Open in app

Dakota Lillie

264 Followers About Follow

Django & React: JWT Authentication

Dakota Lillie Apr 12, 2018 · 12 min read

Recently I've been building an app using a combination of React for the frontend and Django for the backend. I wasn't previously all too familiar with Django, but this seemed like a good opportunity to teach myself, and having experience with Ruby on Rails has made the process a little easier. However, Django and Rails have their fair share of differences, and one of the things which I've had the most trouble implementing is user authentication. So I'm sharing what I've learned, in the hopes that you too might find it useful!

working with Django 2.0.4, and React 16.3.

The Basic Premise: Sessions vs. Tokens

There are many different potential approaches to implementing authentication. Here I'll just cover two of the most common ones: *session authentication* (via cookies), and *token authentication*. There are plenty of articles around the internet differentiating the two, but I'll give a quick summary of what they are and how they work.

Session authentication is stateful, which means when a user logs in, data pertaining to their authenticated status gets stored either in memory or a database. This data is collectively referred to as a *session*, and to facilitate its access, a cookie with the user's session ID is sent back to the client and stored in the browser. Then, next time the client makes a request, that cookie is included in the request and the server searches for a session that matches the cookie's session ID. If there's a match, then the backend proceeds to process the request. When the user logs out, another request has to be made to the server so that the relevant session data is destroyed, along with the cookie in the browser's storage.

Session authentication was for a long time considered the preferred approach, and it remains widely used. However, it suffers from several notable drawbacks. Most relevant to our particular circumstance is the fact that cookies are tied to a particular domain, which leads to significant CORS headaches when the front and back ends are decoupled.

That leads us to token authentication. Tokens are key-value pairs which usually live in the local storage of your browser. In this regard they are similar to cookies — however, where session authentication was stateful, token authorization is stateless, meaning there's no record kept on the server of which users are logged in, how many tokens have been issued etc. Instead, tokens are generated by means of a complex encryption process which, when reversed and decrypted, authenticates the user.

This is a very broad generalization of the methodology employed by JSON Web Tokens (JWTs for short). JWTs are regarded as the gold standard in authentication right now,

Setting Up Django

First off, we need to set up our virtual environment (if that's unfamiliar to you, I wrote a whole blog post about it!). I'm going to use pipenv here—make sure you have it installed, then navigate to the directory you want your project to be in and run:

```
pipenv install
```

Once that's done, activate the virtual environment with:

```
pipenv shell
```

Now we're going to need to install some packages, including Django, Django REST framework (hereafter referred to as the DRF), Django REST framework JWT, and Django CORS headers. The DRF is what we'll be layering on top of Django to turn our project into an API, while Django REST framework JWT gives us the ability to use JWT tokens for our app, and Django CORS headers is necessary to avoid CORS issues:

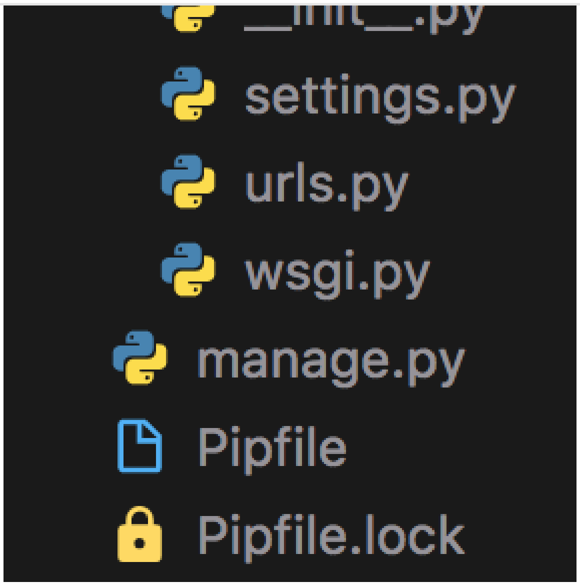
```
pipenv install django
pipenv install djangorestframework
pipenv install djangorestframework-jwt
pipenv install django-cors-headers
```

Once this is done, we're ready to create the Django project. Run the following:

```
django-admin startproject mysite .
```

Note the period at the end there — that denotes that we want to create the project with the current directory as as the root, rather than putting it in a new subdirectory. At this point, your project structure should look something like this:

Open in app



Now we need to adjust some of our project's settings. Go into `settings.py` and make the following modifications:

```
# ...

INSTALLED_APPS = [
    # ...
    'rest_framework',
    'corsheaders',
]

MIDDLEWARE = [
    # ...
    'corsheaders.middleware.CorsMiddleware', # Note that this needs to be placed above CommonMi
    'django.middleware.common.CommonMiddleware', # This should already exist
    # ...
]

#...

REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_jwt.authentication.JSONWebTokenAuthentication',
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.BasicAuthentication',
    ),
    'localhost:3000',
)
```

Let's examine what we've done here: first, we've registered the DRF and CORS headers packages with our project by adding them to `INSTALLED_APPS` . Then, we added a piece of custom CORS middleware, making sure to place it place it above any middleware that generates responses such as Django's `CommonMiddleware` . We then customized some of the default settings for the DRF: first, we set the `DEFAULT_PERMISSION_CLASSES` , which in this case will require a request to be authenticated before it is processed unless specified otherwise. Then, we set the `DEFAULT_AUTHENTICATION_CLASSES` , which determines which authentication methods the server will try when it receives a request, in descending order. Finally, we added `localhost:3000` to the `CORS_ORIGIN_WHITELIST` , since that's where the requests from our React app will be coming from.

The DRF JWT package provides us with a default view for decoding received JWTs. We can add that to `urls.py` :

```
#...

from rest_framework_jwt.views import obtain_jwt_token

urlpatterns = [
    #...
    path('token-auth/', obtain_jwt_token)
]
```

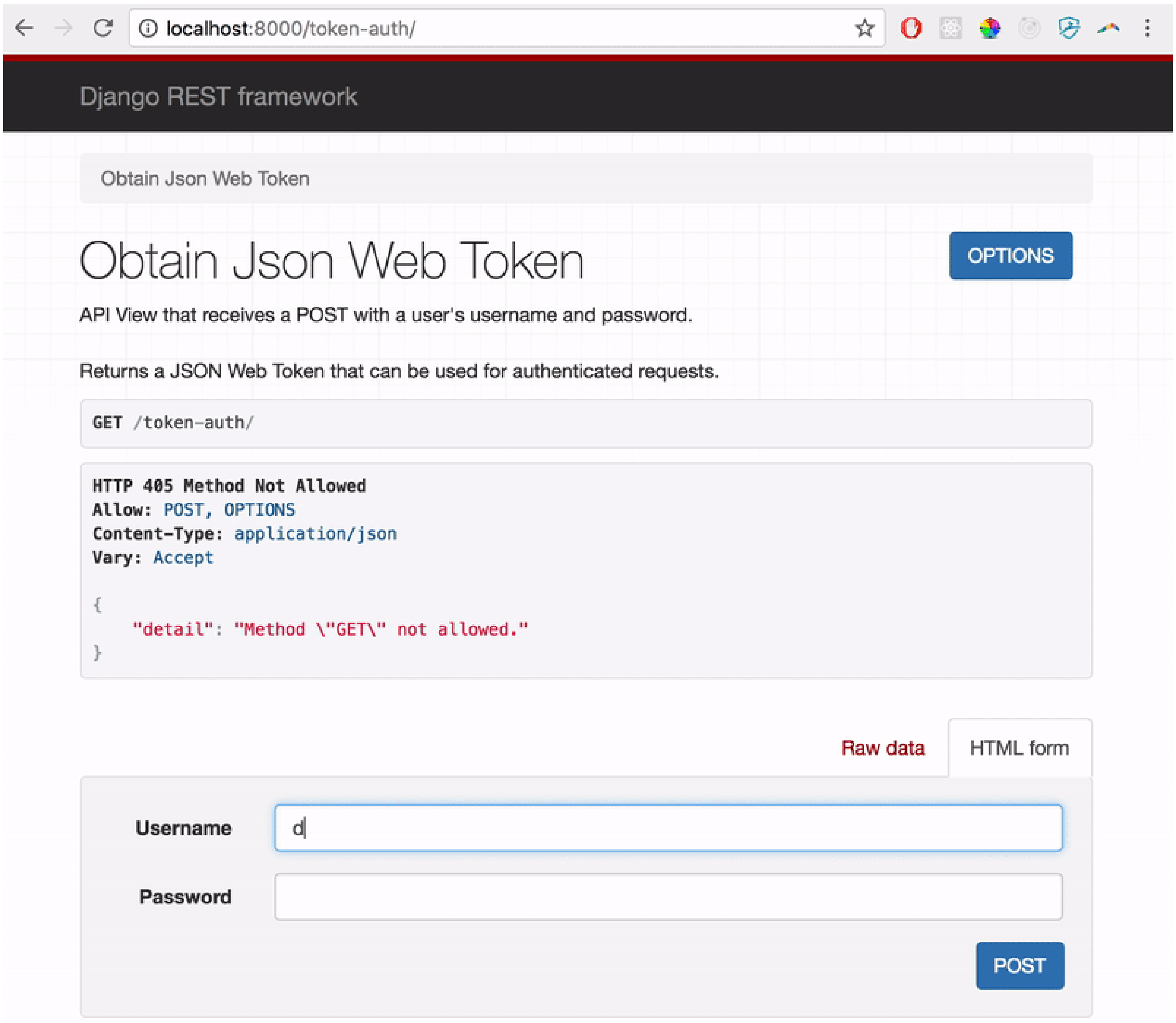
And we're already almost ready to test whether or not this works! But before we can, we need to create a user, and before do that, we need to apply our migrations. Run the following:

```
python manage.py migrate
python manage.py createsuperuser
```

Fill in all the fields and you should be good to go. Now if you start your development server:

```
python manage.py runserver
```

and navigate to http://localhost:8000/token-auth/, you should see an html form there with username and password fields (this is a convenience provided by the DRF… isn't it awesome?). Fill in these fields and you should see the JWT itself displayed right there on the page.



API View that receives a POST with a user's username and password.

Returns a JSON Web Token that can be used for authenticated requests.

```
GET /token-auth/
```

```
HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

Username	d
Password	

This takes care of logging in but we still need our users to be able to sign up. Fortunately, Django comes with a built in `User` model that we can use (which is easy enough to customize, should you need to do so). All we need to do is create the view for it. But if we're making a view, we're going to need an app to put it in. So let's do that now:

```
python manage.py startapp core
```

I'm calling the app "core", but you can of course call it whatever you want. Before we do anything else, let's register the app in `settings.py`:

```
# ...

INSTALLED_APPS = [
    # ...
    'core.apps.CoreConfig'
]

# ...
```

settings.py hosted with ♡ by GitHub view raw

Now we need to take a few steps that might seem circuitous at first but which I promise will make sense by the end. First, we need to create a couple serializers for our `User` model. These serializers will be responsible for serializing/unserializing the `User` model into and out of various formats, primarily JSON in our case. Go ahead and create a new `core/serializers.py` file, and fill it with the following:

```
from rest_framework import serializers
from rest_framework_jwt.settings import api_settings
from django.contrib.auth.models import User


class UserSerializer(serializers.ModelSerializer):

    class Meta:
        model = User
```

```python
class UserSerializerWithToken(serializers.ModelSerializer):

    token = serializers.SerializerMethodField()
    password = serializers.CharField(write_only=True)

    def get_token(self, obj):
        jwt_payload_handler = api_settings.JWT_PAYLOAD_HANDLER
        jwt_encode_handler = api_settings.JWT_ENCODE_HANDLER

        payload = jwt_payload_handler(obj)
        token = jwt_encode_handler(payload)
        return token

    def create(self, validated_data):
        password = validated_data.pop('password', None)
        instance = self.Meta.model(**validated_data)
        if password is not None:
            instance.set_password(password)
        instance.save()
        return instance

    class Meta:
        model = User
        fields = ('token', 'username', 'password')
```

serializers.py hosted with ♡ by GitHub view raw

The reason we're making two different serializers for the model is because we'll be using the `UserSerializerWithToken` for handling signups. When a user signs up, we want the response from the server to include both their relevant user data (in this case, just the username), as well as the token, which will be stored in the browser for further authentication. But we don't need the token every time we request a user's data — just when signing up. Thus, separate serializers.

That's the 'why', but let's take a closer look at the code for the 'how'. Both serializers inherit from `rest_framework.serializers.ModelSerializer` , which provides us with a handy shortcut for customizing the serializers according to the model data they'll be working with (otherwise we'd need to spell out every field by hand). In the internal `Meta` class, we indicate which model each serializer will be representing, and which fields from that model we want the serializer to include.

then add a `get_token()` method which handles the manual creation of a new token. It does this using the default settings for payload and encoding handling provided by the JWT package (the payload is the data being tokenized, in this case the user). Finally, we added the custom 'token' field to the `fields` variable in our `Meta` internal class.

We also need to make sure the serializer recognizes and stores the submitted password, but doesn't include it in the returned JSON. So we add the 'password' field to fields, but above that also specify that the password should be write only. Then, we override the serializer's `create()` method, which determines how the object being serialized gets saved to the database. We do this primarily so that we can call the `set_password()` method on the user instance, which is how the password gets properly hashed.

Now we're ready to start configuring our views in `core/views.py`. There are many ways of going about doing this (function-based views, class-based views, or viewsets). Since viewsets can be confusing if you don't understand what's happening internally (and since we don't have enough views here to reap their benefits anyway), I'll use the other forms of views here:

```python
from django.http import HttpResponseRedirect
from django.contrib.auth.models import User
from rest_framework import permissions, status
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework.views import APIView
from .serializers import UserSerializer, UserSerializerWithToken


@api_view(['GET'])
def current_user(request):
    """
    Determine the current user by their token, and return their data
    """

    serializer = UserSerializer(request.user)
    return Response(serializer.data)


class UserList(APIView):
    """
    Create a new user. It's called 'UserList' because normally we'd have a get
    permission_classes = (permissions.AllowAny,)

    def post(self, request, format=None):
        serializer = UserSerializerWithToken(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

views.py hosted with ♡ by GitHub view raw

Let’s walk through this. First, we have the `current_user` function-based view. This view will be used anytime the user revisits the site, reloads the page, or does anything else that causes React to forget its state. React will check if the user has a token stored in the browser, and if a token is found, it’ll make a request to this view. Since we’ve set things up properly, the token will be parsed automatically to check for authentication, and if validated we’ll receive the user object associated with that token in the request’s `user` property. We can then serialize the user object, and return the data from the serializer in the response. This whole function then serves as the input for the `@api_view` decorator, which specifies the request methods this view will respond to (in this case, just GET requests).

Next, we have our class-based `UserList` view. When a request is routed to this view, a `UserSerializerWithToken` serializer object is instantiated with the data the user entered into the signup form. The serializer checks whether or not the data is valid, and if it is, it’ll save the new user and return that user’s data in the response (including the token, since we’re using this particular serializer). Note that we specify the permissions for this class to be `permissions.AllowAny`, because otherwise, the user would have to be logged in before they could sign up, which could be frustrating.

We have our views now, but as of yet still no way of accessing them. To do that, we need to assign them some routes. It’s customary to give each app its own route configuration, so create a new `core/urls.py` file and edit it like so:

```python
from django.urls import path
from .views import current_user, UserList

urlpatterns = [
```

urls.py hosted with ♡ by GitHub view raw

Now we'll hook up this file to the root urls conf by editing that file (`mysite/urls.py`):

```
#...
from django.urls import path, include

urlpatterns = [
    #...
    path('core/', include('core.urls'))
]
```

urls.py hosted with ♡ by GitHub view raw

With that, we're almost good to go — but we still have a problem. Currently, when a user logs in, they receive their token but not any of their user data. To remedy this, we could make a separate request to the `current_user()` view we defined earlier… but that's annoying. Why make multiple requests? Instead, let's customize our JWT settings a bit.

What we're going to need to do is define a custom JWT response payload handler which includes the user's serialized data. Within the `mysite` directory, make a new file called `utils.py` and fill it with the following:

```
from core.serializers import UserSerializer


def my_jwt_response_handler(token, user=None, request=None):
    return {
        'token': token,
        'user': UserSerializer(user, context={'request': request}).data
    }
```

utils.py hosted with ♡ by GitHub view raw

All this is doing is adding a new 'user' field with the user's serialized data when a token is generated. This is going to be our new default JWT response handler, which we can set up by adding a little bit to our `settings.py` file:

Open in app

```
    'JWT_RESPONSE_PAYLOAD_HANDLER': 'mysite.utils.my_jwt_response_handler'
}
```

settings.py hosted with ♡ by GitHub view raw

Now, when a user logs in, they'll get all their user data along with their token… And we should finally be done! On to the frontend.

Setting Up React

We'll be instantiating our React app using create-react-app, so go ahead and get that if you don't have it already. Then run:

```
create-react-app myapp
```

Making sure to call it whatever you please, of course. Change into the directory you just created. I'm mostly going to try to avoid using any additional packages unrelated to the matter at hand, but the one I will use is the `prop-types` package to make sure everything is well documented. So let's go ahead and install that:

```
npm i prop-types -S
```

Create a `components` directory within `src`. We're going to fill this with the basic, essential components we'll need to get our simple auth app up and running, starting with our navbar (really just an unordered list here). Create a new `Nav.js` file within `components` and edit it like so:

```
import React from 'react';
import PropTypes from 'prop-types';

function Nav(props) {
  const logged_out_nav = (
    <ul>
  );

  const logged_in_nav = (
    <ul>
      <li onClick={props.handle_logout}>logout</li>
    </ul>
  );
  return <div>{props.logged_in ? logged_in_nav : logged_out_nav}</div>;
}

export default Nav;

Nav.propTypes = {
  logged_in: PropTypes.bool.isRequired,
  display_form: PropTypes.func.isRequired,
  handle_logout: PropTypes.func.isRequired
};
```

Nav.js hosted with ♡ by GitHub view raw

Basically, there will be two versions of the navbar, one for when the user is logged in and one for when they're not. When logged out, clicking the links in the navbar will call a function which displays the relevant form (for logging in or signing up). All of this is dependent on props which will be passed in by the parent element.

Next let's define some forms for logging in and signing up. Create two new files within `components`, `LoginForm.js` and `SignupForm.js`:

```
import React from 'react';
import PropTypes from 'prop-types';

class LoginForm extends React.Component {
  state = {
    username: '',
    password: ''
  };

  handle_change = e => {
    const name = e.target.name;
    const value = e.target.value;
    this.setState(prevstate => {
      const newState = { ...prevstate };
      newState[name] = value;
  };

  render() {
    return (
      <form onSubmit={e => this.props.handle_login(e, this.state)}>
        <h4>Log In</h4>
        <label htmlFor="username">Username</label>
        <input
          type="text"
          name="username"
          value={this.state.username}
          onChange={this.handle_change}
        />
        <label htmlFor="password">Password</label>
        <input
          type="password"
          name="password"
          value={this.state.password}
          onChange={this.handle_change}
        />
        <input type="submit" />
      </form>
    );
  }
}

export default LoginForm;

LoginForm.propTypes = {
  handle_login: PropTypes.func.isRequired
};
```

```
import React from 'react';
import PropTypes from 'prop-types';

class SignupForm extends React.Component {
  state = {
    username: '',
    password: ''
  };

  handle_change = e => {
    const name = e.target.name;
    const value = e.target.value;
```

```
      newState[name] = value;
      return newState;
    });
  };

  render() {
    return (
      <form onSubmit={e => this.props.handle_signup(e, this.state)}>
        <h4>Sign Up</h4>
        <label htmlFor="username">Username</label>
        <input
          type="text"
          name="username"
          value={this.state.username}
          onChange={this.handle_change}
        />
        <label htmlFor="password">Password</label>
        <input
          type="password"
          name="password"
          value={this.state.password}
          onChange={this.handle_change}
        />
        <input type="submit" />
      </form>
    );
  }
}

export default SignupForm;

SignupForm.propTypes = {
  handle_signup: PropTypes.func.isRequired
};
```

SignupForm is hosted with ♡ by GitHub view raw

These files are virtually identical — the only reason I kept them separate is because normally you'd want to collect more info on a user when they sign up than you'd need when they log in (in fact, you can customize the signup form if you like… Django's `User` model supports `email`, `first_name`, and `last_name` fields in addition to `username` and `password`. Be sure to add the relevant fields to the serializer, too, if you want the data back). Both components are stateful to keep track of the data in their controlled how the form should be processed upon submission.

But where are these props coming from? Well, since this is a very simple example, I just put all the logic in the `App` component. Open up your `App.js` file and make it look like this:

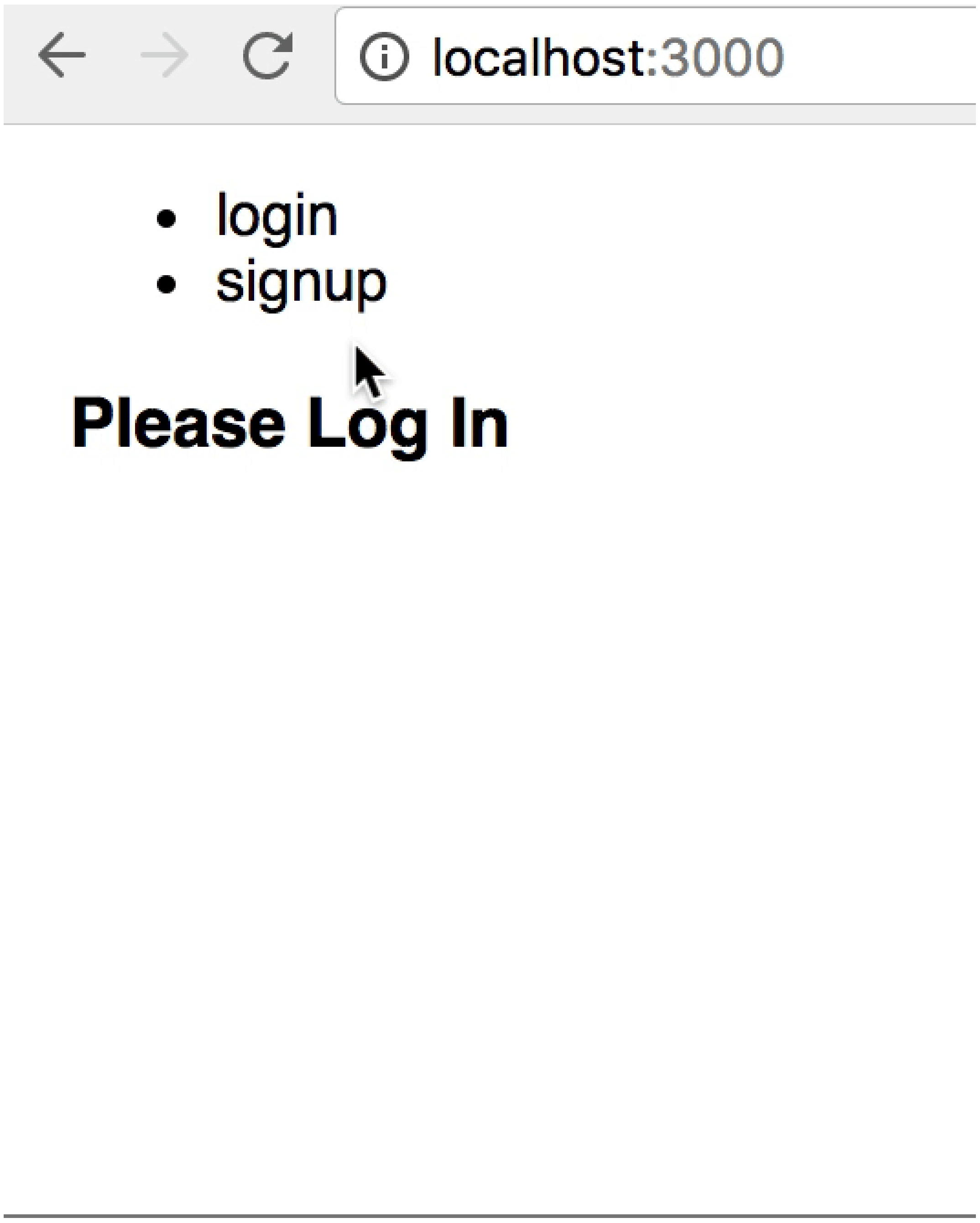
```
import React, { Component } from 'react';
import Nav from './components/Nav';
import LoginForm from './components/LoginForm';
import SignupForm from './components/SignupForm';
import './App.css';

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      displayed_form: '',
      logged_in: localStorage.getItem('token') ? true : false,
      username: ''
    };
  }

  componentDidMount() {
    if (this.state.logged_in) {
      fetch('http://localhost:8000/core/current_user/', {
        headers: {
          Authorization: `JWT ${localStorage.getItem('token')}`
        }
      })
        .then(res => res.json())
        .then(json => {
          this.setState({ username: json.username });
        });
    }
  }

  handle_login = (e, data) => {
    e.preventDefault();
    fetch('http://localhost:8000/token-auth/', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      .then(json => {
        localStorage.setItem('token', json.token);
        this.setState({
          logged_in: true,
          displayed_form: '',
          username: json.user.username
        });
      });
  };

  handle_signup = (e, data) => {
    e.preventDefault();
    fetch('http://localhost:8000/core/users/', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(data)
    })
      .then(res => res.json())
      .then(json => {
        localStorage.setItem('token', json.token);
        this.setState({
          logged_in: true,
          displayed_form: '',
          username: json.username
        });
      });
  };

  handle_logout = () => {
    localStorage.removeItem('token');
    this.setState({ logged_in: false, username: '' });
  };

  display_form = form => {
    this.setState({
      displayed_form: form
    });
  };

  render() {
    let form;
    switch (this.state.displayed_form) {
      case 'login':
        form = <SignupForm handle_signup={this.handle_signup} />;
        break;
      default:
        form = null;
    }

    return (
      <div className="App">
        <Nav
          logged_in={this.state.logged_in}
          display_form={this.display_form}
          handle_logout={this.handle_logout}
        />
        {form}
        <h3>
          {this.state.logged_in
            ? `Hello, ${this.state.username}`
            : 'Please Log In'}
        </h3>
      </div>
    );
  }
}

export default App;
```

This where the magic happens, so let's parse it one bit at a time. We start with the `constructor()`, where the component's state is initialized and the `logged_in` property is determined based on whether or not a token can be found in local storage. We then move on to the `componentDidMount()` lifecycle method, where if a token has been found, we make a request to the `current_user()` view we defined in Django. Notice that we specify the `Authorization` header in the format 'JWT <token>'. Each request to the API which requires the user to be authenticated will need to include this header, in this format, in order for the request to be processed. Then, we parse the response as JSON, and add the user's username to the component's state.

In the `handle_login()` method, we make a POST request to the `obtain_jwt_token` view that we tried out before… only this time, because we changed the default response payload handler, the response from this viewpoint will include the user's serialized

In both of these cases, the token is stored into local storage once the response has been parsed into JSON. Finally, we have the `handle_logout()` method, which simply deletes the token from local storage (no request necessary).

And that's about it! Everything else, in the `display_form()` and `render()` methods, just handles the UI. Run `npm start` (and maybe reboot your Django server if you haven't already), and you should see a simple but functional app with authentication capabilities.



Please Log In

```css
.App {
  margin-left: 20px;
}

input {
  display: block;
  margin: 5px 0 5px;
}
```

App.css hosted with ♡ by GitHub view raw

Conclusion

This is obviously a very simple example —Normally, I'd incorporate a lot more error handling and form validations, as well as use Redux thunks to make the requests to the API. But since none of that is necessary to understand how to incorporate JWT authentication, so I opted to omit all of that here. One additional point: JWT encoding/decoding involves the use of a secret key, which defaults to the `SECRET_KEY` constant defined in `settings.py` . If you're deploying an app that uses JWT to production, be sure to change this, or at least hide the secret key from Github.

I certainly learned a lot in writing this, and I hope you did too!

Sources

Home - Django REST framework
Django, API, REST, Home
www.django-rest-framework.org

Django REST framework JWT
JSON Web Token Authentication support for Django REST Framework
getblimp.github.io

User Registration/Authentication with Django, Django Rest Framework, React, and Redux

This is the website of Iheanyi Ekechukwu, a New York City-based Software Engineer. /> <link href=

iheanyi.com

Django Django Rest Framework Jwt Authentication