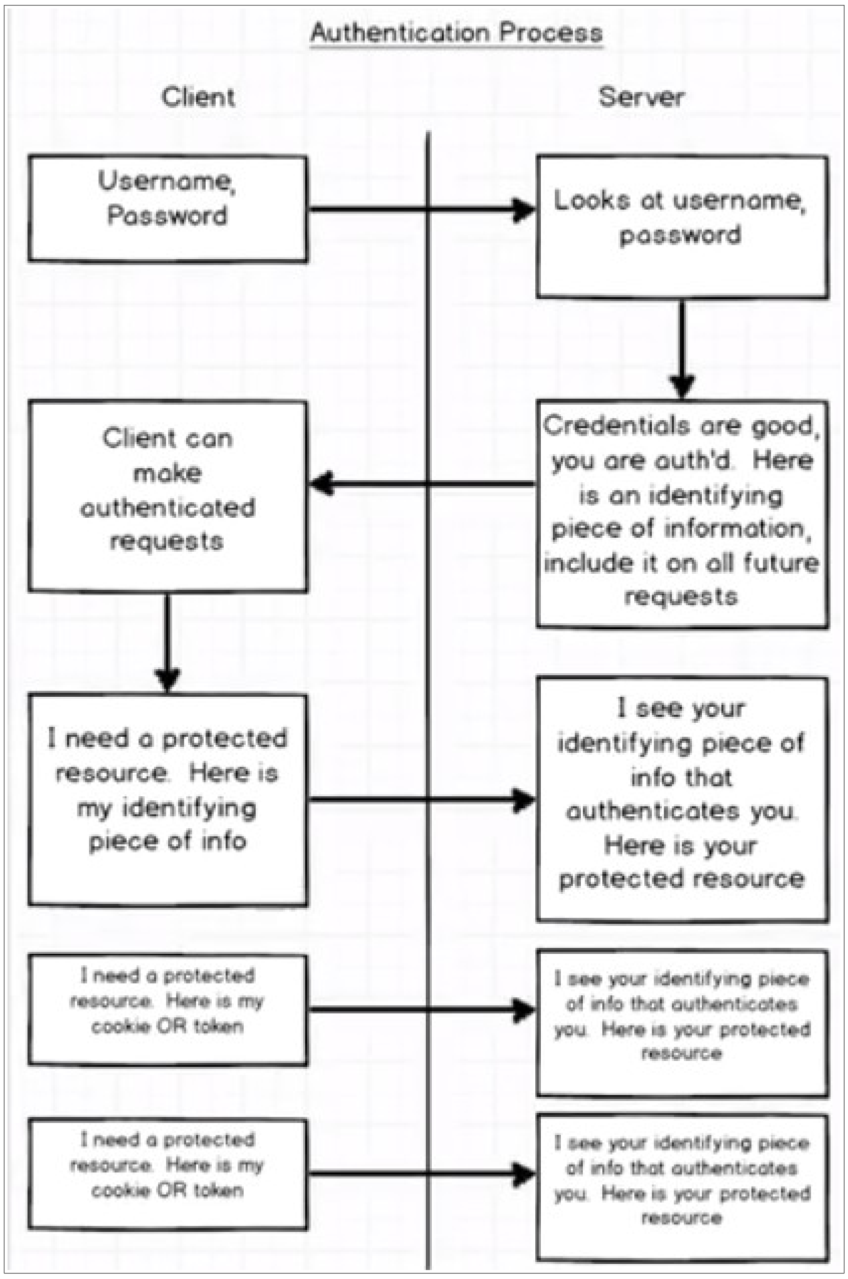
Server-Side Auth:

Introduction to Authentication:

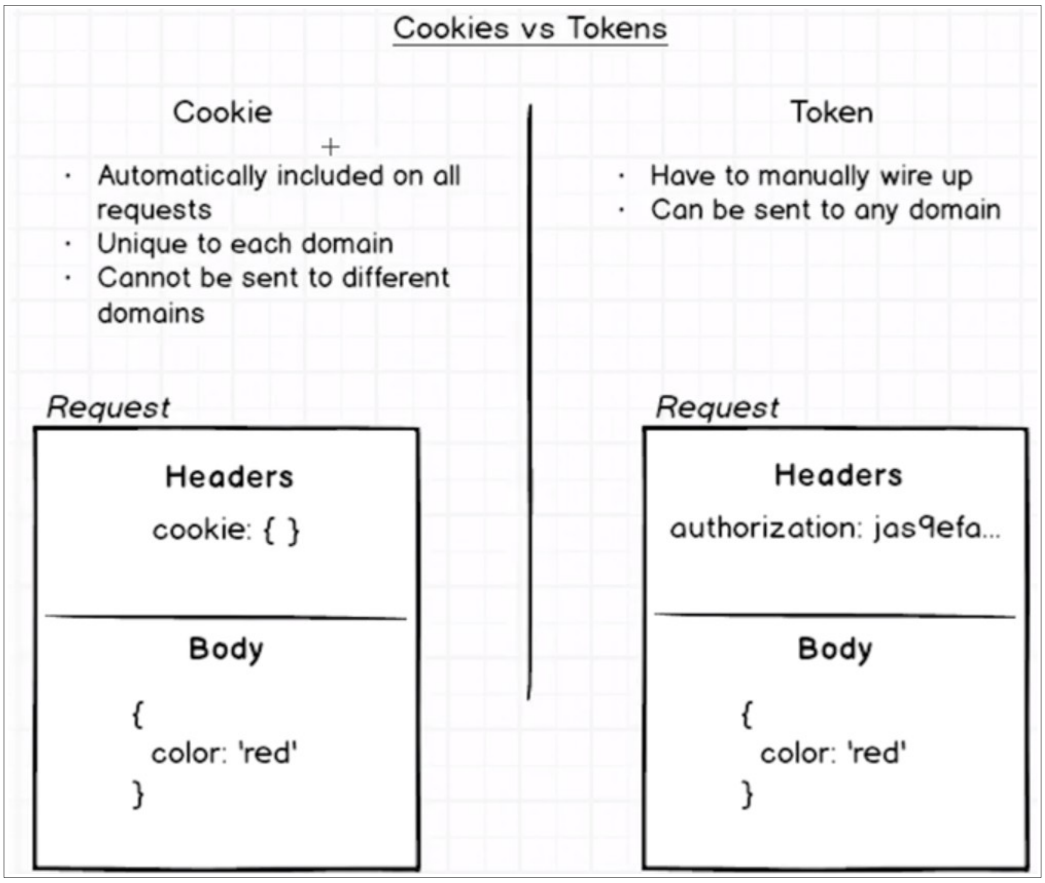
We will start by setting up a server for authentication – what is the best backend for this task? The simple answer is that there is none – all we care about for our React App is that our backend can serve JSON. The following diagram is an overview of the authentication process:


Authentication Process
Client
Server
Username, Password
Looks at username, password
Credentials are good, you are auth'd. Here is an identifying piece of information, include it on all future requests
Client can make authenticated requests
I need a protected resource. Here is my identifying piece of info
I see your identifying piece of info that authenticates you. Here is your protected resource
I need a protected resource. Here is my cookie OR token
I see your identifying piece of info that authenticates you. Here is your protected resource
I need a protected resource. Here is my cookie OR token
I see your identifying piece of info that authenticates you. Here is your protected resource

- After a user supplies a username/password through the client, the server will check and if they're good; supply the user with an identifying piece of information to include in all future requests.
- It is important that the user, when authenticated, is given a piece of info that will identify them on all future requests. When the client has that piece of information, they can make authenticated requests.
- The key is the exchange of a username/password for some identifying token, or piece of info. The client needs to include this on all future requests.

Cookies vs Tokens:

How do we include that identifying piece of information, assuming we supplied a correct username/ password. We can include this info via **cookies** or a **manually set token**:

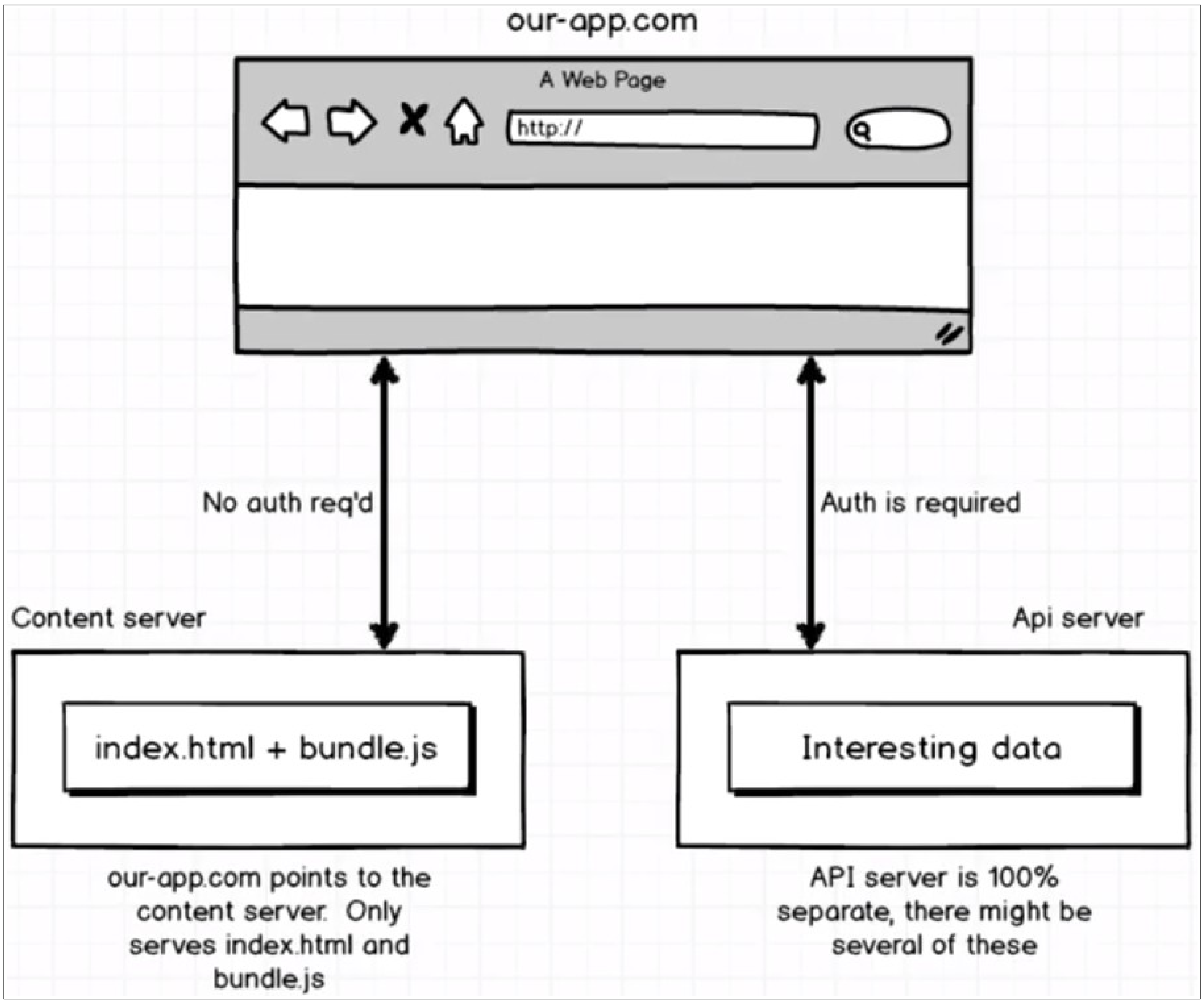


A cookie is included with all HTTP requests that we make and is included by default in our browser. The purpose of cookies is to **introduce state into the stateless http protocol**. Cookies are included by default on all HTTP requests, included as a header on the request message. A server can choose to place information on a users cookie, that identifies them uniquely to that particular server. Then on all follow-up requests that that user made, we see that cookie, identifying them to the user. Cookies that are tied to Google.com cannot be shared with Ebay.com – this gives our requests more security – if we go to a malicious website, it can't lift the users cookie and hijack their identifying info. **Cookies cannot be sent to different domains**.

Opposed to cookies, is the concept of tokens, they can be used in place of cookies. They need to be manually wired up, as in, when we make a request, we need manually include some information in the request header. The benefit of tokens is that they can be sent to any domain that we wish. If we are on Google.com, and make an authenticated request to a different domain, we can do so using a token. This is very useful if we make an app hosted on different servers hosted on different domains, but we want our user to be authenticated across different domains, because of this they're better for scaling. We are going to progress using a token-based approach as this is the trending method in industry.

Scalable Architecture:

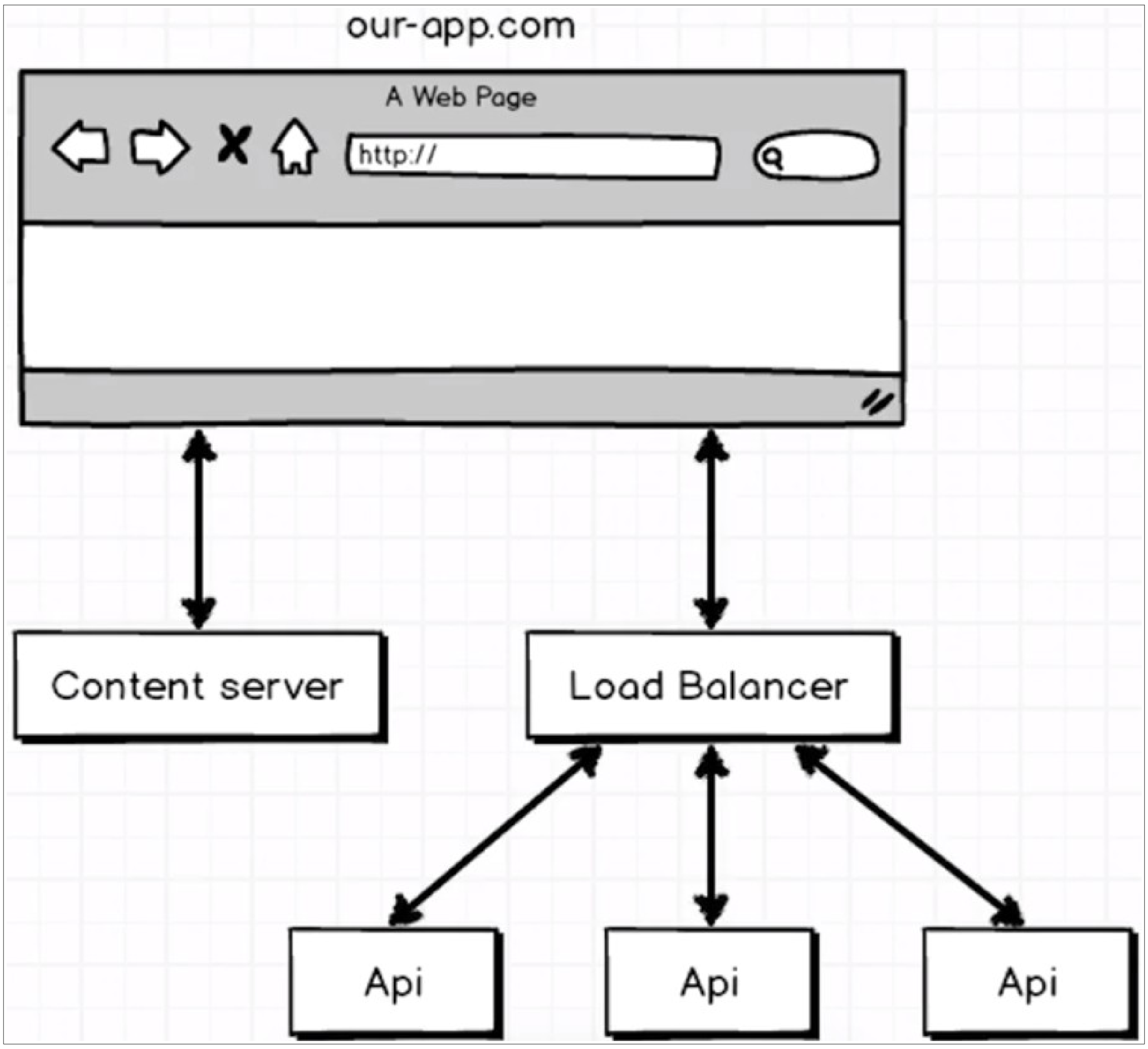
Lets take a look at a diagram of the server structure we will use in our Application:


our-app.com
A Web Page
http://
No auth req'd
Auth is required
Content server
Api server
index.html + bundle.js
Interesting data
our-app.com points to the content server. Only serves index.html and bundle.js
API server is 100% separate, there might be several of these

Lets imagine that a user visits our website, “our-app.com” - where they receive an index.html page and a bundle.js file. This will be hosted on a content server, whos only purpose is to serve these two files. The benefit here is that we can make this server very simple and very stupid, we can spin it up on the fly. This server is easily redistrubuted, lightweight and doesn’t require a lot of resources.

All the interesting data will be protected behind an authentication barrier – which is 100% seperate and there might be several such API servers. This cannot be accessed with cookies, only tokens can go acrosss domains. For example, our content server might have a mobile and web-version, and they both access the same API server. It is therefore useful to have the API server hosted as its own independent unit. This keeps the backend and frontend seperate.

It is also much easier to scale using this approach. Lets say that our content server only needs to serve the webapp to 1000 people, but we have a mobile app that has 5 million users and has huge demand. If we keep the content and API server seperate, we can make the API server capable of handling millions of requests without needing to touch the front-end at all.

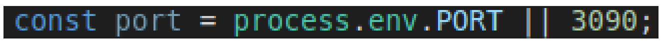

our-app.com
A Web Page
http://
Content server
Load Balancer
Api
Api
Api

Server Setup:

We will set up the backend API server using Node and Express. We will install the following dependencies:

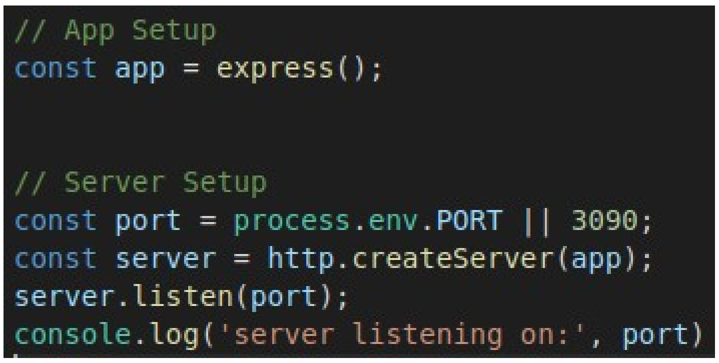
- express, mongoose, morgan, body-parser, nodemon

We will use require syntax for importing modules due to our current node version. We will use a dynamic port – the syntax below states "if there is a PORT variable defined in process.env, use that. Otherwise, use 3090. This will be useful for deployment.



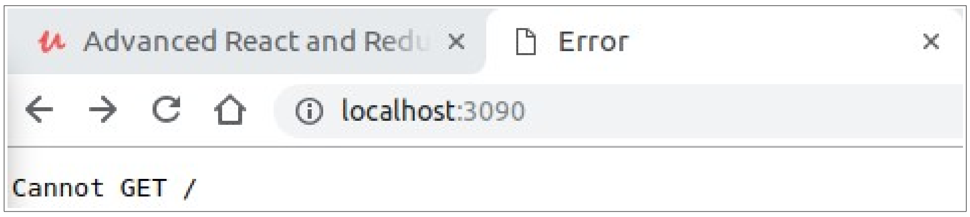
```
const port = process.env.PORT || 3090;
```

Next, we will create our server using the HTTP library. HTTP is a default node library which works with incoming HTTP requests. We will wrap the express app in a http.createServer method which will listen for incoming requests and forward them on to express.



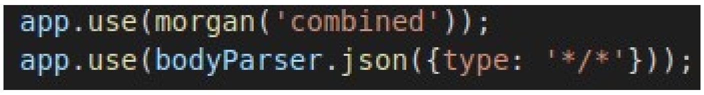
```
// App Setup
const app = express();


// Server Setup
const port = process.env.PORT || 3090;
const server = http.createServer(app);
server.listen(port);
console.log('server listening on:', port)
```

Now, when we visit **localhost:3090**, we will see the following:



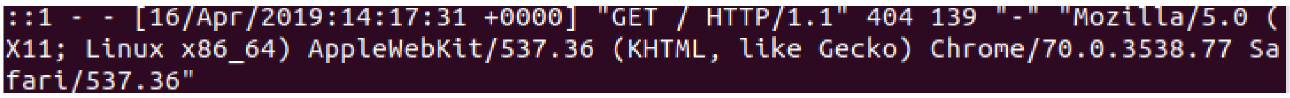
```
Cannot GET /
```

This is the server telling us "I have nothing to show you at the route '/' ", which is to be expected. Next, we will tell our server to use both **morgan** and **bodyParser** middleware, which will pass incoming requests to before they reach the express app.



```
app.use(morgan('combined'));
app.use(bodyParser.json({type: '*/*'}));
```

Morgan is a logging framework – see what is logged when we refresh the page.

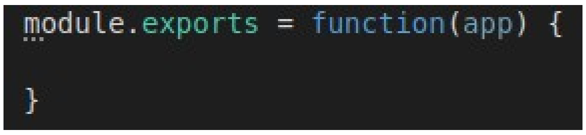


```
::1 - - [16/Apr/2019:14:17:31 +0000] "GET / HTTP/1.1" 404 139 "-" "Mozilla/5.0 (
X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.77 Sa
fari/537.36"
```

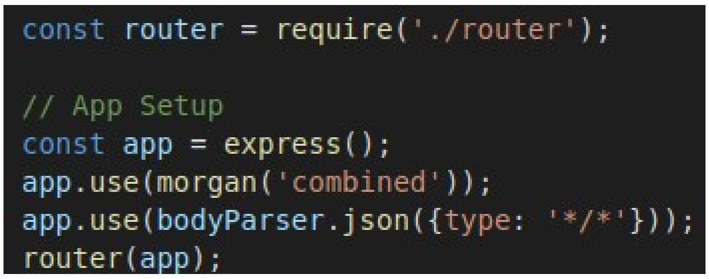
See that it logs the time, route and response message above. This will be used mostly for debugging. **BodyParser** is used to parse incoming requests into JSON, no matter what the request type is – which can cause trouble when we try post a file, so we need to be aware of that.

<u>Routing Setup:</u>

We will set the routing up in a router.js file, defining it as a function export, and import it into the root index.js file – immediately calling it with our express app. The router is basically the brains of our entire application. Here we will define the routes that a user can visit.

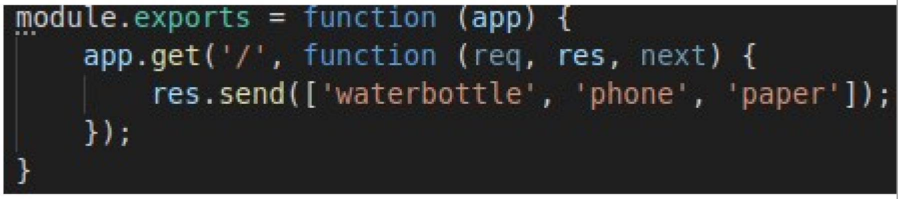


```
module.exports = function(app) {

}
```

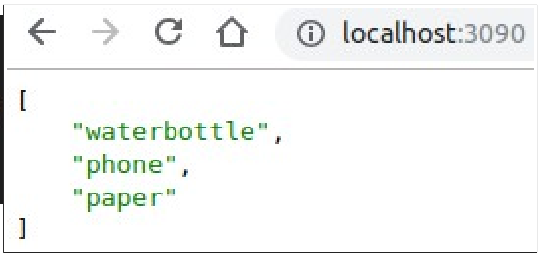


```
const router = require('./router');

// App Setup
const app = express();
app.use(morgan('combined'));
app.use(bodyParser.json({type: '*/*'}));
router(app);
```

When we are defining the routes, we will use the app.get method – wihch is passed a route, and a callback function which is passed a **req**, **res** and **next** argument. Req represents the incoming HTTP request, and has all the data about the request, res represents the response we will send back to whoever made the request. Next is mainly used for error handling.



```
module.exports = function (app) {
    app.get('/', function (req, res, next) {
        res.send(['waterbottle', 'phone', 'paper']);
    });
}
```

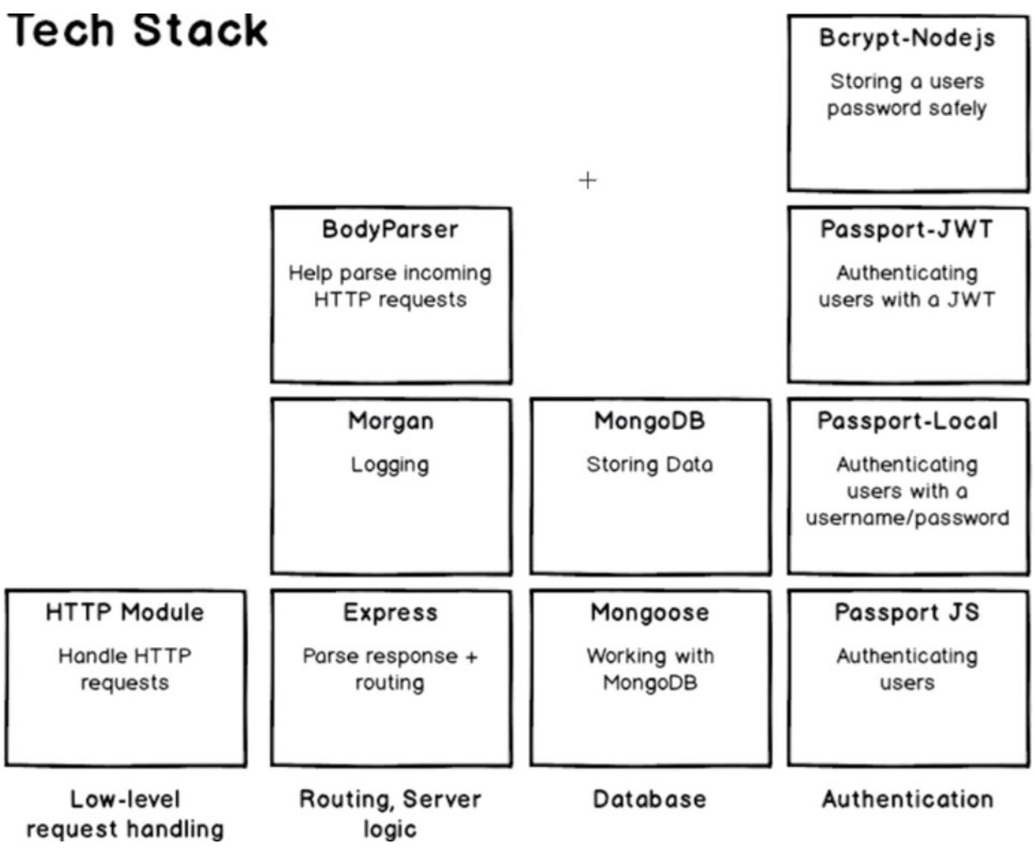


localhost:3090

```
[
    "waterbottle",
    "phone",
    "paper"
]
```

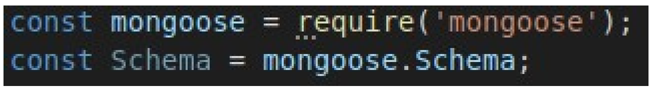
<u>Mongoose Models:</u>

This is where the basic backend setup is complete, and we will get started working on the more advanced concepts:



So far, we have used the HTTP module, Express, Morgan and BodyParser. Now we need move on to the right hand side of the diagram above. Here we will work on a **mongodb database** and how we connect to it. We will use the **Mongoose ORM** library to interact with the database. To make use of mongoose, we will create a **user-model** that represents a user. Each user will have two attributes, an email and a password. We need to tell mongoose about this, making sure that the user model has an email and password field.

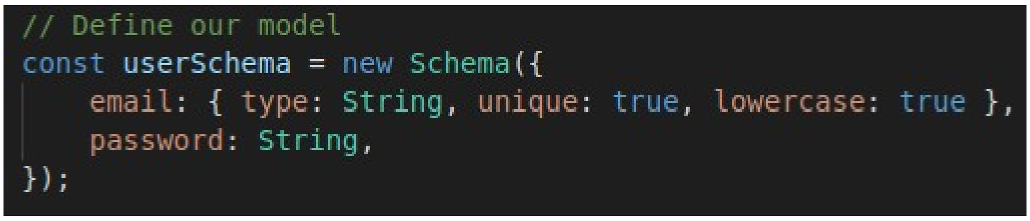
We will create a models directory with a user.js file, where we will make a local definition of what a user is, this will let us tell mongoose how to handle a user. Here, we will import mongoose using the require syntax, and define a Schema object, which we use to tell mongoose about the different fields a model will have.



```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
```

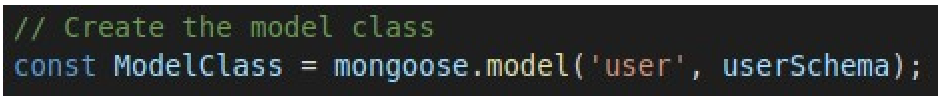
Next, we will define the information to be stored in each instance of the model – this set of instructions is known as the **Schema**. We can do this by creating a variable which has a new isntance of the Schema class inherited from mongoose. We pass an object into this Schema and here we can use email and password as keys, and their respective types as values.

Note that we have passed in an options object for the email, which will make sure that emails must be unique, so no two users can register the same email, also we convert the input to lowercase as mongoose will assume that Greg@mail.com and greg@mail.com are unique, which we don't want:



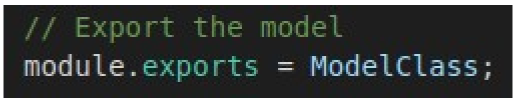
```
// Define our model
const userSchema = new Schema({
    email: { type: String, unique: true, lowercase: true },
    password: String,
});
```

Now that we have defined the template (or Schema) for each document, we need to load this into the database. To do this, we define a **mongoose.model class**, and we pass the name and associated schema in as arguments for the constructor:



```
// Create the model class
const ModelClass = mongoose.model('user', userSchema);
```

Finally, we can export the thie mongoose.model class:



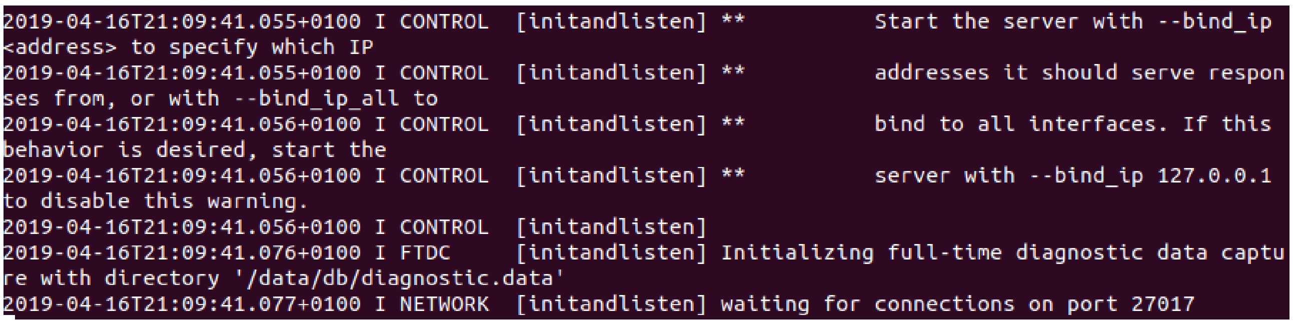
```
// Export the model
module.exports = ModelClass;
```

MongoDB Setup:

Thus far, we have just been defining our models using the JavaScript mongoose library. We still havn’t done anything related to our MongoDB database yet. Now we need to create a database, and connect it to our project, making use of our userSchema. We can find instructions for installing mongoDB for linux here:

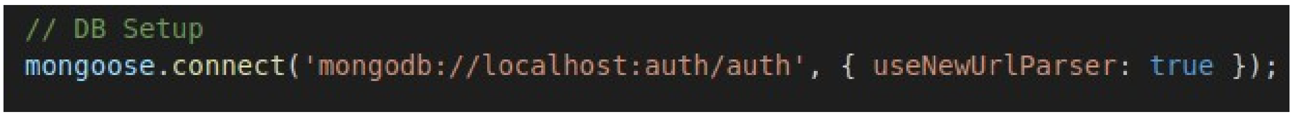
https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/

After installing MongoDB we need to make a /data/db folder and configure its permissions to read/ write. Then we can run the **mongod** command and should see the following:



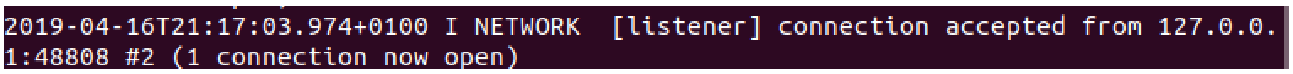
```
2019-04-16T21:09:41.055+0100 I CONTROL  [initandlisten] **          Start the server with --bind_ip
<address> to specify which IP
2019-04-16T21:09:41.055+0100 I CONTROL  [initandlisten] **          addresses it should serve respon
ses from, or with --bind_ip_all to
2019-04-16T21:09:41.056+0100 I CONTROL  [initandlisten] **          bind to all interfaces. If this
behavior is desired, start the
2019-04-16T21:09:41.056+0100 I CONTROL  [initandlisten] **          server with --bind_ip 127.0.0.1
to disable this warning.
2019-04-16T21:09:41.056+0100 I CONTROL  [initandlisten]
2019-04-16T21:09:41.076+0100 I FTDC     [initandlisten] Initializing full-time diagnostic data captu
re with directory '/data/db/diagnostic.data'
2019-04-16T21:09:41.077+0100 I NETWORK  [initandlisten] waiting for connections on port 27017
```

Every time we want to use mongo we need to run that mongod command. We could configure options to run it at login automatically but the manual start is fine for now. Now that we have mongo running, we need to connect to our mongodb server using mongoose. We can do this in our index.js file:



```
// DB Setup
mongoose.connect('mongodb://localhost:auth/auth', { useNewUrlParser: true });
```

This will connect to our local server, and create a file called auth in an auth folder inside our database. Now when we run our node server, we will see the following:

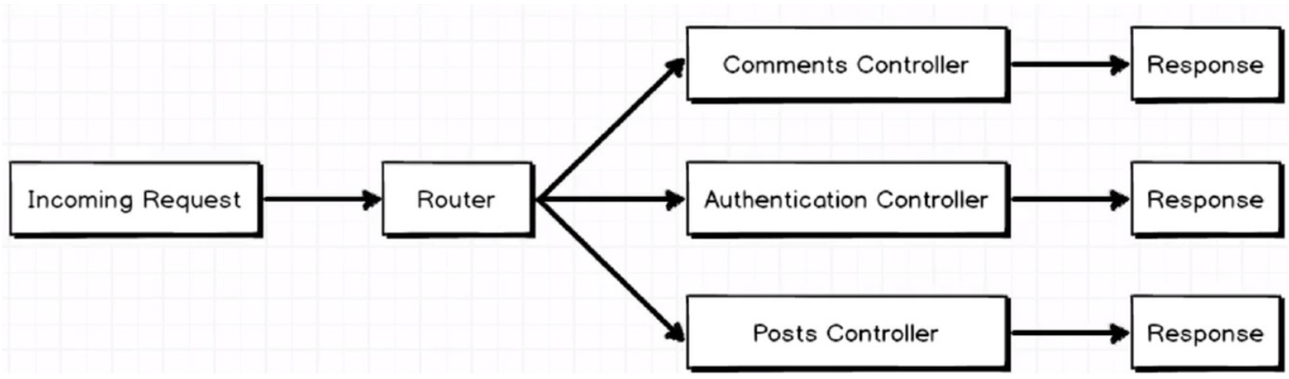


```
2019-04-16T21:17:03.974+0100 I NETWORK  [listener] connection accepted from 127.0.0.
1:48808 #2 (1 connection now open)
```

Now our server is connected with our MongoDB instance.

Authentication Controller:

To prevent the router.js file growing too cumbersome, we will set up the logic for our authentication in a seperate file.



An incoming request will always go to the router, but the router will then decide which controller it sends the request to, where the response will be formulated. Each controller will be individually responsible for creating a response.

We will set up the route handler for the signup route. This will be a post request to the route '/signup'. We will abstract the route logic to the authentication.js file as follows:

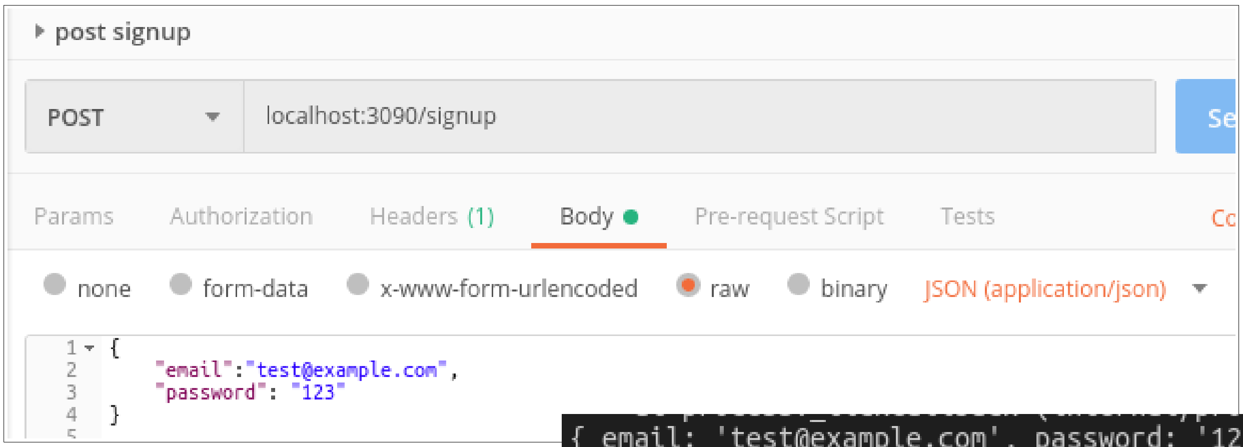
```
module.exports = function (app) {
    app.get('/signup', Authentication.signup);
}
```

```
exports.signup = function(req, res, next) {

}
```

Our goal is to now read-in a user if it is passed in this post route, check for duplicate records, and if none exist, save the user record. The obectives are as follows:

```
exports.signup = function (req, res, next) {
    // See if a user with the given email exists
    console.log(req.body);

    // If a duplicate exists, return an error

    // If a user with the provided email doesnt exist, create and save user record

    // Respond to request indicating the user was created

}
```

The email and password will be accessible through **req.body** – as can be seen in the console log for the following postman request:



```
localhost:3090/signup
```

```
{
    "email":"test@example.com",
    "password": "123"
}
```

```
{ email: 'test@example.com', password: '123' }
```

This is how we will get access to our data:

```
const email = req.body.email;
const password = req.body.password;
```

Now, we need to check if a user exists in the database with this email – we can do this with mongoose. We can import the user class (model), and then use the **.findOne** method to search the collection for other documents with the email provided. This function takes a callback which is invoked upon completion of the search, where an error or existingUser is passed depending on the search result:

```
User.findOne({ email: email }, function (err, existingUser) {
    if (err) { return next(err); }
```

This error callback will be invoked in the case that there was a problem connecting to the database, it doesn’t consider an existing user being found to be an error. This is handled below, where we return from the route handler, and send a 422 error message, signifying an unprocessable entity:

```
if (existingUser) {
    return res.status(422).send({ error: 'Email is in use' });
}
```

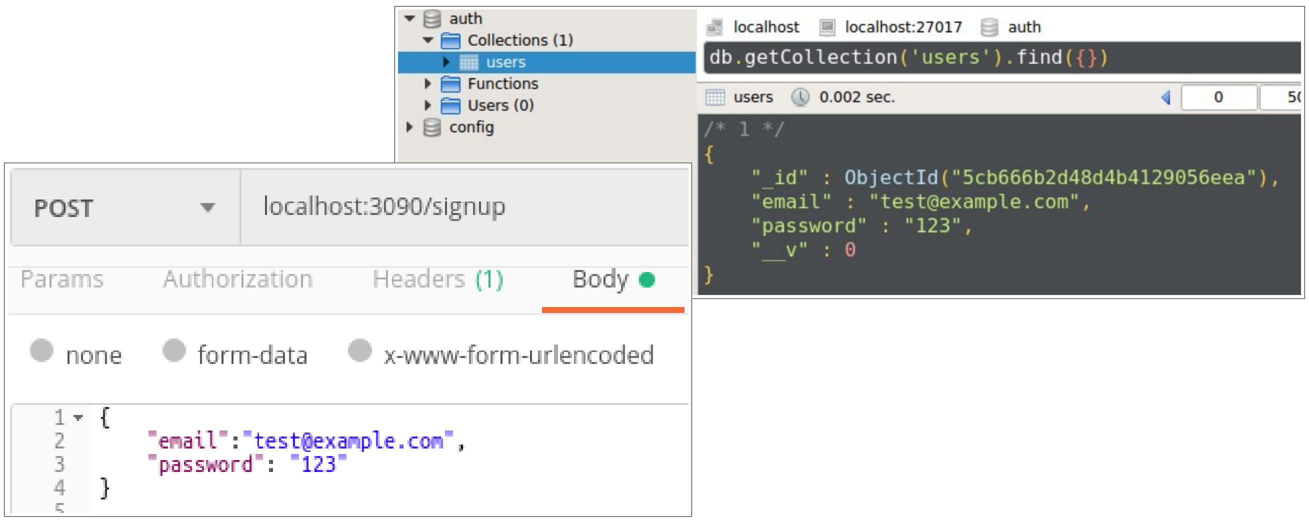
If no existing user is found we can create a new document (instance of the User class), and save it. Recall that saving (or any process involving interacting with the database) is asynchronous, and as such it will take a callback.

```
const user = new User({
    email: email,
    password: password
});
user.save(function (err) {
    if (err) { return next(err); }
});
```

Finally, after saving the document we can send a success message back as a response:

```
res.json({ success: true })
```

Now, making the following request will yield the following in the database:



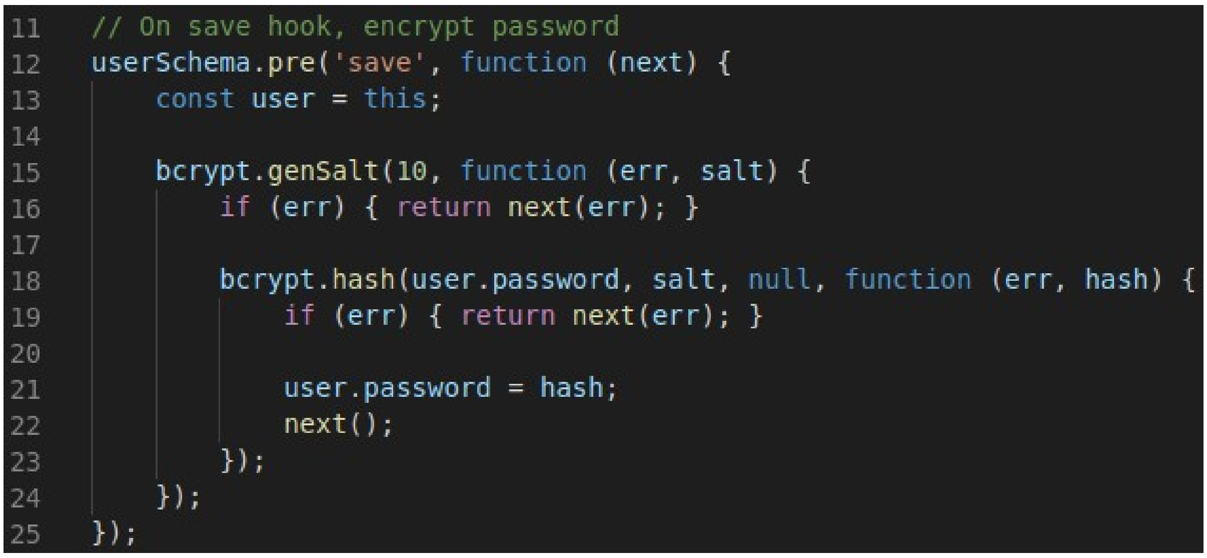
```
db.getCollection('users').find({})
```

```
/* 1 */
{
    "_id" : ObjectId("5cb666b2d48d4b4129056eea"),
    "email" : "test@example.com",
    "password" : "123",
    "__v" : 0
}
```

localhost:3090/signup

```
{
    "email":"test@example.com",
    "password": "123"
}
```

The issue with our signup route thus far is that the passwords are stored in plain text. This is a major problem. We will need to hash the password with **bcrypt**, to ensure the password is encrypted before saving. **npm install --save bcrypt-nodejs**.

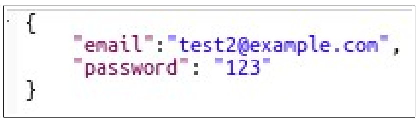


```
11  // On save hook, encrypt password
12  userSchema.pre('save', function (next) {
13      const user = this;
14
15      bcrypt.genSalt(10, function (err, salt) {
16          if (err) { return next(err); }
17
18          bcrypt.hash(user.password, salt, null, function (err, hash) {
19              if (err) { return next(err); }
20
21              user.password = hash;
22              next();
23          });
24      });
25  });
```

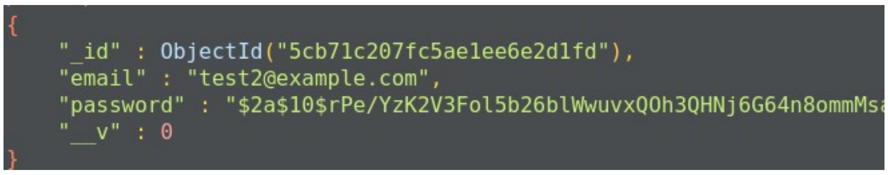
We can implement encryption using a pre-save hook, as above. We will break this code step by step as follows:

- Line 12 – before saving a model, run this function
- Line 13 – get access to user model
- Line 15 – generate a salt, this is an async process (takes some time), so run the callback when finished.
- Line 18 – hash (encrypt) the password, running the callback when finished.
- Line 21 – overwrite the plain text password with the encrypted one.

Now when we make a post request to the API server with the following data, we will get the following user model saved to the collection:

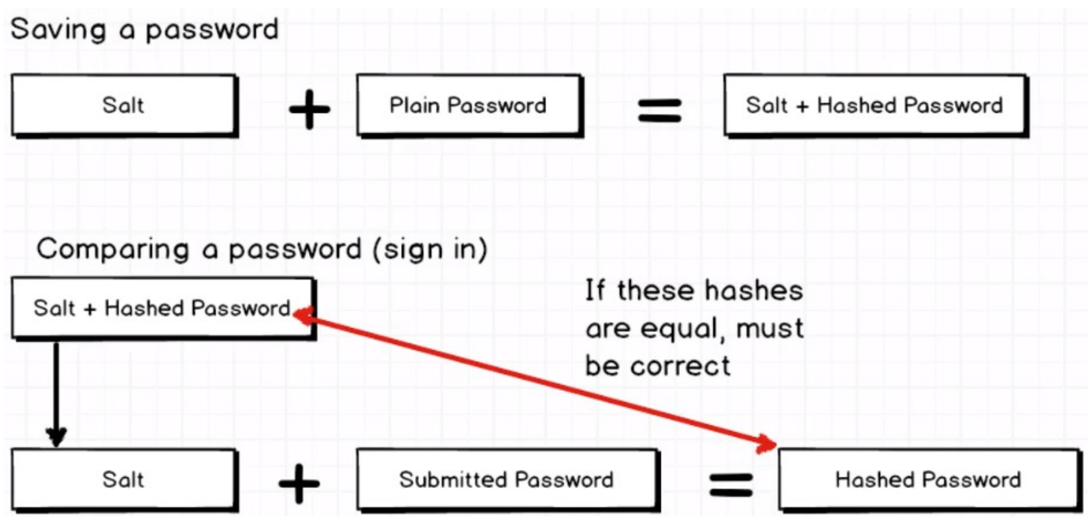


```
{
    "email":"test2@example.com",
    "password": "123"
}
```



```
{
    "_id" : ObjectId("5cb71c207fc5ae1ee6e2d1fd"),
    "email" : "test2@example.com",
    "password" : "$2a$10$rPe/YzK2V3Fol5b26blWwuvxQOh3QHNj6G64n8ommMs
    "__v" : 0
}
```

How does bcrypt do this?



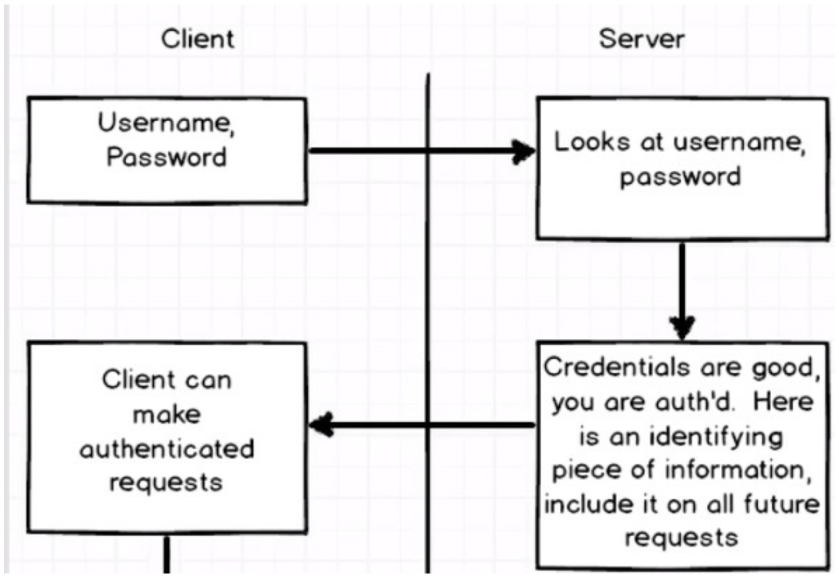
Theres two steps to bcrypt – saving a password (which we just wrote), and comparing an entered password to that in the database.

When we save a password, we generate a salt – a random string of characters. By combining a salt and a plain password, we get a salted + hashed password. So the long string saved in our database contains both the password and salt.

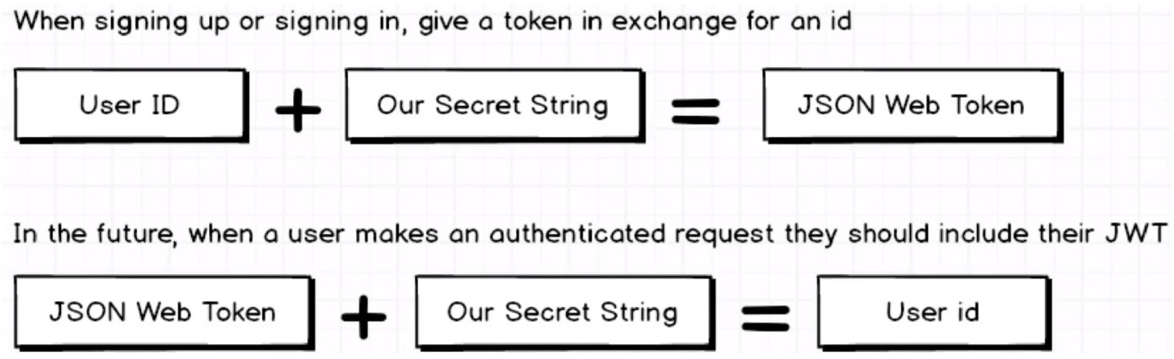
<u>JWT Overview:</u>

We now know how to create a user account and securely save their credentials to our database. But we will haven't dealt with signing in for existing users. Here, bcrypt makes a lot more sense, which we will disucss below.

Recall in our authentication controller, we simply return **{success: true}** as a response when a user signs up, but this doesn't fit with our earlier plan. We want to give the user more than this simple response. We need to give this newly created userr an identifying piece of information which can be included in future requests, allowing them to make authenticated requests for secure info.


Client
Server
Username, Password
Looks at username, password
Credentials are good, you are auth'd. Here is an identifying piece of information, include it on all future requests
Client can make authenticated requests

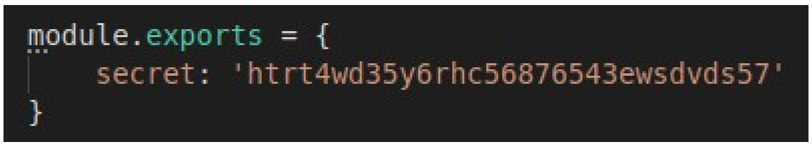
At the end of the signup handler, we now consider them to be logged in, and need to send back that identifying token.


When signing up or signing in, give a token in exchange for an id
User ID
+
Our Secret String
=
JSON Web Token
In the future, when a user makes an authenticated request they should include their JWT
JSON Web Token
+
Our Secret String
=
User id

The token we want to give the user is known as a **JSON-Web-Token (JWT)**. How do we create this token and use it? On top of the above diagram, we take the users id, and combine it with some secret string, and this is used to produce a JWT.

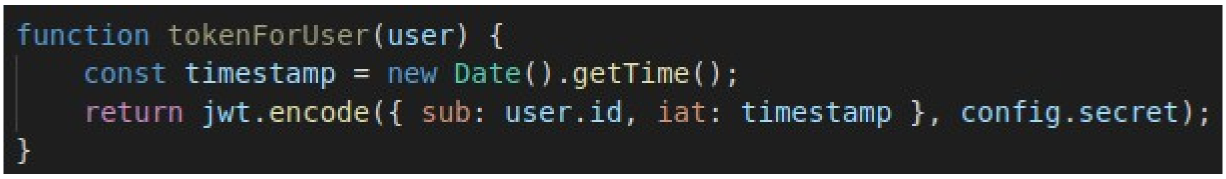
In the future, the user can include this JWT with their requests and we can decrypt it since we know the secret string to retrieve the userId – we can not consider the user authenticated. The good thing is that even if a malicious user gains access to a client's JWT, they won't know the secret string so will not be able to decrypt the token. It is **very important** that we never share this secret string (eg. Posting to our github etc.).

Creating a JWT: We will take the userId and combine it with our secret string to give us our web token. We will use the **jwt-simple** library to do this, rather than doing all the logic from scratch. To create the token, we need to house a secret string in a **config.js** file, and add it to our **.gitignore** file.



```
module.exports = {
    secret: 'htrt4wd35y6rhc56876543ewsdvds57'
}
```

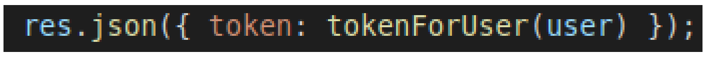
We can import jwt-simple and the secret string into the authentication.js file. We can then define a function **tokenForUser** which takes a user argument. This function makes use of the **jwt-simple.encode(object, secret)** method. We can place any information we want in to be encoded. We could use the users email, though this can change over time leading to old useless tokens. So, we shoud only encode the users ID. We should assign the Id to a **sub** key, with **user.id** as the value.



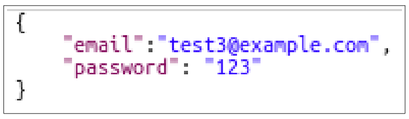
```
function tokenForUser(user) {
    const timestamp = new Date().getTime();
    return jwt.encode({ sub: user.id, iat: timestamp }, config.secret);
}
```

As a convention, JWTs have a sub property, which is short for subject. They also typically have a **iat** (issues at time) property – which is a timestamp for when the token was generated.

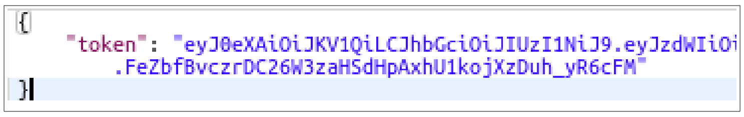
Now, instead of return a simple success message when a user signs up, we can return the token:



```
res.json({ token: tokenForUser(user) });
```



```
{
    "email":"test3@example.com",
    "password": "123"
}
```

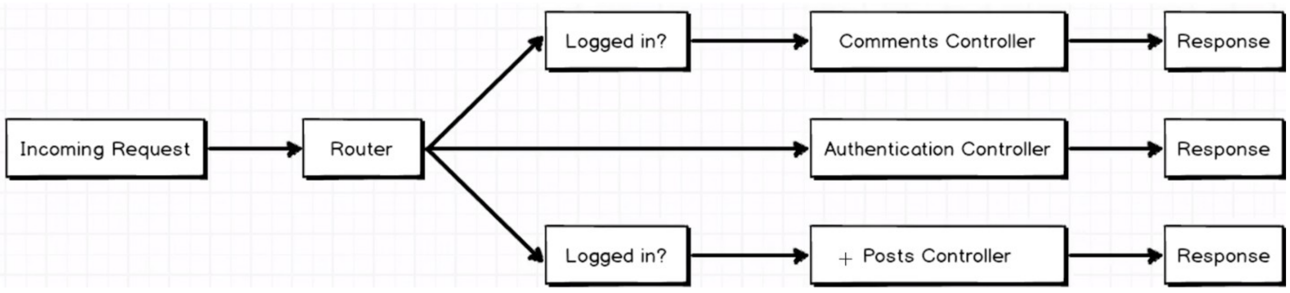


```
{
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOi
        .FeZbfBvczrDC26W3zaHSdHpAxhU1kojXzDuh_yR6cFM"
}
```

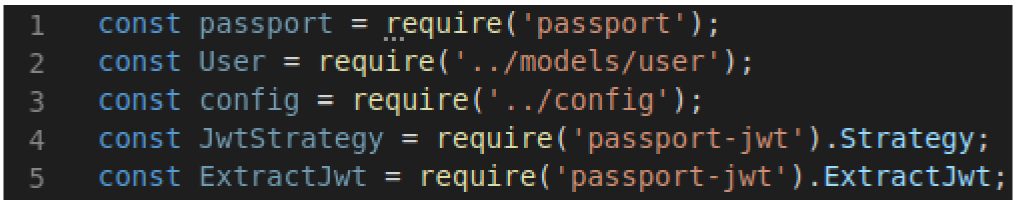
Installing Passport:

We are now able to return a JWT token whenever a user signs up for our application. We take their email and password and store them in the database (with the password salted + hashed). There's two main tasks ahead

1 – **signing in**.
2 – **verify a user is auth when trying to view a protected resource**.

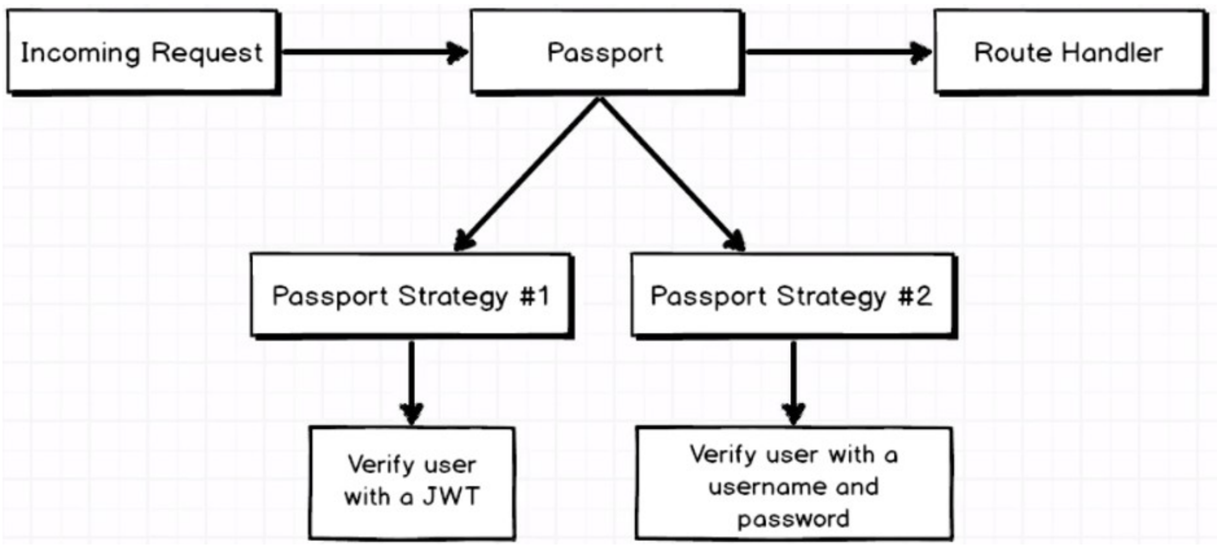


We need to built a service to check if a user is logged in, do they have a valid JWT token in the request? We only want to check if the user is logged in for particular routes. We will do this with two libs called **passport** and **passport-jwt**. Since we are using JWT tokens for authentication, we will make use of the passport-jwt strategy from the passport ecosystem**:**



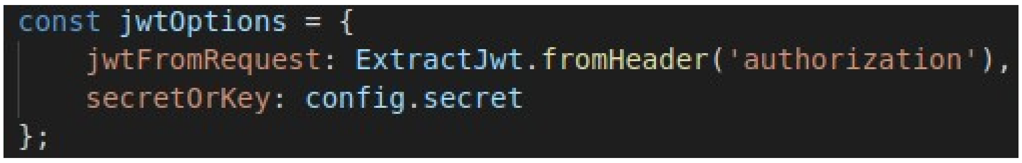
```
const passport = require('passport');
const User = require('../models/user');
const config = require('../config');
const JwtStrategy = require('passport-jwt').Strategy;
const ExtractJwt = require('passport-jwt').ExtractJwt;
```

We import the files outlined above. What is the JwtStrategy that we imported from passport-jwt?



We might have an incoming request, which will encounter the passport library, where we will answer the question of whether the user is logged in, then it will go to the route handler. Passport isn’t just a library, it is more of an eco-system formed of many strategies, two outlined in the diagram above. A strategy is a method for authenticating a user. Above, we imported the passport-jwt strategy. We could use a different strategy to verify a user with a username and password. There are different strategies for google/ facebook/ github login etc.

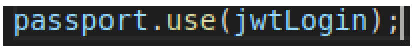
How do we set this strategy up? First we will need to configure the options for the JwtStrategy we wish to use, telling our handler where to look for the JWT (could be in header, body etc), and also we need to give it the secret so it can decrypt the JWT token, giving it access to the user Id etc.



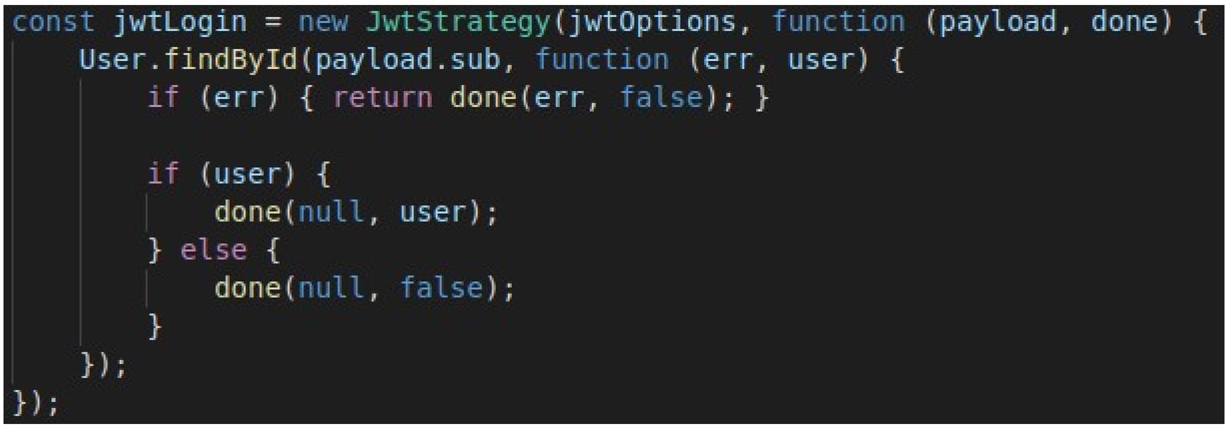
```
const jwtOptions = {
    jwtFromRequest: ExtractJwt.fromHeader('authorization'),
    secretOrKey: config.secret
};
```

The next step is to pass this options object into an instance of JwtStrategy. This takes a 2nd callback argument, which is passed the payload (decrypted JWT), and done. We then can search the database for a user with a matching ID.

Finally, we can register this strategy with passport:

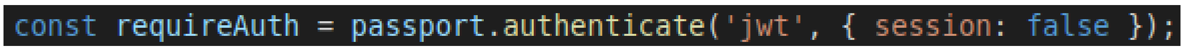


```
passport.use(jwtLogin);
```

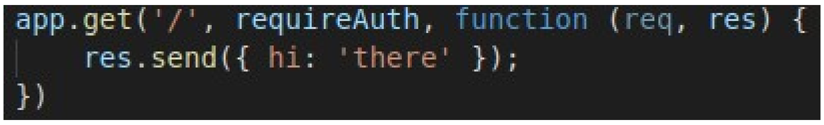


```
const jwtLogin = new JwtStrategy(jwtOptions, function (payload, done) {
    User.findById(payload.sub, function (err, user) {
        if (err) { return done(err, false); }

        if (user) {
            done(null, user);
        } else {
            done(null, false);
        }
    });
});
```

Next we need to apply this middleware to the relevant routes. We can start with the signup route in the router.js file. Registering middleware to a route is simple, just place the middleware in the function body for the **get( )** method (or whatever method is relevant):

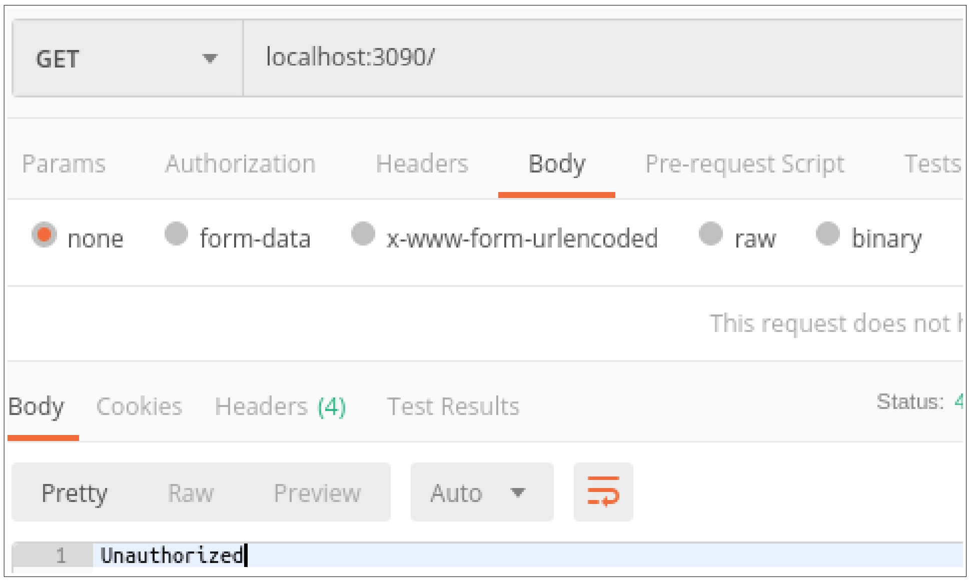


```
const requireAuth = passport.authenticate('jwt', { session: false });
```



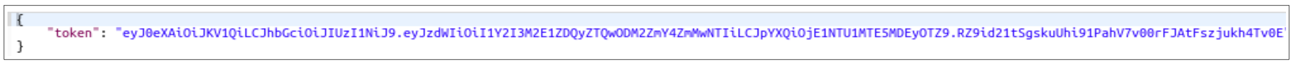
```
app.get('/', requireAuth, function (req, res) {
    res.send({ hi: 'there' });
})
```

By simply registering this middleware in the route handler, we have a **protected route**. If we make a request to this route now, without an appropriate JWT token in the header, we will get the following response:



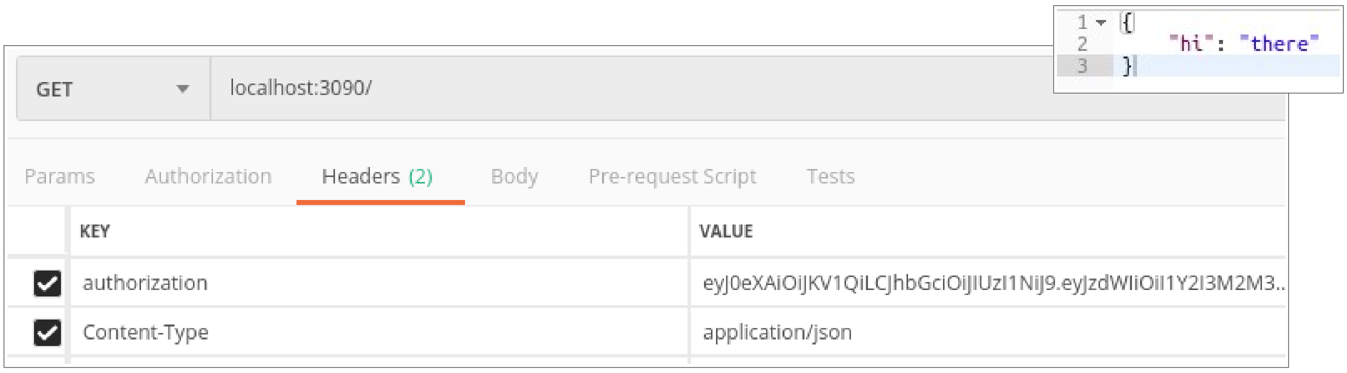
```
Unauthorized
```

We can manually wire up a token by signing up for an account and taking the token (which we still have as the response):



```
{
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiI1Y2I3M2E1ZDQyZTQwODM2ZmY4ZmMwNTIiLCJpYXQiOjE1NTU1MTE5MDEyOTZ9.RZ9id21tSgskuUhi91PahV7v00rFJAtFszjukh4Tv0E'
}
```

We can copy and paste this token in the header of a '/' get request, and we shouldn't see the unauthorized response:



GET localhost:3090/

Params Authorization Headers (2) Body Pre-request Script Tests

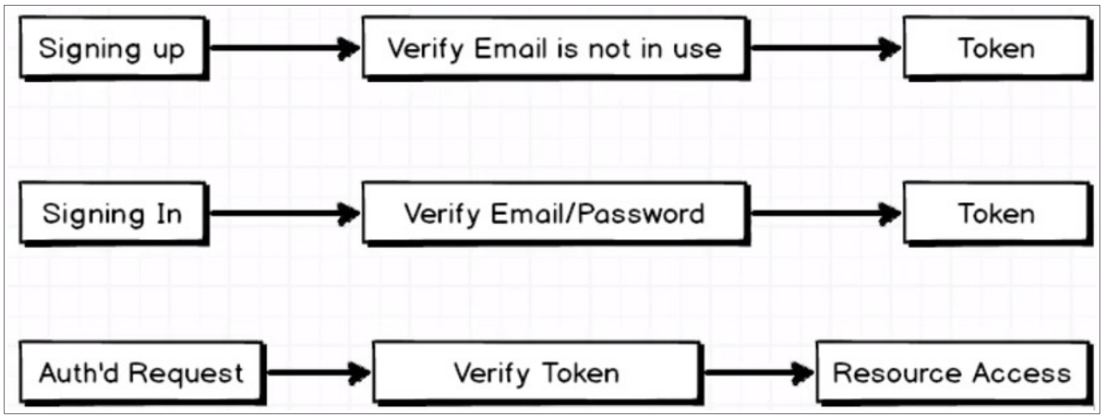
KEY	VALUE
authorization	eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOiI1Y2I3M2M3..
Content-Type	application/json

```
{
    "hi": "there"
}
```

<u>Signing in with Local Strategy:</u>

Thus far, we are able to enable our users to sign-up, issuing them with a JWT after storing their password (encrypted) and email in the database. Furthermore, we can make an authenticated request to the '/' route, where the route is only accessible if a decrypted JWT included in the header contains a sub property that corresponds to the id of a user in the database.

Now, we need to implement a way for a user to log in, and get a token in exchange for correct credentials. Here, we will be receiving an email and password, rather than a token. We need to auth this email and password, and return a token upon success. We will use a local strategy to do this (local as in – local database). If they're sucessfully authenticated we will send them to some particular route where they will receive a JWT token. The diagram below outlines why two different strategies are necessary for giving users tokens. The top strategy is the passport-JWT strategy, whilst the middle represents our local strategy where we want to verify the email and password of pre-existing users.

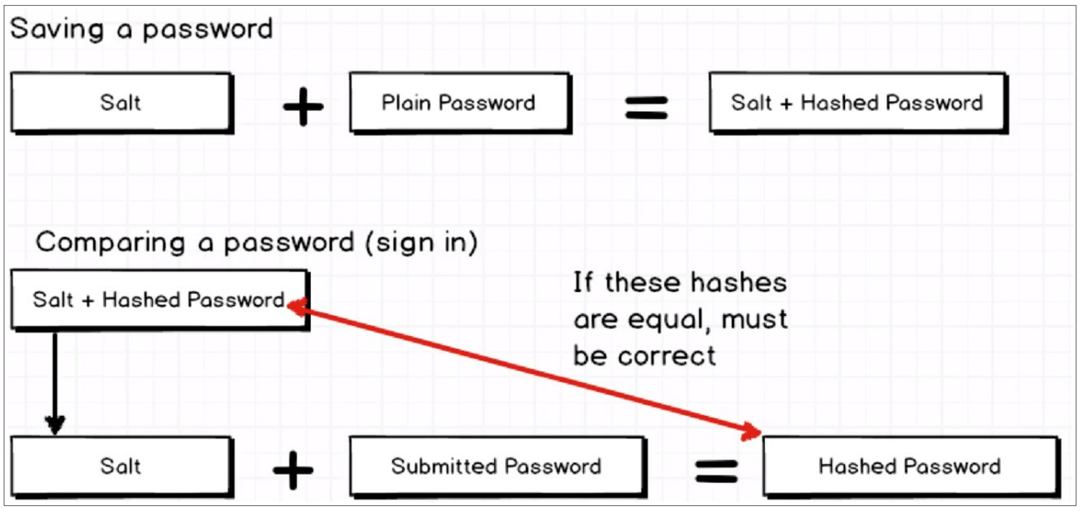


To get started, first we will need to install the **passport-local** module. Then we can implement the strategy as follows:

```
// Create local strategy
const localOptions = { usernameField: 'email' };
const localLogin = new LocalStrategy(localOptions, function (email, password, done) {
    // Verify this username and password, call done with the user
    // if it is the correct username and password
    // Otherwise, call done with false
});
```

We pass in an options object to the LocalSrategy constructor – here we need to define an object with usernameField corresponding to a value of ‘email’. LocalStrategy will automatically parse the incoming request body and detect the username and password. We aren’t using a username however, so we need to tell it instead to check for email in its place. The 2nd argument that goes into the constructor is a callback function which is passed the email, password and done callback.

Now, it is important to look at the callback function passed into the LocalStrategy constructor – it receives an email, password, and done arguments. This password needs to be checked for a match with **user.password**, which is salted and hashed in our database. We will therefore need to make sure we salt and hash the input provided, and if the hashes match, then we can authenticate the user.



We don’t ever actually decrypt a saved password, rather we just salt and hash submitted passwords and **compare hashes**. We can define a custom method for our userSchema which makes use of the **becrypt.compare** method, which will take a candidate password and callback as arguments. If the comparison runs successfully, bcrypt.compare will pass in true / false to the callback function.

```
userSchema.methods.comparePassword = function (candidatePassword, callback) {
    bcrypt.compare(candidatePassword, this.password, function (err, isMatch) {
        if (err) { return callback(err); }
        callback(null, isMatch);
    })
}
```

Now we can implement this method to check the password submitted in our localStrategy:

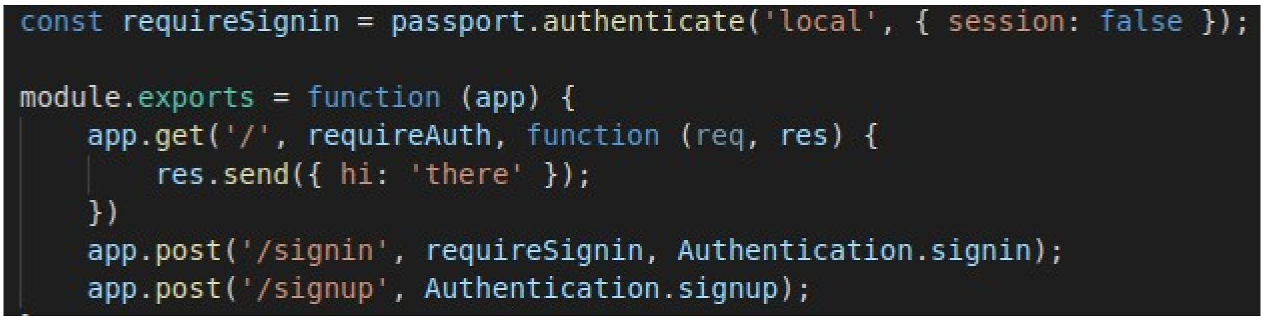
```
user.comparePassword(password, function (err, isMatch) {
    if (err) { return done(err); }
    if (!isMAtch) { return done(null, false); }

    return done(null, user);
});
```

Note we pass in the submitted password and a callback as arguments (comparePassword takes two args). Finally, we tell passport to use the localLogin strategy:

```
passport.use(localLogin);
```

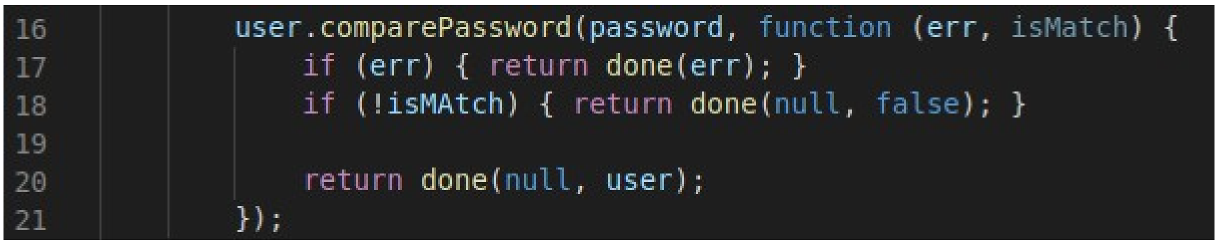
Now we can define a sign-in route, which users can access when they provided correct credentials, exchanging them for a JWT. We need to make sure this route is protected, or anyone could manually visit this url. We want to make sure the users can only access this route if they provided the correct email and password, using the local strategy that we just created.

To do this, we will make another helper, similar to requireAuth, that will intercept the request, and check the email and password provied using the localStrategy.



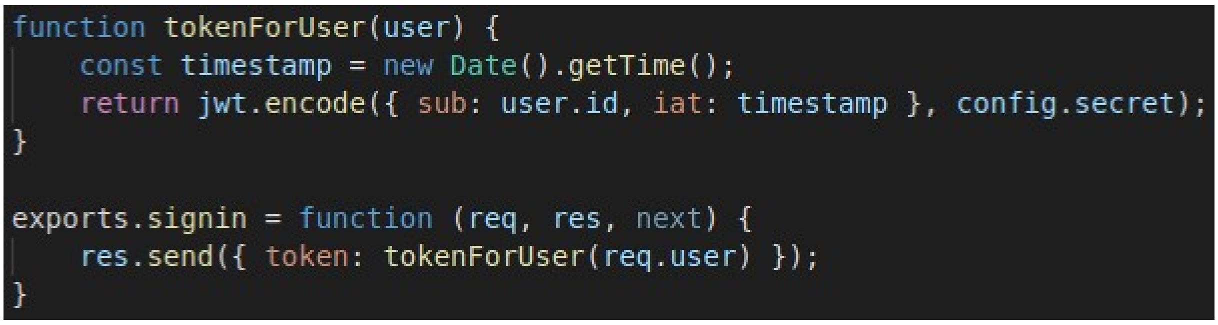
```javascript
const requireSignin = passport.authenticate('local', { session: false });

module.exports = function (app) {
    app.get('/', requireAuth, function (req, res) {
        res.send({ hi: 'there' });
    })
    app.post('/signin', requireSignin, Authentication.signin);
    app.post('/signup', Authentication.signup);
```

After the user has passed the localStrategies check (requireSignin), their request will be passed to the **Authentication.signin** middleware, in wihch we will need to give them a token. To give the user a token, we need to get access to the current user model. Luckily the **done** callback in our localStrategy takes the user model and assigns it to req.user automatically (line 20).

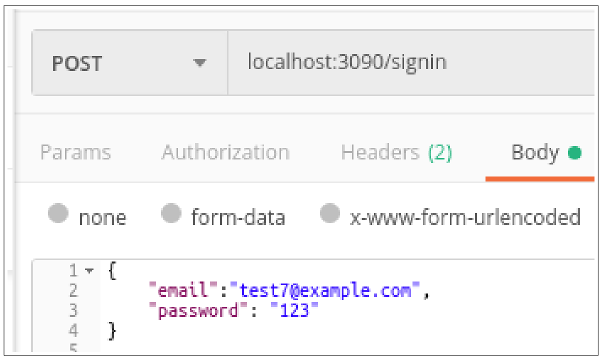


```javascript
16          user.comparePassword(password, function (err, isMatch) {
17              if (err) { return done(err); }
18              if (!isMAtch) { return done(null, false); }
19
20              return done(null, user);
21          });
```

We can therefore access the currently auth'd user via req.user. We can therefore use our helper function **tokenForUser**, and get a token back.



```javascript
function tokenForUser(user) {
    const timestamp = new Date().getTime();
    return jwt.encode({ sub: user.id, iat: timestamp }, config.secret);
}

exports.signin = function (req, res, next) {
    res.send({ token: tokenForUser(req.user) });
}
```

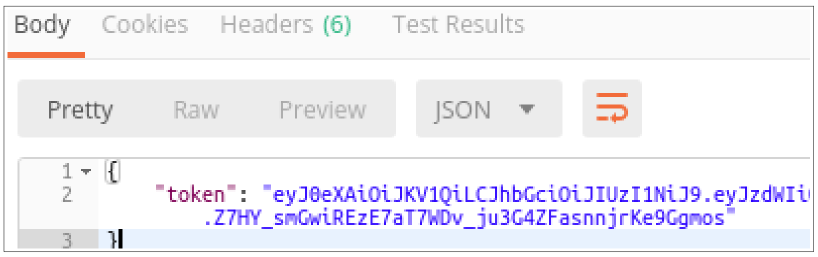


POST localhost:3090/signin

Params | Authorization | Headers (2) | Body ●

none | form-data | x-www-form-urlencoded

```json
{
    "email":"test7@example.com",
    "password": "123"
}
```



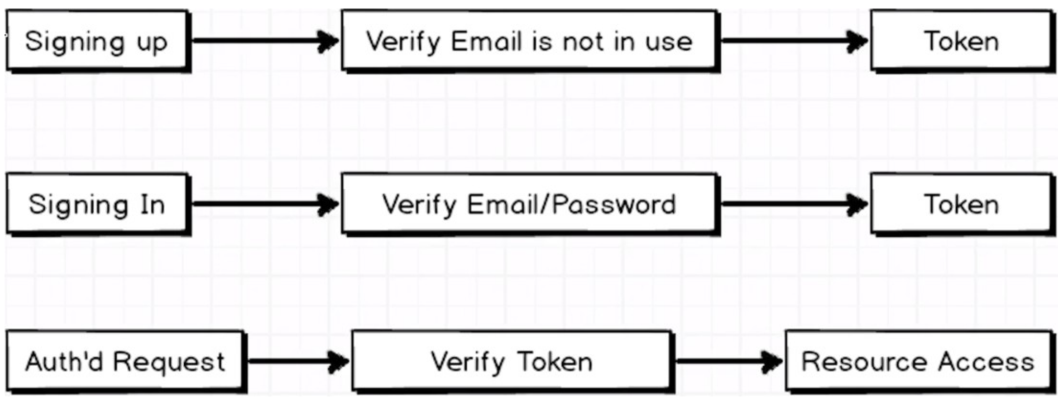
Body | Cookies | Headers (6) | Test Results

Pretty | Raw | Preview | JSON

```json
{
    "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIi
        .Z7HY_smGwiREzE7aT7WDv_ju3G4ZFasnnjrKe9Ggmos"
}
```

Server Review:

The key to our server flow is outlined in the diagram below – we have three possible flows coming in to our server., either signing up, signing in, or trying to access some protected resource.

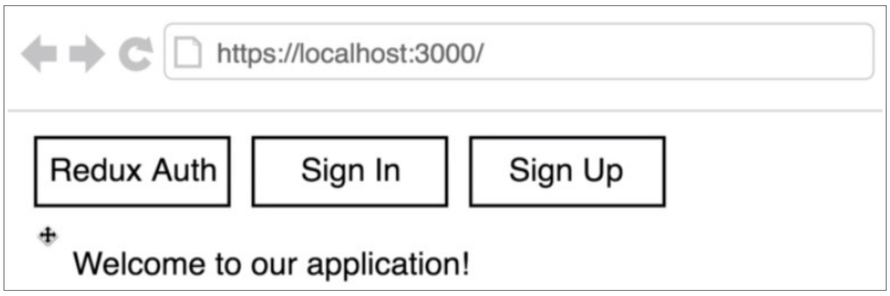


1. When a user is first signing up, we verified that the email was unique, then gave the user an identifying token. We encoded the users ID, and a timestamp into the token.
2. When a pre-existing user signs in, we verified that their email and password were correct using the loalStrategy plubin for passport. When we verified their identify, we supplied them with a token.
3. When a user makes a request for a protected resource, we verify their token, and once it is deemed valid we respond with the resource.

Client Side Auth:

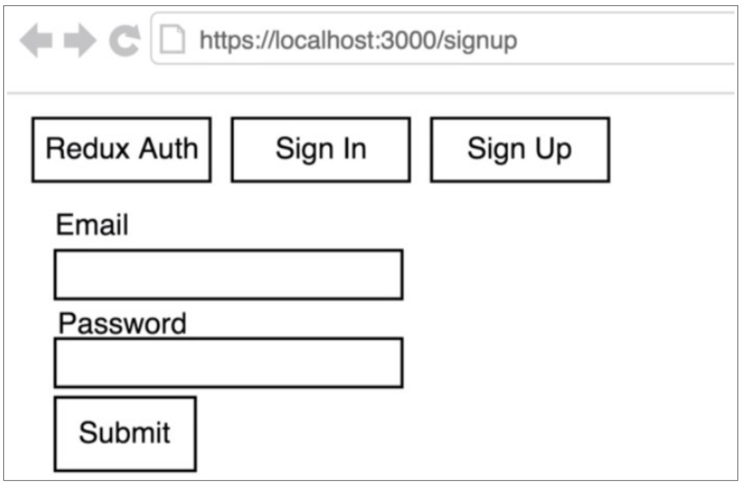
Client Overview:

Now that we have our backend put together, we will work on a front-end to communicate with our API server.



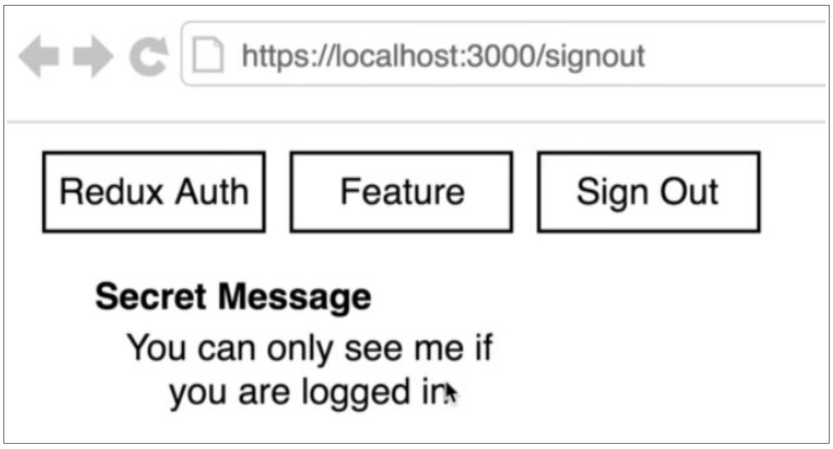
Our App will have pretty basic styling, but the focus of the application is to interact with our backend server, creating an account, and signing in. When a user visits the page, they will see the three buttons above, the “Redux Auth” button will return the user back to the home page of localhost:3000/. The other two buttons will allow new users to sign up with an email address and password, and allow pre-existing users to sign in.

If a user tries to sign up, we will show them the form below. There will be a field for both the email address and password. When the user clicks submit, we will take that info and send it to the backend server where a new account will be generated. The form for signing in will be pretty much identical. In both these cases, the server will provide the client with a JWT token.



When the user tries to sign out, they can be brought to a signout route, where we will need to do something with the token to reflect their signing out.

Most importantly, the user, when signed in, will gain access to the feature page, which they will only be able to see once signed into the application. The secret message will be retrieved from the back-end server, and will not be acessible unless the user is signed into the application.



Secret Message

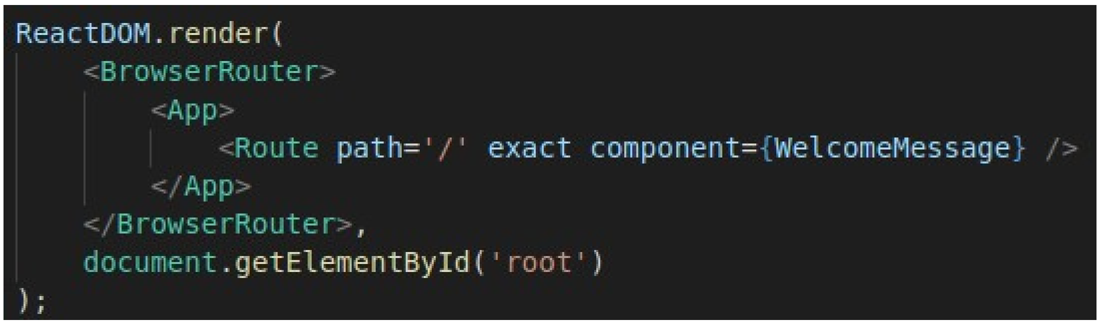
You can only see me if you are logged in

To get started, we will install the following libraries:

- react-router-dom
- redux
- react-redux
- axios
- redux-thunk

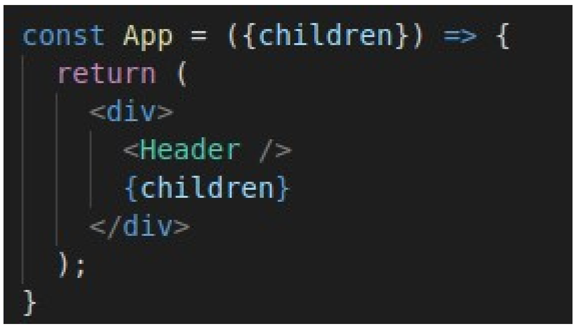
We can make a quick scaffold of the landing page with some basic routing as follows:

1. We will implement the router in the index.js file.



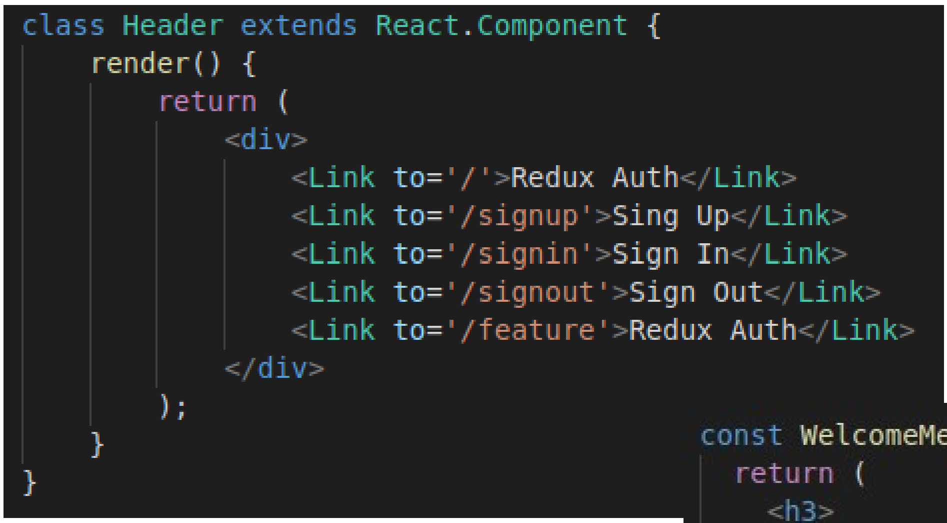
```
ReactDOM.render(
    <BrowserRouter>
        <App>
            <Route path='/' exact component={WelcomeMessage} />
        </App>
    </BrowserRouter>,
    document.getElementById('root')
);
```

2. The App component will house both the Route component (available through props.children), and the Header:

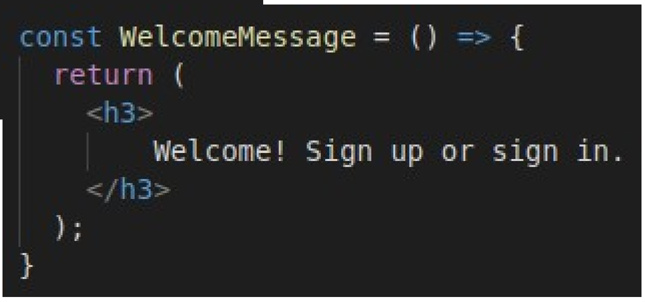


```
const App = ({children}) => {
  return (
    <div>
      <Header />
      {children}
    </div>
  );
}
```

The Header componenet will house the different Link components, controlling the URL and which components can be shown. Due to the Route config in index.js, the welcome message will only be shown at the ‘/’ URL.



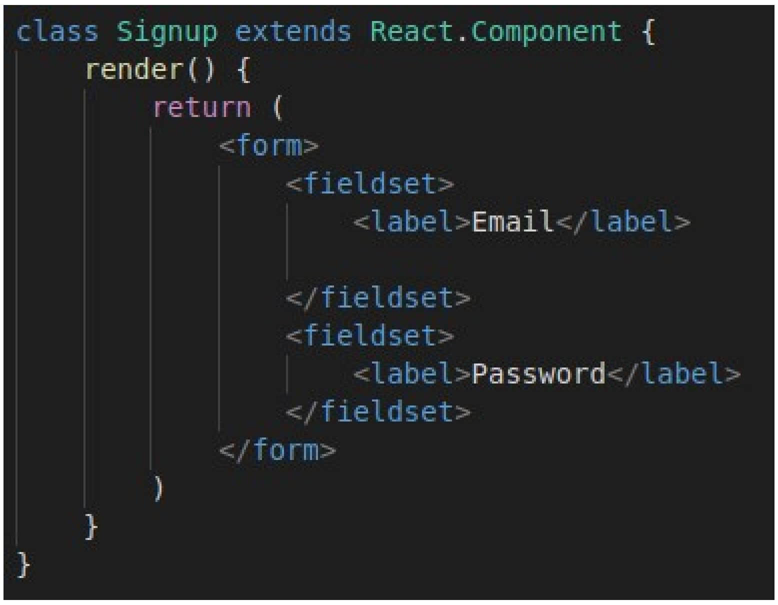
```
class Header extends React.Component {
    render() {
        return (
            <div>
                <Link to='/'>Redux Auth</Link>
                <Link to='/signup'>Sing Up</Link>
                <Link to='/signin'>Sign In</Link>
                <Link to='/signout'>Sign Out</Link>
                <Link to='/feature'>Redux Auth</Link>
            </div>
        );
    }
}
```



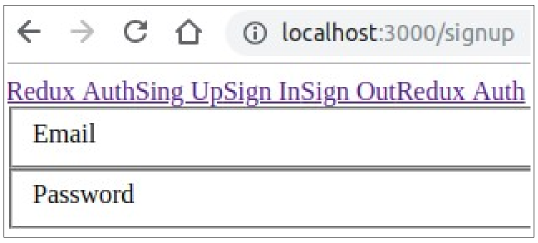
```
const WelcomeMessage = () => {
  return (
    <h3>
        Welcome! Sign up or sign in.
    </h3>
  );
}
```

Scaffolding the Singup Form:

Now we can get started on the signup form. We can make a new folder in the components directory where we can house all the auth-related files for the header. Note the use of **fieldset** elements for housing related elements in a form. We will deal with the text inputs later.



```
class Signup extends React.Component {
    render() {
        return (
            <form>
                <fieldset>
                    <label>Email</label>

                </fieldset>
                <fieldset>
                    <label>Password</label>
                </fieldset>
            </form>
        )
    }
}
```

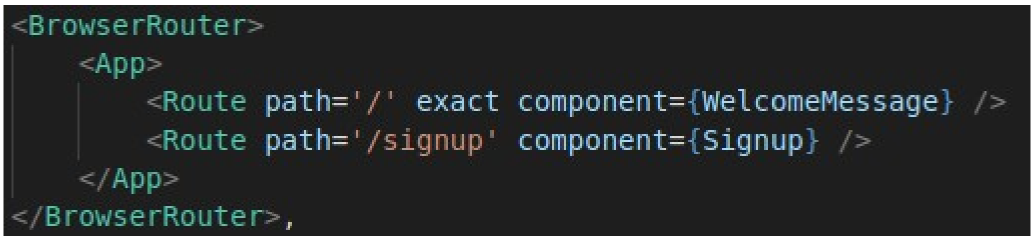


localhost:3000/signup

Redux AuthSing UpSign InSign OutRedux Auth

Email

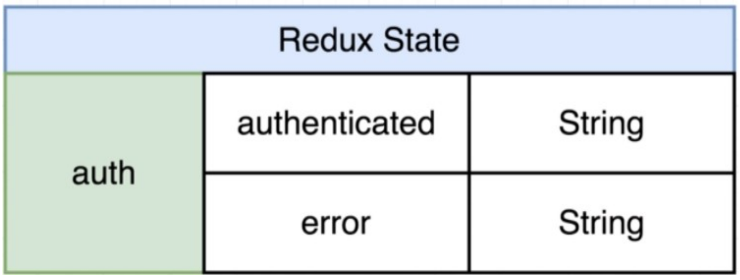
Password



```
<BrowserRouter>
    <App>
        <Route path='/' exact component={WelcomeMessage} />
        <Route path='/signup' component={Signup} />
    </App>
</BrowserRouter>,
```

<u>Setting up Redux:</u>

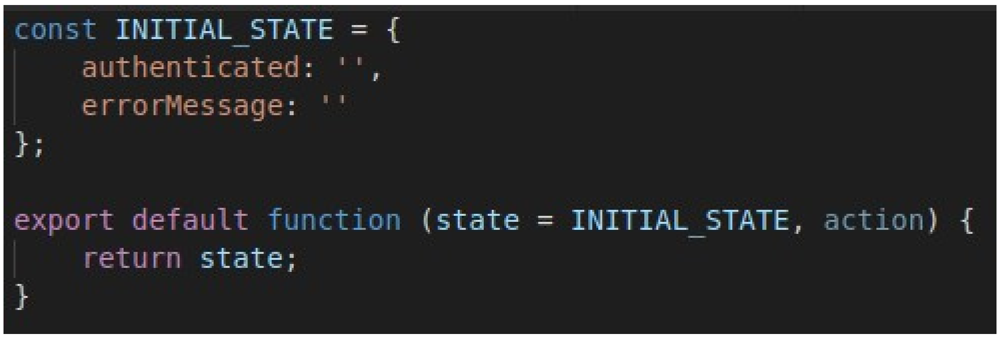
Lets quickly set up redux then discuss the different reducers this application will need.



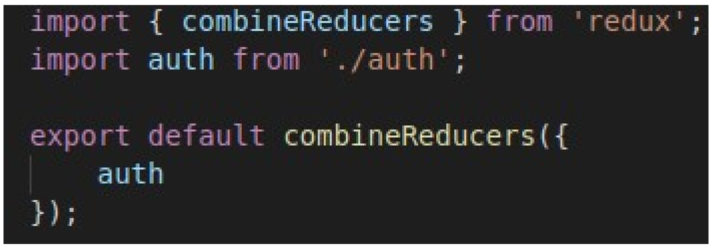
Redux State		
auth	authenticated	String
	error	String

Since this application is quite simple, we will only need an “authenticated” piece of state, which will have two possible properties. The authenticated properties value depends on the presence of a JWT. We will assign our JWT to the auth property. If its an empty string, we will consider the user not signed in. We will also keep an error property in state, which will signify if there is an error in a users signup/signin request.

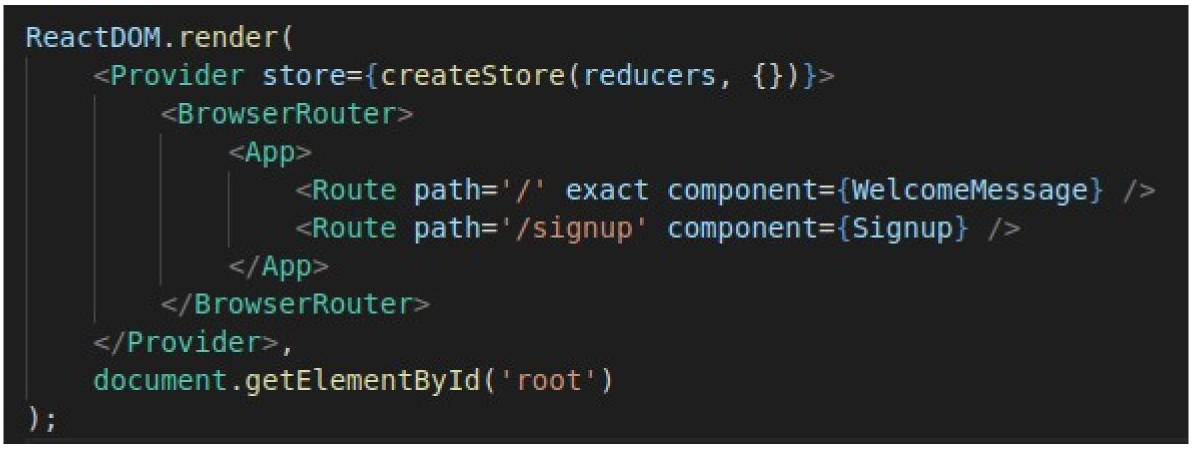
We can write some boilerplate for the auth reducer, and add it to a combineReducers call:



```
const INITIAL_STATE = {
    authenticated: '',
    errorMessage: ''
};

export default function (state = INITIAL_STATE, action) {
    return state;
}
```



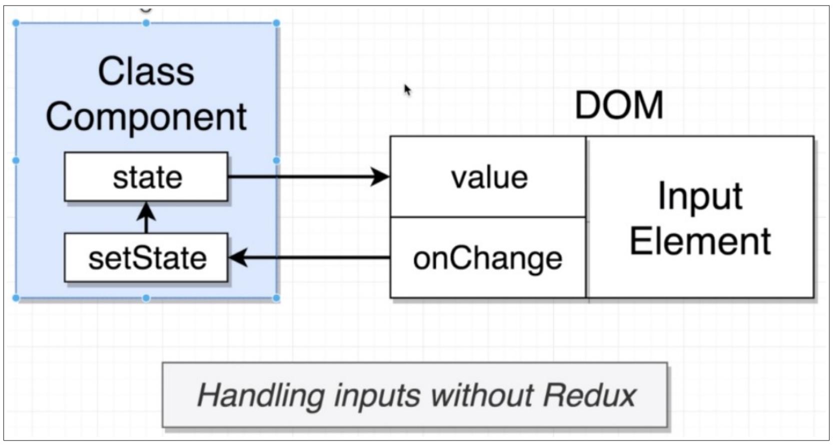
```
import { combineReducers } from 'redux';
import auth from './auth';

export default combineReducers({
    auth
});
```

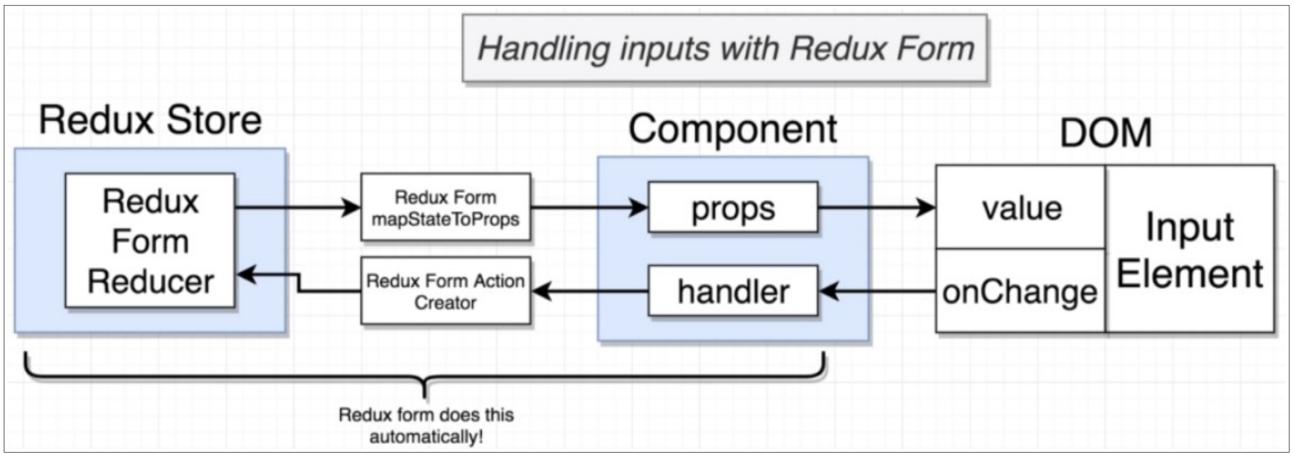


```
ReactDOM.render(
    <Provider store={createStore(reducers, {})}>
        <BrowserRouter>
            <App>
                <Route path='/' exact component={WelcomeMessage} />
                <Route path='/signup' component={Signup} />
            </App>
        </BrowserRouter>
    </Provider>,
    document.getElementById('root')
);
```

ReduxForm for Signup:

With Redux, we want to hold all of the data inside of the redux store, and any time a user changes an element, we will call an action creator which will interact with that redux store. Compare the two diagrams below, where we show a pure React method of handling data, vs Redux-Forms method.

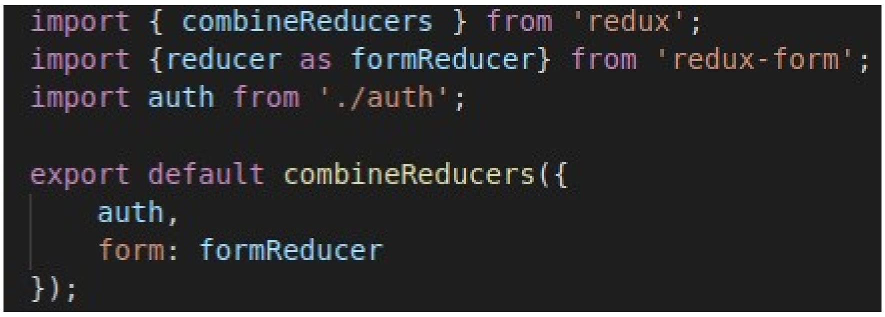



Handling inputs with Redux Form
Redux Store
Redux Form Reducer
Redux Form mapStateToProps
Redux Form Action Creator
Component
props
handler
DOM
value
onChange
Input Element
Redux form does this automatically!

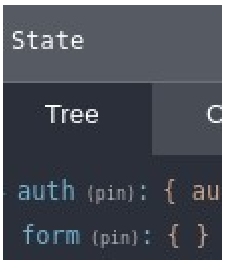
On the left hand side of the redux-form diagram, we have a redux form reducer which is used to communicate with the componend via mapStateToProps. This allows for the component to use its props to change the DOM elements value. We assign an action creator to the onChange handler for the element. All changes to the State are handled by communicating via the components props (we can use **mapStateToProps** and **connect** to make sure the component has access to both the current state and action creators).

In short, we don't let the DOM hold information about our app. Rather we let Redux hold this information and push it to the DOM. The good thing about redux form is that it does the jobs of the reducer and action creator for us automatically.

Lets get started wiring up redux form to our project. First, go to **reducers/index.js**. We need to add an out-of-the-box redux-form reducer to our combineReducers call. Now we will see 'form' in our state, which will be managed by formReducer imported from redux-form.

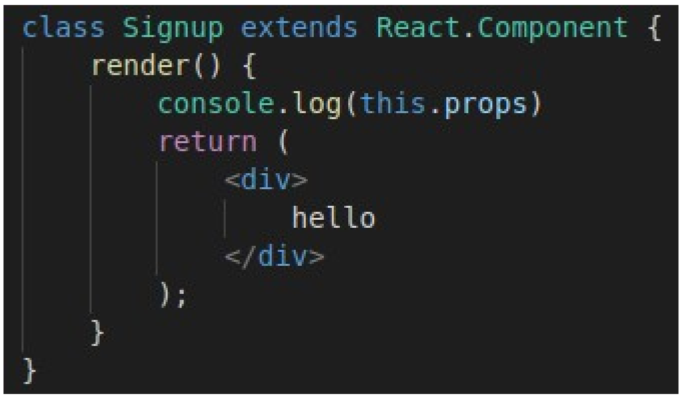


```
import { combineReducers } from 'redux';
import {reducer as formReducer} from 'redux-form';
import auth from './auth';

export default combineReducers({
    auth,
    form: formReducer
});
```

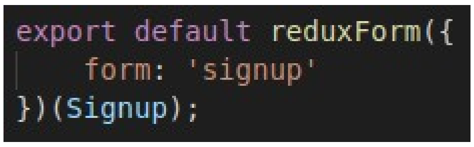


```
State
  Tree    C
auth (pin): { au
form (pin): { }
```

Now we can get started on the form component. First we will import **Field** and **reduxForm** helpers from redux-form. We will use reduxForm similarly to **connect**, wrapping the export. This takes a config object as an argument, with the name of this particular form.

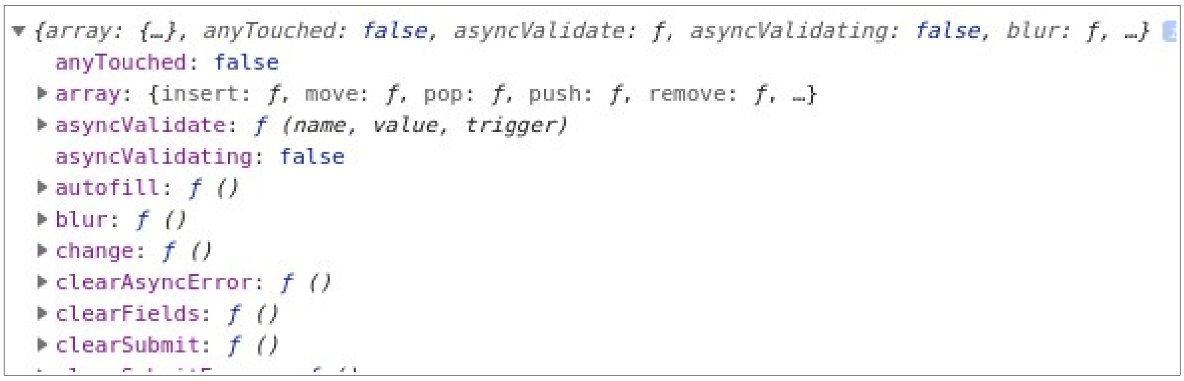


```
class Signup extends React.Component {
    render() {
        console.log(this.props)
        return (
            <div>
                hello
            </div>
        );
    }
}
```



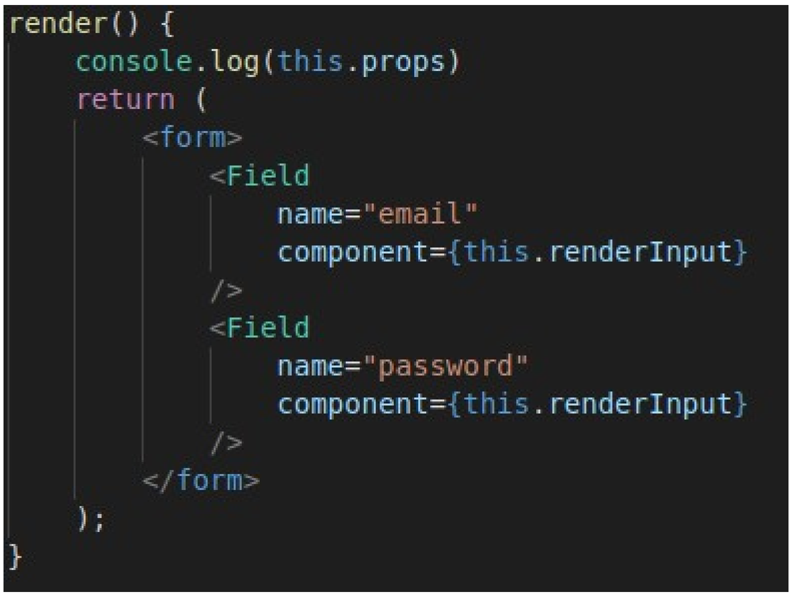
```
export default reduxForm({
    form: 'signup'
})(Signup);
```

Just by doing this, our form will get access to a bunch of new props for free, which we can see logged out below:

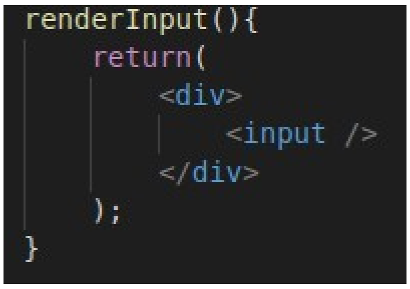


```
▼{array: {…}, anyTouched: false, asyncValidate: f, asyncValidating: false, blur: f, …}
   anyTouched: false
  ▶array: {insert: f, move: f, pop: f, push: f, remove: f, …}
  ▶asyncValidate: f (name, value, trigger)
   asyncValidating: false
  ▶autofill: f ()
  ▶blur: f ()
  ▶change: f ()
  ▶clearAsyncError: f ()
  ▶clearFields: f ()
  ▶clearSubmit: f ()
```

Next, we can write the JSX for the form. We will wrap the form in **<form>** tags, and then use the redux-form **<Field/>** wrapper around the actual elements. The Field element has a mandatory name prop, which decribes the property this element is responsible for. We also need to tell the Field which component to render – it doesn't render anything by default, its used purely for communication to the form state. For the component prop, we will pass a reference to a helper function for rendering the JSX. The Field element provides a bunch of props to its children to help with handling events.

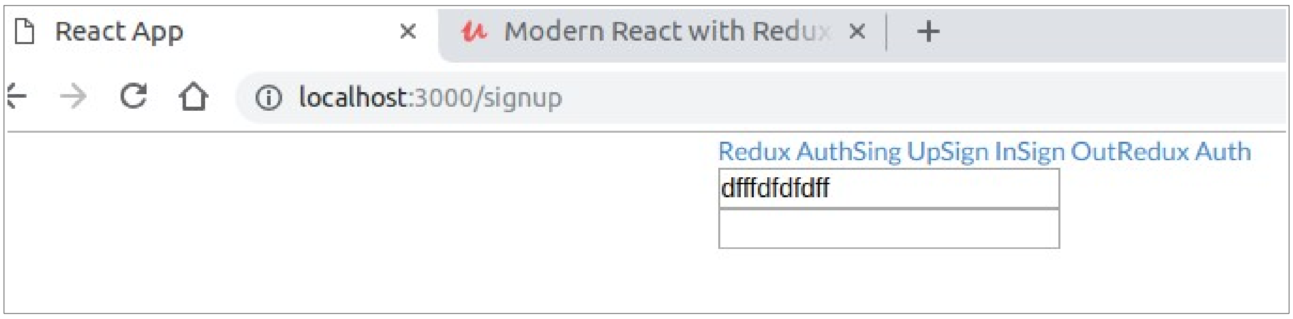


```
render() {
    console.log(this.props)
    return (
        <form>
            <Field
                name="email"
                component={this.renderInput}
            />
            <Field
                name="password"
                component={this.renderInput}
            />
        </form>
    );
}
```



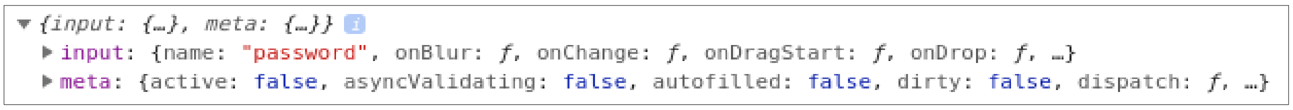
```
renderInput(){
    return(
        <div>
            <input />
        </div>
    );
}
```

This will now be enough to see the following in the app:

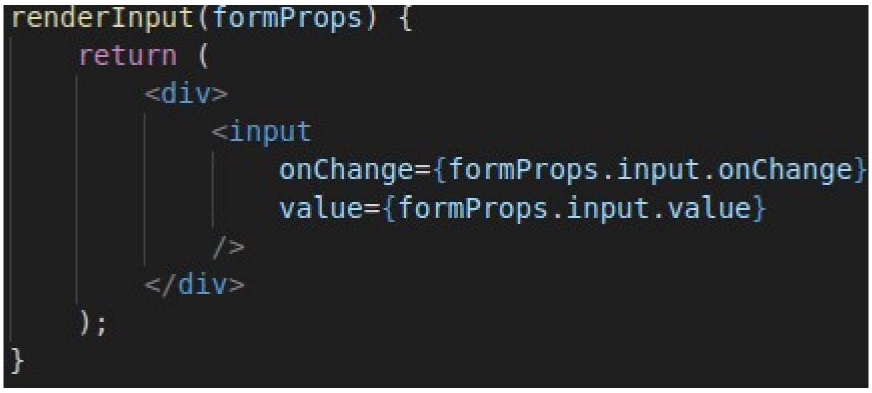


It is important to note, however, that this still isn't a controlled element. We still have to make sure that when a user types into the form input elements, it triggers an action creator which updates the state, which is communicated back to the element so it knows what to display.

We need to make sure the input elements onChange handler refers to the handler provided by the field element:

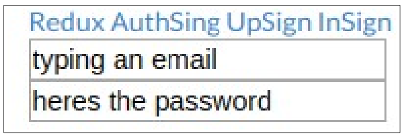


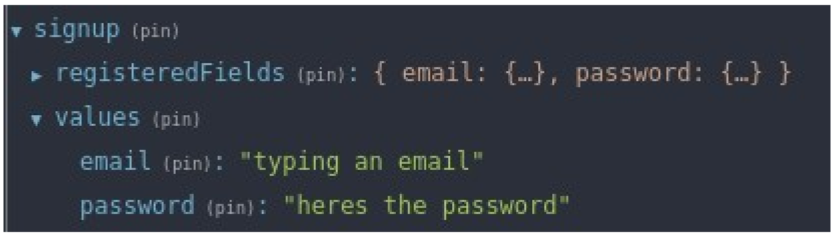
```
▼ {input: {…}, meta: {…}}
  ▶ input: {name: "password", onBlur: ƒ, onChange: ƒ, onDragStart: ƒ, onDrop: ƒ, …}
  ▶ meta: {active: false, asyncValidating: false, autofilled: false, dirty: false, dispatch: ƒ, …}
```



```
renderInput(formProps) {
    return (
        <div>
            <input
                onChange={formProps.input.onChange}
                value={formProps.input.value}
            />
        </div>
    );
}
```

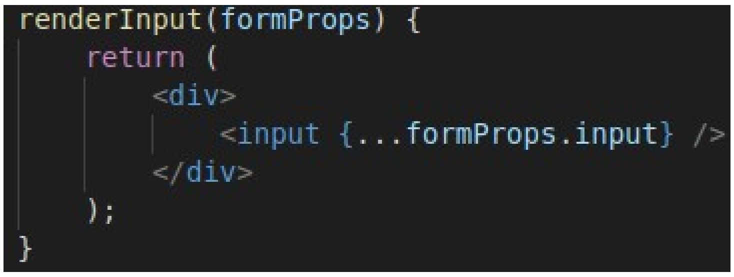
Now when we type in the input boxes, we will get the following updates in state:





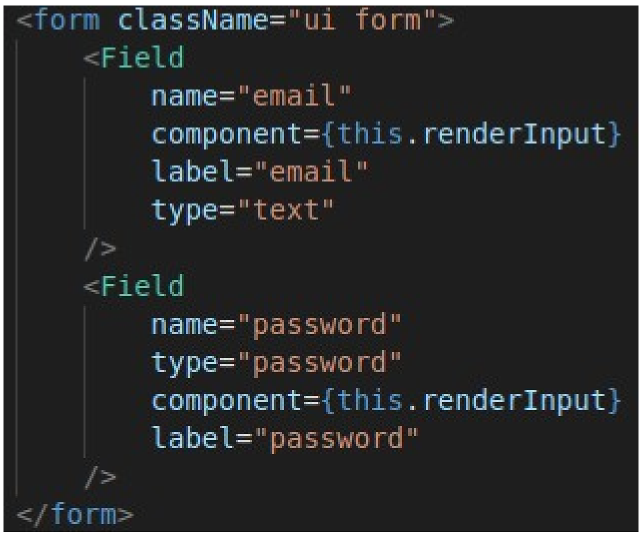
```
▾ signup (pin)
  ▸ registeredFields (pin): { email: {…}, password: {…} }
  ▾ values (pin)
      email (pin): "typing an email"
      password (pin): "heres the password"
```

We can use es6 rest/spread syntax to copy the formProps object passed into the renderInput function, and use it as props in a single-line expression:

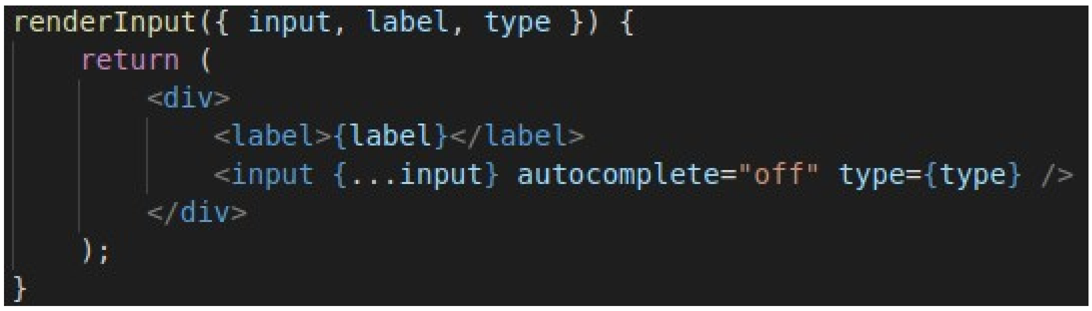


```
renderInput(formProps) {
    return (
        <div>
            <input {...formProps.input} />
        </div>
    );
}
```

The next thing we need to do is ensure the labels are rendered correctly beside each input element. This can be done by providing an additional prop to the Field – label. Field won't know what to do with this so it will just pass it to its children which we can make use of in the renderInput function as follows:



```
<form className="ui form">
    <Field
        name="email"
        component={this.renderInput}
        label="email"
        type="text"
    />
    <Field
        name="password"
        type="password"
        component={this.renderInput}
        label="password"
    />
</form>
```

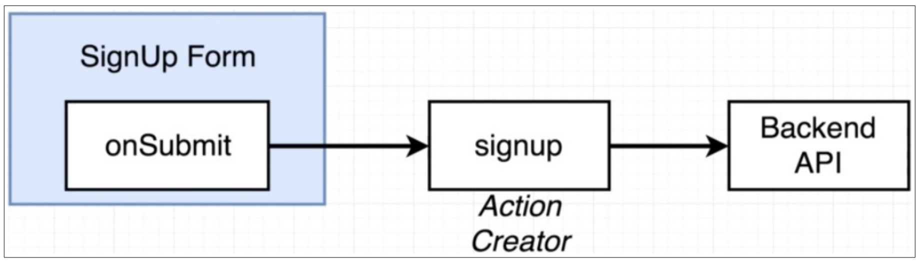


```
renderInput({ input, label, type }) {
    return (
        <div>
            <label>{label}</label>
            <input {...input} autocomplete="off" type={type} />
        </div>
    );
}
```

Now that we have the email and password stored into the state, how do we take this data and send an API request to the backend API?

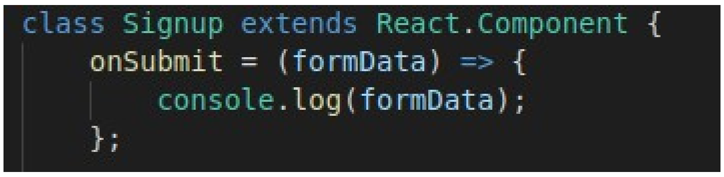
Handling Form Submission:

We have our Signup form component and we will have an onSubmit handler which is called when the user submits the form. We can give the component access to an action creator which will contact API when this onSubmit handler is triggered – which will exchange the email and password for a JWT.

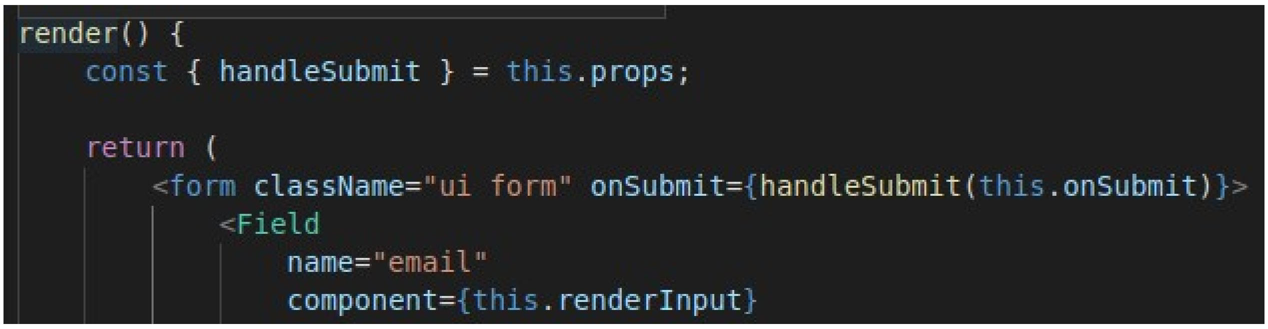


We can define a class method which will take the formData as an object, and console log it out. We have a function in our props **handleSubmit** provided by reduxForm, in which we need to call this onSubmit function.

We can destructure reduxForm's **handleSubmit** function from **this.props** which is passed to the component. Next, we will need to reference this function inside the form's onSubmit handler, passing the function we want to use into the argument for handleSubmit. Redux-Form's handleSubmit function will automatically take the data from the form and passes it into the function it refers to:

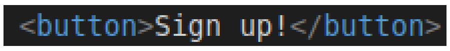


```
class Signup extends React.Component {
    onSubmit = (formData) => {
        console.log(formData);
    };
```

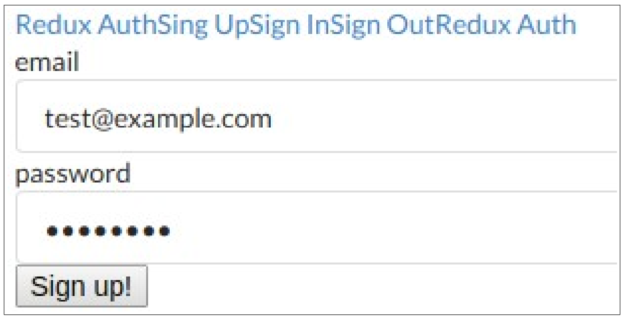


```
render() {
    const { handleSubmit } = this.props;

    return (
        <form className="ui form" onSubmit={handleSubmit(this.onSubmit)}>
            <Field
                name="email"
                component={this.renderInput}
```

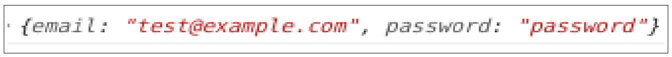
Lastly, we need to make sure there is a button before the closing form tag, so the user can click to submit the button itself:



```
<button>Sign up!</button>
```



Redux AuthSing UpSign InSign OutRedux Auth
email
test@example.com
password
••••••••
Sign up!

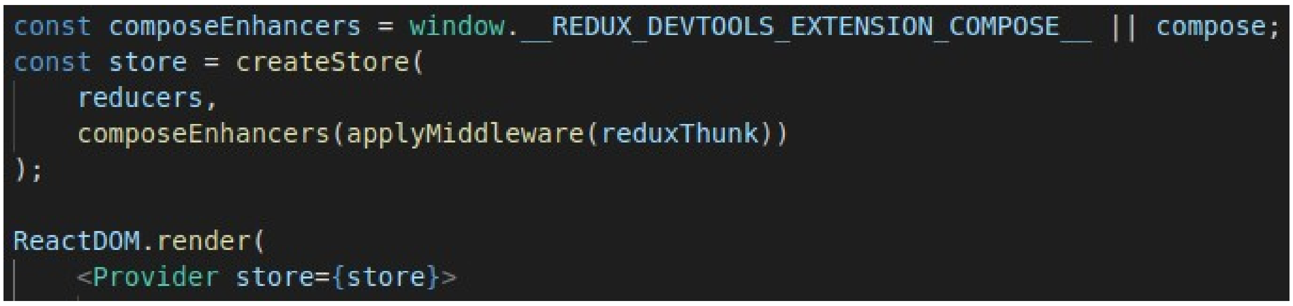


```
{email: "test@example.com", password: "password"}
```

Redux Thunk:

Before we set up our auth action creator, we need to utilize **redux-thunk** which will allow us to return functions from actions, which will automatically be passed **getState** and **dispatch** arguments, allowing us to carry out an async request and **dispatch** the action after **await**ing the response.

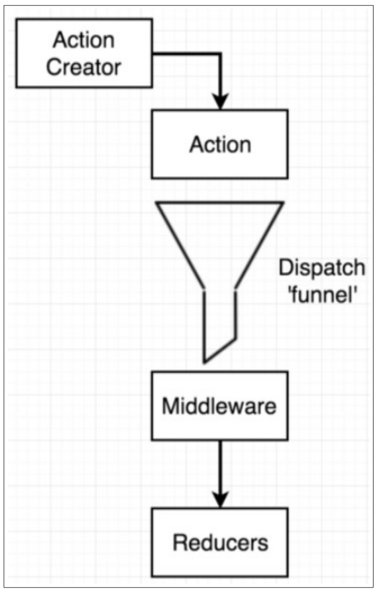
To wire up this middleware, we use the **applyMiddleware** function which we imported from Redux, and pass it in as an argument into the **createStore** call. Note the composeEnchancers call which is used for setting up redux dev tools.



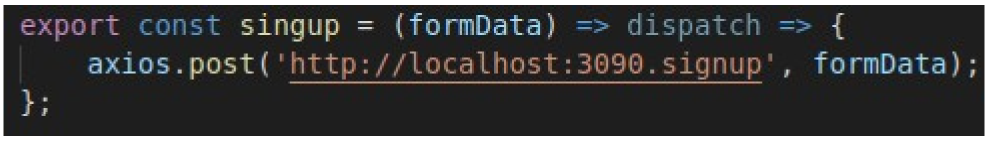
```
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;
const store = createStore(
    reducers,
    composeEnhancers(applyMiddleware(reduxThunk))
);

ReactDOM.render(
    <Provider store={store}>
```

Redux' dispatch function is how we initiate change in our appliation. It works as follows:

- Action creator returns an action.
- Redux sees this action and dispatches it to the reducer.
- It can be intercepted by middleware on the way.
- Redux thunk is a middleware which will check if a function is returned by an action, in which case it passes the dispatch and getState arguments in, allowing us to make async requests.



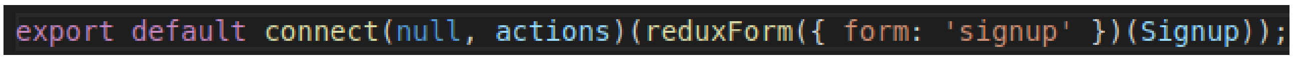
We can define the following action creator, and wire it up to the Signup component using **connect** and **mapDispatchToProps**.



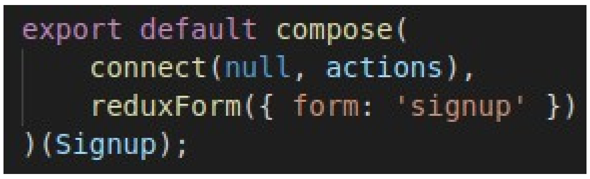
```
export const singup = (formData) => dispatch => {
    axios.post('http://localhost:3090.signup', formData);
};
```

Wrapping the export in both **reduxForm** and a **connect** helper makes for nasty syntax, so we can do the formWrapping and connect using a helper function provided by redux – the **compose** function.

Compose lets us to wrap a component in multiple higher order components with much cleaner syntax. To sue compose, we just wrap the export in a compose function and add the different HOCs that we wish to apply as arguments into compose. Compare the following:

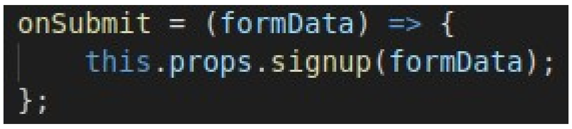


```
export default connect(null, actions)(reduxForm({ form: 'signup' })(Signup));
```



```
export default compose(
    connect(null, actions),
    reduxForm({ form: 'signup' })
)(Signup);
```

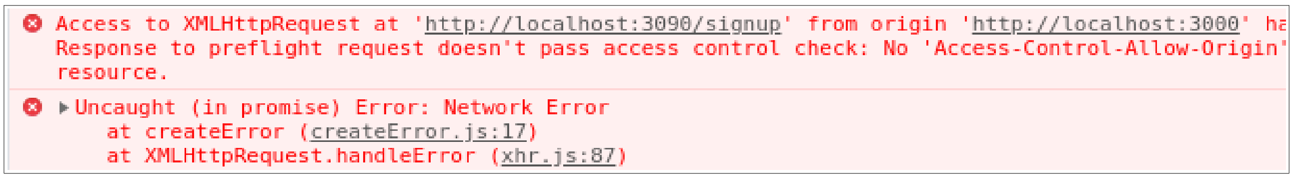
Now we will have access to the signup action creator inside the props of the Signup component, and can thus call it inside the **onSubmit** handler:



```
onSubmit = (formData) => {
    this.props.signup(formData);
};
```

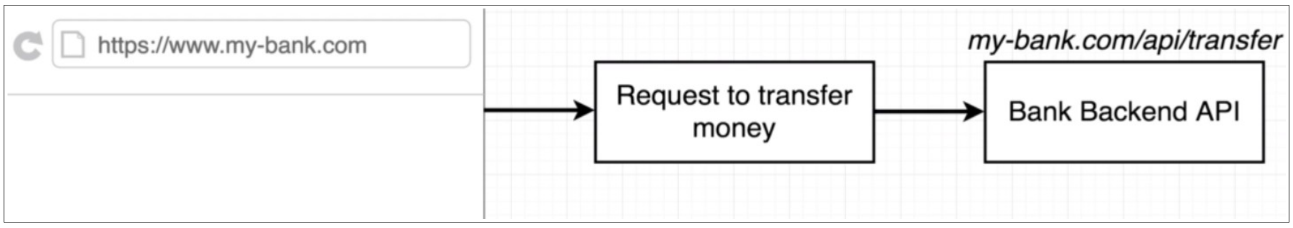
CORS Errors:

Now, when we submit the data, the signup action creator is successfully called, but the network request, and the request is attempted, but is blocked by a **CORS error**. This stands for cross-origin-resource-sharing.

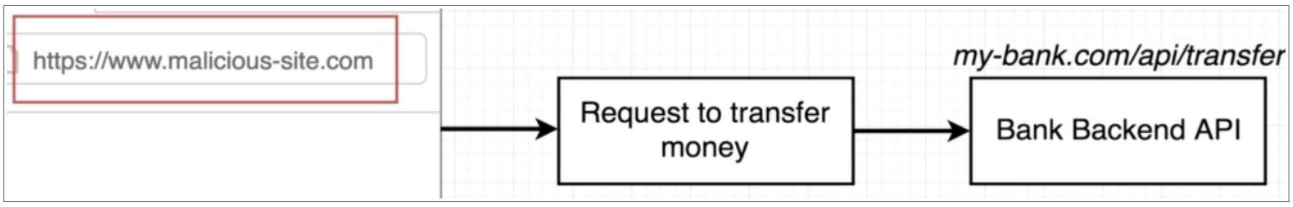


```
Access to XMLHttpRequest at 'http://localhost:3090/signup' from origin 'http://localhost:3000' ha
Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin'
resource.
▶Uncaught (in promise) Error: Network Error
    at createError (createError.js:17)
    at XMLHttpRequest.handleError (xhr.js:87)
```

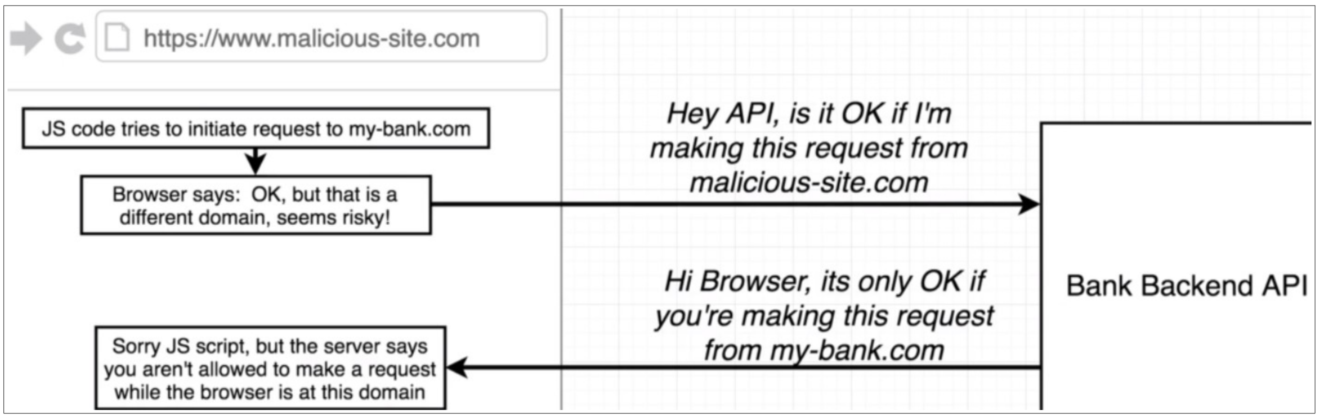
CORS is a security feature implemented inside the browser. Why are we getting this error? Imagine we were working on a backend banking API, and some malicious person creates a new website, and tries to trick our users to visit it. The user uses the malicious button to make the exact same request.



The user enters the same credentials, and makes a request to the same domain. The only thing that is different is that the user is at a new domain. We want to prevent this impersonator website making this request. This is prevented using CORS restrictions.



How does this work behid the scenes?


https://www.malicious-site.com
JS code tries to initiate request to my-bank.com
Browser says: OK, but that is a different domain, seems risky!
Hey API, is it OK if I'm making this request from malicious-site.com
Hi Browser, its only OK if you're making this request from my-bank.com
Bank Backend API
Sorry JS script, but the server says you aren't allowed to make a request while the browser is at this domain

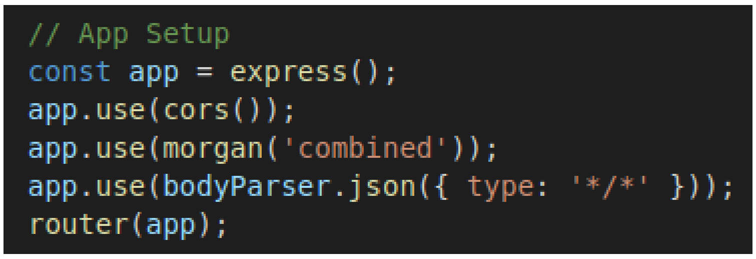
- The user visits the imposter website
- JS loads up inside the browser.
- The user makes a request to the banking server from this malicious website.

- The banks backend server will see the request is being made from a different domain (or subdomain or port) to the one the backend is currently at.
- The browser will view this as suspicious, and it will ask the bank's backend if the origin domain is okay.
- The backend will have a list of approved domains that it will allow resource sharing between.
- If the original request didn't come from an approved domain, the request will be denied.

The browser thus makes a 'preflight' request, and asks the API is the request's domain okay? The express API for the backend rejects this request since it is on a different domain/port. The browse then tells our JS that the request was blocked, and an error is thrown. This feature is hard-coded into browsers. Thats why postman was allowed to make requests to different APIs, Postman doesn't have these security features implemented.

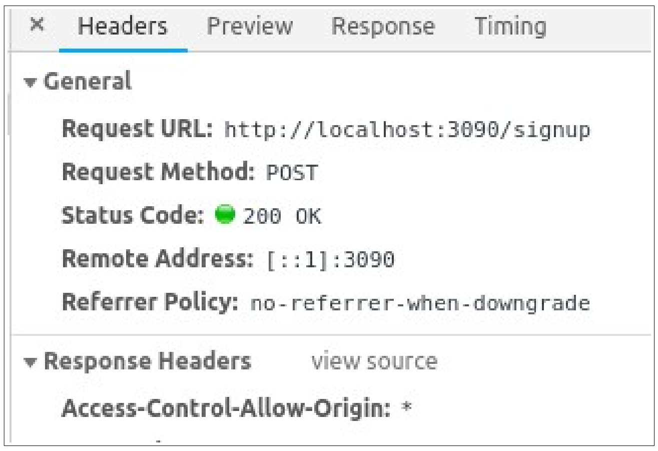
To make our application work, we will need to change the way out backend API views cross-origin requests. We need to install the **cors** package into our server directory, and tell our application to make use of this middleware. If we want to just approve a specific domain (instead of any domain) we can do this too, detailed in the docs:

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

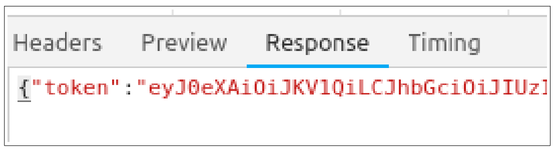


```
// App Setup
const app = express();
app.use(cors());
app.use(morgan('combined'));
app.use(bodyParser.json({ type: '*/*' }));
router(app);
```

Now if we resubmit the form request from our client, we can see the request is issued successfully, and the JWT in the response tab.



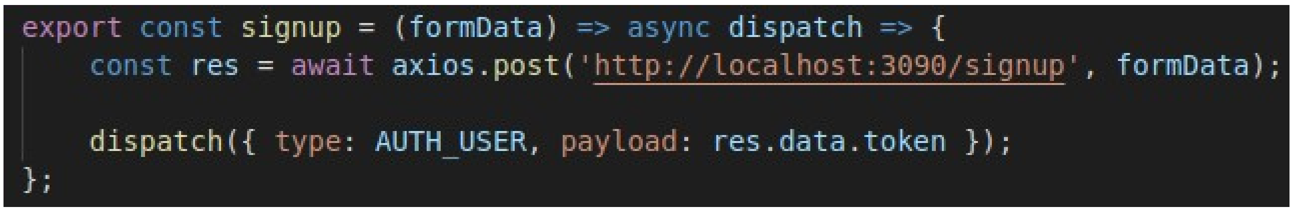
```
▾ General
    Request URL: http://localhost:3090/signup
    Request Method: POST
    Status Code: ● 200 OK
    Remote Address: [::1]:3090
    Referrer Policy: no-referrer-when-downgrade
▾ Response Headers
    Access-Control-Allow-Origin: *
```



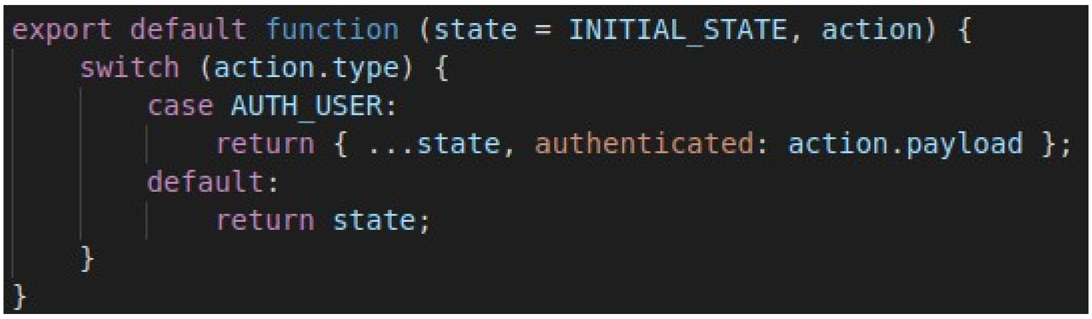
```
{"token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI
```

<u>Handling JWTs</u>:

Now that we have access to the JWT in the action creators HTTP response, we can configure a dispatch and a reducer to collect this data in the application state.

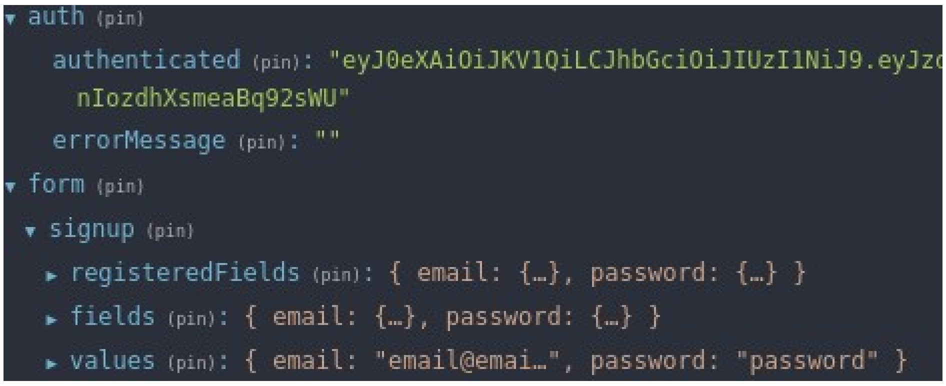


```
export const signup = (formData) => async dispatch => {
    const res = await axios.post('http://localhost:3090/signup', formData);

    dispatch({ type: AUTH_USER, payload: res.data.token });
};
```



```
export default function (state = INITIAL_STATE, action) {
    switch (action.type) {
        case AUTH_USER:
            return { ...state, authenticated: action.payload };
        default:
            return state;
    }
}
```

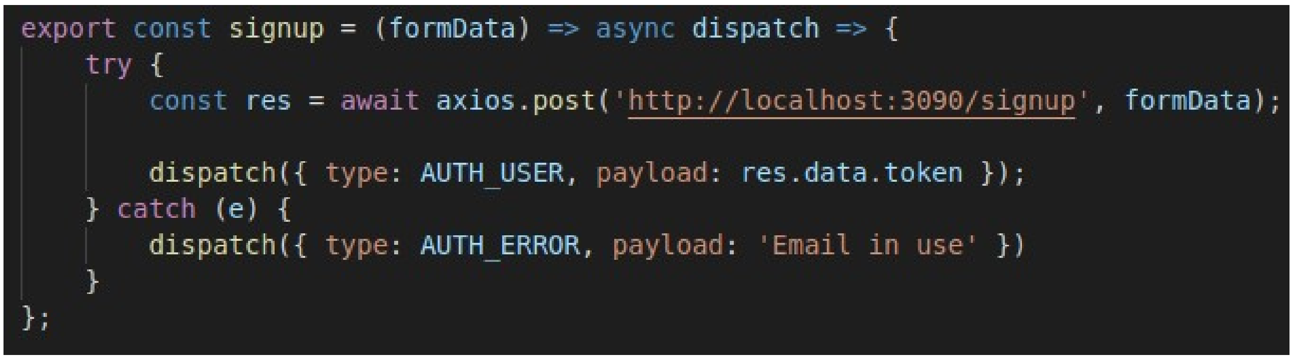
Now when a user signs up sucessfully, they will get the following saved to their state:



```
▾ auth (pin)
    authenticated (pin): "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzc
      nIozdhXsmeaBq92sWU"
    errorMessage (pin): ""
▾ form (pin)
  ▾ signup (pin)
    ▸ registeredFields (pin): { email: {…}, password: {…} }
    ▸ fields (pin): { email: {…}, password: {…} }
    ▸ values (pin): { email: "email@emai…", password: "password" }
```

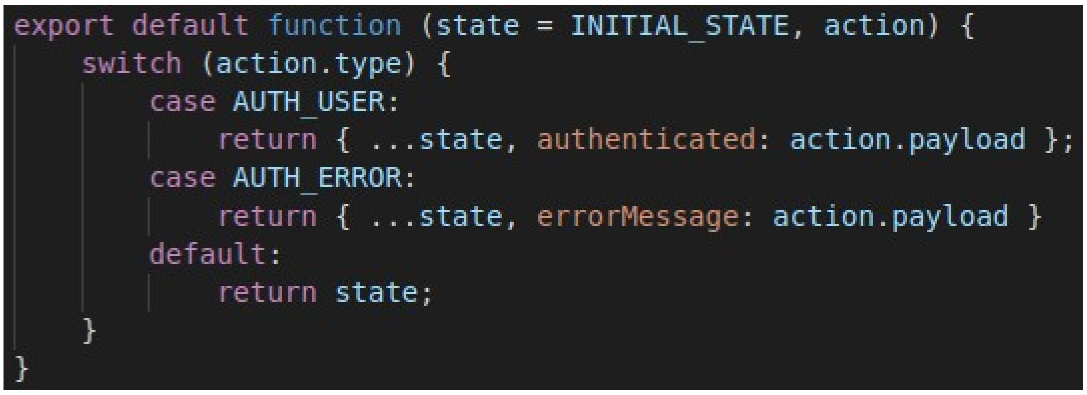
What about in the case where there is an error? In this case, we should dispatch an action where we notify the user. We can add this to our action creator with a **try/catch** statement, and tell our auth reducer to look for this error dispatch:

Action creator with try catch statement:

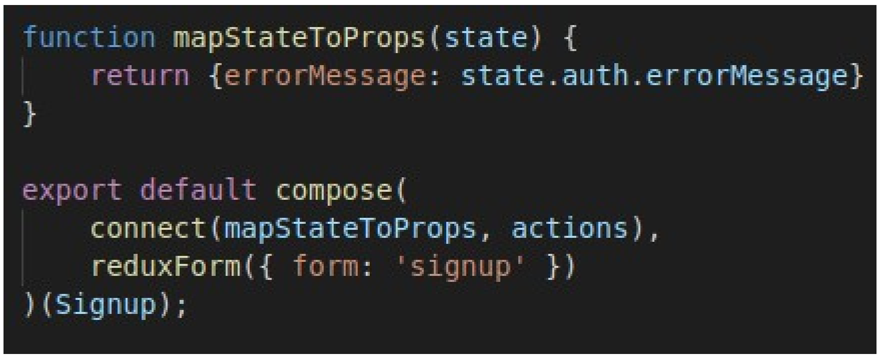


```
export const signup = (formData) => async dispatch => {
    try {
        const res = await axios.post('http://localhost:3090/signup', formData);

        dispatch({ type: AUTH_USER, payload: res.data.token });
    } catch (e) {
        dispatch({ type: AUTH_ERROR, payload: 'Email in use' })
    }
};
```

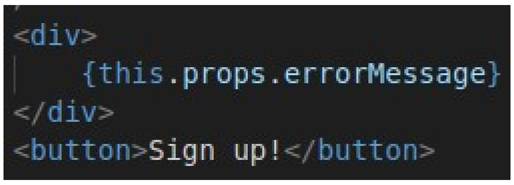
Auth reducer:

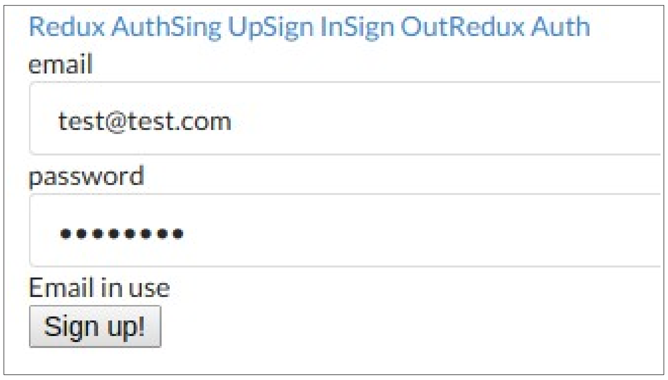

```
export default function (state = INITIAL_STATE, action) {
    switch (action.type) {
        case AUTH_USER:
            return { ...state, authenticated: action.payload };
        case AUTH_ERROR:
            return { ...state, errorMessage: action.payload }
        default:
            return state;
    }
}
```

Now we just need to pull this piece of state off the redux store (**mapStateToProps**) and provide it to the Signup component:


```
function mapStateToProps(state) {
    return {errorMessage: state.auth.errorMessage}
}

export default compose(
    connect(mapStateToProps, actions),
    reduxForm({ form: 'signup' })
)(Signup);
```

Finally, we need to make sure that if this error message is present, we display it in the form:


```
<div>
    {this.props.errorMessage}
</div>
<button>Sign up!</button>
```


Redux AuthSing UpSign InSign OutRedux Auth

email

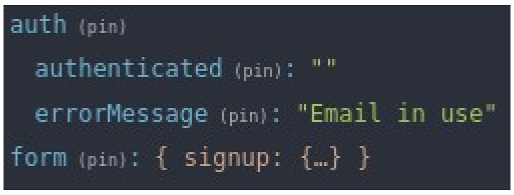
test@test.com

password

•••••••••

Email in use

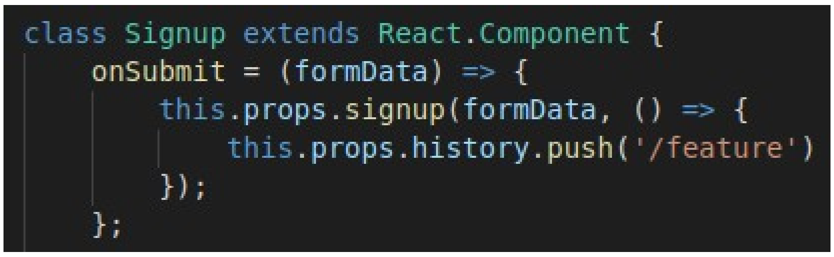
Sign up!


```
auth (pin)
  authenticated (pin): ""
  errorMessage (pin): "Email in use"
form (pin): { signup: {…} }
```

Redirect on Signup:

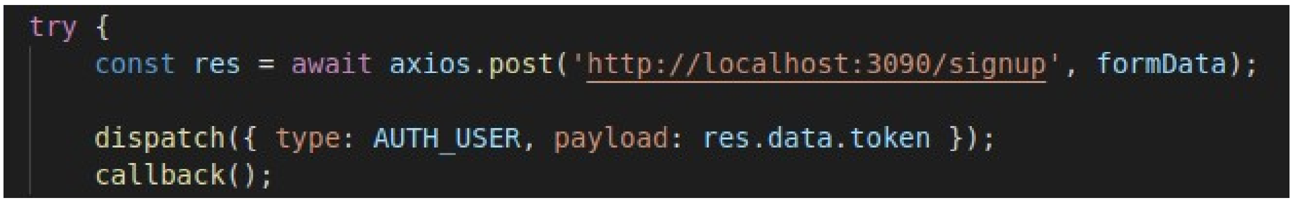
The next phase is to redirect the user when they have signed up sucessfully to the feature route. This route is supposed to be protected from users who aren't signed into the application. As a 2nd argument to the signup action creator (called in the component onSubmit handler), we can pass a callback which is invoked when a user sucessfully signs up.

We can do this with the **history** object provided by react router. We will send the users to a Feature component with some secret data fetched by the backend API:



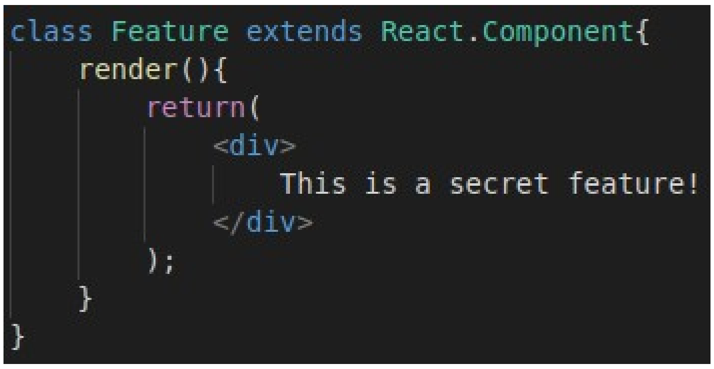
```
class Signup extends React.Component {
    onSubmit = (formData) => {
        this.props.signup(formData, () => {
            this.props.history.push('/feature')
        });
    };
```

We can then return to our action creator and make sure it invokes the callback after the user is sucessfully authenticated, inside the try statement:



```
try {
    const res = await axios.post('http://localhost:3090/signup', formData);

    dispatch({ type: AUTH_USER, payload: res.data.token });
    callback();
```

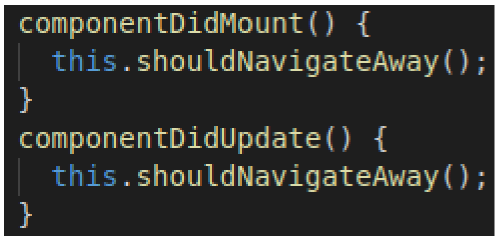
This will correctly redirect a user in when they sign up.



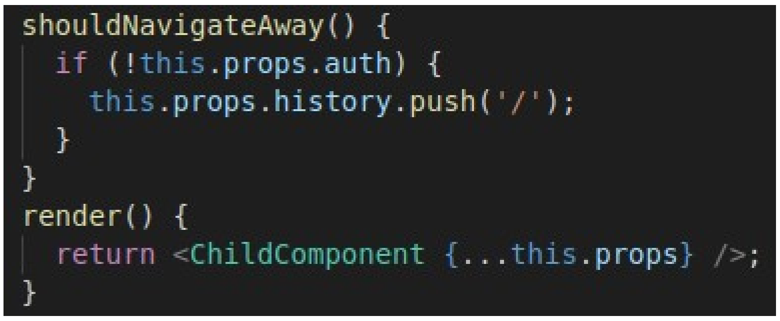
```
class Feature extends React.Component{
    render(){
        return(
            <div>
                This is a secret feature!
            </div>
        );
    }
}
```

We will make this a class method because we will later want to fetch some secret data with this components **componentDidMount** lifycycle method. Currently, we have an issue as users can manually navigate to this URL whether they are signed in or not. We can fix this using a higher order component that will restrict access to this page if the user is not signed in.

This HOC will render a child component only if the **state.auth.authenticated** is truthy, otherwise it will redirect the users to '/':



```
componentDidMount() {
  this.shouldNavigateAway();
}
componentDidUpdate() {
  this.shouldNavigateAway();
}
```



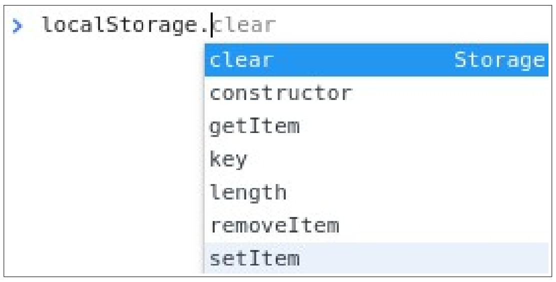
```
shouldNavigateAway() {
  if (!this.props.auth) {
    this.props.history.push('/');
  }
}
render() {
  return <ChildComponent {...this.props} />;
}
```

Persisting Login State:

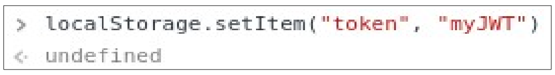
Now that we have implemented client-side authentication for viewing the '/feature' route, we will notice something interesting – if we refresh the page, we will lose our auth priviledges and be redirected back to the root directory!

This is because when we refresh the page we dump the current state and start fresh – our app has no idea that the user is still logged in after refreshing. To work around this, we will utilize a browser feature – **local store**, which we have access to inside of our browser. This is a great place to store small pieces of information, such as the JWT we get back after authenticating.

We can access localStorage within the chrome console as follows – note the different methods available:

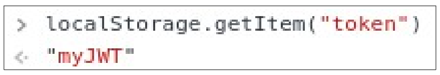


```
> localStorage.clear
                clear           Storage
                constructor
                getItem
                key
                length
                removeItem
                setItem
```



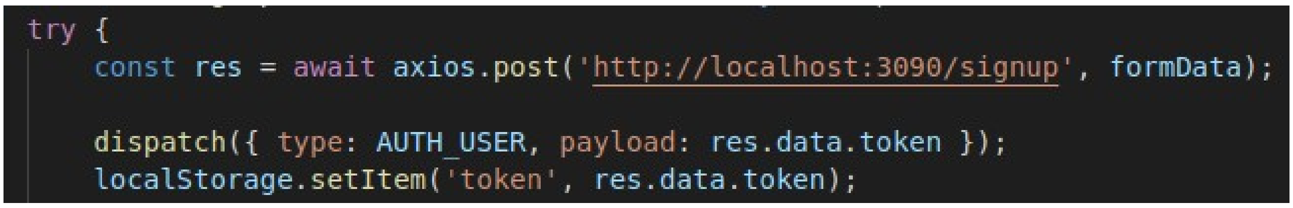
```
> localStorage.setItem("token", "myJWT")
<· undefined
```

After refresh:



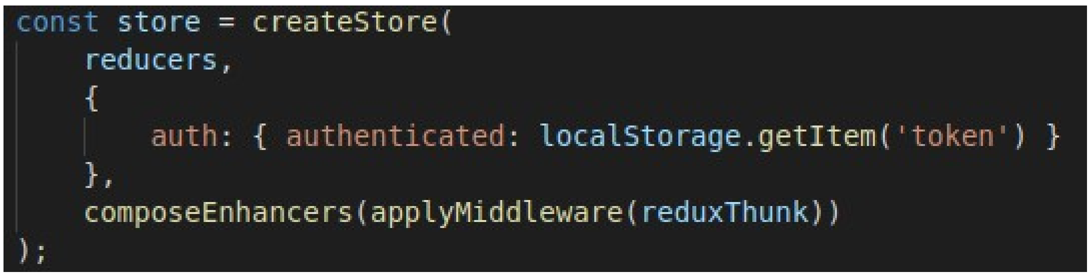
```
> localStorage.getItem("token")
<· "myJWT"
```

We can use **localStorage.setItem** to save a JWT to the local browser memory after a user signs up.



```
try {
    const res = await axios.post('http://localhost:3090/signup', formData);

    dispatch({ type: AUTH_USER, payload: res.data.token });
    localStorage.setItem('token', res.data.token);
```

Then when the app starts up we can use **getItem** to check to see do they have a JWT in their local storage. We can do this using the **initialState** which we set in the root index.js:



```
const store = createStore(
    reducers,
    {
        auth: { authenticated: localStorage.getItem('token') }
    },
    composeEnhancers(applyMiddleware(reduxThunk))
);
```

Now when the app first boots up, our app will pull that token out of local storage to authenticate a user.

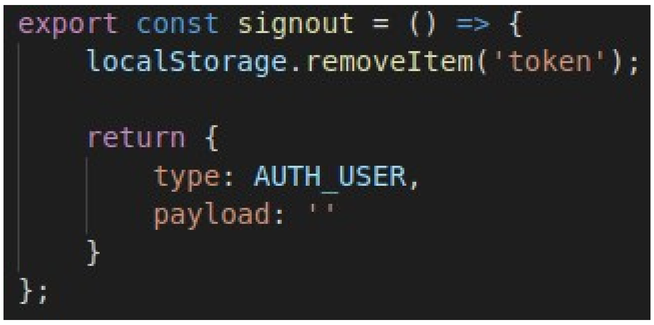
<u>Signing A User Out:</u>

Now that we can persist the login state, we need to make sure users can also log out. When a user visits the '/signout' route, we want to carry out the following 2 stage process:

- reach into localStorage and clear the token data.
- change the authenticated state to false or ''.

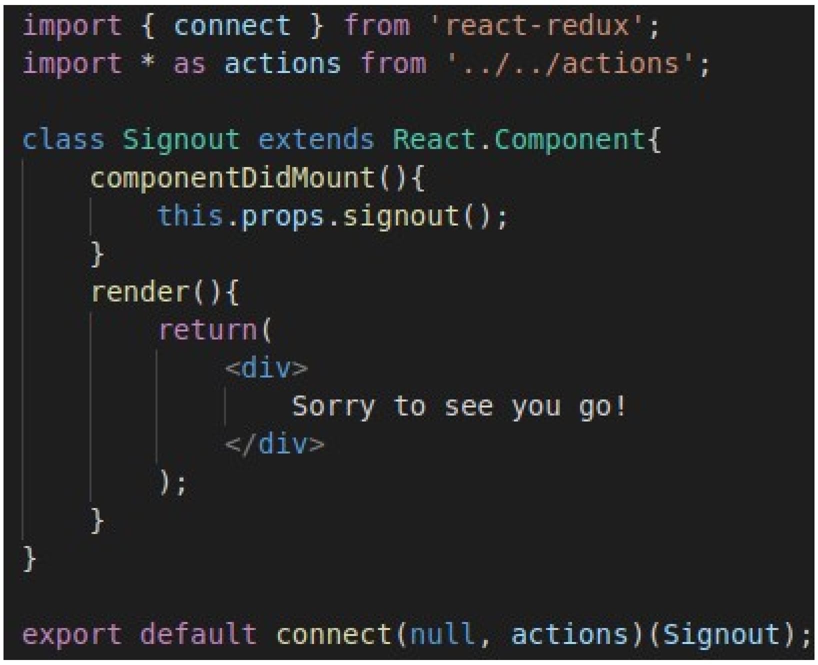
We will make this a class-based component as we will need to take advantage of lifycycle methods. We will set up this component in the **components/auth** directory and add a Route for it in App.js. Next, we will create an action creator to take care of the entire signout process, which we will fire on the Signout componentDidMount method.

This will be a normal synchronous action creator, and we will simply clear the localStorage using the **localStorage.removeItem('token')** method. Then we will return an action which reuses the **AUTH_USER** action type to reset the auth state to an empty string.



```
export const signout = () => {
    localStorage.removeItem('token');

    return {
        type: AUTH_USER,
        payload: ''
    }
};
```

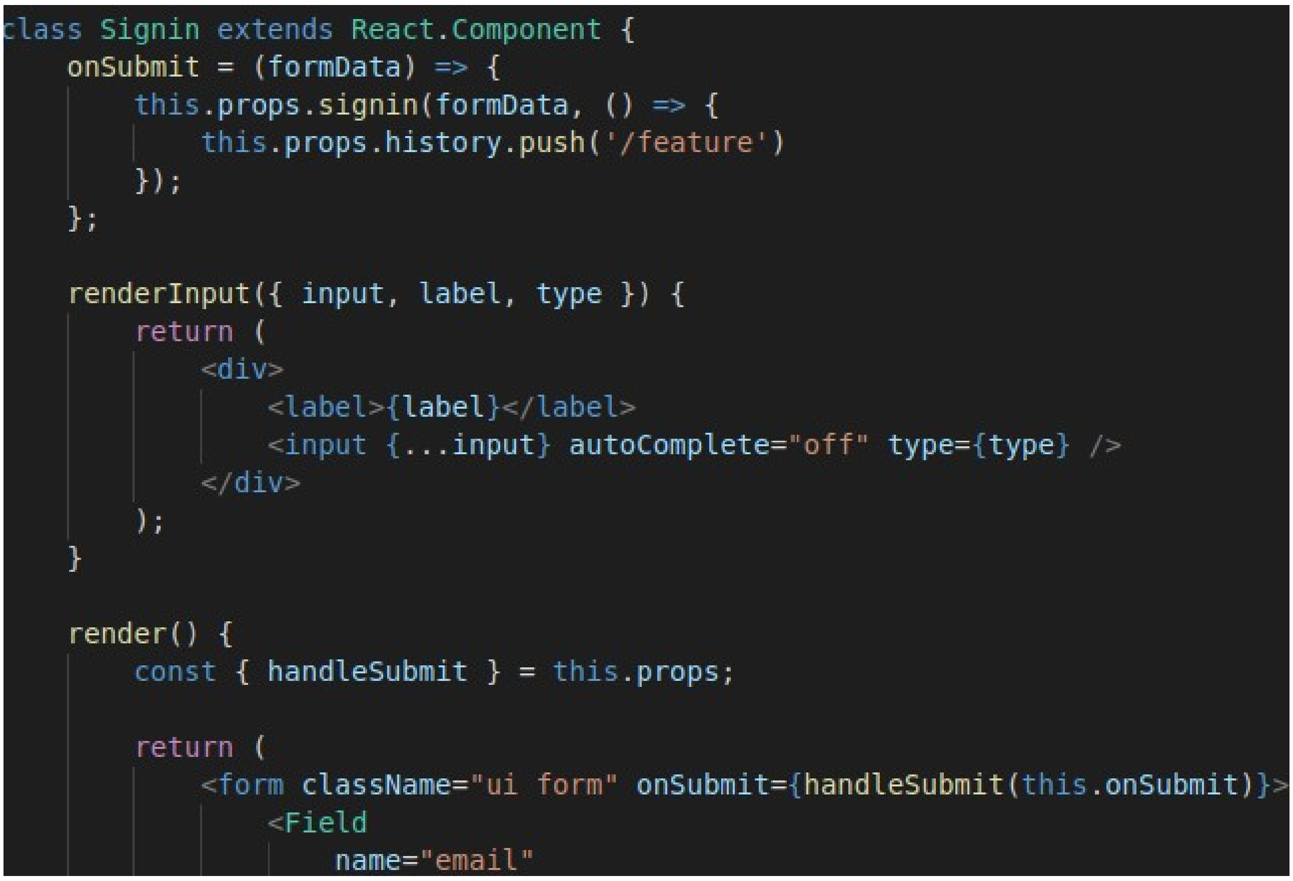
Now we can wire up the Signout component to give it access to the action creators. We do this with a connect statement, and then invoke this action creator in its **componentDidMount** lifycycle method.



```
import { connect } from 'react-redux';
import * as actions from '../../actions';

class Signout extends React.Component{
    componentDidMount(){
        this.props.signout();
    }
    render(){
        return(
            <div>
                Sorry to see you go!
            </div>
        );
    }
}

export default connect(null, actions)(Signout);
```

Sign In Form:

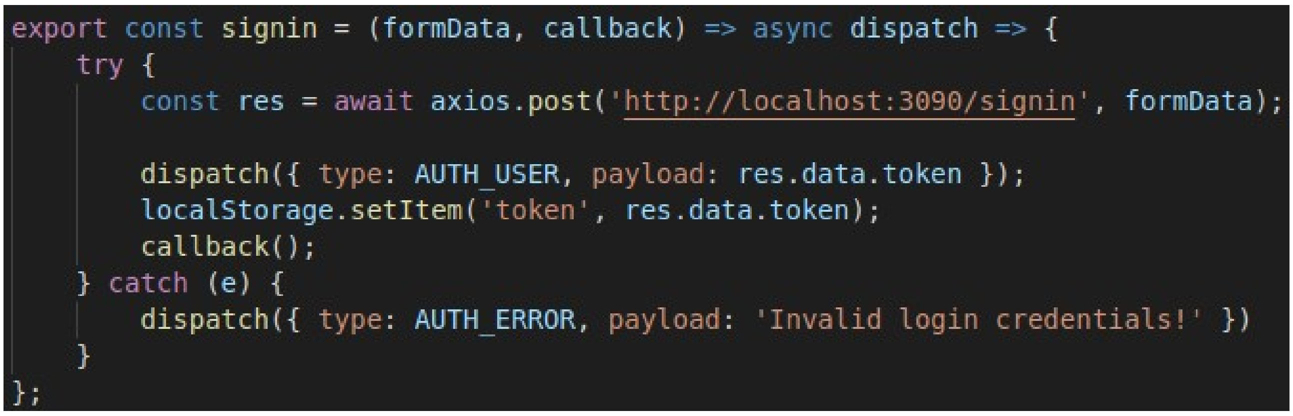
This is going to be largely identical to the Signup form, and we will reuse the code, just changing the names of the component, and action creator (which we will define shortly). We can also make route mapping for this component.



```
lass Signin extends React.Component {
    onSubmit = (formData) => {
        this.props.signin(formData, () => {
            this.props.history.push('/feature')
        });
    };

    renderInput({ input, label, type }) {
        return (
            <div>
                <label>{label}</label>
                <input {...input} autoComplete="off" type={type} />
            </div>
        );
    }

    render() {
        const { handleSubmit } = this.props;

        return (
            <form className="ui form" onSubmit={handleSubmit(this.onSubmit)}>
                <Field
                    name="email"
```

This is identical to the Signup form, incluiding the connect/ mapStateToProps etc. Note the onSubmit has a new action creator, but redirects to the same route after auth is completed.

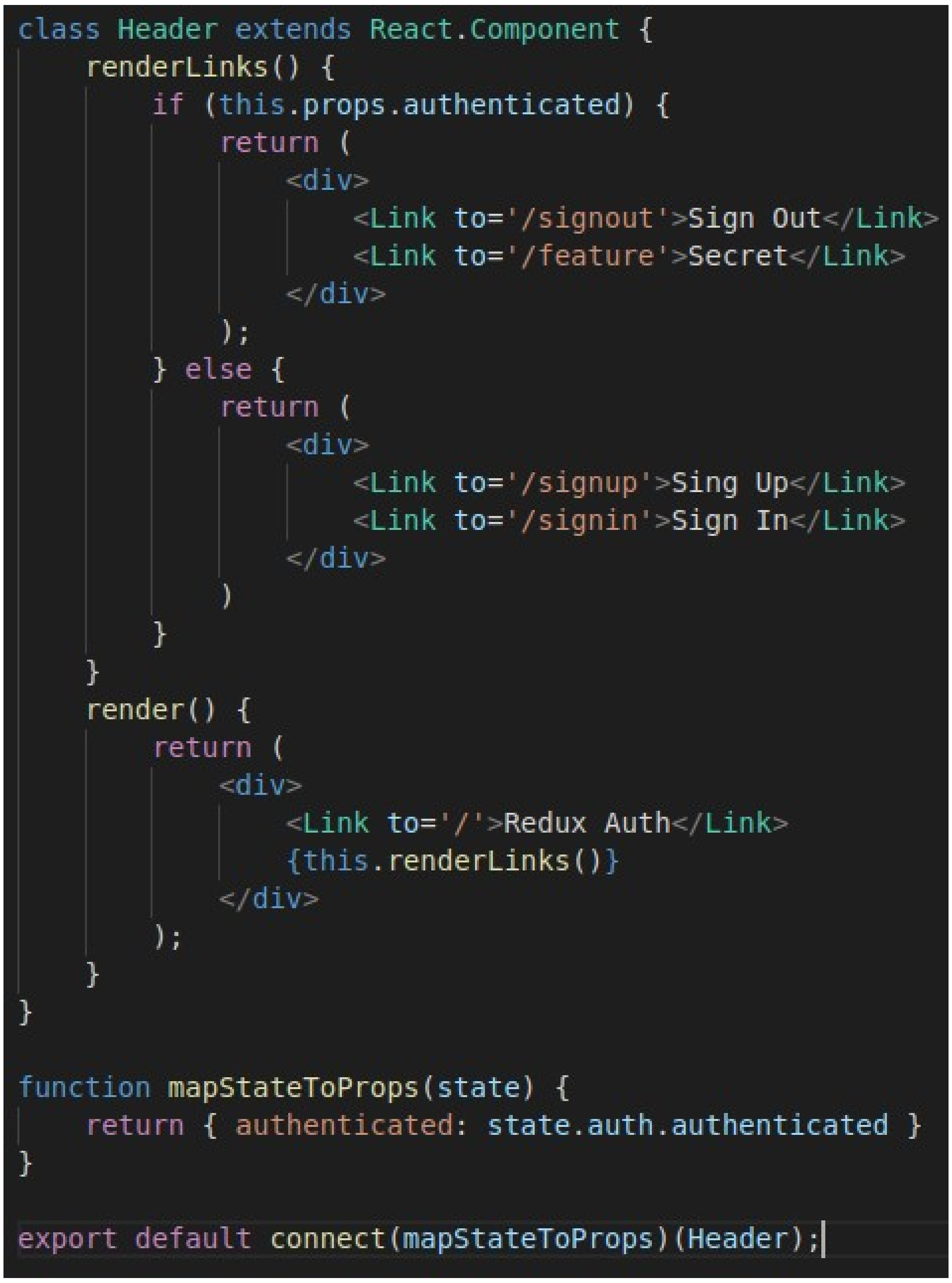
This action cretator is almost identical to the signup action creator, but we change the name and route it makes a request to:



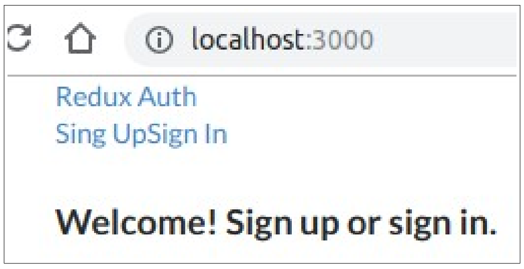
```
export const signin = (formData, callback) => async dispatch => {
    try {
        const res = await axios.post('http://localhost:3090/signin', formData);

        dispatch({ type: AUTH_USER, payload: res.data.token });
        localStorage.setItem('token', res.data.token);
        callback();
    } catch (e) {
        dispatch({ type: AUTH_ERROR, payload: 'Invalid login credentials!' })
    }
};
```

Conditional Rendering in the Header:

We will define a helper method inside the Header.js file, **renderLinks**, which will display the appropriate links depending on the auth status of the user.



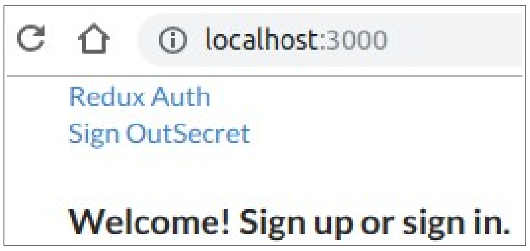
```
class Header extends React.Component {
    renderLinks() {
        if (this.props.authenticated) {
            return (
                <div>
                    <Link to='/signout'>Sign Out</Link>
                    <Link to='/feature'>Secret</Link>
                </div>
            );
        } else {
            return (
                <div>
                    <Link to='/signup'>Sing Up</Link>
                    <Link to='/signin'>Sign In</Link>
                </div>
            )
        }
    }
    render() {
        return (
            <div>
                <Link to='/'>Redux Auth</Link>
                {this.renderLinks()}
            </div>
        );
    }
}

function mapStateToProps(state) {
    return { authenticated: state.auth.authenticated }
}

export default connect(mapStateToProps)(Header);
```



localhost:3000

Redux Auth
Sing UpSign In

Welcome! Sign up or sign in.



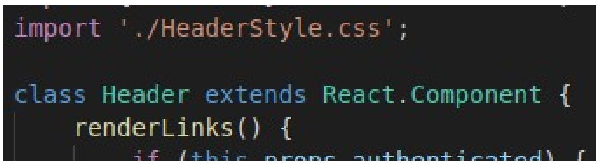
localhost:3000

Redux Auth
Sign OutSecret

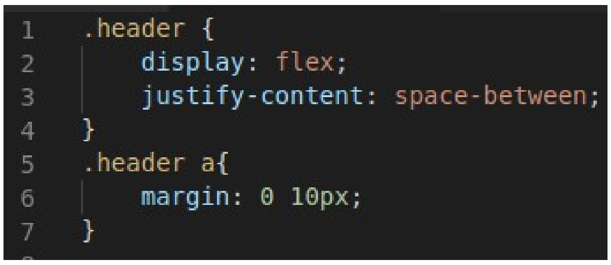
Welcome! Sign up or sign in.

<u>Header Styling:</u>

Here, we will add some CSS to aid with the presentation of the header.

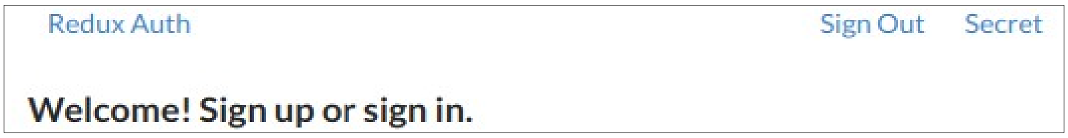


```
import './HeaderStyle.css';

class Header extends React.Component {
    renderLinks() {
```



```
.header {
    display: flex;
    justify-content: space-between;
}
.header a{
    margin: 0 10px;
}
```

Note when importing a CSS file, there is no variable associated with it, so we simply use **import ‘/HeaderStyle.css’**.



Welcome! Sign up or sign in.