Django Rest Framework with React Tutorial

Jan 16, 2018

In this tutorial we will build a *Todo List app* with a Django Rest Framework backend and a React frontend. If you've already used Django before then you'll be amazed at how little code it requires to transform a Django app into a REST API that can be consumed by a frontend.

In this tutorial I assume you have basic familiarity with Django. If not, I've written an entire book, Django For Beginners, that walks you through building multiple apps with Django. It also covers proper local development configuration with Python 3 and pipenv.

Complete source code can be found here on Github.

Setup

We start by building a standard Django app:

- install Django with `pipenv`

- create a new project
- create a new app
- configure our local database

In a new command line console, enter the following commands. You don't have to place your code on the `Desktop`–it can live anywhere on your computer–but you do need to place it in a dedicated directory.

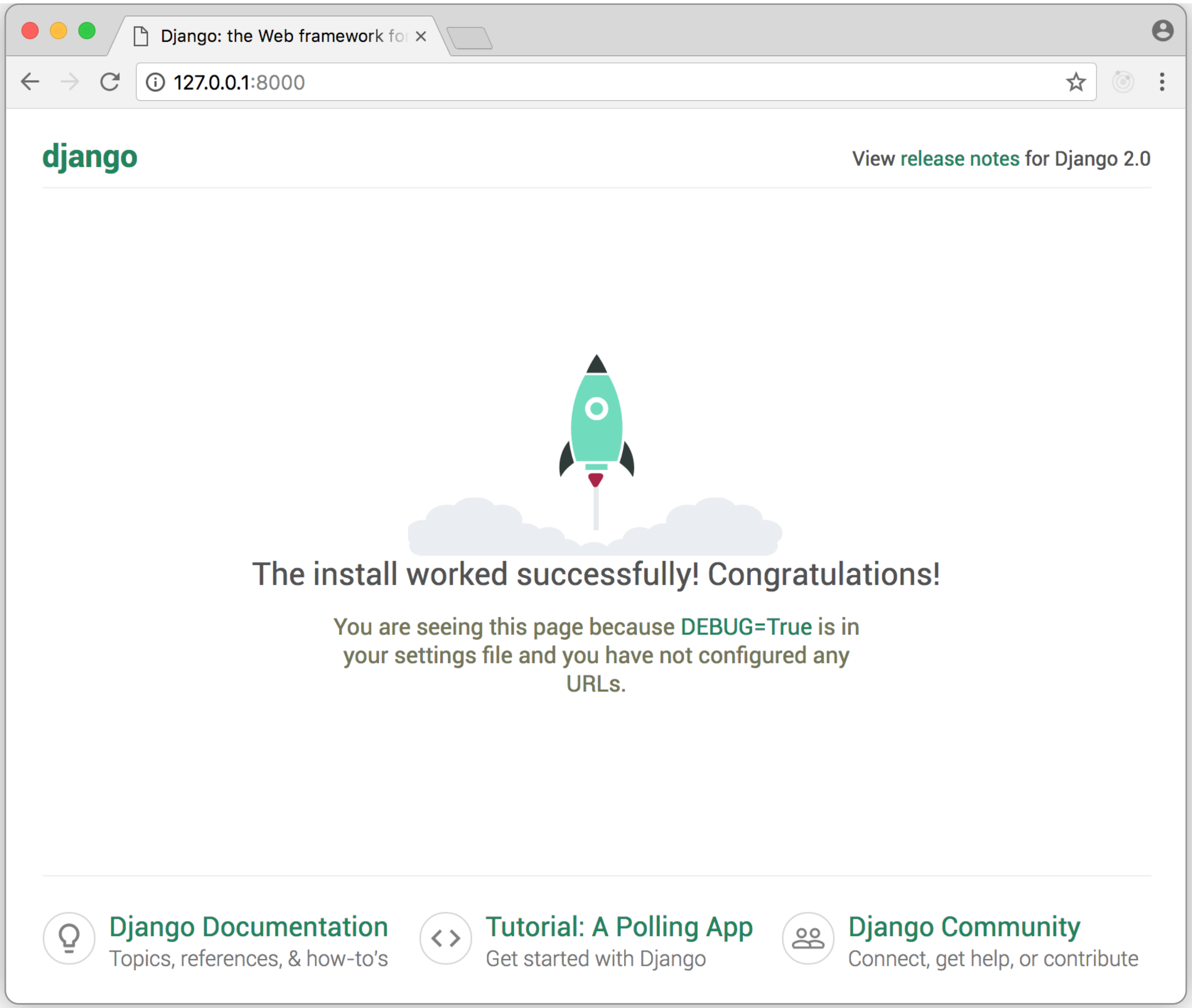
```
$ cd ~/Desktop
$ mkdir todo && cd todo
$ mkdir backend && cd backend
$ pipenv install django
$ pipenv shell
(backend) $ django-admin startproject todo_api .
(backend) $ python manage.py startapp todos
(backend) $ python manage.py migrate
```

Note that the actual virtual environment name will be `backend-XXX` where the `XXX` will be different for each of us. I've shortened this to `backend` here throughout the tutorial.

Next add our `todos` app to the `INSTALLED_APPS` settings in our `settings.py` file.

```python
# todo_api/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'todos', # new line
]
```

If you run `python manage.py runserver` you'll see our project is successfully installed and ready to use.


Django: the Web framework fo ×

127.0.0.1:8000

django

View release notes for Django 2.0

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Django Documentation
Topics, references, & how-to's

Tutorial: A Polling App
Get started with Django

Django Community
Connect, get help, or contribute

Models

Our *todo list* database model is deliberately quite basic and has only two fields: `title` and `description`.

```python
# todos/models.py
from django.db import models


class Todo(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()

    def __str__(self):
        """A string representation of the model."""
        return self.title
```

We have updated our model so it's time for Django's two-step dance of making a new migrations file and then migrating the database with it. Type `Control-c` to stop our local server. Then run these two commands:

```
(backend) $ python manage.py makemigrations todos
(backend) $ python manage.py migrate todos
```

We can use the built-in Django *admin* app to interact with our data but first we need to add our `Todo` model to it.

```
# todos/admin.py
from django.contrib import admin

from .models import Todo

admin.site.register(Todo)
```

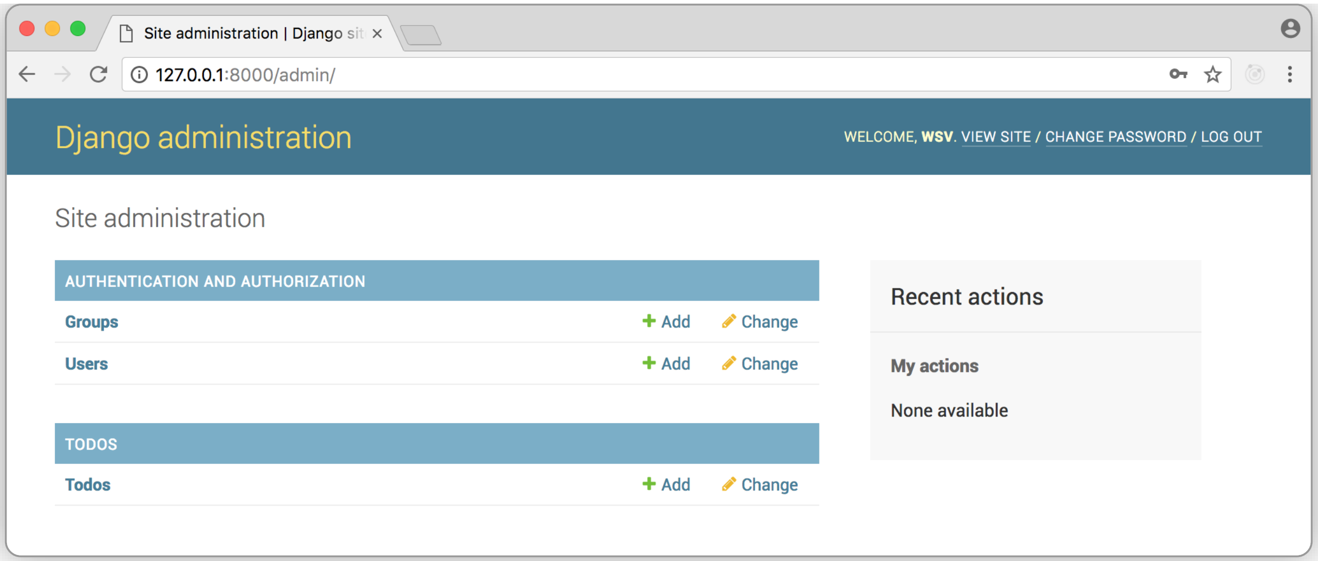
Now create a `superuser` account so we can login to the admin.

```
(backend) $ python manage.py createsuperuser
```

And then start up the local server again:

```
(backend) $ python manage.py runserver
```

If you navigate to http://127.0.0.1:8000/admin/ you can now login.


Site administration | Django sit
127.0.0.1:8000/admin/
Django administration
WELCOME, WSV. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

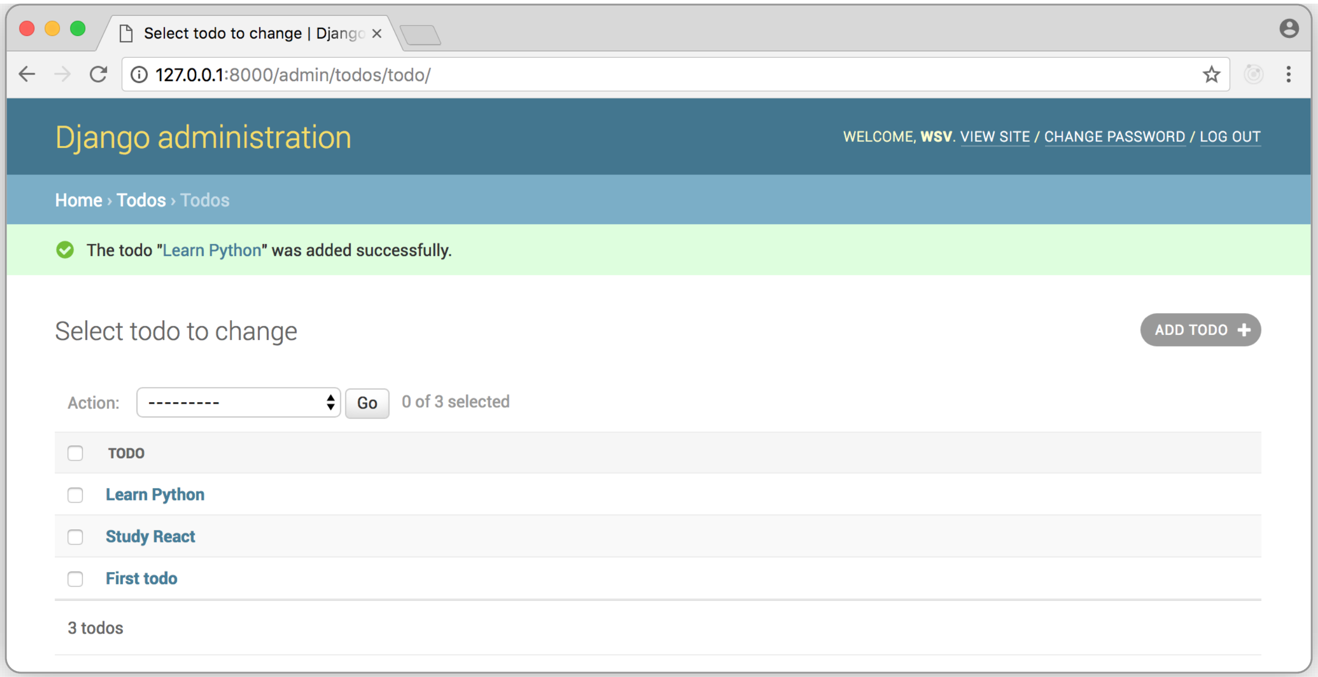
TODOS		
Todos	+ Add	Change

Recent actions

My actions

None available

Click on `Todos` and create 3 new todo items, making sure to add a title and description for both. Here's what mine look like:


Select todo to change | Django

127.0.0.1:8000/admin/todos/todo/

Django administration

WELCOME, WSV. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Todos › Todos

The todo "Learn Python" was added successfully.

Select todo to change

ADD TODO

Action: --------- Go 0 of 3 selected

TODO
Learn Python
Study React
First todo

3 todos

Note: You could also use the `django shell` to add content to our database, but in my experience teaching the graphical nature of the admin app is easier for beginners to understand.

We've done quite a bit of work in a very small amount of code. Our new project is created, we added and configured a `todos` app, updated our database models, and used the admin to populate it.

Tests

Time for some tests before we proceed further. In Django for Beginners I cover testing in-depth. Here I'll just give you the code to use which simply tests that we can add content to our `title` and `description` fields.

```
# todos/tests.py
from django.test import TestCase
from .models import Todo


class TodoModelTest(TestCase):

    @classmethod
    def setUpTestData(cls):
        Todo.objects.create(title='first todo')
        Todo.objects.create(description='a description here')

    def test_title_content(self):
        todo = Todo.objects.get(id=1)
        expected_object_name = f'{todo.title}'
        self.assertEquals(expected_object_name, 'first todo')

    def test_description_content(self):
        todo = Todo.objects.get(id=2)
        expected_object_name = f'{todo.description}'
        self.assertEquals(expected_object_name, 'a description her
```

To confirm that our tests are working quit the local server `Control+c` and run `python manage.py test`.

```
(backend) $ python manage.py test
```

At this point in a traditional Django project we would start adding `urls.py` files, views, and templates for our app. **But we won't do that here!** We're simply creating an API that sends this data "over the wire" so someone else can consume it. Time for Django Rest Framework!

Django Rest Framework

The first step is to install Django Rest Framework (DRF) with `pipenv`.

```
(backend) $ pipenv install djangorestframework
```

We need to make two updates to our `settings.py` file to configure DRF.

The first is to add `rest_framework` to our `INSTALLED_APPS`. Then at the bottom of the file we'll start adding configurations for DRF. In a real-world project it's very important to have proper permissions on your API. But to keep things simple we'll allow anyone to make changes to the API since it will only be consumed locally.

```
# todo_api/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'rest_framework',

    'todos',
]

REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny',
    ]
}
```

It's time to decide how we want our API URLs to look. We'll have it so our endpoint to list all todos is `api/` and a detail view which returns all information of a single todo object will be at `api/<id>` where `<id>` represents the internal id/primary key Django automatically sets for us. Therefore the URL route for the first todo item should be `/api/1`, the second will be `/api/2`, and so on.

Start by updating our project-level `urls.py` file. We just want it to refer to the `todos` app at the subdomain `api/`.

```
# todo_api/urls.py
from django.contrib import admin
from django.urls import include, path


urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('todos.urls')),
]
```

Then create our app-level `urls.py` file.

```
(backend) $ touch todos/urls.py
```

And update it with the code below.

```
# todos/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.ListTodo.as_view()),
    path('<int:pk>/', views.DetailTodo.as_view()),
]
```

Note that we haven't created the two views referenced here: `ListTodo` and `DetailTodo`. That's up next!

Serializers

Let's review where we are so far. We have a project and app, a database model, and a url scheme. We want our URL endpoints to return data in a JSON format, which means we need a serializer.

If you're new to APIs, this is the part where confusion strikes. What, what's a serializer? The key point is that when all our data was contained within a Django project–frontend and backend–we could just pass information around without really thinking about it. But with our API, **all we output** is the data.

The serializer translates our data into a format, JSON, that can efficiently be sent over the internet and then consumed by another party, which is our frontend. But we could also expose an “external” API that other developers could access too, in addition to this “internal” API we’re building for ourselves. It’s all about the permissions we set.

To make our first serializer, create a new `serializers.py` file in our `todos` app.

```
(backend) $ touch todos/serializers.py
```

Then update it with the following code.

```python
# todos/serializers.py
from rest_framework import serializers
from .models import Todo


class TodoSerializer(serializers.ModelSerializer):
    class Meta:
        fields = (
            'id',
            'title',
            'description',
        )
        model = Todo
```

We import `serializers` from DRF as well as our `models.py` file. Next we create a class `TodoSerializer`. The format here is **very** similar to how we create model classes. We're specifying which model to use and the specific fields on it we want to expose. Remember that `id` is created automatically by Django so we didn't have to define it in our `Todo` model but we will use it in our detail view.

Views

In regular Django **views** are used to customize what data to send to the templates. In DRF our views do the same thing for our serialized data.

The syntax of DRF views is **very similar** to regular Django views. They also have generic views for common use cases.

We'll import DRFs generic views at the top of the file. Then import both our `models.py` and `serializers.py` files.

Our two views will *list* all our todos and provide a *detail* view of a single todo object. DRF has the generic views `ListCreateAPIView` and `RetrieveUpdateDestroyAPIView` just for this use case.

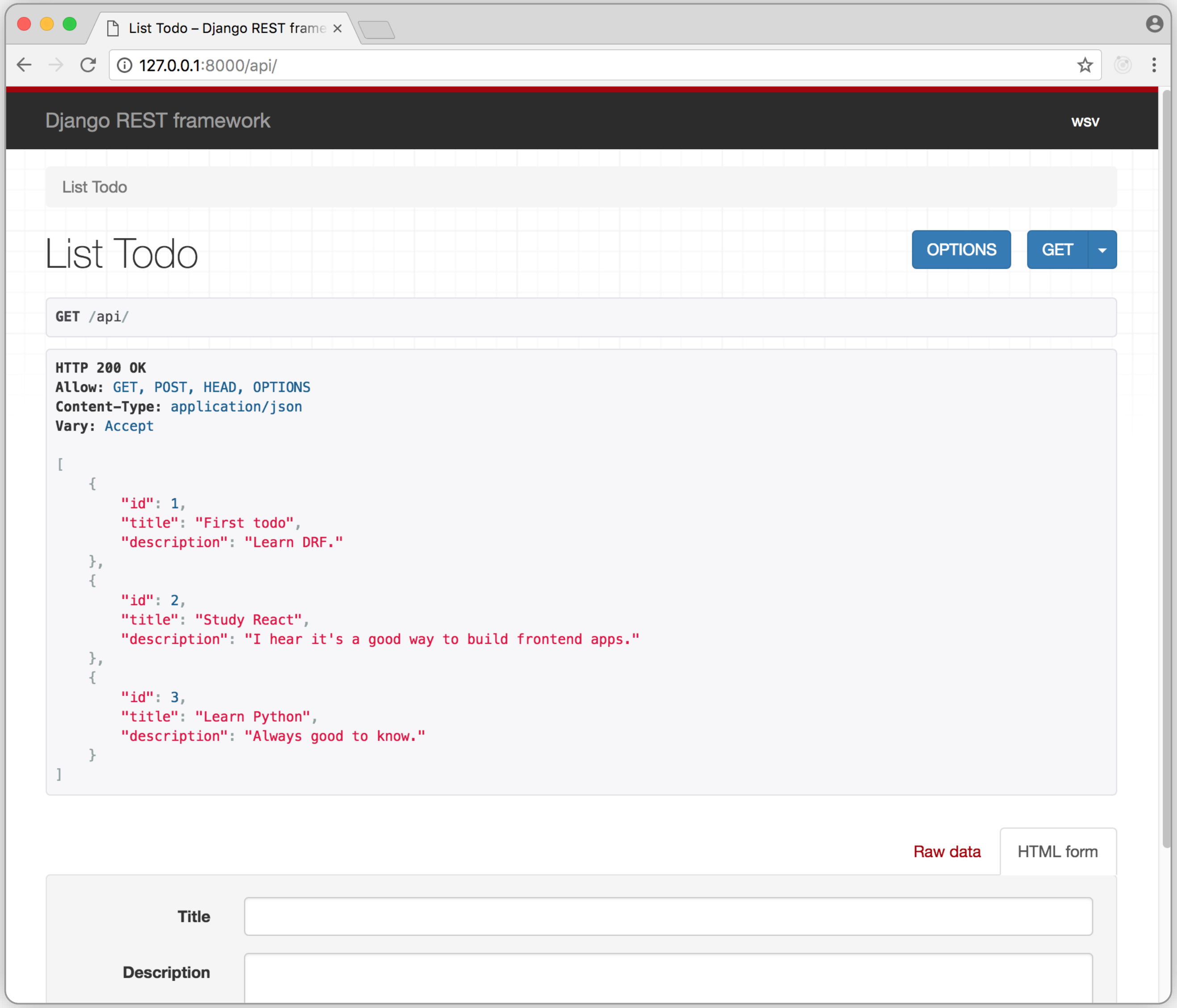
```
# todos/views.py
from rest_framework import generics

from .models import Todo
from .serializers import TodoSerializer


class ListTodo(generics.ListCreateAPIView):
    queryset = Todo.objects.all()
    serializer_class = TodoSerializer


class DetailTodo(generics.RetrieveUpdateDestroyAPIView):
    queryset = Todo.objects.all()
    serializer_class = TodoSerializer
```

And we're done! Our API is ready to consume.

Browsable API

At this point we could use cURL to try out our API or a 3rd party tool like Postman, but DRF isn't done yet with its cool features. It also features a browsable API that we can view and interact with.

Run the local server with `python manage.py runserver` and navigate to http://127.0.0.1:8000/api/ to see our list views. This contains the actual JSON available at this endpoint.



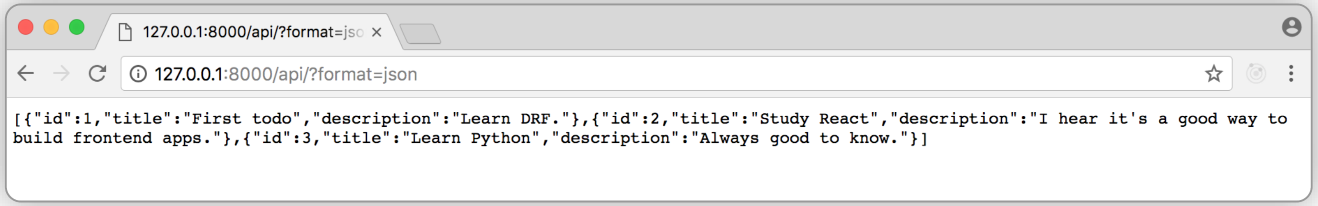
List Todo

```
GET /api/
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "title": "First todo",
        "description": "Learn DRF."
    },
    {
        "id": 2,
        "title": "Study React",
        "description": "I hear it's a good way to build frontend apps."
    },
    {
        "id": 3,
        "title": "Learn Python",
        "description": "Always good to know."
    }
]
```

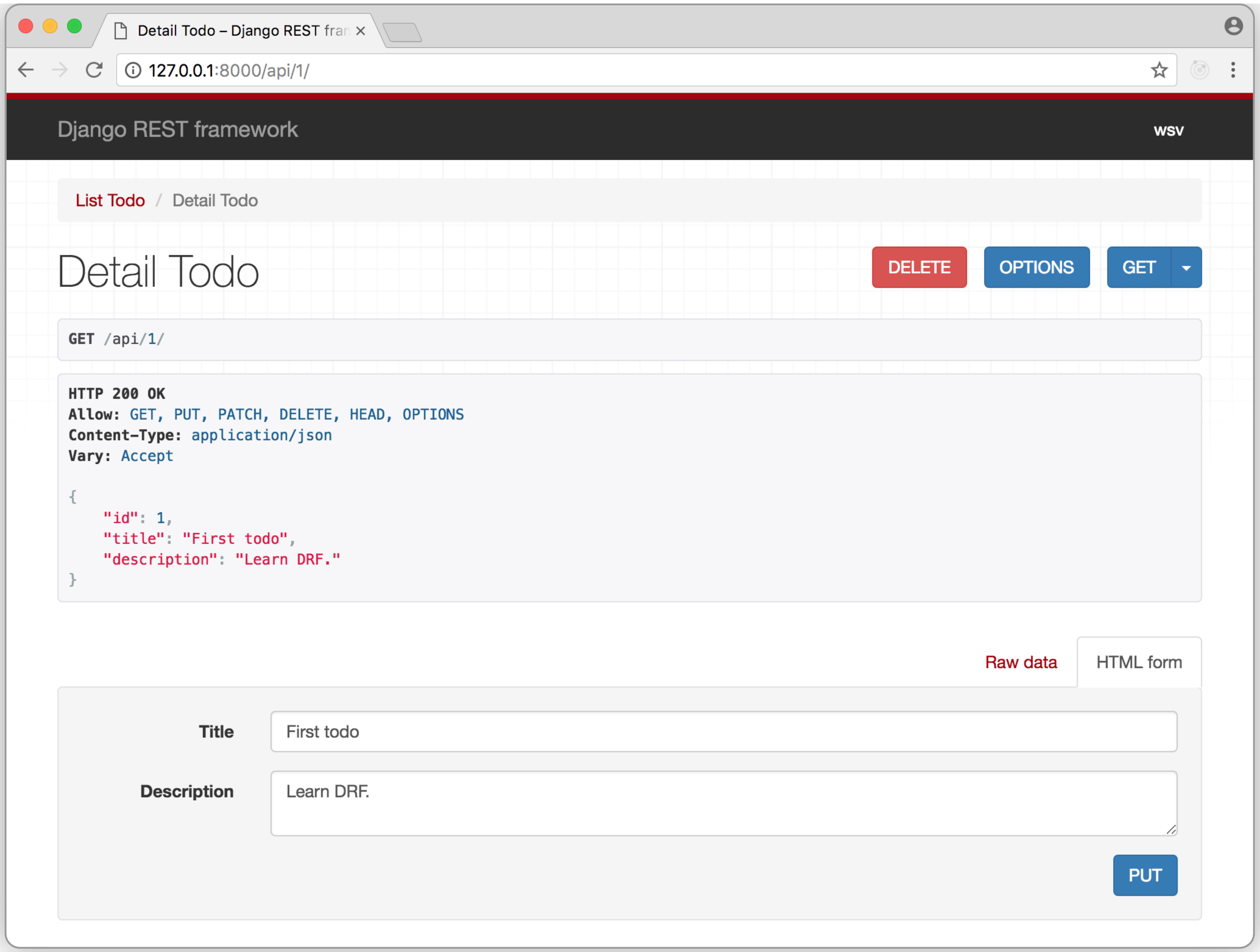
You can even use the form at the bottom to add new content.

To see the raw JSON rather than this view, click on the "GET" button in the upper right corner and select `JSON`.


127.0.0.1:8000/api/?format=jsc
127.0.0.1:8000/api/?format=json

```
[{"id":1,"title":"First todo","description":"Learn DRF."},{"id":2,"title":"Study React","description":"I hear it's a good way to
build frontend apps."},{"id":3,"title":"Learn Python","description":"Always good to know."}]
```

We can also see a detail views and edit the contents with the form at the bottom. For example, our first post is at http://127.0.0.1:8000/api/1/.



```
GET /api/1/
```

```
HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 1,
    "title": "First todo",
    "description": "Learn DRF."
}
```

Title	First todo
Description	Learn DRF.

CORS

There's one last step we need to do and that's deal with Cross-Origin Resource Sharing (CORS). Whenever a client interacts with an API hosted on a **different domain** there are potential security issues. CORS requires the server to include specific HTTP headers that allow the browser to determine if and when cross-domain requests should be allowed.

The easiest way to handle this–and the one recommended by DRF–is to use middleware that will automatically include the appropriate HTTP headers based on our settings.

The recommended package is django-cors-headers which can be easily added to our existing project.

First quit our server `Control+c` and then install `django-cors-headers` with Pipenv.

```
(backend) $ pipenv install django-cors-headers
```

Then update our `settings.py` file in three places:

- add `corsheaders` to the `INSTALLED_APPS`
- add two new middlewares that need to appear at the top!
- create a `CORS_ORIGIN_WHITELIST`

```
# todo_api/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'rest_framework',
    'corsheaders', # new

    'todos',
]

MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware', # new
    'django.middleware.common.CommonMiddleware', # new
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

CORS_ORIGIN_WHITELIST = (
    'localhost:3000/'
)
```

Note: We're using the domain `localhost:3000/` because that's the default local port for React.

And that's it! Now our backend is complete. Make sure the server is running.

```
(backend) $ python manage.py runserver
```

Now it's time to build our frontend in React.

Front end

We're using React in this tutorial but our backend doesn't care what frontend framework is used to consume our *Todo list* API. We could just as easily choose Vue, Angular, or something else.

The fastest way to build a new React app is to use the excellent create-react-app which does not require any build configuration. It just works.

Open up a new command line console so there are now **two consoles open**. Leave our existing backend open and still running our local server for our DRF API.

In the new console install `create-react-app` globally with the following command.

```
$ npm install -g create-react-app
```

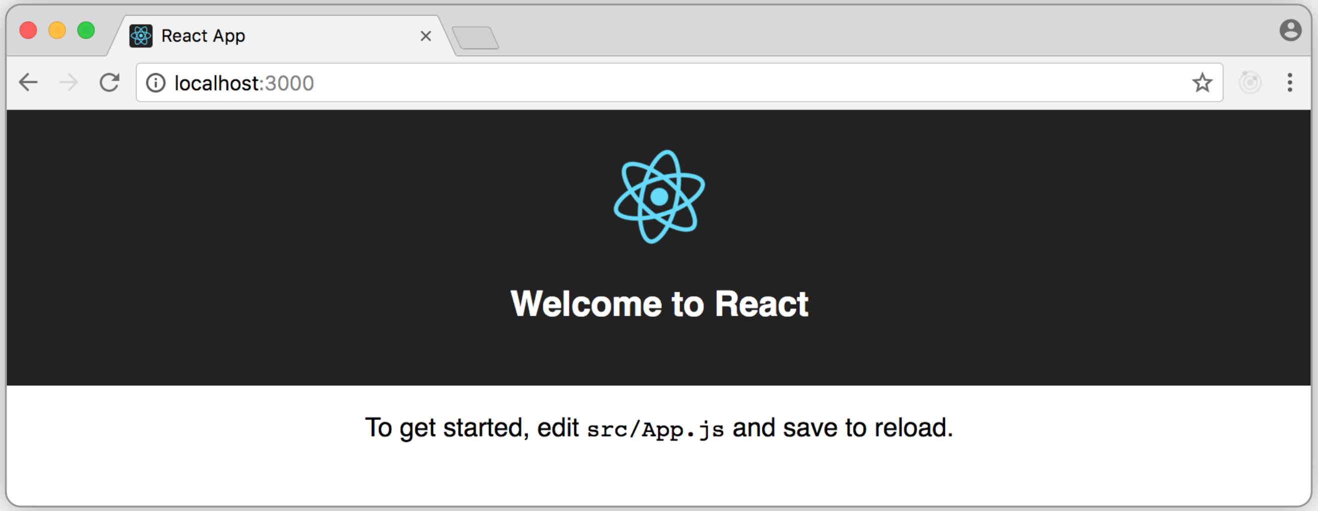
Make sure we're in the correct directory by navigating to the Desktop and then creating a `frontend` directory.

```
$ cd ~/Desktop
$ create-react-app frontend
$ cd frontend
```

We can now run our React app with the command `npm start`.

```
$ npm start
```

And navigate to http://localhost:3000/.



To get started, edit `src/App.js` and save to reload.

Isn’t that cool?

Since this tutorial is not about React I’m just going to give you the necessary code. We only need to update the `App.js` file.

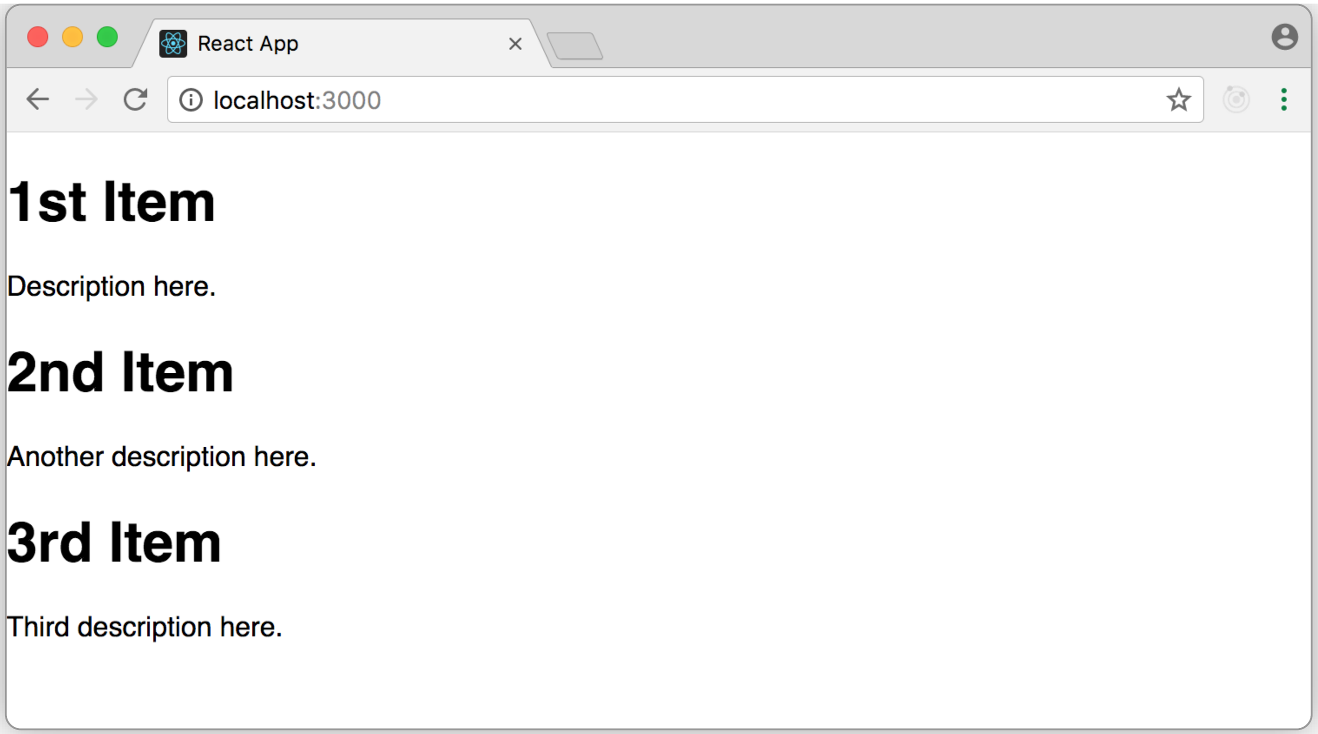
Our `/api` endpoint is in JSON. Mine looks like this:

```
[
  {
    "id":1,
    "title":"First todo",
    "description":"Learn DRF."
  },
  {
    "id":2,
    "title":"Study React",
    "description":"I hear it's a good way to build frontend apps."
  },
  {
    "id":3,
    "title":"Learn Python",
    "description":"Always good to know."
  }
]
```

We can mock that up in our React app in a variable `list`, load that list into our state, and then use `map()` to display all the items. Here’s the code.

```
// App.js
import React, { Component } from 'react';

const list = [
  {
    'id': 1,
    'title': '1st Item',
    'description': 'Description here.'
  },
  {
    'id': 2,
    'title': '2nd Item',
    'description': 'Another description here.'
  },
  {
    'id': 3,
    'title': '3rd Item',
    'description': 'Third description here.'
  }
];

class App extends Component {
  constructor(props) {
    super(props);
    super(props);
    this.state = { list };
  }

  render() {
    return (
      <div>
        {this.state.list.map(item => (
          <div>
            <h1>{item.title}</h1>
            <span>{item.description}</span>
          </div>
        ))}
      </div>
    );
  }
}

export default App;
```

You should now see our todos listed out on the homepage at http://localhost:3000/.


React App

localhost:3000

1st Item

Description here.

2nd Item

Another description here.

3rd Item

Third description here.

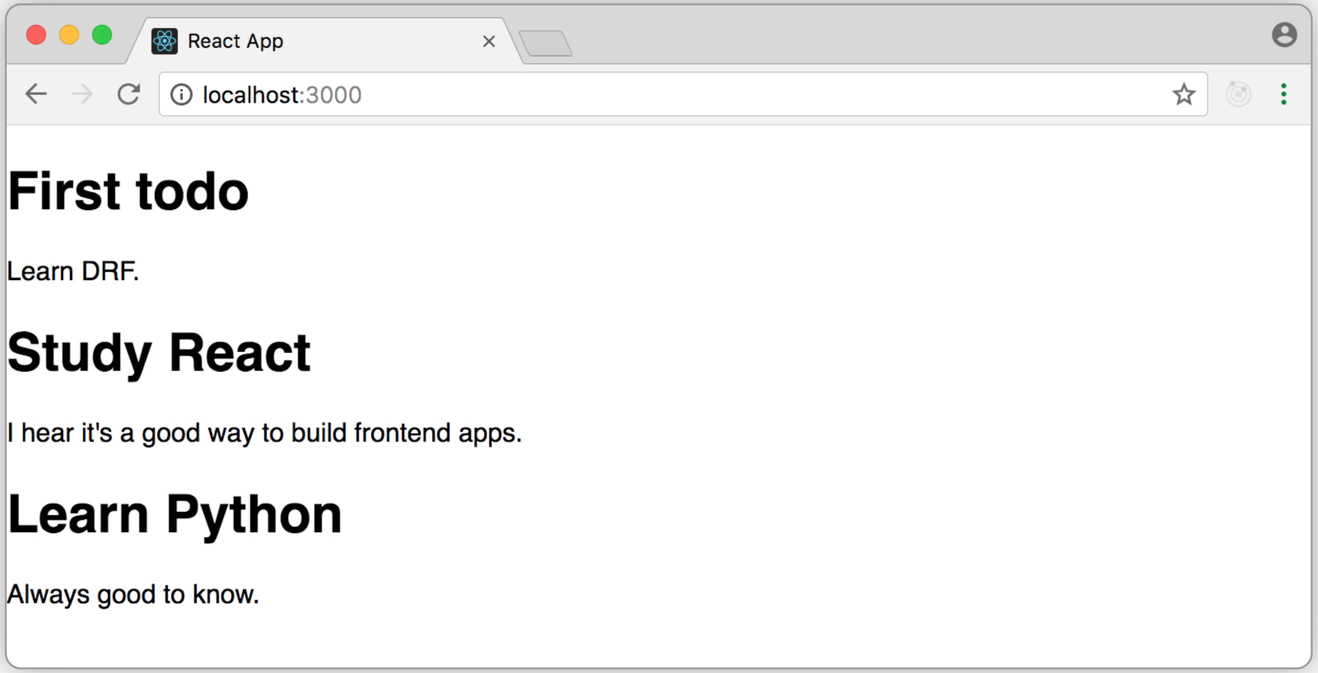
DRF + React

Now let's hook into our DRF API for real. Our endpoints are at the local host of `http://127.0.0.1:8000/`.

In our `App.js` file we'll load this JSON data into our *state* before we render any components. We can do this with `componentDidMount`. Even cooler, we can use `async/await` so it's done asynchronously. Then we output it as before, this time only showing the `title` and not the `description` from our DRF backend.

```
// App.js
// App.js
import React, { Component } from 'react';

class App extends Component {
  state = {
    todos: []
  };

  async componentDidMount() {
    try {
      const res = await fetch('http://127.0.0.1:8000/api/');
      const todos = await res.json();
      this.setState({
        todos
      });
    } catch (e) {
      console.log(e);
    }
  }

  render() {
    return (
      <div>
        {this.state.todos.map(item => (
          <div key={item.id}>
            <h1>{item.title}</h1>
            <span>{item.description}</span>
          </div>
          ))}
        </div>
      );
    }
  }

export default App;
```

If you look again at http://localhost:3000/ React has automatically updated the page for us.



First todo

Learn DRF.

Study React

I hear it's a good way to build frontend apps.

Learn Python

Always good to know.

There's much more we can do to improve our React app. For example, if we wanted to have links to detail pages we could use React Router but that's beyond the scope of this tutorial.

Conclusion

Django React Framework allows us to transform Django apps into robust APIs that can be consumed by frontend frameworks like React. It doesn't take much code to do this and both Django and Django React Framework are very well documented.

Want to learn how to build RESTful APIs with Python and Django? Check out my book REST APIs with Django.

Subscribe for more articles on Python & JavaScript

Your Email

Sign Up

will@wsvincent.com