Implement User Auth in a Django & React app with Knox

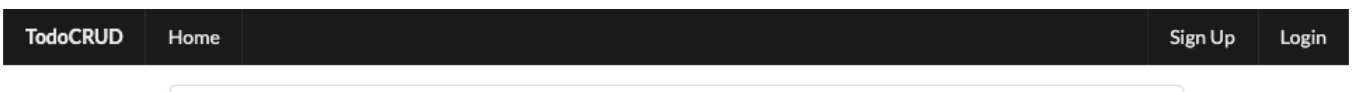
Add Token-based Authentication with Django-rest-knox to an app built with Django and React/Redux

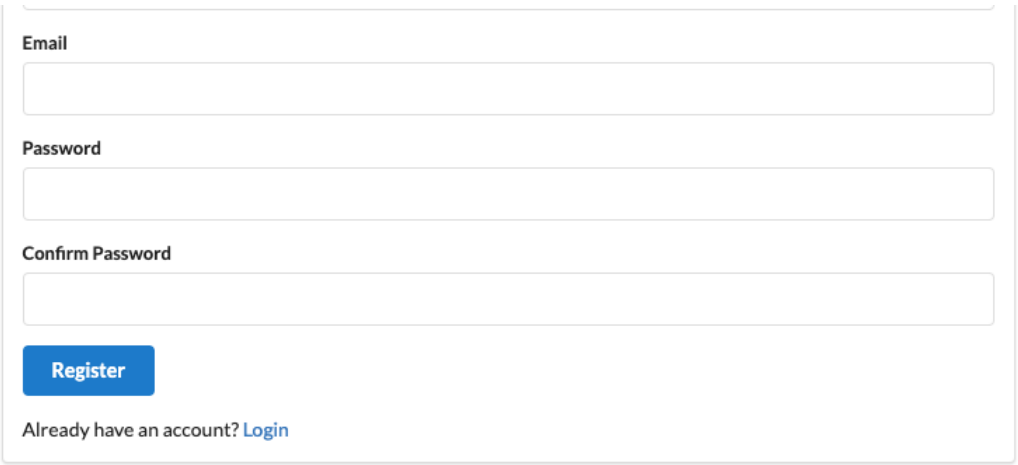
Koji Mochizuki

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.





Register

Login

Logout

Table of Contents

- 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

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:

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
```

```python
# from todos.models import Todo  # remove


class TodoViewSet(viewsets.ModelViewSet):
    # queryset = Todo.objects.all()  # remove
    serializer_class = TodoSerializer
    permission_classes = [permissions.IsAuthenticated]  # added

    def get_queryset(self):  # added
        return self.request.user.todos.all()

    def perform_create(self, serializer):  # added
        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:

```python
# todocrud/settings.py

INSTALLED_APPS = [
    'accounts.apps.AccountsConfig',  # added
    'frontend.apps.FrontendConfig',
    'todos.apps.TodosConfig',
    'rest_framework',
    'knox',  # added
    # ...
```

```
REST_FRAMEWORK = {
    # 'DEFAULT_PERMISSION_CLASSES': [  # remove
    #     'rest_framework.permissions.AllowAny'
    # ],
    'DEFAULT_AUTHENTICATION_CLASSES': (  # added
        'knox.auth.TokenAuthentication',
    ),
    'DATETIME_FORMAT': "%m/%d/%Y %H:%M:%S",
}
```

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

```
# accounts/api/serializers.py

from django.contrib.auth import authenticate
from django.contrib.auth.models import User

from rest_framework import serializers
```

```python

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ('id', 'username', 'email')


class RegisterSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ('id', 'username', 'email', 'password')
        extra_kwargs = {'password': {'write_only': True}}

    def create(self, validated_data):
        user = User.objects.create_user(
            validated_data['username'],
            validated_data['email'],
            validated_data['password']
        )
        return user


class LoginSerializer(serializers.Serializer):
    username = serializers.CharField()
    password = serializers.CharField()

    def validate(self, data):
        user = authenticate(**data)
        if user and user.is_active:
            return user
        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

```python
# accounts/api/views.py

from rest_framework import generics, permissions
from rest_framework.response import Response

```

```python
from .serializers import UserSerializer, RegisterSerializer, LoginSerializer


class UserAPIView(generics.RetrieveAPIView):
    permission_classes = [
        permissions.IsAuthenticated,
    ]
    serializer_class = UserSerializer

    def get_object(self):
        return self.request.user


class RegisterAPIView(generics.GenericAPIView):
    serializer_class = RegisterSerializer

    def post(self, request, *args, **kwargs):
        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        user = serializer.save()
        return Response({
            "user": UserSerializer(user, context=self.get_serializer_context()).data,
            "token": AuthToken.objects.create(user)[1]
        })


class LoginAPIView(generics.GenericAPIView):
    serializer_class = LoginSerializer

    def post(self, request, *args, **kwargs):
        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        user = serializer.validated_data
        return Response({
            "user": UserSerializer(user, context=self.get_serializer_context()).data,
            "token": AuthToken.objects.create(user)[1]
        })
```

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

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

from knox.views import LogoutView

from .views import UserAPIView, RegisterAPIView, LoginAPIView

urlpatterns = [
    path('', include('knox.urls')),
    path('user', UserAPIView.as_view()),
    path('register', RegisterAPIView.as_view()),
    path('login', LoginAPIView.as_view()),
    path('logout', LogoutView.as_view(), name='knox_logout')
]
```

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:

```python
# todocrud/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('', include('frontend.urls')),
    path('api/', include('todos.api.urls')),
    path('api/auth/', include('accounts.api.urls')),  # added
    path('admin/', admin.site.urls),
]
```

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
```

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:

```
# requests/api.http

# Send Request
POST http://127.0.0.1:8000/api/auth/register
Content-Type: application/json

{
    "username": "John",
    "email": "john@example.com",
    "password": "1234"
}
```

api.http hosted with ❤️ by GitHub view raw

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

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:

```
# requests/api.http

# Send Request
GET http://127.0.0.1:8000/api/auth/user
Authorization: Token 5abd2f673838bacb5249006736c0450f3abb09ed5142e4f44947a5fb3c542b04

###

# Send Request
POST http://127.0.0.1:8000/api/auth/register
Content-Type: application/json

{
    "username": "John",
    "email": "john@example.com",
    "password": "1234"
}
```

api.http hosted with ❤️ by GitHub view raw

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

The user data should be returned.

Logout

```
# requests/api.http

# Send Request
POST http://127.0.0.1:8000/api/auth/logout
```

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:

```
# requests/api.http

# Send Request
POST http://127.0.0.1:8000/api/auth/login
Content-Type: application/json

{
    "username": "Mary",
    "password": "1234"
}
```

api.http hosted with ❤️ by GitHub view raw

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

Login with Correct data

Next, we will send the correct data:

```
# Send Request
POST http://127.0.0.1:8000/api/auth/login
Content-Type: application/json

{
    "username": "John",
    "password": "1234"
}
```

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:

```
// frontend/src/actions/types.js

export const USER_LOADING = 'USER_LOADING'; // added
export const USER_LOADED = 'USER_LOADED'; // added
export const AUTH_ERROR = 'AUTH_ERROR'; // added
export const REGISTER_SUCCESS = 'REGISTER_SUCCESS'; // added
export const REGISTER_FAIL = 'REGISTER_FAIL'; // added
export const LOGIN_SUCCESS = 'LOGIN_SUCCESS'; // added
export const LOGIN_FAIL = 'LOGIN_FAIL'; // added
export const LOGOUT_SUCCESS = 'LOGOUT_SUCCESS'; // added
```

types.js hosted with ❤️ by GitHub view raw

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

```
// frontend/src/actions/auth.js

import axios from 'axios';
import { stopSubmit } from 'redux-form';

import {
  USER_LOADING,
  USER_LOADED,
  AUTH_ERROR,
  LOGIN_SUCCESS,
  LOGIN_FAIL
} from './types';

// LOAD USER
export const loadUser = () => async (dispatch, getState) => {
  dispatch({ type: USER_LOADING });

  try {
    const res = await axios.get('/api/auth/user', tokenConfig(getState));
    dispatch({
      type: USER_LOADED,
      payload: res.data
    });
  } catch (err) {
    dispatch({
      type: AUTH_ERROR
    });
  }
};

// LOGIN USER
export const login = ({ username, password }) => async dispatch => {
  // Headers
  const config = {
    headers: {
      'Content-Type': 'application/json'
    }
  };

  // Request Body
  const body = JSON.stringify({ username, password });

```

```
      dispatch({
        type: LOGIN_SUCCESS,
        payload: res.data
      });
    } catch (err) {
      dispatch({
        type: LOGIN_FAIL
      });
      dispatch(stopSubmit('loginForm', err.response.data));
    }
};

// helper function
export const tokenConfig = getState => {
  // Get token
  const token = getState().auth.token;

  // Headers
  const config = {
    headers: {
      'Content-Type': 'application/json'
    }
  };

  if (token) {
    config.headers['Authorization'] = `Token ${token}`;
  }

  return config;
};
```

auth.js is 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:

```
// frontend/src/actions/todos.js

import { tokenConfig } from './auth'; // added
```

```
// GET TODOS
export const getTodos = () => async (dispatch, getState) => {
  const res = await axios.get('/api/todos/', tokenConfig(getState));
  // ...
};

// GET TODO
export const getTodo = id => async (dispatch, getState) => {
  const res = await axios.get(`/api/todos/${id}/`, tokenConfig(getState));
  // ...
};

// ADD TODO
export const addTodo = formValues => async (dispatch, getState) => {
  const res = await axios.post(
    '/api/todos/',
    { ...formValues },
    tokenConfig(getState)
  );
  // ...
};

// DELETE TODO
export const deleteTodo = id => async (dispatch, getState) => {
  await axios.delete(`/api/todos/${id}/`, tokenConfig(getState));
  // ...
};

// EDIT TODO
export const editTodo = (id, formValues) => async (dispatch, getState) => {
  const res = await axios.patch(
    `/api/todos/${id}/`,
    formValues,
    tokenConfig(getState)
  );
  // ...
};
```

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:

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:

```
// frontend/src/reducers/index.js

import auth from './auth'; // added

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

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:

```
// frontend/src/components/auth/LoginForm.js

import React, { Component } from 'react';
import { Link, Redirect } from 'react-router-dom';
import { connect } from 'react-redux';
import { Field, reduxForm } from 'redux-form';
import { login } from '../../actions/auth';

class LoginForm extends Component {
  renderField = ({ input, label, type, meta: { touched, error } }) => {
    return (
      <div className={`field ${touched && error ? 'error' : ''}`}>
        <label>{label}</label>
        <input {...input} type={type} />
        {touched && error && (
          <span className='ui pointing red basic label'>{error}</span>
        )}
      </div>
    );
  };

```

```
      <div className='field'>
        <input type={type} />
        {error && <div className='ui red message'>{error}</div>}
      </div>
    );
  };

  onSubmit = formValues => {
    this.props.login(formValues);
  };

  render() {
    if (this.props.isAuthenticated) {
      return <Redirect to='/' />;
    }
    return (
      <div className='ui container'>
        <div className='ui segment'>
          <form
            onSubmit={this.props.handleSubmit(this.onSubmit)}
            className='ui form'
          >
            <Field
              name='username'
              type='text'
              component={this.renderField}
              label='Username'
            />
            <Field
              name='password'
              type='password'
              component={this.renderField}
              label='Password'
            />
            <Field
              name='non_field_errors'
              type='hidden'
              component={this.hiddenField}
            />
            <button className='ui primary button'>Login</button>
          </form>
          <p style={{ marginTop: '1rem' }}>
            Don't have an account? <Link to='/register'>Register</Link>
          </p>
        </div>
      </div>
```

```
}

const mapStateToProps = state => ({
  isAuthenticated: state.auth.isAuthenticated
});

LoginForm = connect(
  mapStateToProps,
  { login }
)(LoginForm);

export default reduxForm({
  form: 'loginForm'
})(LoginForm);
```

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:

```
// frontend/src/components/common/PrivateRoute.js

import React from 'react';
import { Route, Redirect } from 'react-router-dom';
import { connect } from 'react-redux';

const PrivateRoute = ({ component: Component, auth, ...rest }) => (
  <Route
    {...rest}
    render={props => {
      if (auth.isLoading) {
        return <div>Loading...</div>;
      } else if (!auth.isAuthenticated) {
        return <Redirect to='/login' />;
      } else {
```

```
    }}
  />
);

const mapStateToProps = state => ({
  auth: state.auth
});

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:

```
// frontend/src/components/App.js

import LoginForm from './auth/LoginForm'; // added
import PrivateRoute from './common/PrivateRoute'; // added

import { loadUser } from '../actions/auth'; // added

class App extends Component {
  // added
  componentDidMount() {
    store.dispatch(loadUser());
  }

  render() {
    return (
      <Provider store={store}>
        <Router history={history}>
          <Header />
          <Switch>
            <PrivateRoute exact path='/' component={Dashboard} /> // updated
            <Route exact path='/delete/:id' component={TodoDelete} />
            <Route exact path='/edit/:id' component={TodoEdit} />
            <Route exact path='/login' component={LoginForm} /> // added
          </Switch>
        </Router>
      </Provider>
    );
```

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

Non-field errors are also displayed correctly.

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:

Reducers

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

```
// frontend/src/reducers/auth.js

// added LOGOUT_SUCCESS
import {
  USER_LOADING,
  USER_LOADED,
  AUTH_ERROR,
  LOGIN_SUCCESS,
  LOGIN_FAIL,
  LOGOUT_SUCCESS
} from '../actions/types';

export default function(state = initialState, action) {
  switch (action.type) {
    // ...
    case AUTH_ERROR:
    case LOGIN_FAIL:
    case LOGOUT_SUCCESS: // added
      localStorage.removeItem('token');
      return {
        ...state,
        isLoading: false,
        isAuthenticated: false,
        user: null,
        token: null
      };
    // ...
  }
}
```

auth.js hosted with ❤ by GitHub view raw

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:

```
// frontend/src/reducers/index.js

import { LOGOUT_SUCCESS } from '../actions/types'; // added

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

const appReducer = combineReducers({
  form: formReducer,
  todos,
  auth
});

const rootReducer = (state, action) => {
  if (action.type === LOGOUT_SUCCESS) {
    state = undefined;
  }
  return appReducer(state, action);
};

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:

```
// frontend/src/components/layout/Headers.js

import React, { Component } from 'react';
import { Link } from 'react-router-dom'; // 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'>
34           Login
35         </Link>
36       </div>
37     );
38
39     // updated
40     return (
41       <div className='ui inverted menu' style={{ borderRadius: '0' }}>
42         <Link to='/' className='header item'>
43           TodoCRUD
44         </Link>
45         <Link to='/' className='item'>
46           Home
47         </Link>
48         {isAuthenticated ? userLinks : guestLinks}
49       </div>
50     );
51   }
```

```
// added
const mapStateToProps = state => ({
  auth: state.auth
});

// updated
export default connect(
  mapStateToProps,
  { logout }
)(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:

```
// frontend/src/actions/auth.js

import {
  USER_LOADING,
  USER_LOADED,
  AUTH_ERROR,
  REGISTER_SUCCESS, // added
  REGISTER_FAIL, // added
  LOGIN_SUCCESS,
  LOGIN_FAIL,
  LOGOUT_SUCCESS
} from './types';

// REGISTER USER
export const register = ({ username, email, password }) => async dispatch => {
  // Headers
  const config = {
    headers: {
      'Content-Type': 'application/json'
    }
  };

  // Request Body
  const body = JSON.stringify({ username, email, password });

  try {
    const res = await axios.post('/api/auth/register', body, config);
    dispatch({
      type: REGISTER_SUCCESS,
      payload: res.data
    });
  } catch (err) {
    dispatch({
      type: REGISTER_FAIL
```

```
  }
};
```

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:

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:

```
// frontend/src/components/auth/RegisterForm.js

import React, { Component } from 'react';
import { Link, Redirect } from 'react-router-dom';
import { connect } from 'react-redux';
import { Field, reduxForm } from 'redux-form';
import { register } from '../../actions/auth';

class RegisterForm extends Component {
  renderField = ({ input, label, type, meta: { touched, error } }) => {
    return (
      <div className={`field ${touched && error ? 'error' : ''}`}>
        <label>{label}</label>
        <input {...input} type={type} />
        {touched && error && (
          <span className='ui pointing red basic label'>{error}</span>
        )}
      </div>
    );
  };

  onSubmit = formValues => {
    this.props.register(formValues);
  };

  render() {
```

```
    }
    return (
      <div className='ui container'>
        <div className='ui segment'>
          <form
            onSubmit={this.props.handleSubmit(this.onSubmit)}
            className='ui form'
          >
            <Field
              name='username'
              type='text'
              component={this.renderField}
              label='Username'
              validate={[required, minLength3, maxLength15]}
            />
            <Field
              name='email'
              type='email'
              component={this.renderField}
              label='Email'
              validate={required}
            />
            <Field
              name='password'
              type='password'
              component={this.renderField}
              label='Password'
              validate={required}
            />
            <Field
              name='password2'
              type='password'
              component={this.renderField}
              label='Confirm Password'
              validate={[required, passwordsMatch]}
            />
            <button className='ui primary button'>Register</button>
          </form>
          <p style={{ marginTop: '1rem' }}>
            Already have an account? <Link to='/login'>Login</Link>
          </p>
        </div>
      </div>
    );
  }
}
```

```
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);
```

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 {
```

```
  render() {
    return (
      <Provider store={store}>
        <Router history={history}>
          <Header />
          <Switch>
            <PrivateRoute exact path='/' component={Dashboard} />
            <Route exact path='/delete/:id' component={TodoDelete} />
            <Route exact path='/edit/:id' component={TodoEdit} />
            <Route exact path='/register' component={RegisterForm} /> // added
            <Route exact path='/login' component={LoginForm} />
          </Switch>
        </Router>
      </Provider>
    );
  }
}
```

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!

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

```python
# frontend/urls.py

urlpatterns = [
    path('', index),
    path('login', index),  # added
    path('register', index),  # added
    path('edit/<int:pk>', TodoDetailView.as_view()),
    path('delete/<int:pk>', TodoDetailView.as_view()),
]
```

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!