

Build a CRUD Todo app with Django and React/Redux

Tutorial for Beginners



Koji Mochizuki

Jun 29, 2019 · 10 min read ★



Photo by Anete Lūsina on Unsplash

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

Task

Add

<input type="checkbox"/>	buy milk	06/12/2019 04:22:16	<button>Delete</button>
<input type="checkbox"/>	wash the car	06/12/2019 04:25:46	<button>Delete</button>
<input type="checkbox"/>	pay bills	06/12/2019 04:26:53	<button>Delete</button>
<input type="checkbox"/>	tidy my room	06/12/2019 04:27:59	<button>Delete</button>
<input type="checkbox"/>	call John	06/12/2019 04:30:28	<button>Delete</button>

CRUD Todo App

Table of Contents

- Setting up Django
 - Setting up React
 - Getting data from the API and displaying the list
 - Creating Form and adding a new Todo
 - Creating a Header
 - Removing Todos
 - Editing Todos
- . . .

Setting up Django

Creating a virtual environment with Pipenv

First, we will create the folder for the project and navigate into it:

Let's create a virtual environment by running this command:

```
$ pipenv --python 3
```

If you don't have Pipenv installed yet, please install it by running this command:

```
$ pip install pipenv
```

We will install the packages we need:

```
$ pipenv install django djangorestframework
```

Creating a new project and some apps

In this tutorial, we will create a project named “todocrud”. We can leave out the extra folder which is automatically created by adding a dot to the end of the command and running:

```
$ django-admin startproject todocrud .
```

Next, we will create two apps. One is for the backend, the other is for the frontend:

```
$ python manage.py startapp todos
$ python manage.py startapp frontend
```

We will open the `settings.py` file in the project directory, and configure to use the apps we created and Django REST framework:

```
1 # settings.py
2
```

```
7     'django.contrib.admin',
8     'django.contrib.auth',
9     'django.contrib.contenttypes',
10    'django.contrib.sessions',
11    'django.contrib.messages',
12    'django.contrib.staticfiles',
13 ]
14
15 REST_FRAMEWORK = { # added
16     'DEFAULT_PERMISSION_CLASSES': [
17         'rest_framework.permissions.AllowAny'
18     ],
19     'DATETIME_FORMAT': "%m/%d/%Y %H:%M:%S",
20 }
```

settings.py hosted with ❤ by GitHub

[view raw](#)

We can specify the output format of a date and time by including `DATETIME_FORMAT` in a configuration dictionary named `REST_FRAMEWORK`.

Let's apply migrations and start the development server:

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

Visit `http://127.0.0.1:8000/` with your browser. If you see the page a rocket taking off, it worked well!

Writing backend's modules

First, we will create a simple model. Open the `models.py` file and write the following code:

```
1 # todos/models.py
2
3 from django.db import models
4
5
6 class Todo(models.Model):
7     task = models.CharField(max_length=255)
```

Next, we will build a simple model-backed API using REST framework. Let's create a new folder named `api` and create new files `__init__.py`, `serializers.py`, `views.py` and `urls.py` in it:

```
todos/
  api/
    __init__.py
    serializers.py
    urls.py
    views.py
```

Since the `api` is a module, we need to include `__init__.py` file.

Let's define the API representation in the `serializers.py` file:

```
1 # todos/api/serializers.py
2
3 from rest_framework import serializers
4
5 from todos.models import Todo
6
7
8 class TodoSerializer(serializers.ModelSerializer):
9     class Meta:
10         model = Todo
11         fields = '__all__'
```

The `ModelSerializer` class will create fields that correspond to the Model fields.

Next, we will define the view behavior in the `api/views.py` file:

```
1 # todos/api/views.py
2
3 from rest_framework import viewsets
```

```
9 class TodoViewSet(viewsets.ModelViewSet):
10     queryset = Todo.objects.all()
11     serializer_class = TodoSerializer
```

views.py hosted with ❤ by GitHub

[view raw](#)

We will finally write the URL configuration using Routers :

```
1 # todos/api/urls.py
2
3 from rest_framework import routers
4
5 from .views import TodoViewSet
6
7 router = routers.DefaultRouter()
8 router.register('todos', TodoViewSet, 'todos')
9 # router.register('<The URL prefix>', <The viewset class>, '<The URL name>')
10
11 urlpatterns = router.urls
```

urls.py hosted with ❤ by GitHub

[view raw](#)

We use three arguments to the `register()` method, but the third argument is not required.

Writing frontend's modules

In the frontend, all we have to do is write simple views and URL patterns.

Open the `frontend/views.py` file and create the two views:

```
1 # frontend/views.py
2
3 from django.shortcuts import render
4 from django.views.generic.detail import DetailView
5
6 from todos.models import Todo
7
8
9 def index(request):
10     return render(request, 'frontend/index.html')
```

[views.py](#) hosted with ❤ by GitHub[view raw](#)

We will create the `frontend/index.html` file later. Don't worry about it now.

Add a new `urls.py` file to the same directory and create the URL conf:

```
1 # frontend/urls.py
2
3 from django.urls import path
4
5 from .views import index, TodoDetailView
6
7 urlpatterns = [
8     path('', index),
9     path('edit/<int:pk>', TodoDetailView.as_view()),
10    path('delete/<int:pk>', TodoDetailView.as_view()),
11 ]
```

[urls.py](#) hosted with ❤ by GitHub[view raw](#)

As you can see above, the `index` view is for the index page and the `TodoDetailView` is called when we request a specific object.

Wire up the URLs

We will include the frontend's and backend's 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('admin/', admin.site.urls),
10 ]
```

[urls.py](#) hosted with ❤ by GitHub[view raw](#)

As a final part of this chapter, we will make a new migration file and apply changes to our databases by running the commands below:

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

Setting up React

Creating directories

First of all, let's create all of the directories we need:

```
$ mkdir -p ./frontend/src/{components,actions,reducers}  
$ mkdir -p ./frontend/{static,templates}/frontend
```

The above command should create the directories as follows:

```
frontend/  
  src/  
    actions/  
    components/  
    reducers/  
  static/  
    frontend/  
  templates/  
    frontend/
```

Installing packages

We need to create a `package.json` file by running the following command before installing packages:

```
$ npm init -y
```

```
$ npm i -D webpack webpack-cli
$ npm i -D babel-loader @babel/core @babel/preset-env @babel/preset-react @babel/plugin-proposal-class-properties

$ npm i react react-dom react-router-dom
$ npm i redux react-redux redux-thunk redux-devtools-extension
$ npm i redux-form
$ npm i axios
$ npm i lodash
```

Creating config files

Add a file named `.babelrc` to the root directory and configure **Babel**:

```
1 // .babelrc
2
3 {
4   "presets": [
5     [
6       "@babel/preset-env",
7       {
8         "targets": {
9           "node": "current"
10        },
11        "useBuiltIns": "usage",
12        "corejs": 3
13      }
14    ],
15    "@babel/preset-react"
16  ],
17  "plugins": ["@babel/plugin-proposal-class-properties"]
18 }
```

.babelrc hosted with ❤ by GitHub

[view raw](#)

We can use **Async/Await** with **Babel** by writing as above.

Secondary, add a file named `webpack.config.js` to the same directory and write a configuration for `webpack`:

```
1 // webpack.config.js
```

```
7     test: /\.js|jsx$/,
8     exclude: /node_modules/,
9     use: [
10       {
11         loader: 'babel-loader'
12       }
13     ]
14   }
15 };
```

webpack.config.js hosted with ❤ by GitHub

[view raw](#)

Additionally, we need to rewrite the "scripts" property of the `package.json` file:

```
1 // package.json
2
3 {
4   // ...
5   "scripts": {
6     "dev": "webpack --mode development --watch ./frontend/src/index.js --output ./frontend/main.js",
7     "build": "webpack --mode production ./frontend/src/index.js --output ./frontend/main.js"
8   },
9   // ...
10 }
```

package.jsonc hosted with ❤ by GitHub

[view raw](#)

Two new scripts have been defined. We can run scripts with `npm run dev` for development or `npm run build` for production. When these scripts are run, webpack bundles modules and output the `main.js` file.

Creating principal files

Let's create three principal files and render the first word.

We will create a file named `index.js` that is called first when we run the React application:

```
1 // frontend/src/index.js
2
3 import App from './components/App';
```

Next, we will create a file named `App.js` that is a parent component:

```
1 // frontend/src/components/App.js
2
3 import React, { Component } from 'react';
4 import ReactDOM from 'react-dom';
5
6 class App extends Component {
7   render() {
8     return (
9       <div>
10         <h1>ToDoCRUD</h1>
11       </div>
12     );
13   }
14 }
15
16 ReactDOM.render(<App />, document.querySelector('#app'));
```

App.js hosted with ❤ by GitHub

[view raw](#)

We will finally create a template file named `index.html` that is specified in the `views.py` file:

```
1 <!-- templates/frontend/index.html -->
2
3 <!DOCTYPE html>
4 <html lang="en">
5
6 <head>
7   <meta charset="UTF-8">
8   <meta name="viewport" content="width=device-width, initial-scale=1.0">
9   <meta http-equiv="X-UA-Compatible" content="ie=edge">
10  <!-- semantic-ui CDN -->
11  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.4.1/
12    <title>ToDoCRUD</title>
13 </head>
14
15 <body>
16   <div id='app'></div>
17
```

In this tutorial, we will use Semantic UI as a CSS framework.

Place the wrapper for rendering the App component and the bundled script into the `<body>` tag.

Checking the display

Let's see whether it is displayed correctly.

Open another terminal and run the script:

```
$ npm run dev
```

The `main.js` file should be generated in the `static/frontend` directory.

Then, start the dev server and visit `http://127.0.0.1:8000/`:

```
$ python manage.py runserver
```

If the word “ToDoCRUD” is displayed, everything is going well so far :)

Getting data from the API and displaying the list

It is time to use **Redux**. We will create **Actions**, **Reducers** and **Store**.

Actions

Let's define all the **type properties** in advance. Add a new file named `types.js` into the `src/actions` directory:

```
1 // actions/types.js
2
3 export const GET TODOS = 'GET_TODOS';
4 export const GET TODO = 'GET_TODO'.
```

In order to create actions, we need to define **Action Creators**. Add a new file named `todos.js` into the `src/actions` directory:

```
1 // actions/todos.js
2
3 import axios from 'axios';
4 import { GET_TODOS } from './types';
5
6 // GET TODOS
7 export const getTodos = () => async dispatch => {
8     const res = await axios.get('/api/todos/');
9     dispatch({
10         type: GET_TODOS,
11         payload: res.data
12     });
13 }
```

[todos.js hosted with ❤ by GitHub](#)

[view raw](#)

Reducers

Reducers specify how the application's state changes in response to actions sent to the store.

That is the role of **Reducers**. Add a new file named `todos.js` into the `src/reducers` directory and write a child reducer:

```
1 // reducers/todos.js
2
3 import _ from 'lodash';
4 import { GET_TODOS } from '../actions/types';
5
6 export default (state = {}, action) => {
7     switch (action.type) {
8         case GET_TODOS:
9             return {
10                 ...state,
11                 ..._.mapKeys(action.payload, 'id')
12             };
13     }
14 }
```

Lodash is a JavaScript utility library. It is not a requirement, but it can cut down on development time and make your codebase smaller.

Let's create the parent reducer to put together every child reducer using `combineReducers()`. Add a new file named `index.js` into the `src/reducers` directory:

```
1 // reducers/index.js
2
3 import { combineReducers } from 'redux';
4 import { reducer as formReducer } from 'redux-form';
5 import todos from './todos';
6
7 export default combineReducers({
8   form: formReducer,
9   todos
10});
```

[index.js](#) hosted with ❤ by GitHub

[view raw](#)

In order to use `redux-form`, we need to include its reducer in the `combineReducers` function.

Store

The **Store** is an object to hold the **state** of our application. In addition, we will use the recommended middleware **Redux Thunk** to write async logic that interacts with the store. Let's create a new file named `store.js` in the `src` directory:

```
1 // fronted/src/store.js
2
3 import { createStore, applyMiddleware } from 'redux';
4 import { composeWithDevTools } from 'redux-devtools-extension';
5 import reduxThunk from 'redux-thunk';
6 import rootReducer from './reducers';
7
8 const store = createStore(
9   rootReducer,
10  composeWithDevTools(applyMiddleware(reduxThunk))
11);
```

Use of **Redux DevTools** is optional, but it is very useful because it visualizes the state changes of Redux. I will omit how to use it here, but it is highly recommended.

Components

First, create a new folder named `todos` in the `components` directory. And then, add a new file named `TodoList.js` into the folder we created:

```
1 // components/todos/TodoList.js
2
3 import React, { Component } from 'react';
4 import { connect } from 'react-redux';
5 import { getTodos } from '../../actions/todos';
6
7 class TodoList extends Component {
8     componentDidMount() {
9         this.props.getTodos();
10    }
11
12    render() {
13        return (
14            <div className='ui relaxed divided list' style={{ marginTop: '2rem' }}>
15                {this.props.todos.map(todo => (
16                    <div className='item' key={todo.id}>
17                        <i className='large calendar outline middle aligned icon' />
18                        <div className='content'>
19                            <a className='header'>{todo.task}</a>
20                            <div className='description'>{todo.created_at}</div>
21                        </div>
22                    </div>
23                )));
24            </div>
25        );
26    }
27 }
28
29 const mapStateToProps = state => ({
30     todos: Object.values(state.todos)
31 });
32
33 export default connect(
34     mapStateToProps,
```

We use Semantic UI list to decorate the list.

The `connect()` function connects this component to the store. It accepts `mapStateToProps` as the first argument, Action Creators as the second argument. We will be able to use the store state as **Props** by specifying `mapStateToProps`.

We will create a new file named `Dashboard.js` in the same directory. It is just a container for `TodoList` and a form we will create in the next chapter:

```
1 // components/todos/Dashboard.js
2
3 import React, { Component } from 'react';
4 import TodoList from './TodoList';
5
6 class Dashboard extends Component {
7   render() {
8     return (
9       <div className='ui container'>
10         <div>Todo Create Form</div>
11         <TodoList />
12       </div>
13     );
14   }
15 }
16
17 export default Dashboard;
```

Dashboard.js hosted with ❤ by GitHub

[view raw](#)

Open the `App.js` file and update as follows:

```
1 // components/App.js
2
3 import Dashboard from './todos/Dashboard'; // added
4
5 import { Provider } from 'react-redux'; // added
6 import store from '../store'; // added
7
8 class App extends Component {
9   render() {
10   }
```

```
15     }
16 }
```

App.js hosted with ❤ by GitHub

[view raw](#)

The `Provider` makes the store available to the component nested inside of it.

Checking the display

First, visit `http://127.0.0.1:8000/api/todos/` and create several objects. And then, visit `http://127.0.0.1:8000/`.

You should see a simple list of the objects you created. Did it work?

Creating Form and adding a new Todo

Actions

Open the `actions/todos.js` file, and add a new action creator:

```
1 // actions/todos.js
2
3 import { reset } from 'redux-form'; // added
4 import { GET_TODOS, ADD_TODO } from './types'; // added ADD_TODO
5
6 // ADD TODO
7 export const addTodo = formValues => async dispatch => {
8   const res = await axios.post('/api/todos/', { ...formValues });
9   dispatch({
10     type: ADD_TODO,
11     payload: res.data
12   });
13   dispatch(reset('todoForm'));
14 }
```

todos.js hosted with ❤ by GitHub

[view raw](#)

Dispatching `reset('formName')` clears our form after we submission succeeds. We will specify the form name later in the Form component.

Reducers

```

3 import { GET_TODOS, ADD_TODO } from '../actions/types'; // added ADD_TODO
4
5 export default (state = {}, action) => {
6   switch (action.type) {
7     // ...
8     case ADD_TODO: // added
9       return {
10         ...state,
11         [action.payload.id]: action.payload
12       };
13     // ...
14   }
15 };

```

todos.js hosted with ❤ by GitHub

[view raw](#)

Components

Let's create a Form component. We will create a Form separately as a reusable component so that it can also be used for editing. Create a new file named `TodoForm.js` in the `components/todos` directory:

```

1 // components/todos/TodoForm.js
2
3 import React, { Component } from 'react';
4 import { Field, reduxForm } from 'redux-form';
5
6 class TodoForm extends Component {
7   renderField = ({ input, label, meta: { touched, error } }) => {
8     return (
9       <div className={`field ${touched && error ? 'error' : ''}`}>
10         <label>{label}</label>
11         <input {...input} autoComplete='off' />
12         {touched && error && (
13           <span className='ui pointing red basic label'>{error}</span>
14         )}
15       </div>
16     );
17   };
18
19   onSubmit = formValues => {
20     this.props.onSubmit(formValues);

```

```

25      <div className='ui segment'>
26          <form
27              onSubmit={this.props.handleSubmit(this.onSubmit)}
28                  className='ui form error'
29          >
30              <Field name='task' component={this.renderField} label='Task' />
31              <button className='ui primary button'>Add</button>
32          </form>
33      </div>
34  );
35 }
36 }
37
38 const validate = formValues => {
39     const errors = {};
40
41     if (!formValues.task) {
42         errors.task = 'Please enter at least 1 character';
43     }
44
45     return errors;
46 };
47
48 export default reduxForm({
49     form: 'todoForm',
50     touchOnBlur: false,
51     validate
52 })(TodoForm);

```

[TodoForm.js](#) hosted with ❤ by GitHub

[view raw](#)

The tutorial would be lengthy, so I will leave out how to use **Redux Form**. To understand how the Redux Form works, it is a good idea to try to customize your form referring to the documentation.

'todoForm' is the name of this form. That is what we used in the action creator `addTodo`.

When we click in the textbox and then remove the focus, it displays a validation error, so specify `touchOnBlur: false` to disable it.

Next, let's create a component for adding new todos. Create a new file named

```

3 import React, { Component } from 'react';
4 import { connect } from 'react-redux';
5 import { addTodo } from '../../actions/todos';
6 import TodoForm from './TodoForm';
7
8 class TodoCreate extends Component {
9   onSubmit = formValues => {
10     this.props.addTodo(formValues);
11   };
12
13   render() {
14     return (
15       <div style={{ marginTop: '2rem' }}>
16         <TodoForm destroyOnUnmount={false} onSubmit={this.onSubmit} />
17       </div>
18     );
19   }
20 }
21
22 export default connect(
23   null,
24   { addTodo }
25 )(TodoCreate);

```

[TodoCreate.js](#) hosted with ❤ by GitHub

[view raw](#)

All we have to do is render the `TodoForm`. By setting `destroyOnUnmount` to `false`, we can disable that the Redux Form automatically destroys a form state in the Redux store when the component is unmounted. It is for displaying the form state in an editing form.

If we don't need to specify a `mapStateToProps` function, set `null` into `connect()`.

Let's view and test the form. Open the `Dashboard.js` file, and update as follows:

```

1 // components/todos/Dashboard.js
2
3 import TodoCreate from './TodoCreate'; // added
4
5 class Dashboard extends Component {
6   render() {

```

```
12     );
13 }
14 }
15
16 export default Dashboard;
```

Dashboard.js hosted with ❤ by GitHub

[view raw](#)

Creating a Header

Let's take a break and create a header. Create a new folder named `layout`, and then add a new file name `Header.js` into it:

```
1 // components/layout/Header.js
2
3 import React, { Component } from 'react';
4
5 class Header extends Component {
6   render() {
7     return (
8       <div className='ui inverted menu' style={{ borderRadius: '0' }}>
9         <a className='header item'>TodoCRUD</a>
10        <a className='item'>Home</a>
11      </div>
12    );
13  }
14 }
15
16 export default Header;
```

Header.js hosted with ❤ by GitHub

[view raw](#)

Open the `App.js` file and nest the `Header` component:

```
1 // components/App.js
2
3 import Header from './layout/Header'; // added
4
5 class App extends Component {
6   render() {
7     return (
```

```
12      };
13  }
14 }
```

App.js hosted with ❤ by GitHub

[view raw](#)

In this tutorial, the header is just an ornament.

Removing Todos

First, we will create a `history` object using the `history` package. We can use it for changing the current location. Create a new file named `history.js` in the `frontend/src` directory, and write the code below:

```
1 // frontend/src/history.js
2
3 import { createBrowserHistory } from 'history';
4
5 export default createBrowserHistory();
```

history.js hosted with ❤ by GitHub

[view raw](#)

Actions

Open the `actions/todos.js` file, and add two new action creators:

```
1 // actions/todos.js
2
3 import history from '../history'; // added
4 import { GET_TODOS, GET_TODO, ADD_TODO, DELETE_TODO } from './types'; // added GET_TODOS
5
6 // GET TODO
7 export const getTodo = id => async dispatch => { // added
8   const res = await axios.get(`/api/todos/${id}/`);
9   dispatch({
10     type: GET_TODO,
11     payload: res.data
12   });
13 };
14
15 // DELETE TODO
16 export const deleteTodo = id => async dispatch => { // added
```

```
21  });
22  history.push('/');
```

todos.js hosted with ❤ by GitHub

[view raw](#)

We have created `getTodo` to get a specific object and `deleteTodo` to delete the object.

We are going to create a modal window for confirmation of deletion later. The `history.push('/')` method automatically takes us from the modal window to the index page after removing an object.

Reducers

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

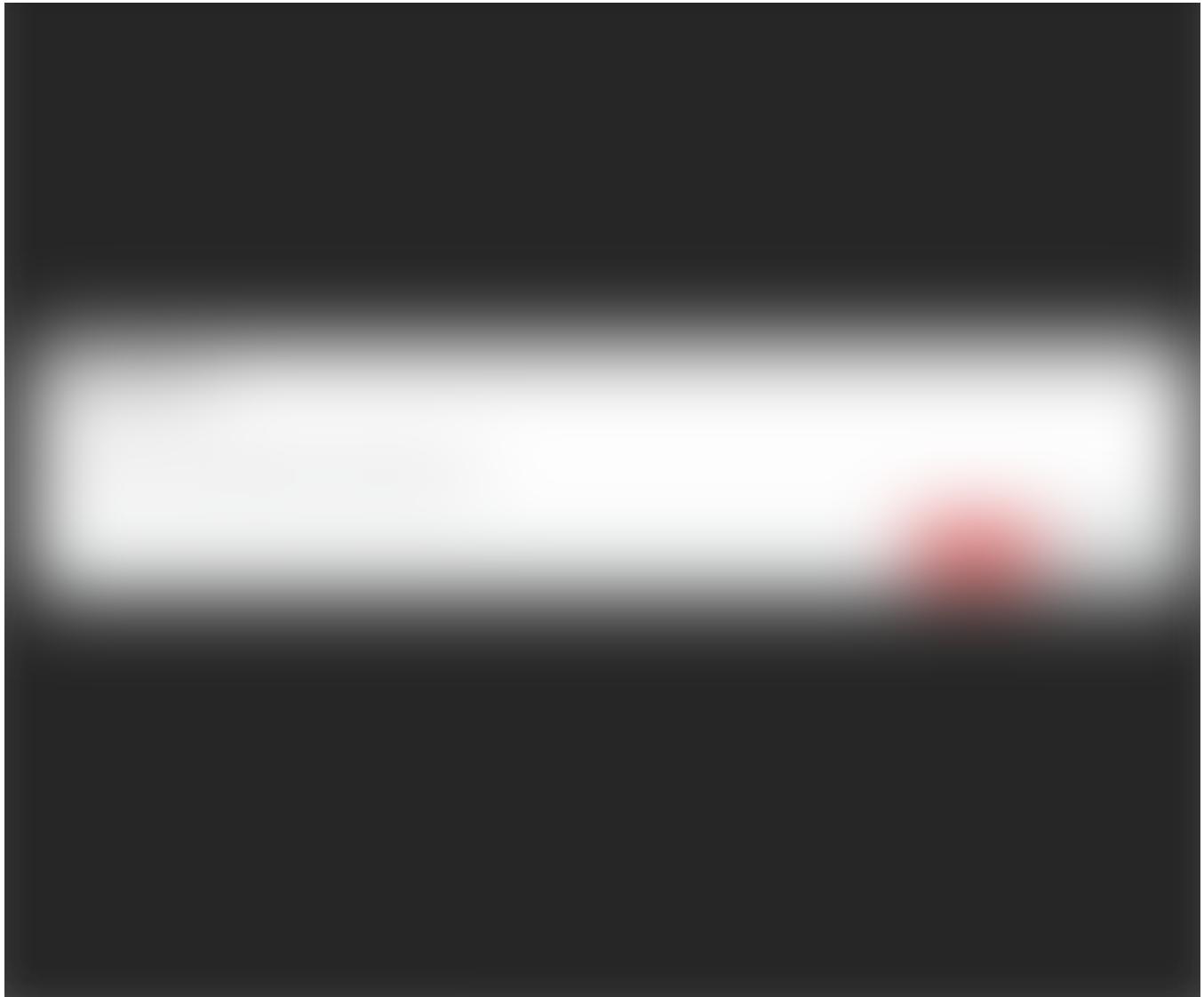
```
1 // reducers/todos.js
2
3 import _ from 'lodash'; // added
4 import { GET_TODOS, GET_TODO, ADD_TODO, DELETE_TODO } from '../actions/types'; // added
5
6 export default (state = {}, action) => {
7   switch (action.type) {
8     // ...
9     case GET_TODO: // added
10    case ADD_TODO:
11      return {
12        ...state,
13        [action.payload.id]: action.payload
14      };
15    case DELETE_TODO: // added
16      return _.omit(state, action.payload);
17    // ...
18  }
19};
```

todos.js hosted with ❤ by GitHub

[view raw](#)

The `GET_TODO` action is the same as the `ADD_TODO` action, so we only need to set the case. For the `DELETE_TODO` action, use Lodash again as a shortcut.

Components



The modal window for confirmation of the deletion

Create a new file named `Modal.js` in the `components/layout` directory, and write as follows:

```
1 // components/layout/Modal.js
2
3 import React from 'react';
4 import ReactDOM from 'react-dom';
5
6 const Modal = props => {
7   return ReactDOM.createPortal(
8     <div onClick={props.onDismiss} className='ui active dimmer'>
9       <div onClick={e => e.stopPropagation()} className='ui active modal'>
10         <div className='header'>{props.title}</div>
11         <div className='content'>{props.content}</div>
```

```
10      );
11  };
12
13  export default Modal;
```

Modal.js hosted with ❤ by GitHub

[view raw](#)

To render the `Modal` component outside the DOM hierarchy of the parent component, create a portal using `createPortal()`. The first argument is the renderable child element and the second argument is the DOM element to render.

And then, open the `index.html` file and add a container for the Modal inside the `<body>` tag:

```
1  <!-- templates/frontend/index.html -->
2
3  <body>
4    <div id='app'></div>
5    <div id="modal"></div>
6
7    {% load static %}
8    <script src="{% static 'frontend/main.js' %}"></script>
9  </body>
```

index.html hosted with ❤ by GitHub

[view raw](#)

Next, we will create a new component `TodoDelete.js` in the `components/todos` directory:

```
1  // components/todos/TodoDelete.js
2
3  import React, { Component, Fragment } from 'react';
4  import { connect } from 'react-redux';
5  import { Link } from 'react-router-dom';
6  import Modal from '../layout/Modal';
7  import history from '../../history';
8  import { getTodo, deleteTodo } from '../../../../../actions/todos';
9
10 class TodoDelete extends Component {
11   componentDidMount() {
12     this.props.getTodo(this.props.match.params.id);
```

```

17         return 'Are you sure you want to delete this task?';
18     }
19     return `Are you sure you want to delete the task: ${this.props.todo.task}`;
20 }
21
22 renderActions() {
23     const { id } = this.props.match.params;
24     return (
25         <Fragment>
26             <button
27                 onClick={() => this.props.deleteTodo(id)}
28                 className='ui negative button'
29             >
30                 Delete
31             </button>
32             <Link to='/' className='ui button'>
33                 Cancel
34             </Link>
35         </Fragment>
36     );
37 }
38
39 render() {
40     return (
41         <Modal
42             title='Delete Todo'
43             content={this.renderContent()}
44             actions={this.renderActions()}
45             onDismiss={() => history.push('/')}
46         />
47     );
48 }
49 }
50
51 const mapStateToProps = (state, ownProps) => ({
52     todo: state.todos[ownProps.match.params.id]
53 });
54
55 export default connect(
56     mapStateToProps,
57     { getTodo, deleteTodo }
58 )(TodoDelete);

```

the Modal component. `onDismiss` is set to return to the index page when the dim part of the modal window is clicked.

We can retrieve the data from its own props by specifying `ownProps` as the second argument to `mapStateToProps`.

Let's open the `TodoList.js` file and put a delete button:

```
1 // components/todos/TodoList.js
2
3 import { Link } from 'react-router-dom'; // added
4 import { getTodos, deleteTodo } from '../../../../../actions/todos'; // added deleteTodo
5
6 class TodoList extends Component {
7     // ...
8
9     render() {
10         return (
11             <div className='ui relaxed divided list' style={{ marginTop: '2rem' }}>
12                 {this.props.todos.map(todo =>
13                     <div className='item' key={todo.id}>
14                         <div className='right floated content'> // added
15                             <Link
16                                 to={`/delete/${todo.id}`}
17                                 className='small ui negative basic button'
18                             >
19                                 Delete
20                             </Link>
21                         </div>
22                         <i className='large calendar outline middle aligned icon' />
23                         <div className='content'>
24                             <a className='header'>{todo.task}</a>
25                             <div className='description'>{todo.created_at}</div>
26                         </div>
27                     </div>
28                 ))}
29             </div>
30         );
31     }
32 }
33
34 //
```

```
39 })(TodoList);
```

TodoList.js hosted with ❤ by GitHub

[view raw](#)

Finally, we need to configure routing using **React Router**. Open the `App.js` file and configure as follows:

```
1 // components/App.js
2
3 import { Router, Route, Switch } from 'react-router-dom'; // added
4
5 import history from '../history'; // added
6 import TodoDelete from './todos/TodoDelete'; // added
7
8 class App extends Component {
9   render() {
10     return (
11       <Provider store={store}>
12         <Router history={history}>
13           <Header />
14           <Switch>
15             <Route exact path='/' component={Dashboard} />
16             <Route exact path='/delete/:id' component={TodoDelete} />
17           </Switch>
18         </Router>
19       </Provider>
20     );
21   }
22 }
```

App.js hosted with ❤ by GitHub

[view raw](#)

The reason for using `Router` instead of **BrowserRouter** is explained in the REACT TRAINING document as follows:

The most common use-case for using the low-level <Router> is to synchronize a custom history with a state management lib like Redux or Mobx. Note that this is not required to use state management libs alongside React Router, it's only for deep integration.

The `exact` parameter specified in `Route` returns a route only if the `path` exactly matches the current URL.

Editing Todos

This is the last chapter. We are almost done, so let's keep going!

Actions

Open the `actions/todos.js` file, and add a new action creator:

```
1 // actions/todos.js
2
3 import { GET_TODOS, GET_TODO, ADD_TODO, DELETE_TODO, EDIT_TODO } from './types'; // added
4
5 // EDIT TODO
6 export const editTodo = (id, formValues) => async dispatch => {
7   const res = await axios.patch(`api/todos/${id}/`, formValues);
8   dispatch({
9     type: EDIT_TODO,
10    payload: res.data
11  });
12  history.push('/');
13};
```

[todos.js](#) hosted with ❤ by GitHub

[view raw](#)

Reducers

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

```
1 // reducers/todos.js
2
3 import {
4   GET_TODOS,
5   GET_TODO,
6   ADD_TODO,
7   DELETE_TODO,
8   EDIT_TODO // added
9 } from '../actions/types';
10
11 export default (state = {}, action) => {
12   switch (action.type) {
13     // ...
14     case GET_TODO:
15     case ADD_TODO:
```

```
20      };
21      // ...
22  }
23 };
```

[todos.js](#) hosted with ❤ by GitHub

[view raw](#)

Components

Create a new component `TodoEdit.js` in the `components/todos` directory:

```
1 // components/todos/TodoEdit.js
2
3 import _ from 'lodash';
4 import React, { Component } from 'react';
5 import { connect } from 'react-redux';
6 import { getTodo, editTodo } from '../../actions/todos';
7 import TodoForm from './TodoForm';
8
9 class TodoEdit extends Component {
10   componentDidMount() {
11     this.props.getTodo(this.props.match.params.id);
12   }
13
14   onSubmit = formValues => {
15     this.props.editTodo(this.props.match.params.id, formValues);
16   };
17
18   render() {
19     return (
20       <div className='ui container'>
21         <h2 style={{ marginTop: '2rem' }}>Edit Todo</h2>
22         <TodoForm
23           initialValues={_.pick(this.props.todo, 'task')}
24           enableReinitialize={true}
25           onSubmit={this.onSubmit}
26         />
27       </div>
28     );
29   }
30 }
31
32 const mapStateToProps = (state, ownProps) => ({
```

```
  ...
  import {TodoEdit} from './todos/TodoEdit';
38  { getTodo, editTodo }
39 ) (TodoEdit);
```

TodoEdit.js hosted with ❤ by GitHub

[view raw](#)

Specify an object in `initialValues`. We can get only the value of `task` using the `_.pick` function of **Lodash**. In addition, set `enableReinitialize` to `true` so that we can also get the value when the page is reloaded. Pass these optional properties to the `TodoForm`.

Open the `TodoList.js` file and update `{todo.task}` as follows:

```
1 // components/todos/TodoList.js
2
3 <Link to={`/edit/${todo.id}`} className='header'>
4   {todo.task}
5 </Link>
```

TodoList.js hosted with ❤ by GitHub

[view raw](#)

Let's add the new component to the `App.js` file:

```
1 // components/App.js
2
3 import TodoEdit from './todos/TodoEdit'; // added
4
5 class App extends Component {
6   render() {
7     return (
8       <Provider store={store}>
9         <Router history={history}>
10           <Header />
11           <Switch>
12             <Route exact path='/' component={Dashboard} />
13             <Route exact path='/delete/:id' component={TodoDelete} />
14             <Route exact path='/edit/:id' component={TodoEdit} /> // added
15           </Switch>
16         </Router>
17       </Provider>
18     );
19 }
```

Finally, let's change the text of the button on the edit form from "Add" to "Update".

Open the `TodoForm.js` file and update as follows:

```
1 // components/todos/TodoForm.js
2
3 class TodoForm extends Component {
4     // ...
5
6     render() {
7         const btnText = `${this.props.initialValues ? 'Update' : 'Add'}`; // added
8         return (
9             <div className='ui segment'>
10                 <form
11                     onSubmit={this.props.handleSubmit(this.onSubmit)}
12                     className='ui form error'
13                 >
14                     <Field name='task' component={this.renderField} label='Task' />
15                     <button className='ui primary button'>{btnText}</button> // updated
16                 </form>
17             </div>
18         );
19     }
20 }
```

[TodoForm.js](#) hosted with ❤ by GitHub

[view raw](#)

Now, click on any task on the index page and try editing:



The edit form

You should have been able to change the value from the form.

Next Step

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

[medium.com](https://medium.com/@mehmetcan.ozturk/implement-user-auth-in-a-django-react-app-with-knox-5a2a2a2a2a2a)

Programming Coding Django React Redux

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