

Implement User Auth in a Django & React app with Knox

Add Token-based Authentication with Django-rest-knox to an app built with Django and React/Redux



Koji Mochizuki [Follow](#)
Jul 27, 2019 · 8 min read ★



Photo by Jon Moore on Unsplash

In the last tutorial, we created a CRUD Todo application with React, Redux, and Django (REST framework). This time, we will implement user authentication, such as Login, for the application. Again, we use Semantic UI for UI design.

Password

Confirm Password

[Register](#)

Already have an account? [Login](#)

Register

[TodoCRUD](#) [Home](#) [Sign Up](#) [Login](#)

Username

Password

[Login](#)

Don't have an account? [Register](#)

Login

[TodoCRUD](#) [Home](#) [John ▾](#)

Task

[Add](#)

- | | | |
|--------------------------|--------------------------------------|------------------------|
| <input type="checkbox"/> | John's Task 1
07/15/2019 09:05:25 | Delete |
| <input type="checkbox"/> | John's Task 2
07/15/2019 09:05:39 | Delete |

[Logout](#)

Table of Contents

- Preparation
- Adding Todo's owner
- Creating a new app for user auth
- Testing authentication using REST Client

- Implementing User Loading and Login
- Creating a Login Form
- Creating PrivateRoute
- Implementing Logout
- Adding items to the navbar
- Implementing Register
- Creating a Register Form
- Updating a URLconf

• • •

Preparation

The first thing we have to do is prepare the base application. If you want to build the base app from scratch, start with the tutorial below:

Build a CRUD Todo app with Django and React/Redux

In this tutorial, we will learn how to build a CRUD Todo application with Django REST framework for the backend, React...

[medium.com](https://medium.com/@mohamedabdullah/build-a-crud-todo-app-with-django-and-react-redux-10f3a2a2a2)

You can also download the base app from my repository on GitHub.

If you cloned the repository on your computer, initialize your local repository:

```
$ rm -rf .git  
$ git init
```

Install the packages:

```
$ pipenv install  
$ npm install
```

Run migrations and start the dev server:

```
$ pipenv shell  
$ python manage.py migrate  
$ python manage.py runserver
```

Open another terminal and run the script:

```
$ npm run dev
```

Visit <http://127.0.0.1:8000/> with your browser. If the Todo creation form is displayed, the base app is ready.

Adding Todo's owner

We will add the `owner` field to the `Todo` model and relate the field to the `User` model that Django provides:

```
1 # todos/models.py  
2  
3 from django.db import models  
4 from django.contrib.auth.models import User # added  
5  
6  
7 class Todo(models.Model):  
8     task = models.CharField(max_length=255)  
9     owner = models.ForeignKey(  
10         User, related_name="todos", on_delete=models.CASCADE, null=True) #added  
11     created_at = models.DateTimeField(auto_now_add=True)  
12  
13     def __str__(self):  
14         return self.task
```

models.py hosted with ❤ by GitHub

[view raw](#)

Apply the changes to our databases by running the following commands:

```
$ python manage.py makemigrations  
$ python manage.py migrate
```

Next, we will set permissions on `TodoViewSet`. And define a method to get only the todos created by each owner. Open the `todos/api/views.py` file:

```
1 # todos/api/views.py
2
3 from rest_framework import viewsets, permissions # added permissions
4
5 from .serializers import TodoSerializer
6 # from todos.models import Todo # remove
7
8
9 class TodoViewSet(viewsets.ModelViewSet):
10     # queryset = Todo.objects.all() # remove
11     serializer_class = TodoSerializer
12     permission_classes = [permissions.IsAuthenticated] # added
13
14     def get_queryset(self): # added
15         return self.request.user.todos.all()
16
17     def perform_create(self, serializer): # added
18         serializer.save(owner=self.request.user)
```

[views.py](#) hosted with ❤ by GitHub

[view raw](#)

Our serializer will now have the `'owner'` field by overriding the `perform_create()` method as above.

Creating a new app for user auth

We will add a new app for user authentication by running the command below:

```
$ python manage.py startapp accounts
```

We will install a new package to use Token Authentication:

```
$ pipenv install django-rest-knox
```

Open the `settings.py` file and enable them:

```
1 # todocrud/settings.py
2
3 INSTALLED_APPS = [
4     'accounts.apps.AccountsConfig', # added
5     'frontend.apps.FrontendConfig',
```

```

6     'todos.apps.TodosConfig',
7     'rest_framework',
8     'knox', # added
9     # ...
10    ]
11
12 REST_FRAMEWORK = {
13     # 'DEFAULT_PERMISSION_CLASSES': [ # remove
14     #     'rest_framework.permissions.AllowAny'
15     # ],
16     'DEFAULT_AUTHENTICATION_CLASSES': ( # added
17         'knox.auth.TokenAuthentication',
18     ),
19     'DATETIME_FORMAT': "%m/%d/%Y %H:%M:%S",
20 }

```

settings.py hosted with ❤ by GitHub

[view raw](#)

We can set permissions on a per-view basis, so remove the default permission policy set globally. We will set the `DEFAULT_AUTHENTICATION_CLASSES` setting instead.

Now that **Knox** has been added, let's run the migrations:

```
$ python manage.py migrate
```

Creating API

First, we will create a new folder named `api`, and create four files in it:

```

accounts/
  api/
    __init__.py
    serializers.py
    urls.py
    views.py

```

Let's create each file.

Serializers

```

1  # accounts/api/serializers.py
2
3  from django.contrib.auth import authenticate
4  from django.contrib.auth.models import User
5
6  from rest_framework import serializers

```

```

7
8     User._meta.get_field('email')._unique = True
9
10
11    class UserSerializer(serializers.ModelSerializer):
12        class Meta:
13            model = User
14            fields = ('id', 'username', 'email')
15
16
17    class RegisterSerializer(serializers.ModelSerializer):
18        class Meta:
19            model = User
20            fields = ('id', 'username', 'email', 'password')
21            extra_kwargs = {'password': {'write_only': True}}
22
23        def create(self, validated_data):
24            user = User.objects.create_user(
25                validated_data['username'],
26                validated_data['email'],
27                validated_data['password']
28            )
29            return user
30
31
32    class LoginSerializer(serializers.Serializer):
33        username = serializers.CharField()
34        password = serializers.CharField()
35
36        def validate(self, data):
37            user = authenticate(**data)
38            if user and user.is_active:
39                return user
40            raise serializers.ValidationError("Incorrect Credentials")

```

serializers.py hosted with ❤ by GitHub

[view raw](#)

I would like to set `User._meta.get_field('email')._unique = True` so that duplicate email addresses can not be registered.

APIViews

```

1  # accounts/api/views.py
2
3  from rest_framework import generics, permissions
4  from rest_framework.response import Response
5
6  from knox.models import AuthToken
7
8  from serializers import UserSerializer, RegisterSerializer, LoginSerializer

```

```

8     from .serializers import UserSerializer, RegisterSerializer, LoginSerializer
9
10
11 class UserAPIView(generics.RetrieveAPIView):
12     permission_classes = [
13         permissions.IsAuthenticated,
14     ]
15     serializer_class = UserSerializer
16
17     def get_object(self):
18         return self.request.user
19
20
21 class RegisterAPIView(generics.GenericAPIView):
22     serializer_class = RegisterSerializer
23
24     def post(self, request, *args, **kwargs):
25         serializer = self.get_serializer(data=request.data)
26         serializer.is_valid(raise_exception=True)
27         user = serializer.save()
28         return Response({
29             "user": UserSerializer(user, context=self.get_serializer_context()).data,
30             "token": AuthToken.objects.create(user)[1]
31         })
32
33
34 class LoginAPIView(generics.GenericAPIView):
35     serializer_class = LoginSerializer
36
37     def post(self, request, *args, **kwargs):
38         serializer = self.get_serializer(data=request.data)
39         serializer.is_valid(raise_exception=True)
40         user = serializer.validated_data
41         return Response({
42             "user": UserSerializer(user, context=self.get_serializer_context()).data,
43             "token": AuthToken.objects.create(user)[1]
44         })

```

views.py hosted with ❤ by GitHub

[view raw](#)

"token": AuthToken.objects.create(user) [1] means that the AuthToken.objects.create() returns a tuple (**instance, token**). So, add [1] and specify the second position.

URLs

```

1 # accounts/api/urls.py
2
3 from django.urls import path, include
4

```

```
5  from knox.views import LogoutView
6
7  from .views import UserAPIView, RegisterAPIView, LoginAPIView
8
9  urlpatterns = [
10     path('', include('knox.urls')),
11     path('user', UserAPIView.as_view()),
12     path('register', RegisterAPIView.as_view()),
13     path('login', LoginAPIView.as_view()),
14     path('logout', LogoutView.as_view(), name='knox_logout')
15 ]
```

urls.py hosted with ❤ by GitHub

[view raw](#)

To logout, use the view provided by **Knox**.

Project's URLconf

We will finally include the accounts URLs to the project's URLconf:

```
1  # todocrud/urls.py
2
3  from django.contrib import admin
4  from django.urls import path, include
5
6  urlpatterns = [
7      path('', include('frontend.urls')),
8      path('api/', include('todos.api.urls')),
9      path('api/auth/', include('accounts.api.urls')), # added
10     path('admin/', admin.site.urls),
11 ]
```

urls.py hosted with ❤ by GitHub

[view raw](#)

Now that we have made changes to the User model, run migrations:

```
$ python manage.py makemigrations
$ python manage.py migrate
```

Testing authentication using REST Client

What is REST Client?

REST Client allows you to send HTTP request and view the response in Visual Studio Code directly.

If you don't use **VS Code**, you can use Postman.

In this tutorial, we will use the **REST Client** to perform authentication tests.

First of all, we will create a new folder named `requests` in the project root directory, and create a file named `api.http` in it:

```
requests/  
    api.http
```

Register

Let's create a user first. Write as follows in the `api.http` file. Then, "Send Request" will be displayed above the HTTP method. Click this to view a response:

```
1 # requests/api.http  
2  
3 # Send Request  
4 POST http://127.0.0.1:8000/api/auth/register  
5 Content-Type: application/json  
6  
7 {  
8     "username": "John",  
9     "email": "john@example.com",  
10    "password": "1234"  
11 }
```

api.http hosted with ❤ by GitHub

[view raw](#)

```
{  
  "user": {  
    "id": 1,  
    "username": "John",  
    "email": "john@example.com"  
  },  
  "token":  
  "5abd2f673838bacb5249006736c0450f3abb09ed5142e4f44947a5fb3c542b04"  
}
```

If the response views as above, it worked.

Load User

To write multiple requests in the same file, place three or more consecutive `#` as separators between requests. Write as follows, and send the GET request with Token:

```

1 # requests/api.http
2
3 # Send Request
4 GET http://127.0.0.1:8000/api/auth/user
5 Authorization: Token 5abd2f673838bacb5249006736c0450f3abb09ed5142e4f44947a5fb3c542b04
6
7 #####
8
9 # Send Request
10 POST http://127.0.0.1:8000/api/auth/register
11 Content-Type: application/json
12
13 {
14     "username": "John",
15     "email": "john@example.com",
16     "password": "1234"
17 }
```

api.http hosted with ❤ by GitHub

[view raw](#)

```
{
  "id": 1,
  "username": "John",
  "email": "john@example.com"
}
```

The user data should be returned.

Logout

```

1 # requests/api.http
2
3 # Send Request
4 POST http://127.0.0.1:8000/api/auth/logout
5 Authorization: Token 5abd2f673838bacb5249006736c0450f3abb09ed5142e4f44947a5fb3c542b04
```

api.http hosted with ❤ by GitHub

[view raw](#)

There is no response, of course. If the request passes, the token used for authentication is removed from the system and can no longer be used.

Try sending the request to get the user data again. An error should be returned as follows:

```
{
  "detail": "Invalid token."
```

```
}
```

Login with Incorrect data

Let's test login authentication. First, we will send incorrect data. An error should be returned:

```
1 # requests/api.http
2
3 # Send Request
4 POST http://127.0.0.1:8000/api/auth/login
5 Content-Type: application/json
6
7 {
8     "username": "Mary",
9     "password": "1234"
10 }
```

api.http hosted with ❤ by GitHub

[view raw](#)

```
{
  "non_field_errors": [
    "Incorrect Credentials"
  ]
}
```

Login with Correct data

Next, we will send the correct data:

```
1 # requests/api.http
2
3 # Send Request
4 POST http://127.0.0.1:8000/api/auth/login
5 Content-Type: application/json
6
7 {
8     "username": "John",
9     "password": "1234"
10 }
```

api.http hosted with ❤ by GitHub

[view raw](#)

```
{
  "user": {
    "id": 1,
```

```

    "username": "John",
    "email": "john@example.com"
},
"token":
"055eb2ea1bf931039433dbd2030f6b9075e067c81e0eda57ca383880244a7015"
}

```

The response should be returned as above.

Finally, try sending the request to get the user data with the newly generated token.

Implementing User Loading and Login

Let's add all the action types needed to implement user authentication first. Open the `types.js` file and add them:

```

1 // frontend/src/actions/types.js
2
3 export const USER_LOADING = 'USER_LOADING'; // added
4 export const USER_LOADED = 'USER_LOADED'; // added
5 export const AUTH_ERROR = 'AUTH_ERROR'; // added
6 export const REGISTER_SUCCESS = 'REGISTER_SUCCESS'; // added
7 export const REGISTER_FAIL = 'REGISTER_FAIL'; // added
8 export const LOGIN_SUCCESS = 'LOGIN_SUCCESS'; // added
9 export const LOGIN_FAIL = 'LOGIN_FAIL'; // added
10 export const LOGOUT_SUCCESS = 'LOGOUT_SUCCESS'; // added

```

[types.js hosted with ❤ by GitHub](#)

[view raw](#)

Action Creators

We will create a new file named `auth.js` in the `actions` directory, and define two action creators:

```

1 // frontend/src/actions/auth.js
2
3 import axios from 'axios';
4 import { stopSubmit } from 'redux-form';
5
6 import {
7   USER_LOADING,
8   USER_LOADED,
9   AUTH_ERROR,
10  LOGIN_SUCCESS,
11  LOGIN_FAIL
12 } from './types';
13
14 // LOAD USER
15 export const loadUser = () => async (dispatch, getState) => {

```

```
16 dispatch({ type: USER_LOADING });
17
18 try {
19     const res = await axios.get('/api/auth/user', tokenConfig(getState));
20     dispatch({
21         type: USER_LOADED,
22         payload: res.data
23     });
24 } catch (err) {
25     dispatch({
26         type: AUTH_ERROR
27     });
28 }
29 };
30
31 // LOGIN USER
32 export const login = ({ username, password }) => async dispatch => {
33     // Headers
34     const config = {
35         headers: {
36             'Content-Type': 'application/json'
37         }
38     };
39
40     // Request Body
41     const body = JSON.stringify({ username, password });
42
43     try {
44         const res = await axios.post('/api/auth/login', body, config);
45         dispatch({
46             type: LOGIN_SUCCESS,
47             payload: res.data
48         });
49     } catch (err) {
50         dispatch({
51             type: LOGIN_FAIL
52         });
53         dispatch(stopSubmit('loginForm', err.response.data));
54     }
55 };
56
57 // helper function
58 export const tokenConfig = getState => {
59     // Get token
60     const token = getState().auth.token;
61
62     // Headers
63     const config = {
64         headers: {
65             'Content-Type': 'application/json'
66         }
67     };
68
69     return {
70         headers: {
71             Authorization: `Bearer ${token}`
72         }
73     };
74 }
75
76 // Action Creators
77 export const loadUser = () => dispatch({ type: USER_LOADING });
78
79 export const loginUser = ({ username, password }) => dispatch => {
80     dispatch(tokenConfig());
81
82     return axios.post('/api/auth/login', { username, password })
83         .then(res => {
84             dispatch({
85                 type: LOGIN_SUCCESS,
86                 payload: res.data
87             });
88         })
89         .catch(err => {
90             dispatch({
91                 type: LOGIN_FAIL
92             });
93             dispatch(stopSubmit('loginForm', err.response.data));
94         });
95 }
96
97 // Thunk Creators
98 export const registerUser = ({ name, email, password }) => dispatch => {
99     dispatch(tokenConfig());
100
101     return axios.post('/api/auth/register', { name, email, password })
102         .then(res => {
103             dispatch({
104                 type: REGISTER_SUCCESS,
105                 payload: res.data
106             });
107         })
108         .catch(err => {
109             dispatch({
110                 type: REGISTER_FAIL
111             });
112             dispatch(stopSubmit('registerForm', err.response.data));
113         });
114 }
```

```

67 };
68
69   if (token) {
70     config.headers['Authorization'] = `Token ${token}`;
71   }
72
73   return config;
74 };

```

auth.js hosted with ❤ by GitHub

[view raw](#)

Create a function named `tokenConfig` as a helper function that gets and sets tokens. This function is also used for todo's action creators.

We can use `stopSubmit()` to pass server-side errors to our Redux Form fields. The `loginForm` is going to be created later. Don't worry about it for now.

We also need to pass token to the todo actions. Open the `actions/todos.js` file and update each action creator:

```

1 // frontend/src/actions/todos.js
2
3 import { tokenConfig } from './auth'; // added
4
5 // GET TODOS
6 export const getTodos = () => async (dispatch, getState) => {
7   const res = await axios.get('/api/todos/', tokenConfig(getState));
8   // ...
9 };
10
11 // GET TODO
12 export const getTodo = id => async (dispatch, getState) => {
13   const res = await axios.get(`/api/todos/${id}/`, tokenConfig(getState));
14   // ...
15 };
16
17 // ADD TODO
18 export const addTodo = formValues => async (dispatch, getState) => {
19   const res = await axios.post(
20     '/api/todos/',
21     { ...formValues },
22     tokenConfig(getState)
23   );
24   // ...
25 };
26
27 // DELETE TODO
28 export const deleteTodo = id => async (dispatch, getState) => {
29   await axios.delete(`/api/todos/${id}/`, tokenConfig(getState));

```

```

30 // ...
31 };
32
33 // EDIT TODO
34 export const editTodo = (id, formValues) => async (dispatch, getState) => {
35   const res = await axios.patch(
36     `/api/todos/${id}`,
37     formValues,
38     tokenConfig(getState)
39   );
40   // ...
41 }

```

[todos.js](#) hosted with ❤ by GitHub

[view raw](#)

Reducers

Next, we will create a new file named `auth.js` in the `reducers` directory, and write an auth reducer:

```

1 // frontend/src/reducers/auth.js
2
3 import {
4   USER_LOADING,
5   USER_LOADED,
6   AUTH_ERROR,
7   LOGIN_SUCCESS,
8   LOGIN_FAIL
9 } from '../actions/types';
10
11 const initialState = {
12   isLoading: false,
13   isAuthenticated: null,
14   user: null,
15   token: localStorage.getItem('token')
16 };
17
18 export default function(state = initialState, action) {
19   switch (action.type) {
20     case USER_LOADING:
21       return {
22         ...state,
23         isLoading: true
24       };
25     case USER_LOADED:
26       return {
27         ...state,
28         isLoading: false,
29         isAuthenticated: true,
30         user: action.payload
31       };

```

```

31   };
32 
33   case LOGIN_SUCCESS:
34     localStorage.setItem('token', action.payload.token);
35     return {
36       ...state,
37       isLoading: false,
38       isAuthenticated: true,
39       ...action.payload
40     };
41 
42   case AUTH_ERROR:
43   case LOGIN_FAIL:
44     localStorage.removeItem('token');
45     return {
46       ...state,
47       isLoading: false,
48       isAuthenticated: false,
49       user: null,
50       token: null
51     };
52 
53   default:
54     return state;
55   }
56 }

```

auth.js hosted with ❤ by GitHub

[view raw](#)

Tokens are stored in a web browser using the `localStorage` property.

Learn more about `localStorage`.

Then, we will add the auth reducer to the parent reducer. Open the `reducers/index.js` file and update it as follows:

```

1 // frontend/src/reducers/index.js
2
3 import auth from './auth'; // added
4
5 export default combineReducers({
6   form: formReducer,
7   todos,
8   auth // added
9 });

```

index.js hosted with ❤ by GitHub

[view raw](#)

Creating a Login Form

Let's create a new component for login using **Redux Form**. Create a new folder named `auth` in the `components` directory, and add a new file named `LoginForm.js` into it:

```
1 // frontend/src/components/auth/LoginForm.js
2
3 import React, { Component } from 'react';
4 import { Link, Redirect } from 'react-router-dom';
5 import { connect } from 'react-redux';
6 import { Field, reduxForm } from 'redux-form';
7 import { login } from '../../actions/auth';
8
9 class LoginForm extends Component {
10   renderField = ({ input, label, type, meta: { touched, error } }) => {
11     return (
12       <div className={`field ${touched && error ? 'error' : ''}`}>
13         <label>{label}</label>
14         <input {...input} type={type} />
15         {touched && error && (
16           <span className='ui pointing red basic label'{error}</span>
17         )}
18       </div>
19     );
20   };
21
22   hiddenField = ({ type, meta: { error } }) => {
23     return (
24       <div className='field'>
25         <input type={type} />
26         {error && <div className='ui red message'{error}</div>}
27       </div>
28     );
29   };
30
31   onSubmit = formValues => {
32     this.props.login(formValues);
33   };
34
35   render() {
36     if (this.props.isAuthenticated) {
37       return <Redirect to='/' />;
38     }
39     return (
40       <div className='ui container'>
41         <div className='ui segment'>
42           <form
43             onSubmit={this.props.handleSubmit(this.onSubmit)}
44             className='ui form'
45           >
46             <Field
47               name='username'
48               type='text'
49               component={this.renderField}
50               label='Username'
51             />
```

```

52     <Field
53         name='password'
54         type='password'
55         component={this.renderField}
56         label='Password'
57     />
58     <Field
59         name='non_field_errors'
60         type='hidden'
61         component={this.hiddenField}
62     />
63     <button className='ui primary button'>Login</button>
64   </form>
65   <p style={{ marginTop: '1rem' }}>
66     Don't have an account? <Link to='/register'>Register</Link>
67   </p>
68   </div>
69 </div>
70 );
71 }
72 }
73
74 const mapStateToProps = state => ({
75   isAuthenticated: state.auth.isAuthenticated
76 });
77
78 LoginForm = connect(
79   mapStateToProps,
80   { login }
81 )(LoginForm);
82
83 export default reduxForm({
84   form: 'loginForm'
85 })(LoginForm);

```

LoginForm.js hosted with ❤ by GitHub

[view raw](#)

If the `username` and `password` do not match the information in the database, Django returns **Non-field errors**. To render this error, we need to have a field named

`'non_field_errors'`.

`'loginForm'` is the name of this form. You can name each form as you like.

Creating PrivateRoute

It is a good idea to redirect users who are not logged in to the login page when they visit the home page. Create a new folder named `common` in the `components` directory, and add a new file named `PrivateRoute.js` into it. And then write the code as follows:

```

1 // frontend/src/components/common/PrivateRoute.js
2
3 import React from 'react';
4 import { Route, Redirect } from 'react-router-dom';
5 import { connect } from 'react-redux';
6
7 const PrivateRoute = ({ component: Component, auth, ...rest }) => (
8   <Route
9     {...rest}
10    render={props => {
11      if (auth.isLoading) {
12        return <div>Loading...</div>;
13      } else if (!auth.isAuthenticated) {
14        return <Redirect to='/login' />;
15      } else {
16        return <Component {...props} />;
17      }
18    }}
19  />
20);
21
22 const mapStateToProps = state => ({
23   auth: state.auth
24 });
25
26 export default connect(mapStateToProps)(PrivateRoute);

```

PrivateRoute.js hosted with ❤ by GitHub

[view raw](#)

For information about the `render` prop function, see the following URL:
<https://reacttraining.com/react-router/web/api/Route/render-func>

Let's open the `App.js` file and make some changes:

```

1 // frontend/src/components/App.js
2
3 import LoginForm from './auth/LoginForm'; // added
4 import PrivateRoute from './common/PrivateRoute'; // added
5
6 import { loadUser } from '../actions/auth'; // added
7
8 class App extends Component {
9   // added
10  componentDidMount() {
11    store.dispatch(loadUser());
12  }
13
14  render() {
15    return (
16      <Provider store={store}>

```

```
17 <Router history={history}>
18   <Header />
19   <Switch>
20     <PrivateRoute exact path='/' component={Dashboard} /> // updated
21     <Route exact path='/delete/:id' component={TodoDelete} />
22     <Route exact path='/edit/:id' component={TodoEdit} />
23     <Route exact path='/login' component={LoginForm} /> // added
24   </Switch>
25 </Router>
26 </Provider>
27 );
28 }
29 }
```

App.js hosted with ❤ by GitHub

[view raw](#)

Now we should be able to log in. Visit <http://127.0.0.1:8000/> with your browser:



Login Form

You should be redirected to the login page. Did it work?



Non-field errors are also displayed correctly.

Finally, try logging in using the account we created during the authentication test. And then, try adding a couple of tasks.

Implementing Logout

Action Creators

Open the `actions/auth.js` file and add a new action creator for logout:

```

1 // frontend/src/actions/auth.js
2
3 // added LOGOUT_SUCCESS
4 import {
5   USER_LOADING,
6   USER_LOADED,
7   AUTH_ERROR,
8   LOGIN_SUCCESS,
9   LOGIN_FAIL,
10  LOGOUT_SUCCESS
11 } from './types';
12
13 // LOGOUT USER
14 export const logout = () => async (dispatch, getState) => {
15   await axios.post('/api/auth/logout', null, tokenConfig(getState));
16   dispatch({
17     type: LOGOUT_SUCCESS
18   });
19 };

```

auth.js hosted with ❤ by GitHub

[view raw](#)

Reducers

Open the `reducers/auth.js` file and add the action to the reducer:

```

1 // frontend/src/reducers/auth.js
2
3 // added LOGOUT_SUCCESS
4 import {
5   USER_LOADING,
6   USER_LOADED,
7   AUTH_ERROR,
8   LOGOUT_SUCCESS
9 } from './types';
10
11 const authReducer = (state = {}, action) => {
12   switch (action.type) {
13     case USER_LOADING:
14       return { ...state, loading: true };
15     case USER_LOADED:
16       return { ...state, loading: false, user: action.payload };
17     case AUTH_ERROR:
18       return { ...state, error: action.payload };
19     case LOGOUT_SUCCESS:
20       return { ...state, token: null, user: null };
21     default:
22       return state;
23   }
24 };

```

```

7   AUTH_ERROR,
8   LOGIN_SUCCESS,
9   LOGIN_FAIL,
10  LOGOUT_SUCCESS
11 } from '../actions/types';

12

13 export default function(state = initialState, action) {
14   switch (action.type) {
15     // ...
16     case AUTH_ERROR:
17     case LOGIN_FAIL:
18     case LOGOUT_SUCCESS: // added
19       localStorage.removeItem('token');
20       return {
21         ...state,
22         isLoading: false,
23         isAuthenticated: false,
24         user: null,
25         token: null
26       };
27     // ...
28   }
29 }

```

auth.js hosted with ❤ by GitHub

[view raw](#)

Now we can log out, but there is one problem. For example, if you log in and out with **Account 1** and then log in with **Account 2**, the todo objects for **Account 1** will also appear in the todo list for **Account 2**. This is because the todo reducer is not initialized.

To solve this problem, customize the root reducer as follows:

```

1 // frontend/src/reducers/index.js
2
3 import { LOGOUT_SUCCESS } from '../actions/types'; // added
4
5 // export default combineReducers({
6 //   form: formReducer,
7 //   todos,
8 //   auth
9 // });
10
11 const appReducer = combineReducers({
12   form: formReducer,
13   todos,
14   auth
15 });
16
17 const rootReducer = (state, action) => {
18   if (action.type === LOGOUT_SUCCESS) {

```

```
19     state = undefined;
20 }
21 return appReducer(state, action);
22 ;
23
24 export default rootReducer;
```

index.js hosted with ❤ by GitHub

[view raw](#)

Now all reducers will be initialized whenever each user logs out.

Adding items to the navbar

Let's add a link to the navigation bar so we can log out. Also, change the display depending on whether you are logged in or not. Open the `Headers.js` file and update it as follows:

```
1 // frontend/src/components/layout/Headers.js
2
3 import React, { Component } from 'react';
4 import { Link } from 'react-router-dom'; // added
5 import { connect } from 'react-redux'; // added
6 import { logout } from '../../actions/auth'; // added
7
8 class Header extends Component {
9   render() {
10     const { user, isAuthenticated } = this.props.auth; // added
11
12     // added
13     const userLinks = (
14       <div className='right menu'>
15         <div className='ui simple dropdown item'>
16           {user ? user.username : ''}
17           <i className='dropdown icon' />
18           <div className='menu'>
19             <a onClick={this.props.logout} className='item'>
20               Logout
21             </a>
22           </div>
23         </div>
24       </div>
25     );
26
27     // added
28     const guestLinks = (
29       <div className='right menu'>
30         <Link to='/register' className='item'>
31           Sign Up
32         </Link>
33         <Link to='/login' className='item'>
```

```
55     <Link to="/login" className="item">
56       Login
57     </Link>
58   </div>
59 );
60
61 // updated
62 return (
63   <div className="ui inverted menu" style={{ borderRadius: '0' }}>
64     <Link to="/" className="header item">
65       TodoCRUD
66     </Link>
67     <Link to="/" className="item">
68       Home
69     </Link>
70     {isAuthenticated ? userLinks : guestLinks}
71   </div>
72 );
73 }
74
75 // added
76 const mapStateToProps = state => ({
77   auth: state.auth
78 });
79
80 // updated
81 export default connect(
82   mapStateToProps,
83   { logout }
84 )(Header);
```

Header.js hosted with ❤ by GitHub

[view raw](#)

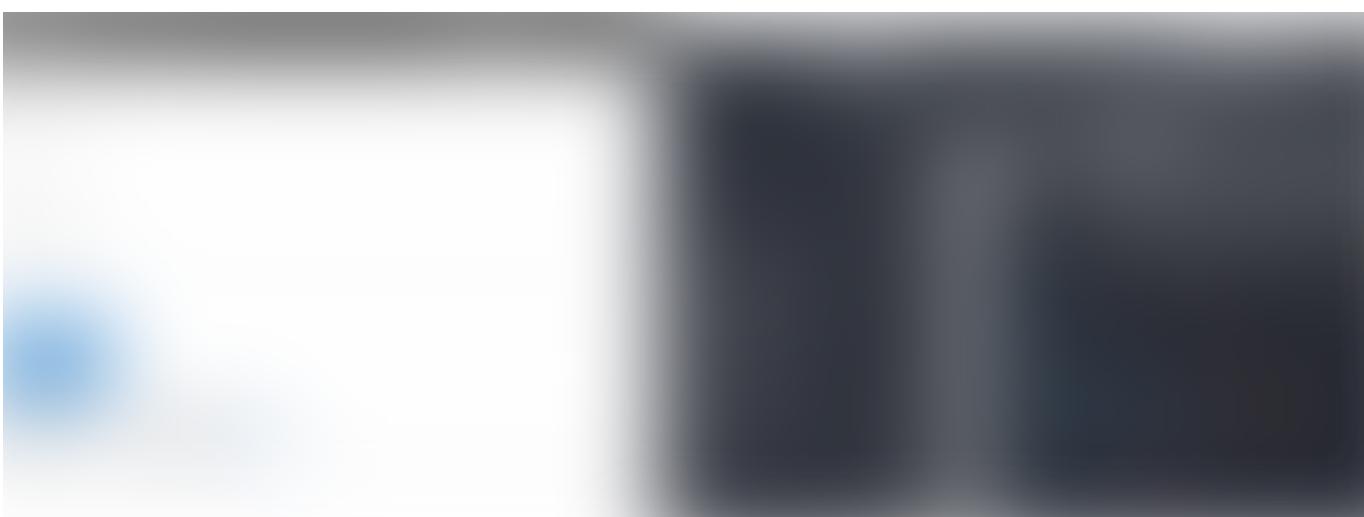
We have also placed `login` and `register` links.

Now, let's try logging out:





Logout



After logging out

Implementing Register

Action Creators

Open the `actions/auth.js` file and add a new action creator for register:

```
1 // frontend/src/actions/auth.js
2
3 import {
4   USER_LOADING,
5   USER_LOADED,
6   AUTH_ERROR,
7   REGISTER_SUCCESS, // added
8   REGISTER_FAIL, // added
9   LOGIN_SUCCESS,
10  LOGIN_FAIL,
11  LOGOUT_SUCCESS
12 } from './types';
13
14 // REGISTER USER
15 export const register = ({ username, email, password }) => async dispatch => {
16   // Headers
17   const config = {
18     headers: {
19       'Content-Type': 'application/json'
20     }
21   };
22 }
```

```

22 // Request Body
23 const body = JSON.stringify({ username, email, password });
24
25
26 try {
27   const res = await axios.post('/api/auth/register', body, config);
28   dispatch({
29     type: REGISTER_SUCCESS,
30     payload: res.data
31   });
32 } catch (err) {
33   dispatch({
34     type: REGISTER_FAIL
35   });
36   dispatch(stopSubmit('registerForm', err.response.data));
37 }
38 };

```

auth.js hosted with ❤ by GitHub

[view raw](#)

We use `stopSubmit()` again here to prevent double registration of users.

Reducers

Open the `reducers/auth.js` file and add the actions to the reducer:

```

1 // frontend/src/reducers/auth.js
2
3 import {
4   USER_LOADING,
5   USER_LOADED,
6   AUTH_ERROR,
7   REGISTER_SUCCESS, // added
8   REGISTER_FAIL, // added
9   LOGIN_SUCCESS,
10  LOGIN_FAIL,
11  LOGOUT_SUCCESS
12 } from '../actions/types';
13
14 export default function(state = initialState, action) {
15   switch (action.type) {
16     // ...
17     case REGISTER_SUCCESS: // added
18     case LOGIN_SUCCESS:
19       localStorage.setItem('token', action.payload.token);
20       return {
21         ...state,
22         isLoading: false,
23         isAuthenticated: true,
24         ...action.payload
25       };
26   }
27   return state;
28 }

```

```

25     };
26 
27     case AUTH_ERROR:
28     case REGISTER_FAIL: // added
29     case LOGIN_FAIL:
30     case LOGOUT_SUCCESS:
31         localStorage.removeItem('token');
32         return {
33             ...state,
34             isLoading: false,
35             isAuthenticated: false,
36             user: null,
37             token: null
38         };
39     // ...
40 }

```

auth.js hosted with ❤ by GitHub

[view raw](#)

It was very easy because we only set the action types.

Creating a Register Form

Let's create a registration form using **Redux Form**. Add a new file named `RegisterForm.js` into the `components/auth` directory:

```

1 // frontend/src/components/auth/RegisterForm.js
2
3 import React, { Component } from 'react';
4 import { Link, Redirect } from 'react-router-dom';
5 import { connect } from 'react-redux';
6 import { Field, reduxForm } from 'redux-form';
7 import { register } from '../../actions/auth';
8
9 class RegisterForm extends Component {
10     renderField = ({ input, label, type, meta: { touched, error } }) => {
11         return (
12             <div className={`field ${touched && error ? 'error' : ''}`}>
13                 <label>{label}</label>
14                 <input {...input} type={type} />
15                 {touched && error && (
16                     <span className='ui pointing red basic label'>{error}</span>
17                 )}
18             </div>
19         );
20     };
21
22     onSubmit = formValues => {
23         this.props.register(formValues);
24     };

```

```
25 render() {
26     if (this.props.isAuthenticated) {
27         return <Redirect to='/' />;
28     }
29     return (
30         <div className='ui container'>
31             <div className='ui segment'>
32                 <form
33                     onSubmit={this.props.handleSubmit(this.onSubmit)}
34                     className='ui form'>
35                     >
36                         <Field
37                             name='username'
38                             type='text'
39                             component={this.renderField}
40                             label='Username'
41                             validate={[required, minLength3, maxLength15]}>
42                         />
43                         <Field
44                             name='email'
45                             type='email'
46                             component={this.renderField}
47                             label='Email'
48                             validate={required}>
49                         />
50                         <Field
51                             name='password'
52                             type='password'
53                             component={this.renderField}
54                             label='Password'
55                             validate={required}>
56                         />
57                         <Field
58                             name='password2'
59                             type='password'
60                             component={this.renderField}
61                             label='Confirm Password'
62                             validate={[required, passwordsMatch]}>
63                         />
64                         <button className='ui primary button'>Register</button>
65                     </form>
66             <p style={{ marginTop: '1rem' }}>
67                 Already have an account? <Link to='/login'>Login</Link>
68             </p>
69         </div>
70     </div>
71 );
72 );
73 }
74 }
```

```

76  const required = value => (value ? undefined : 'Required');
77
78  const minLength = min => value =>
79      value && value.length < min
80          ? `Must be at least ${min} characters`
81          : undefined;
82
83  const minLength3 = minLength(3);
84
85  const maxLength = max => value =>
86      value && value.length > max ? `Must be ${max} characters or less` : undefined;
87
88  const maxLength15 = maxLength(15);
89
90  const passwordsMatch = (value, allValues) =>
91      value !== allValues.password ? 'Passwords do not match' : undefined;
92
93  const mapStateToProps = state => ({
94      isAuthenticated: state.auth.isAuthenticated
95  });
96
97  RegisterForm = connect(
98      mapStateToProps,
99      { register }
100 )(RegisterForm);
101
102 export default reduxForm({
103     form: 'registerForm'
104 })(RegisterForm);

```

RegisterForm.js hosted with ❤ by GitHub

[view raw](#)

The registration form is almost the same as the login form, but we use **Field-Level Validation** in this form.

Learn more about Field-Level Validation.

Password matching can be defined as above using the `allvalues` parameter.

Then we will include the `RegisterForm` in the `App` component:

```

1 // frontend/src/components/App.js
2
3 import RegisterForm from './auth/RegisterForm'; // added
4
5 class App extends Component {
6     // ...
7
8     render() {
9         return (

```

```
10     <Provider store={store}>
11         <Router history={history}>
12             <Header />
13             <Switch>
14                 <PrivateRoute exact path='/' component={Dashboard} />
15                 <Route exact path='/delete/:id' component={TodoDelete} />
16                 <Route exact path='/edit/:id' component={TodoEdit} />
17                 <Route exact path='/register' component={RegisterForm} /> // added
18                 <Route exact path='/login' component={LoginForm} />
19             </Switch>
20         </Router>
21     </Provider>
22 );
23 }
24 }
```

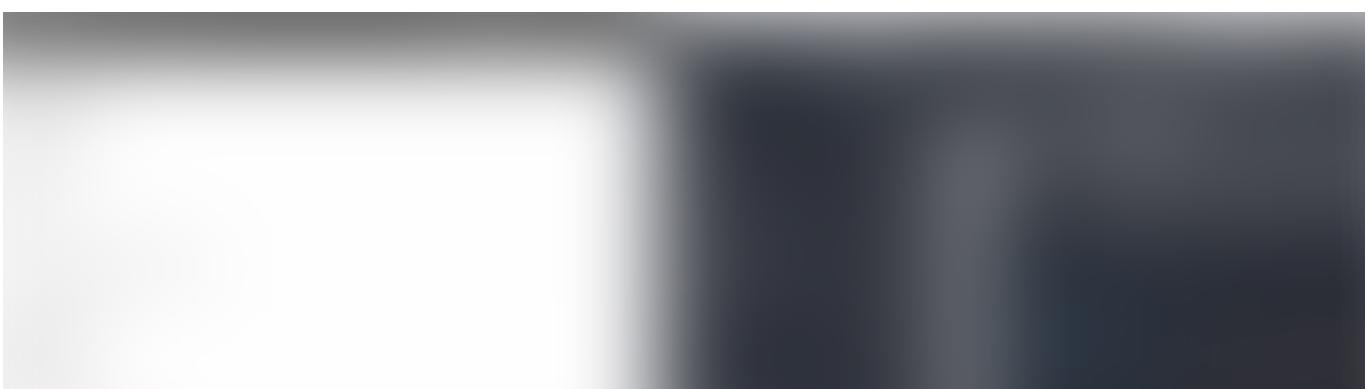
App.js hosted with ❤ by GitHub

[view raw](#)

Now let's test the registration form:



Server-side errors





Passwords do not match

Validations are working properly.

Let's register a new user and add a couple of tasks:



New User

Perfect!

Updating a URLconf

Finally, we will add `login` and `register` URLs to the `frontend/urls.py` file:

```
1 # frontend/urls.py
2
3 urlpatterns = [
4     path('', index),
5     path('login', index), # added
6     path('register', index), # added
```

```
7     path('edit/<int:pk>', TodoDetailView.as_view()),
8     path('delete/<int:pk>', TodoDetailView.as_view()),
9 ]
```

urls.py hosted with ❤ by GitHub

[view raw](#)

Now, even if the browser is updated on the login or registration page, it will be displayed correctly.

This tutorial ends here. The source code is also available on GitHub. Thank you for reading!

[Programming](#) [Coding](#) [Django](#) [React](#) [Redux](#)

[About](#) [Help](#) [Legal](#)