Two server react deployment

With Heroku!

This is what I saw a bunch of students in g44 use with success. Also separate architecture like this is something I have seen used widely with success. Wheter or not you use two server architecture in your own projects, you need to be exposed to it to prepare you in your future career.

Recall that the react client side that you make with `create-react-app` and run with yarn start is actually an app that translates `.jsx` files and other assets into something for a web browser. Therefore, it can be done on a standalone server.

Also, when you create an express app with `express` generator, it make a stand alone server as well.

You can deploy full stack apps with one or two servers. In the two server configuration, one server handles all the webpack stuff to package for a browser. The other runs the API. It is also possible to do this with one server, but that is not covered here. The checklist for the single server configuration is very different.

I will deploy both to Heroku here. *Using Heroku enforces a way of structuring our application* . Other deployment enviroments, such as Amazon Web Services, would offer more options—but at the cost of added complexity.

Also, I will use `yarn` for the whole process, thought the same things can be done with `npm` if you like.

As a side note, deploying separate servers allows you to see what can be done to support React apps in a large environment. Your react apps may have to hit 1, 2, 5, or more APIs *at once* . Time to get ready!

What this document does *not* cover

This documentation does not cover authentication, authorization, or any other security topics.

Initial deployment

You are going to deploy *both* the client and server (API) sides of the app to Heroku right away. Remember, an app doesn't work until it is deployed. By doing deployment early and often, you can get things issues out of the way before they become problems.

API / Server Side: Locally and on Heroku

- Create a repo for the server / API side.
- Create an an express app with the `express` generator. Make sure it is at the root of your repo.
- Add the CORS headers by placing these before the routes. You may need to customize this later, because this is not totally secure, but this will get you going. (`res.header('Access-Control-Allow-Origin', '*')` can have tighter security) You probably need it in `app.js`

```
app.use(function (req, res, next) {
  res.header('Access-Control-Allow-Origin', '*')
  res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept')
  res.header('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, DELETE')
  next()
})

```

- Quick note: in `./bin/www` you probably want a port different than 3000. Something like 8181 for example.

```
var port = normalizePort(process.env.PORT || '8181');
```

- Add a note to remind yourself of 8181

```
function onListening() {
  var addr = server.address();
  var bind = typeof addr === 'string'
    ? 'pipe ' + addr
    : 'port ' + addr.port;
  console.log('Listening on ' + bind);
}
```

- Add a simple hello world route to your server / API app. This will just return some basic JSON. Test this locally on your own computer using httpie, Postman, cURL or another tool of your choice.
- Log into the Heroku web interface and create an app with the name you want for the frontend.
- Then make your express `Procfile` for the server / API side. (This is not needed for the client side below. **Place this is at the root of your server / API repo**

```
echo ‘web: node ./bin/www’ > Procfile
```

- Check your `node` version with

```
node --version
```

- Then make sure you specify your `node` version in your `package.json`

```
“engines”: {
    “node”: “[YOUR NODE VERSION]”
}
```

- Also give a start script in your `package.json`

```
“start”: “node ./bin/www”
```

- Make sure that you have PostgreSQL support installed!

```
heroku addons:create heroku-postgresql
```

- Then do `heroku login`
- Then do `heroku git:remote -a [YOUR APP NAME]`
- Then do `git push heroku master`
- Then do `heroku open`
- Now, test your Heroku deployment. with cURL, Postman or your choice of tool.

Client side: both locally and on Heroku

- Create a separate repo for your user side.
- Create a front-end app with `create-react-app`

- Fire up your test react app locally and navigate to it locally. Make sure that it works.
- When you find that working:

```
heroku login
```

- Now we need to get the buildpack for `create-react-app`

```
heroku create -b https://github.com/mars/create-react-app-buildpack.git
```

- **By default, this will create the heroku git remote for you** and a default app name
- So now you can do

```
git push heroku master
```

- Login to the Heroku web UI and find the name of your app.
- You can also run

```
heroku open
```

Optional: Yikes! I hate the default name

Don't panic! After running this buildpack, you can rename your project from the Heroku command line, like so

```
heroku apps:rename [YOUR AWESOME NEW NAME HERE]
```

Great—what next?

Excellent! You have two separate apps but they are not talking yet—but that is OK. Now we can customize each of them alongside each other.

Obviously, development is an interative process. In this case, that means that you can not work on the server / API side in isolation from the client / React side forever. So what can you do? Simple. Make as much as you can without connecting the apps together.

Make the React app with fake data

Rather than using `fetch` to get data from an API, just fake data in your components and work out the basic UI things.

This doesn't need to be complete, just enough to make sure you have the basics of your UI down.

Make a CRUD server API

Make fake data and try it.

Then make seeds.

Etc.

Make the API and test it with cURL, httpie, Postman, etc

You don't need a front end to test your API! Make a database and work out some CRUD routes. Test them without a UI.

And of course! Keep on deploying.

Now wire them together

- On the **client** side make your environment files in the root of the repo
- `.env.production`

```
REACT_APP_API_URL=https://[YOUR AWESOME API HERE].herokuapp.com
```

- `.env.development`

```
REACT_APP_API_URL=http://localhost:8181
```

- These will give you access to the `process.env.REACT_APP_API_URL` environment variables. Heroku and react will switch them automatically for you.
- On your client side code, where you make URLs, you should reference this environment variable. ssumeing the atch we are going for is `hoarding`, for `GET`

```
async componentDidMount() {
    const res = await fetch(`${process.env.REACT_APP_API_URL}/hoard`)
    const json = await res.json()
    this.setState({
      ...this.state,
      items: json
    })
  }
```

- for `POST`

```
addItem = async (item) => {
    const url = `${process.env.REACT_APP_API_URL}/hoard`
    const opts = {
      method: 'POST',
      body: JSON.stringify(item),
      headers: {
        'Content-Type': 'application/json',
        'Accept': 'application/json'
      }
    }
    await fetch(url, opts)
    const getRes = await
fetch(`${process.env.REACT_APP_API_URL}/hoard`)
    const items = await getRes.json()
    this.setState({
      ...this.state,
      items: items,
      currentItem: -1
    })
  }
```

Whenever you do a `git push origin master` do a `git push heroku master` this will keep your front and back ends in sync. Because react and Heroku know to switch between `.env.production` and `.env.development` you will have both production and development working!