Build a CRUD Todo app with Django and React/Redux

Tutorial for Beginners

Koji Mochizuki Follow
Jun 29, 2019 · 10 min read ★

Photo by Anete Lūsiņa 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.

At the end of this tutorial, we will have the application that looks like this:

CRUD Todo App

Table of Contents

. . .

Setting up Django

Creating a virtual environment with Pipenv

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

```
$ mkdir django-react-todo
$ cd django-react-todo
```

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:

```
# settings.py

INSTALLED_APPS = [
    'frontend.apps.FrontendConfig',  # added
    'todos.apps.TodosConfig',  # added
    'rest_framework',  # added
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]

REST_FRAMEWORK = {  # added
    'DEFAULT_PERMISSION_CLASSES': [
```

```
        'rest_framework.permissions.AllowAny'
    ],
    'DATETIME_FORMAT': "%m/%d/%Y %H:%M:%S",
}
```

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:

```
# todos/models.py

from django.db import models


class Todo(models.Model):
    task = models.CharField(max_length=255)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.task
```

models.py hosted with ♡ by GitHub view raw

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:

```python
# todos/api/serializers.py

from rest_framework import serializers

from todos.models import Todo


class TodoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Todo
        fields = '__all__'
```

serializers.py hosted with ♡ by GitHub view raw

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:

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

from rest_framework import viewsets

from .serializers import TodoSerializer
from todos.models import Todo


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

views.py hosted with ♡ by GitHub view raw

We will finally write the URL configuration using `Routers` :

```python
# todos/api/urls.py

from rest_framework import routers

from .views import TodoViewSet

router = routers.DefaultRouter()
```

```
router.register('todos', TodoViewSet, 'todos')
# router.register('<The URL prefix>', <The viewset class>, '<The URL name>')

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:

```
# frontend/views.py

from django.shortcuts import render
from django.views.generic.detail import DetailView

from todos.models import Todo


def index(request):
    return render(request, 'frontend/index.html')


class TodoDetailView(DetailView):
    model = Todo
    template_name = '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:

```
# frontend/urls.py

from django.urls import path

from .views import index, TodoDetailView

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

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:

```
# 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('admin/', admin.site.urls),
]
```

urls.py hosted with ♡ by GitHub view raw

Although the path for the Django `admin` site is left, we are not going to use it in this tutorial.

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
```

In order to use `npm`, Node.js needs to be installed.

Then, let's install all the packages we use with `npm` command:

```
$ 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**:

```
// .babelrc

{
  "presets": [
    [
      "@babel/preset-env",
      {
        "targets": {
          "node": "current"
        },
        "useBuiltIns": "usage",
        "corejs": 3
```

```
      }
    ],
    "@babel/preset-react"
  ],
  "plugins": ["@babel/plugin-proposal-class-properties"]
}
```

.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` :

```
// webpack.config.js

module.exports = {
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader'
        }
      }
    ]
  }
};
```

webpack.config.js hosted with ♡ by GitHub view raw

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

```
// package.json

{
  // ...
  "scripts": {
    "dev": "webpack --mode development --watch ./frontend/src/index.js --output ./frontend/stati
    "build": "webpack --mode production ./frontend/src/index.js --output ./frontend/static/front
  },
  // ...
}
```

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:

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

import App from './components/App';
```

index.js hosted with ♡ by GitHub view raw

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

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

import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class App extends Component {
  render() {
    return (
      <div>
        <h1>ToDoCRUD</h1>
      </div>
    );
  }
}

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:

```
<!-- templates/frontend/index.html -->

<!DOCTYPE html>
<html lang="en">

<head>
  <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/semantic
12    <title>ToDoCRUD</title>
13  </head>
14
15  <body>
16    <div id='app'></div>
17
18    {% load static %}
19    <script src="{% static 'frontend/main.js' %}"></script>
20  </body>
21
22  </html>
```

index.html hosted with ♡ by GitHub view raw

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:

```
// actions/types.js

export const GET_TODOS = 'GET_TODOS';
export const GET_TODO = 'GET_TODO';
export const ADD_TODO = 'ADD_TODO';
export const DELETE_TODO = 'DELETE_TODO';
export const EDIT_TODO = 'EDIT_TODO';
```

types.js hosted with ♡ by GitHub view raw

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

```
// actions/todos.js

import axios from 'axios';
import { GET_TODOS } from './types';

// GET TODOS
export const getTodos = () => async dispatch => {
  const res = await axios.get('/api/todos/');
  dispatch({
    type: GET_TODOS,
    payload: res.data
  });
};
```

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:

```
// reducers/todos.js

import _ from 'lodash';
import { GET_TODOS } from '../actions/types';

export default (state = {}, action) => {
  switch (action.type) {
```

```
    case GET_TODOS:
      return {
        ...state,
        ..._.mapKeys(action.payload, 'id')
      };
    default:
      return state;
  }
};
```

todos.js hosted with ♡ by GitHub view raw

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:

```
// reducers/index.js

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

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

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:

```
// fronted/src/store.js

import { createStore, applyMiddleware } from 'redux';
import { composeWithDevTools } from 'redux-devtools-extension';
import reduxThunk from 'redux-thunk';
import rootReducer from './reducers';

const store = createStore(
```

```
  rootReducer,
  composeWithDevTools(applyMiddleware(reduxThunk))
);

export default store;
```

store.js hosted with ♡ by GitHub view raw

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:

```
// components/todos/TodoList.js

import React, { Component } from 'react';
import { connect } from 'react-redux';
import { getTodos } from '../../actions/todos';

class TodoList extends Component {
  componentDidMount() {
    this.props.getTodos();
  }

  render() {
    return (
      <div className='ui relaxed divided list' style={{ marginTop: '2rem' }}>
        {this.props.todos.map(todo => (
          <div className='item' key={todo.id}>
            <i className='large calendar outline middle aligned icon' />
            <div className='content'>
              <a className='header'>{todo.task}</a>
              <div className='description'>{todo.created_at}</div>
            </div>
          </div>
        ))}
      </div>
    );
  }
}

const mapStateToProps = state => ({
  todos: Object.values(state.todos)
});

export default connect(
  mapStateToProps,
```

```
  { getTodos }
)(TodoList);
```

TodoList.js hosted with ♡ by GitHub view raw

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:

```
// components/todos/Dashboard.js

import React, { Component } from 'react';
import TodoList from './TodoList';

class Dashboard extends Component {
  render() {
    return (
      <div className='ui container'>
        <div>Todo Create Form</div>
        <TodoList />
      </div>
    );
  }
}

export default Dashboard;
```

Dashboard.js hosted with ♡ by GitHub view raw

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

```
// components/App.js

import Dashboard from './todos/Dashboard'; // added

import { Provider } from 'react-redux'; // added
import store from '../store'; // added

class App extends Component {
  render() {
    return (
      <Provider store={store}>
        <Dashboard />
```

```
      </Provider>
    );
  }
}
```

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:

```
// actions/todos.js

import { reset } from 'redux-form'; // added
import { GET_TODOS, ADD_TODO } from './types'; // added ADD_TODO

// ADD TODO
export const addTodo = formValues => async dispatch => {
  const res = await axios.post('/api/todos/', { ...formValues });
  dispatch({
    type: ADD_TODO,
    payload: res.data
  });
  dispatch(reset('todoForm'));
};
```

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

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

```
// reducers/todos.js

import { GET_TODOS, ADD_TODO } from '../actions/types'; // added ADD_TODO
```

```
export default (state = {}, action) => {
  switch (action.type) {
    // ...
    case ADD_TODO: // added
      return {
        ...state,
        [action.payload.id]: action.payload
      };
    // ...
  }
};
```

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:

```
// components/todos/TodoForm.js

import React, { Component } from 'react';
import { Field, reduxForm } from 'redux-form';

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

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

  render() {
    return (
      <div className='ui segment'>
        <form
          onSubmit={this.props.handleSubmit(this.onSubmit)}
          className='ui form error'
        >
```

```
          <Field name='task' component={this.renderField} label='Task' />
          <button className='ui primary button'>Add</button>
        </form>
      </div>
    );
  }
}

const validate = formValues => {
  const errors = {};

  if (!formValues.task) {
    errors.task = 'Please enter at least 1 character';
  }

  return errors;
};

export default reduxForm({
  form: 'todoForm',
  touchOnBlur: false,
  validate
})(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 `TodoCreate.js` in the `components/todos` directory:

```
// components/todos/TodoCreate.js

import React, { Component } from 'react';
import { connect } from 'react-redux';
import { addTodo } from '../../actions/todos';
import TodoForm from './TodoForm';

class TodoCreate extends Component {
  onSubmit = formValues => {
    this.props.addTodo(formValues);
  };
```

```
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() {
7       return (
8         <div className='ui container'>
9           <TodoCreate /> // added
10          <TodoList />
11        </div>
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:

```
// components/layout/Header.js

import React, { Component } from 'react';

class Header extends Component {
  render() {
    return (
      <div className='ui inverted menu' style={{ borderRadius: '0' }}>
        <a className='header item'>TodoCRUD</a>
        <a className='item'>Home</a>
      </div>
    );
  }
}

export default Header;
```

Header.js hosted with ♡ by GitHub view raw

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

```
// components/App.js

import Header from './layout/Header'; // added

class App extends Component {
  render() {
    return (
      <Provider store={store}>
        <Header /> // added
        <Dashboard />
      </Provider>
    );
  }
}
```

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:

```
// frontend/src/history.js

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_TODO and DELE
5
6  // GET TODO
7  export const getTodo = id => async dispatch => { // added
8    const res = await axios.get(`/api/todos/${id}/`);
9    dispatch({
10     type: GET_TODO,
11     payload: res.data
12   });
13 };
14
15 // DELETE TODO
16 export const deleteTodo = id => async dispatch => { // added
17   await axios.delete(`/api/todos/${id}/`);
18   dispatch({
19     type: DELETE_TODO,
20     payload: id
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 GET_TODO
```

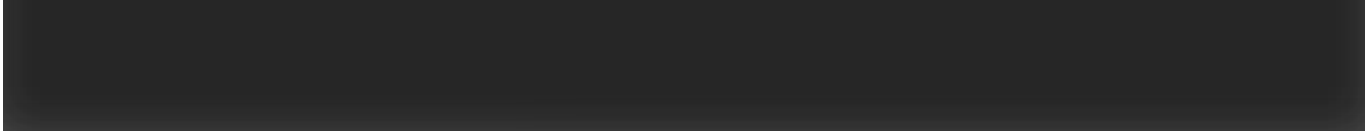
```
export default (state = {}, action) => {
  switch (action.type) {
    // ...
    case GET_TODO: // added
    case ADD_TODO:
      return {
        ...state,
        [action.payload.id]: action.payload
      };
    case DELETE_TODO: // added
      return _.omit(state, action.payload);
    // ...
  }
};
```

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

Let's create the modal window I just mentioned. We will have it that looks like this:



The modal window for confirmation of the deletion

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

```
// components/layout/Modal.js

import React from 'react';
import ReactDOM from 'react-dom';

const Modal = props => {
  return ReactDOM.createPortal(
    <div onClick={props.onDismiss} className='ui active dimmer'>
      <div onClick={e => e.stopPropagation()} className='ui active modal'>
        <div className='header'>{props.title}</div>
        <div className='content'>{props.content}</div>
        <div className='actions'>{props.actions}</div>
      </div>
    </div>,
    document.querySelector('#modal')
  );
};

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:

```
<!-- templates/frontend/index.html -->

<body>
  <div id='app'></div>
  <div id="modal"></div>

  {% load static %}
  <script src="{% static 'frontend/main.js' %}"></script>
</body>
```

index.html hosted with ♡ by GitHub view raw

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

```
// components/todos/TodoDelete.js

import React, { Component, Fragment } from 'react';
import { connect } from 'react-redux';
import { Link } from 'react-router-dom';
import Modal from '../layout/Modal';
import history from '../../history';
import { getTodo, deleteTodo } from '../../actions/todos';

class TodoDelete extends Component {
  componentDidMount() {
    this.props.getTodo(this.props.match.params.id);
  }

  renderContent() {
    if (!this.props.todo) {
      return 'Are you sure you want to delete this task?';
    }
    return `Are you sure you want to delete the task: ${this.props.todo.task}`;
  }

  renderActions() {
    const { id } = this.props.match.params;
    return (
      <Fragment>
        <button
          onClick={() => this.props.deleteTodo(id)}
          className='ui negative button'
        >
          Delete
        </button>
        <Link to='/' className='ui button'>
          Cancel
        </Link>
      </Fragment>
    );
  }

  render() {
    return (
      <Modal
        title='Delete Todo'
        content={this.renderContent()}
        actions={this.renderActions()}
        onDismiss={() => history.push('/')}
      />
    );
```

```
  }
}

const mapStateToProps = (state, ownProps) => ({
  todo: state.todos[ownProps.match.params.id]
});

export default connect(
  mapStateToProps,
  { getTodo, deleteTodo }
)(TodoDelete);
```

TodoDelete.js hosted with ♡ by GitHub view raw

The code is a bit long, but it is not so difficult. Define the helper functions that display the content and the action buttons on the modal window. Then, pass them as Props to 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:

```
// components/todos/TodoList.js

import { Link } from 'react-router-dom'; // added
import { getTodos, deleteTodo } from '../../actions/todos'; // added deleteTodo

class TodoList extends Component {
  // ...

  render() {
    return (
      <div className='ui relaxed divided list' style={{ marginTop: '2rem' }}>
        {this.props.todos.map(todo => (
          <div className='item' key={todo.id}>
            <div className='right floated content'> // added
              <Link
                to={`/delete/${todo.id}`}
                className='small ui negative basic button'
              >
                Delete
              </Link>
            </div>
            <i className='large calendar outline middle aligned icon' />
            <div className='content'>
              <a className='header'>{todo.task}</a>
              <div className='description'>{todo.created_at}</div>
```

```
            </div>
          </div>
        ))}
      </div>
    );
  }
}

// ...

export default connect(
  mapStateToProps,
  { getTodos, deleteTodo } // added deleteTodo
)(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:

```
// components/App.js

import { Router, Route, Switch } from 'react-router-dom'; // added

import history from '../history'; // added
import TodoDelete from './todos/TodoDelete'; // added

class App extends Component {
  render() {
    return (
      <Provider store={store}>
        <Router history={history}>
          <Header />
          <Switch>
            <Route exact path='/' component={Dashboard} />
            <Route exact path='/delete/:id' component={TodoDelete} />
          </Switch>
        </Router>
      </Provider>
    );
  }
}
```

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.

That concludes this chapter. Try deleting some objects and see if it works.

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:

```
// actions/todos.js

import { GET_TODOS, GET_TODO, ADD_TODO, DELETE_TODO, EDIT_TODO } from './types'; // added EDIT_T

// EDIT TODO
export const editTodo = (id, formValues) => async dispatch => {
  const res = await axios.patch(`/api/todos/${id}/`, formValues);
  dispatch({
    type: EDIT_TODO,
    payload: res.data
  });
  history.push('/');
};
```

todos.js hosted with ♡ by GitHub view raw

Reducers

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

```
// reducers/todos.js

import {
  GET_TODOS,
  GET_TODO,
  ADD_TODO,
  DELETE_TODO,
  EDIT_TODO // added
} from '../actions/types';

export default (state = {}, action) => {
```

```
  switch (action.type) {
    // ...
    case GET_TODO:
    case ADD_TODO:
    case EDIT_TODO: // added
      return {
        ...state,
        [action.payload.id]: action.payload
      };
    // ...
  }
};
```

todos.js hosted with ♡ by GitHub view raw

Components

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

```
// components/todos/TodoEdit.js

import _ from 'lodash';
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { getTodo, editTodo } from '../../actions/todos';
import TodoForm from './TodoForm';

class TodoEdit extends Component {
  componentDidMount() {
    this.props.getTodo(this.props.match.params.id);
  }

  onSubmit = formValues => {
    this.props.editTodo(this.props.match.params.id, formValues);
  };

  render() {
    return (
      <div className='ui container'>
        <h2 style={{ marginTop: '2rem' }}>Edit Todo</h2>
        <TodoForm
          initialValues={_.pick(this.props.todo, 'task')}
          enableReinitialize={true}
          onSubmit={this.onSubmit}
        />
      </div>
    );
  }
}

const mapStateToProps = (state, ownProps) => ({
```

```
  todo: state.todos[ownProps.match.params.id]
});

export default connect(
  mapStateToProps,
  { getTodo, editTodo }
)(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 `<a className='header'>{todo.task}</a>` as follows:

```
// components/todos/TodoList.js

<Link to={`/edit/${todo.id}`} className='header'>
  {todo.task}
</Link>
```

TodoList.js hosted with ♡ by GitHub view raw

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

```
// components/App.js

import TodoEdit from './todos/TodoEdit'; // added

class App extends Component {
  render() {
    return (
      <Provider store={store}>
        <Router history={history}>
          <Header />
          <Switch>
            <Route exact path='/' component={Dashboard} />
            <Route exact path='/delete/:id' component={TodoDelete} />
            <Route exact path='/edit/:id' component={TodoEdit} /> // added
          </Switch>
        </Router>
      </Provider>
    );
  }
}
```

App.js hosted with ♡ by GitHub view raw

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:

```
// components/todos/TodoForm.js

class TodoForm extends Component {
  // ...

  render() {
    const btnText = `${this.props.initialValues ? 'Update' : 'Add'}`; // added
    return (
      <div className='ui segment'>
        <form
          onSubmit={this.props.handleSubmit(this.onSubmit)}
          className='ui form error'
        >
          <Field name='task' component={this.renderField} label='Task' />
          <button className='ui primary button'>{btnText}</button> // updated
        </form>
      </div>
    );
  }
}
```

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.

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

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

Programming Coding Django React Redux