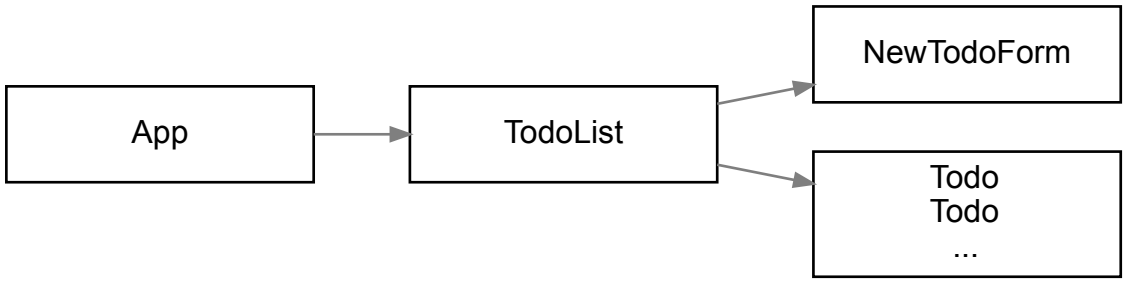
We have a demo running at https://jobly-app.herokuapp.com/ <https://jobly-app.herokuapp.com/>. Take a tour and note the features.

You can register as a new user and explore the site or log in as our test user, "rithmtest" (password: "rithmtest").

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

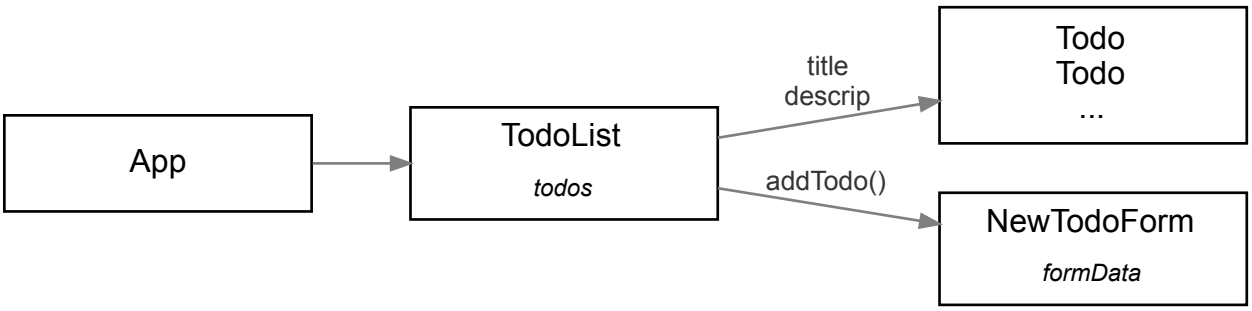
It can be very helpful to sketch out a hierarchy of components, especially for larger apps, like Jobly.

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.

> **Warning: STOP AND GET A REVIEW**
>
> We'd really like to see your thinking here.
>
> Once you've received your review, we **strongly suggest** you look at our solution's component design <our-design.html>.

> You don't *have* to design your app like ours, but we think it may be useful for you to match our component structure. This will make it easier for read our solution if you'd like help from there.

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:

src/JoblyApi.js

```
class JoblyApi {
  static async request(endpoint, params = {}, verb = "get") {
    console.debug("API Call:", endpoint, params, verb);

    const _token = ( // for now, hardcode token for "testing"
        "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InRlc" +
        "3RpbmciLCJpc19hZG1pbiI6ZmFsc2UsImlhdCI6MTU1MzcwMzE1M30." +
        "COmFETEsTxN_VfIlgIKw0bYJLkvbRQNgO1XCSE8NZ0U");

    const data = (verb === "get")
      ? { params: { _token, ...params } } // GET
      : { _token, ...params };            // POST, PATCH

    const req = axios[verb](`${BASE_URL}/${endpoint}`, data);

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

  // Indiv API routes

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

```
    }
}
```

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 "secret").

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.

> **Warning: STOP AND GET A CODE REVIEW**
>
> Please let us see your code at this point!

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 searchbox, 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.

> **Warning: STOP AND GET A CODE REVIEW**
>
> Please let us see your code at this point!

Step Six: Authentication

FIXME: remove localStorage parts from this, since that's in next section.

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

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, setting token, etc). The logic should be more centrally organized, in the ***App*** component or a specialized authentication 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.

You need to add the following to your app:

- A route that that lets the user log in. This will use the login endpoint on the server.

  Rather than keeping this login token in React state, store in your browser's ***localStorage***. This way, if the user refreshes their page or closes the the browser window, they'll stay logged in (more on this in the next step).

Edit the ***JoblyApi*** file to extract the token from ***localStorage***, rather than using that hardcoded “testuser” token.

Make your navigation bar only show only the login link if the user isn’t logged in. Make it show a “logout” link when they are, along with the other links.

- The signup process is similar to the login process: the fields gathered are different, and the backend endpoint is different, but the process is the same: call the endpoint, get the token, and store in localStorage.
- Have homepage show login and signup buttons if the user isn’t logged in.
- Have the navigation bar show either login/signup or logout buttons.

  Figure out how logout should work.

> **Warning: STOP AND GET A CODE REVIEW**
>
> Please let us see your code at this point!

Step Seven: Protecting Routes

FIXME: split these into two steps: protecting routes, and remembering token in localStorage.

Once you can log in and sign up, a new problem emerges: what happens when you hard refresh the page? You need to make sure the app can recover your login status.

To handle this problem in your top-level ***App*** component, add a ***localStorage*** check inside of ***useEffect***. If there’s a valid token in ***localStorage***, then ping the API to get all of the information on the current user and store it in the ***App*** component’s state. This will let you pass current info down as a prop to any descendant component, too.

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.

As a bonus, you can write a ***useLocalStorage*** hook to manage the user data in local storage!

Step Eight: Remebering Login Token in localStorage

TODO: This should be moved here

> **Warning: STOP AND GET A CODE REVIEW**
>
> Please let us see your code at this point!

Step Nine: Profile Page

Add a feature where the logged-in user can edit their profile.

Step Ten: 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.

> **Warning: STOP AND GET A CODE REVIEW**
>
> Please let us see your code at this point!

Step Eleven: 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 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
```

> **Warning: STOP AND GET A CODE REVIEW**
>
> Congratulations! You've reached the end of the main part of the exercise. Please let us take a look at your work and provide a code review.

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 <./further-study.html>

Solution

View our solution <./solution>