# Beginning Django API & React

Build Django 4 Web APIs with React Full Stack Applications


Greg Lim, Daniel Correa

# Beginning Django API with React

Greg Lim - Daniel Correa





First Edition: February 2022

Co-Author: Daniel Correa

# Table of Contents

# Preface

## About this book

In this book, we take you on a fun, hands-on and pragmatic journey to learning Django API React stack development. You'll start building your first Django API React stack app within minutes. Every chapter is written in a bite-sized manner and straight to the point as we don’t want to waste your time (and most certainly ours) on the content you don't need. In the end, you will have the skills to create a Todo app and deploy it to the Internet.

In the course of this book, we will cover:

- Chapter 1: Introduction
- Chapter 2: Installing Python and Django
- Chapter 3: Understanding the Project Structure
- Chapter 4: Creating Our Backend Server
- Chapter 5: Serialization
- Chapter 6: URLs and Class-based Views
- Chapter 7: Creating Todos via the ListCreateAPIView
- Chapter 8: Permissions
- Chapter 9: Other C.R.U.D. Operations
- Chapter 10: Completing a Todo
- Chapter 11: Authentication – Sign Up
- Chapter 12: Authentication – Log In Tokens
- Chapter 13: Introduction to React
- Chapter 14: Create Navigation Header Bar
- Chapter 15: Defining Our Routes
- Chapter 16: TodoDataService - Connecting to the Backend
- Chapter 17: TodoDataService - Login Component
- Chapter 18: TodosList Component
- Chapter 19: Adding and Editing Todos
- Chapter 20: Deleting a Todo
- Chapter 21: Completing a Todo
- Chapter 22: Deployment
- Chapter 23: Hosting and Deploying our React Frontend

The goal of this book is to teach you Django API React stack development in a manageable way without overwhelming you. We focus only on the essentials and cover the material in a hands-on practice manner for you to code along.

## Working Through This Book

This book is purposely broken down into short chapters where the development process of each chapter will center on different essential topics. The book takes a practical hands on approach to learning through practice. You learn best when you code along with the examples in the book.

## Requirements

No previous knowledge on Django or React development is required, but you should have basic programming knowledge. It will be a helpful advantage if you could read through my Beginning_Django and React book first which will provide you will better insight and deeper knowledge into the various technologies. But even if you have not done so, you should still be able to follow along.

## Getting Book Updates and Source Code

To receive updated versions of the book and the source code, send a mail to support@i-ducate.com. I try to update my books to use the latest version of software, libraries and will update the codes/content in this book.

# Chapter 1: Introduction

Welcome to Beginning Django API with React! This book focuses on the key tasks and concepts to get you started to learn and build a RESTful web API with Django and Django REST Framework, one of the most popular and customizable ways to build web APIs. In the second part of the book, we then show how to create a frontend using React to connect to the API. In all, the book is designed for readers who don ’ t need all the details about Django or React at this point in the learning curve but concentrate on what you really need to know.

If you are brand new to Django, I recommend first reading my book, Beginning Django (contact support@i-ducate.com) where we learn about the basics of models, URLs, views, authentication, deployment and more.

### *Why create an API?*

With Django, we can already create a complete web application. So why bother to create an API with Django and then create separate frontends when we can do it all with Django? Now, say your app becomes a hit. Users love your app so much that they want an iOS and Android version of it. So, you have to create the same functionality of your app in two more different languages (Swift and Kotlin for example).

You might also hear of new frontend frameworks like React, Angular, Vue to make your app more dynamic and modern. You want to try them out.

How can we better achieve both objectives? The answer is to create a REST API. With Django REST Framework, we can create a common API to serve information to different frontends like Vue, React, Android, iOS etc. We need only to create the logic in the API and then have each of the frontends connect to it. This makes it more scalable and reliable since we just have one code base and database to serve the information.

### *Django Rest Framework*

The Django Rest Framework (DRF) library enables us to easily create Django APIs that return JSON. DRF takes care of the heavy lifting of transforming database models into a RESTful API. For example, it provides CRUD operations, authentication modules, JSON serializers, routing, and many more. It purposefully mimics many of Django ’ s traditional conventions making it much faster to learn. With a minimal amount of code, DRF can transform any existing Django application into a web API. These will be illustrated as we build our sample application throughout this book.

## The App We Will Be Building

We will build a Todos app which lets users log in, create, view, edit and complete todos (fig. 1a, 1b).

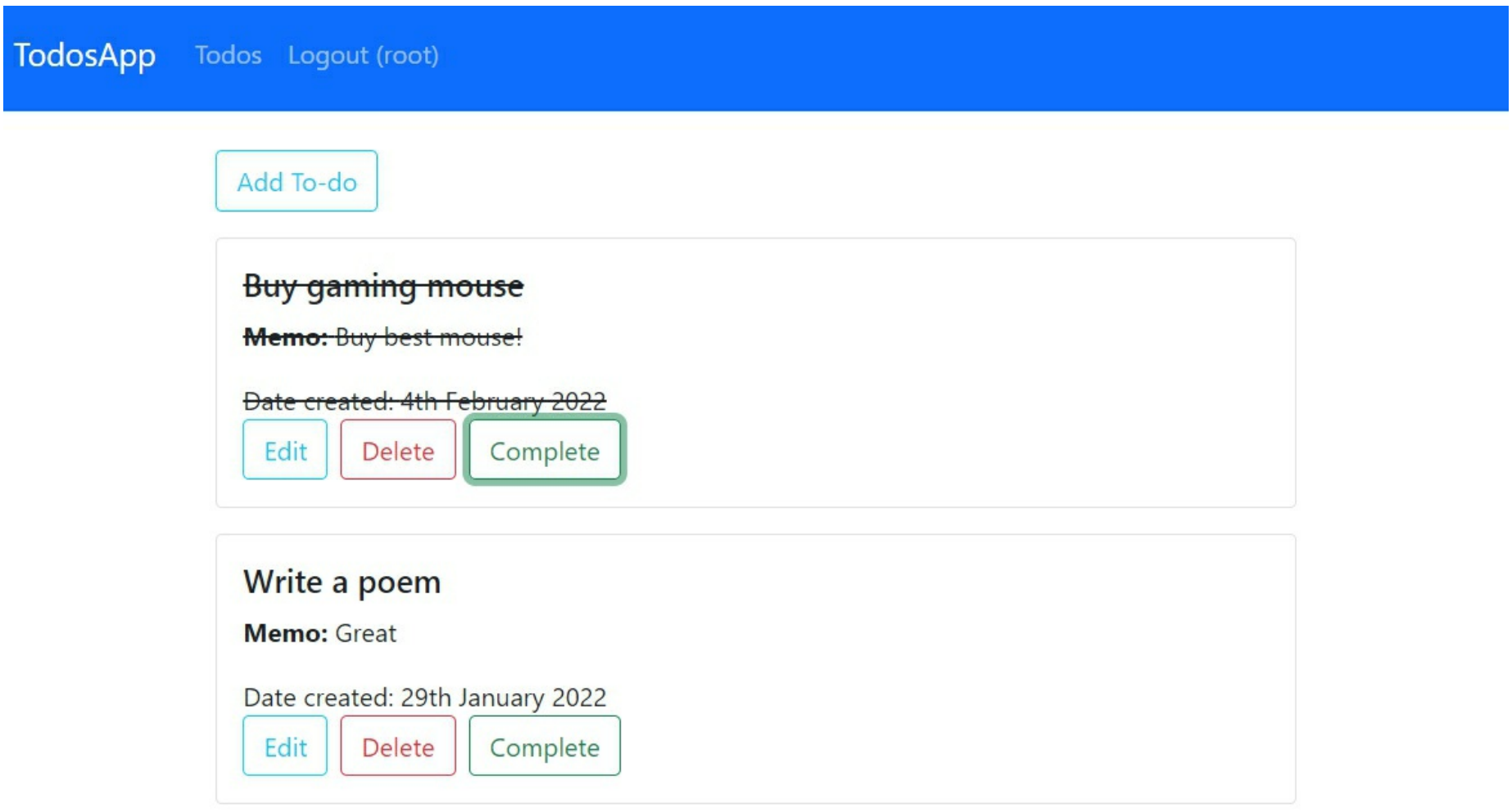


Figure 1a – Home Page listing todos

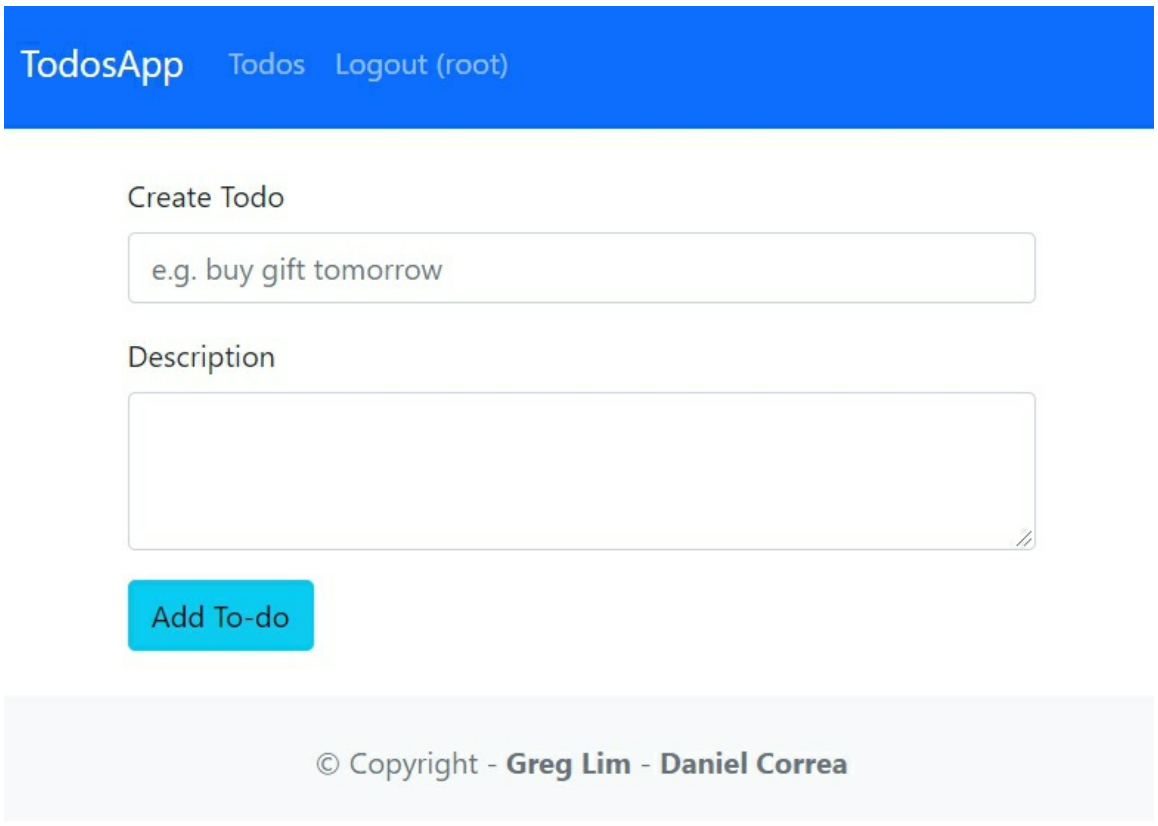

Figure 1b – Create Todo

Users can see the list of todos in the home page and post/edit/delete their own todos if they are logged in. They can also mark their todos as ‘ complete ’ . They will not be able to view other user ’ s todos. Through this app, we will learn a lot of concepts and solidify our Django API and React knowledge.

We will first create the backend of the app using Django and Django Rest Framework (DRF). After that, we will create the frontend with React and connect the frontend to the backend to complete our Django API – React app. In the last chapter, we will deploy our Django backend on PythonAnywhere, and React frontend on Netlify, to run both backend and frontend in the cloud.

So, the overall structure of our app will be:

- the Django runs the backend server and exposes an API. Hosted on PythonAnywhere.
- the React frontend calls the API and renders the user interface on the client ’ s browser. Hosted on Netlify.

In the next chapter, let ’ s begin by installing Python and Django on our machine.

# Chapter 2: Installing Python and Django

Because Django is a Python web framework, we first have to install Python.

## *Installing Python*

Let ’ s check if we have Python installed and what version it is.

If you are using a Mac, open your Terminal. If you are using Windows, open your Command Prompt. For convenience, I will address both the Terminal and Command Prompt as ‘ Terminal ’ throughout the book.

We will need to check if we have at least Python 3.8 to use Django 4. To do so, go to your Terminal and run:

**Execute in Terminal**

```
python3 --version
```

(or *python* on Windows)

This shows the version of Python you installed. Make sure that the version is at least 3.8. If it is not so, get the latest version of Python by going to *python.org*. Go to ‘ Downloads ’ and install the version for your OS.

After the installation, run `python3 --version` again and it should reflect the latest version of Python e.g. Python 3.10.0 (at time of book ’ s writing).

## *Installing Django*

We will be using *pip* to install Django. *pip* is the standard package manager for Python to install and manage packages not part of the standard Python library.

First check if you have *pip* installed by going to the Terminal and running:

**Execute in Terminal**

```
pip3
```

(or *pip* on Windows)

If you have *pip* installed, it should display a list of *pip* commands.

To install Django, run the command:

**Execute in Terminal**

```
pip3 install django
```

This will retrieve the latest Django code and install it in your machine. After installation, close and re-open your Terminal.

Ensure you have installed Django by running:

**Execute in Terminal**

```
python3 -m django
```

It will show you all the options you can use (fig. 2.1):

```
MacBook-Air:~ user$ python3 -m django

Type 'python -m django help <subcommand>' fo

Available subcommands:

[django]
    check
    compilemessages
    createcachetable
    dbshell
    diffsettings
    dumpdata
    flush
    inspectdb
    loaddata
    makemessages
    makemigrations
    migrate
    runserver
    sendtestemail
    shell
    showmigrations
    sqlflush
    sqlmigrate
    sqlsequencereset
    squashmigrations
    startapp
    startproject
    test
    testserver
Note that only Django core commands are liste
nfigured (error: Requested setting INSTALLED_
ured. You must either define the environment
r call settings.configure() before accessing
```

Figure 2.1

Along the course of the book, you will progressively be introduced to some of the options. For now, we will use the *startproject* option to create a new project.

In Terminal, navigate to a location on your computer where you want to store your Django project e.g. Desktop. Create a new folder ‘ todoapp ’ with:

**Execute in Terminal**

```
mkdir todoapp
```

‘ cd ’ to that folder:

**Execute in Terminal**

```
cd todoapp
```

In *todoapp*, run:

**Analyze Code**

```
python3 -m django startproject <project_name>
```

In our case, we want to name our project ‘ backend ’ . We run:

**Execute in Terminal**

```
python3 -m django startproject backend
```

A ‘ backend ’ folder will be created. We named the Django project *backend* because it serves as the backend of the Django-React todo app stack. Later on, the React frontend will be contained in a *frontend* folder. The eventual structure will look something like:

todoapp
- → backend
- → Django …
- → frontend
- → React …

In the next chapter, we will look inside the *backend* folder that Django has created for us and understand it better.

# Chapter 3: Understanding the Project Structure

Let ’ s look at the project files created for us. Open the project folder *backend* in a code editor (I will be using the VSCode editor in this book - https://code.visualstudio.com/).

## *manage.py*

You will see a file *manage.py* which we should not tinker. *manage.py* helps us do administrative things. For example:

**Analyze Code**

```
python3 manage.py runserver
```

to start the local web server. We will later illustrate more of its administrative functions e.g. creating a new app – *python3 manage.py startapp*

## *db.sqlite3*

We also have the *db.sqlite3* file that contains our database. We touch more on this file in the *Models* chapter.

## *backend* folder

You will find another folder of the same name *backend*. To avoid confusion and distinguish between the two folders, we will keep the inner *backend* folder as it is and rename the outer folder to *todobackend* (fig. 3.1).

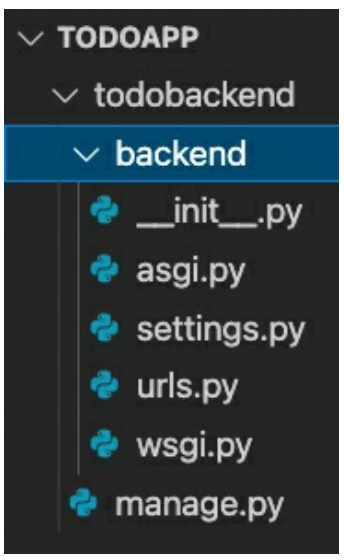


Figure 3.1

In it, you might have a *__pycache__* folder that stores compiled bytecode when we generate our project. You can largely ignore this folder. Its purpose is to make your project start a little faster by caching the compiled code so that it can be readily executed.

*__init__.py* specifies what to run when Django launches for the first time. Again, we can ignore this file.

*asgi.py* allows for an optional Asynchronous Server Gateway Interface to run. *wsgi.py* which stands for Web Server Gateway Interface helps Django serve our web pages. Both files are used when deploying our app. We will revisit them later when we deploy our app.

## *urls.py*

This is the file which tells Django which pages to render in response to a browser or URL request. For example, when someone enters a url http://localhost:8000/123, the request comes into *urls.py* and gets routed to a page based on the paths specified there. We will later add paths to this file and better understand how it works.

## *settings.py*

The *settings.py* file is an important file controlling our project ’ s settings. It contains several properties:

- BASE_DIR determines where on your machine the project is situated at.
- SECRET_KEY is used when you have data flowing in and out of your website. Do not ever share this with others.
- DEBUG – Our site can run in debug or not. In debug mode, we get detailed information on errors. For e.g. if I run http://localhost:8000/123 in the browser, I will see a 404 page not found with details on the error

(fig. 3.2).

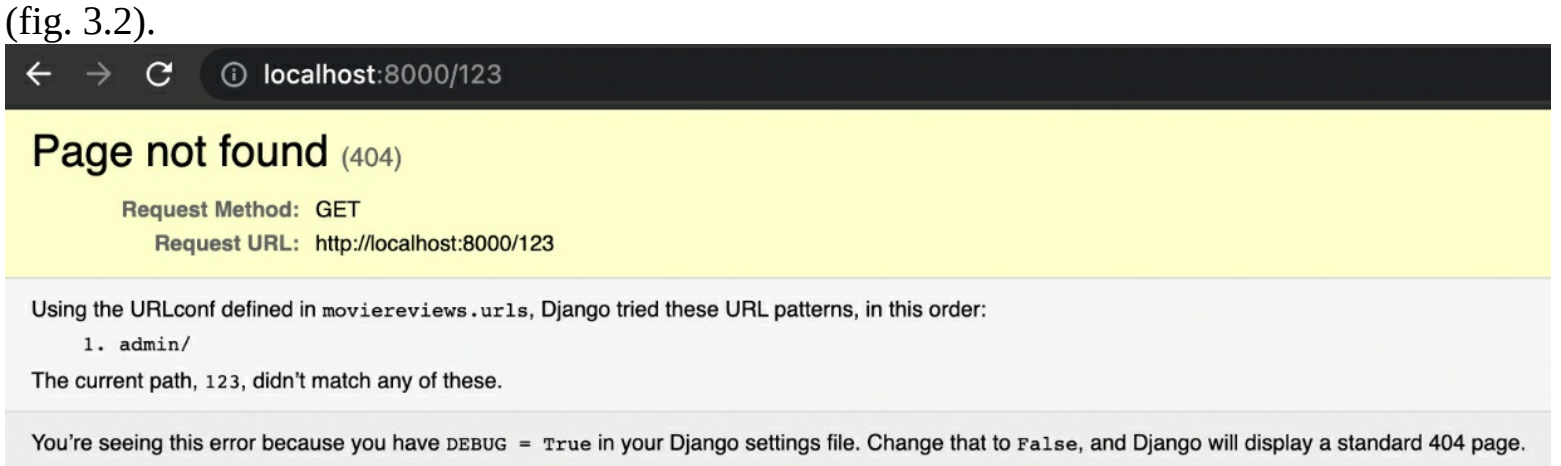


Figure 3.2

If DEBUG = False, we will see a generic 404 page without error details. While developing our project, we set DEBUG = True to help us with debugging. When deploying our app to production, we should set DEBUG to False.

- INSTALLED_APPS allow us to bring in different pieces of code into our project. We will see this in action later.
- MIDDLEWARE are built-in Django functions to help in our project.
- ROOT_URLCONF specify where our URLs are.
- TEMPLATES help us return HTML code.
- AUTH_PASSWORD_VALIDATORS allow us to specify validations we want on passwords. E.g. a minimum length.

There are some other properties in *settings.py* like LANGUAGE_CODE and TIME_ZONE but we have focused on the more important ones. We will later revisit this file and see how relevant it is in developing our site. Let ' s next create our backend server!

# Chapter 4: Creating Our Backend Server

Now, it ’ s time to create the backend server! A single Django project can contain one or more *apps* that work together to power an API. Django uses the concept of projects and apps to keep code clean and readable.

In the *todobackend* folder, create a new app ‘todo’ with the command:

**Execute in Terminal**

```
python3 manage.py startapp todo
```

A new folder *todo* will be added to the project. As we progress along the book, we will explain the files inside the folder.

Though our new app exists in our Django project, Django doesn ’ t ‘ know ’ it till we explicitly add it. To do so, we specify it in *settings.py*. So go to … */backend/settings.py*, under INSTALLED_APPS, add the app name (shown in **bold**):

**Modify Bold Code**

```
…
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'todo',
]
…
```

## Models

Working with databases in Django involves working with models. We create a database model (e.g. todo, book, blog post, movie) and Django turns these models into a database table for us. In … */backend/todo*, you have the file *models.py* where we create our models. Let’s create our todo model by filling in the following:

### … */todo/models.py*:

**Replace Entire Code**

```
from django.db import models
from django.contrib.auth.models import User


class Todo(models.Model):
    title = models.CharField(max_length=100)
    memo = models.TextField(blank=True)


    #set to current time
    created = models.DateTimeField(auto_now_add=True)
  completed = models.BooleanField(default=False)


  #user who posted this
    user = models.ForeignKey(User, on_delete=models.CASCADE)


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

### *Code Explanation*

**Analyze Code**

```
from django.db import models
```

Django imports a module *models* to help us build database models which ‘ model ’ the characteristics of the data in the database. In our case, we created a todo model to store the title, memo, time of creation, time of completion and user who created the todo.

**Analyze Code**

```
class Todo(models.Model)
```

*class Todo* inherits from the *Model* class. The *Model* class allows us to interact with the database, create a table, retrieve and make changes to data in the database.

Note that we use Django ’ s built in User model (imported at the second line from the top) as the creator of the todo.

**Analyze Code**

```
title = models.CharField(max_length=100)
memo = models.TextField(blank=True)


#set to current time
 created = models.DateTimeField(auto_now_add=True)
completed = models.BooleanField(default=False)


#user who posted this
 user = models.ForeignKey(User, on_delete=models.CASCADE)
```

We then have the properties of the model. Notice that the properties have types like *CharField*, *TextField*, *DateTimeField*. Django provides many other model fields to support common types like dates, integers, emails, etc. To have a complete documentation of the kinds of types and how to use them, refer to the *Model* field reference in the Django documentation (https://docs.djangoproject.com/en/4.0/ref/models/fields/).

If any of the above feels new to you, it will be good to pause here and review Beginning Django (contact support@i-ducate.com) for more explanation of traditional Django.

**Analyze Code**

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

We also include a *__str__* method so that the title of a todo will display in the admin later on.

With this model, we can create a Todo table in our database.

## Migrations

Whenever we create a new model or make changes to it in *models.py*, we need to update Django in a two-step process. First, in /todoapp/todobackend/, we create a migration file with the *makemigrations* command:

**Execute in Terminal**

```
python3 manage.py makemigrations
```

This generates the SQL commands (migrations) for preinstalled apps in our INSTALLED_APPS setting. The SQL commands are not yet executed but are just a record of all changes to our models. The migrations are stored in an auto-generated folder *migrations* (fig. 1).

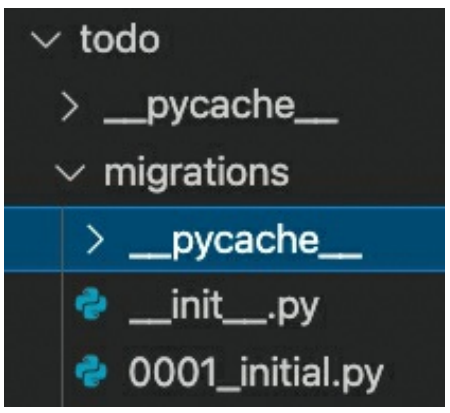


Figure 1

Second, we build the actual database with the following:

**Execute in Terminal**

```
python3 manage.py migrate
```

The *migrate* command creates an initial database based on Django ’ s default settings and executes the SQL commands in the migrations file. Notice there is a *db.sqlite3* file in the project folder. The file represents our SQLite database. It is created the first time we run *migrate*. *migrate* syncs the database with the current state of any database models in the project and listed in INSTALLED_APPS. It is thus essential that after we update a model, we need to run *migrate*.

In summary, each time you make changes to a model file, you have to run:

**Analyze Code**

```
python3 manage.py makemigrations
python3 manage.py migrate
```

* Contact support@i-ducate.com for the source code of the completed project

# Adding todos to the database

We will add some todo data via the build-in admin site. Django has a powerful built-in admin interface which provides a visual way of managing all aspects of a Django project, e.g. users, making changes to model data. To access the Admin site, start your local Django server by going to …/todobackend/ and running in the Terminal:

**Execute in Terminal**

```
python3 manage.py runserver
```

In the browser, go to http://localhost:8000/admin/ where you will see a log in form (fig. 2).

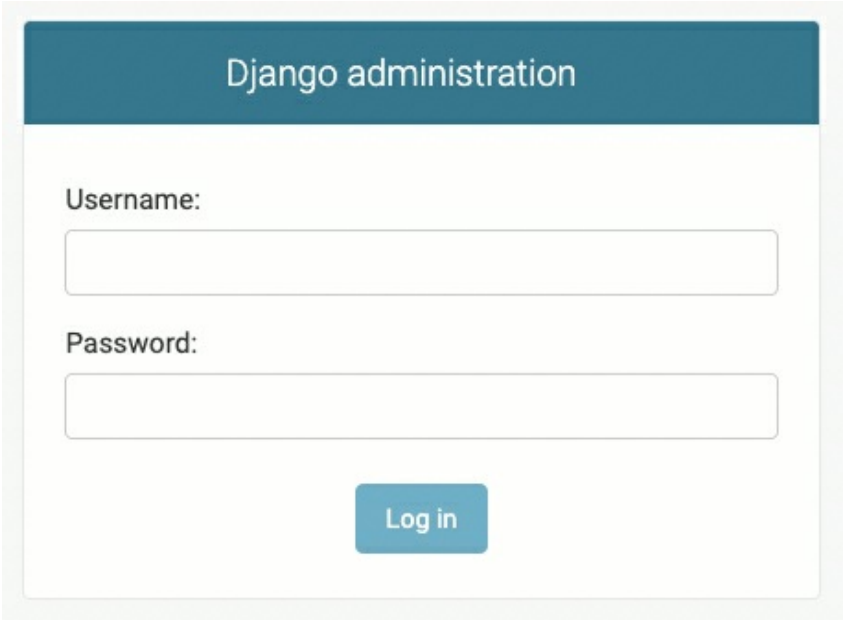


Figure 2

With what username and password do we log in to Admin? We will have to first create a superuser in the Terminal. Open a new Terminal, and in … /todobackend/, run:

**Execute in Terminal**

```
python3 manage.py createsuperuser
```

You will then be asked to specify a username, email and password. Note that anyone can access the admin path on your site, so make sure your password is something secure.

We next have to add the Todo model to admin by going to …/*todo/admin.py*, and register our Todo model with:

**Modify Bold Code**

```
from django.contrib import admin
from .models import Todo


# Register your models here.
admin.site.register(Todo)
```

When you save your file and log back into http://localhost:8000/admin/. The Todo model will now show up (fig. 3).


Figure 3

Try adding a todo object by clicking on ‘ +Add ’ . You will be brought to a Add Todo Form (fig. 4).

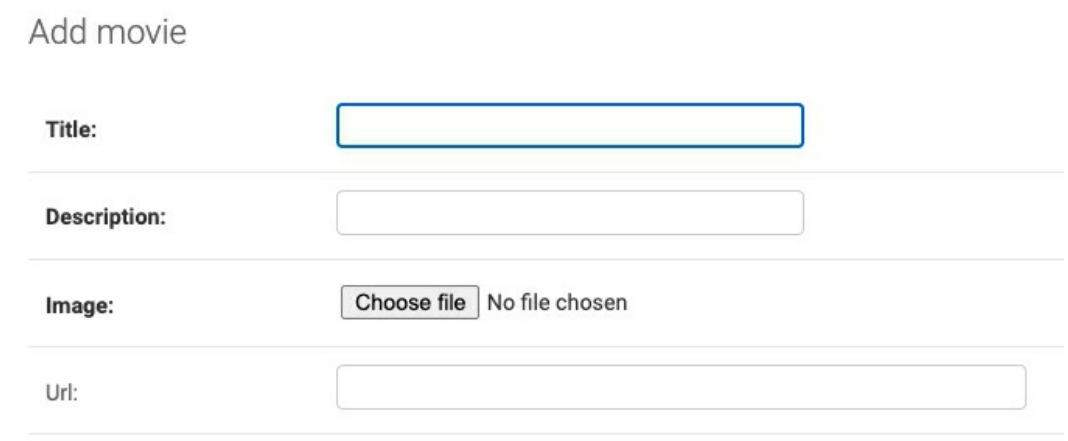


Figure 4

Try adding a todo and hit ‘ Save ’ . Note that you have to specify a user (fig. 5).

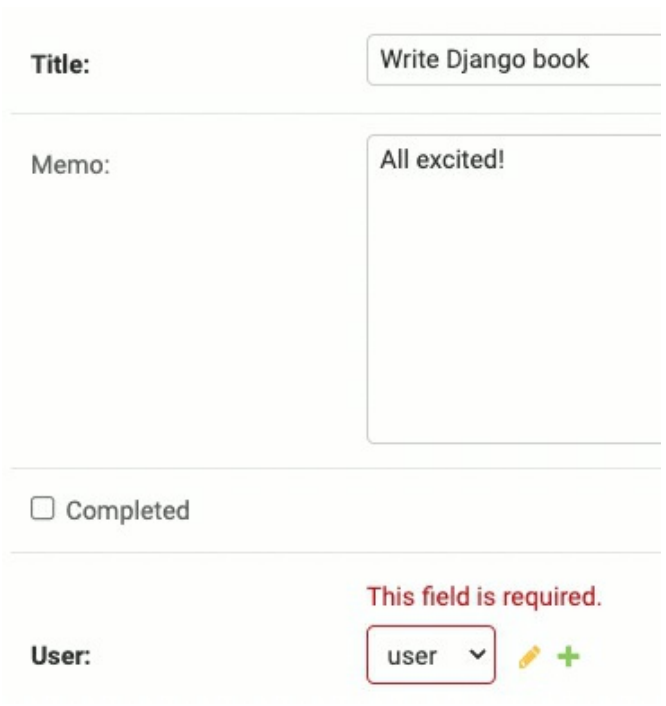


Figure 5

Your todo object will be saved to the database and reflected in the admin (fig. 6).

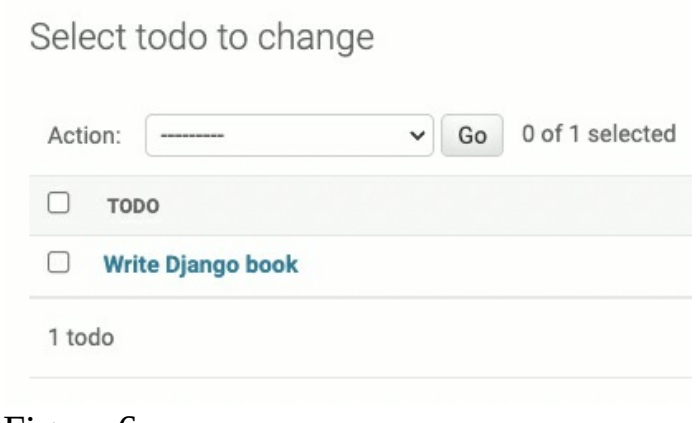


Figure 6

Now that we know how to add todo objects to our database via admin, let ’ s carry on with building our API.

# Chapter 5: Serialization

## *Django REST Framework*

We will use the Django REST Framework to help build our API. You can build APIs on your own in Django but it will take a lot of work. The Django Rest Framework helps us quickly build reliable and secure APIs. There are a ton of documentation in the DRF site ((https://www.django-rest-framework.org/) but this book will show you the best parts of the framework.

## *Installation*

Install DRF using pip:

**Execute in Terminal**

```
pip3 install djangorestframework
```

Next, add 'rest_framework' to your INSTALLED_APPS setting in *../backend/settings.py*:

**Modify Bold Code**

```
...
INSTALLED_APPS = [
    ...
    'todo',
    'rest_framework',
]
...
```

## *Adding the API app*

We create a dedicated *api* app to contain API specific files. So in the Terminal, under *../todobackend/*, run the command:

**Execute in Terminal**

```
python manage.py startapp api
```

We'll need to add our new *api* app to INSTALLED_APPS. In *todobackend/backend/settings.py*, add:

**Modify Bold Code**

```
...
INSTALLED_APPS = [
    ...
    'todo',
    'rest_framework',
    'api',
]
...
```

To get started on our API, we declare Serializers which translate data from the Todo Model instances into JSON objects that is easy to consume over the Internet. The JSON objects are outputted at API endpoint urls. Django REST framework shops with a powerful built-in serializer class that we can quickly extend with some code.

In the *api* folder, create the file *serializers.py* with the following code:

**Add Code**

```
from rest_framework import serializers
from todo.models import Todo


class TodoSerializer(serializers.ModelSerializer):
     created = serializers.ReadOnlyField()
     completed = serializers.ReadOnlyField()


    class Meta:
        model = Todo
        fields = ['id','title','memo','created','completed']
```

## *Code Explanation*

**Analyze Code**

```
class TodoSerializer(serializers.ModelSerializer):
```

We extend DRF's *ModelSerializer* into a *TodoSerializer* class. *ModelSerializer* provides an API to create serializers from your models.

**Analyze Code**

```
class Meta:
    model = Todo
    fields = ['id','title','memo','created','completed']
```

Under *class Meta*, we specify our database model *Todo* and the fields we want to expose i.e. ['id','title','memo','created','completed']. Django REST Framework then magically transforms our model data into JSON, exposing these fields from our Todo model.

Fields not specified here will not be exposed in the API. Remember that ‘id’ is created automatically by Django so we didn’t have to define it in Todo model. But we will use it in our API.

Often, an underlying database model will have more fields than what needs to be exposed. DRF’s serializer class’s ability to include/exclude fields in our API is a great feature and makes it straightforward to control this.

**Analyze Code**

```
# auto populated by app. User can’t manipulate
  created = serializers.ReadOnlyField()
  completed = serializers.ReadOnlyField()
```

Additionally, we specify that created and completed fields are read only. I.e., they cannot be edited by a user (because they ought to be auto-populated when a todo is created and when it is marked as complete).
Now that we have our first serializer, let ’ s see how to setup URLs for our API endpoints in the next chapter.

# Chapter 6: URLs and Class-based Views

For the API app to have its own *urls.py* to contain the paths for the API end points, we add the below to *todobackend/backend/urls.py*:

**Modify Bold Code**

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


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

This will forward all API related paths e.g. *localhost:8000/api/** to the API app.
*Creating our first route*

In */todobackend/api/*, create a new file *urls.py* which will contain the routes of the different endpoints. Add in the below code:

**Add Code**

```
from django.urls import path
from . import views


urlpatterns = [
    path('todos/', views.TodoList.as_view()),
]
```

We have one route '/todos' as a demonstration. What we want to achieve is that when a request is made to *localhost:8000/api/todos*, you should get a JSON response with the list of todo instances. But how do we achieve this?

The answer is in *views.TodoList.as_view()*.

In traditional full stack Django projects, views are used to customize what data to send to the HTML templates. In our API project, we use class-based generic views (from DRF) to similarly send customized serialized data but this time, to the API endpoints instead of to templates.

*views.TodoList* is an instance of a class-based generic view. Django's *generic views* help us quickly write views (without having to write too much repetitive code) to do common tasks like:

- Display a list of objects, e.g. list of todos.
- Display detail pages for a single object. E.g. detail page of a todo.
- Allow users to create, update, and delete objects – with or without authorization.

Things will be clearer as we implement our *TodoList* generic view. Thus, in *todobackend/api/views.py*, replace the code with the following:

**Replace Entire Code**

```
from rest_framework import generics
from .serializers import TodoSerializer
from todo.models import Todo


class TodoList(generics.ListAPIView):
    # ListAPIView requires two mandatory attributes, serializer_class and
    # queryset.
    # We specify TodoSerializer which we have earlier implemented
    serializer_class = TodoSerializer


    def get_queryset(self):
            user = self.request.user
        return Todo.objects.filter(user=user).order_by('-created')
```

## *Code Explanation*

**Analyze Code**

```
from rest_framework import generics
```

We import DRF ' s *generics* class of views.

**Analyze Code**

```
class TodoList(generics.ListAPIView):
```

We then create *TodoList* that uses *generics.ListAPIView*. *ListAPIView* is a built-in generic class which creates a read-only endpoint for model instances. *ListAPIView* requires two mandatory attributes which are serializer_class and queryset.
When we specify *TodoSerializer* as the serializer class, we create a read-only endpoint for todo instances. There are many other generic views available and we explore them later.

**Analyze Code**

```
def get_queryset(self):
        user = self.request.user
    return Todo.objects.filter(user=user).order_by('-created')
```

get_queryset returns the queryset of todo objects for the view. In our case, we specify the query set as all todos which match the user. Additionally, we order the todos by the created date i.e. we show the latest todo first. You can customize get_queryset to return the set of todos that you want.

## *Running our App*

In Terminal, *cd* to the *todobackend* directory and run *python3 manage.py runserver* to test run your app.

Go to localhost:8000/api/todos and you can see the data in JSON format visualized (fig. 1)! Note you must be logged in the admin section with your admin user, before accessing the previous route.

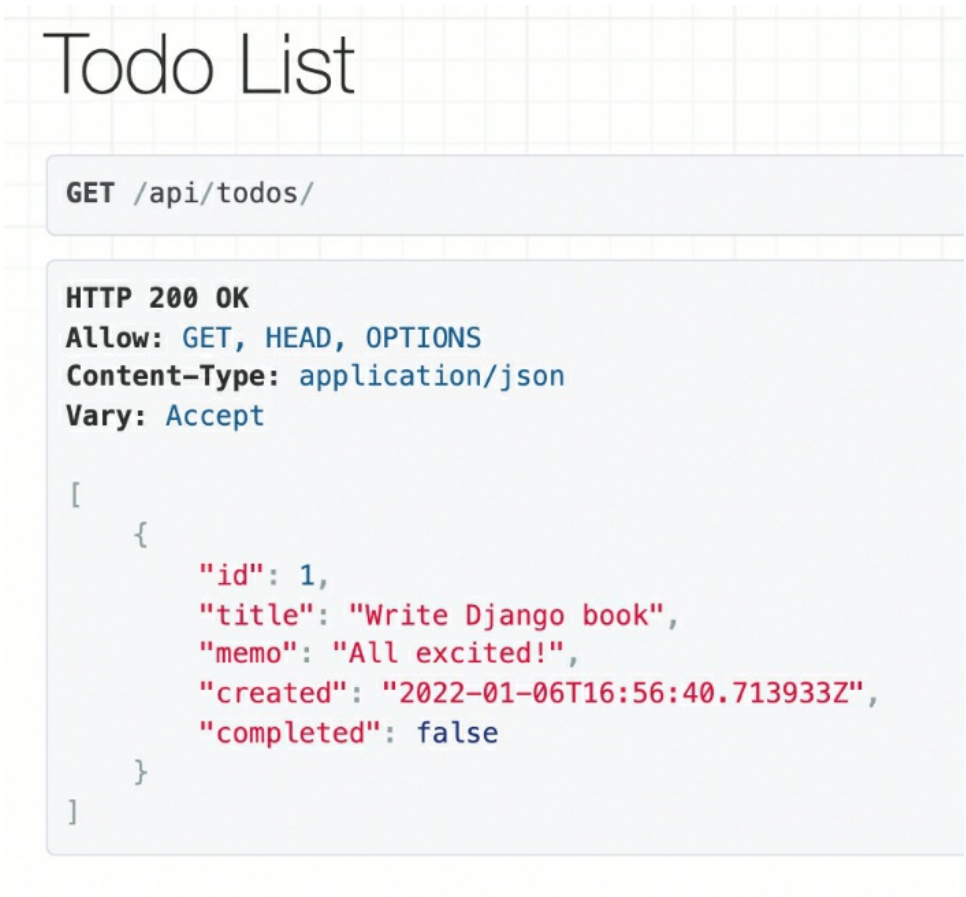


Figure 1

DRF provides this powerful visualization by default. There are lots of other functionality in this page that we will explore throughout the book. As you add more todos via the Admin, they will show up in the API endpoint as you reload the page.

And you just created your first API endpoint with Django Rest Framework! See how DRF generic views help us to quickly implement these endpoints with little code?

But what about endpoints that allow us to create, update or delete? We will explore them in the following chapters.

# Chapter 7: Creating Todos via the ListCreateAPIView

The *ListAPIView* we used in the previous chapter provides us with a read-only endpoint of todo model instances. To have a read-write endpoint, we use *ListCreateAPIView*. It is similar to *ListAPIView* but allows for creation as well.

In *todobackend/api/views.py*, add the following in **bold**:

**Modify Bold Code**

```
...


class TodoList**Create**(generics.List**Create**APIView):
    # ListAPIView requires two mandatory attributes, serializer_class and
    # queryset.
    # We specify TodoSerializer which we have earlier implemented
    serializer_class = TodoSerializer


    def get_queryset(self):
        user = self.request.user
        return Todo.objects.filter(user=user).order_by('-created')
```

All we have to do is just change *ListAPIView* to *ListCreateAPIView*, and automatically, the *write* operation is allowed for the given API end point! This is the advantage of a fully featured framework like Django REST Framework. The functionality is available and just works. Developers don't have to write much code and implement it themselves.

Note that we have to update our view name in *todobackend/api/urls.py*:

**Modify Bold Code**

```
...
urlpatterns = [
    path('todos/', views.TodoList**Create**.as_view()),
]
```

To test our API, go to *localhost:8000/api/todos* on the browser, and you will now see a form/raw data (JSON) below (fig. 1) that you can enter to create a new todo.

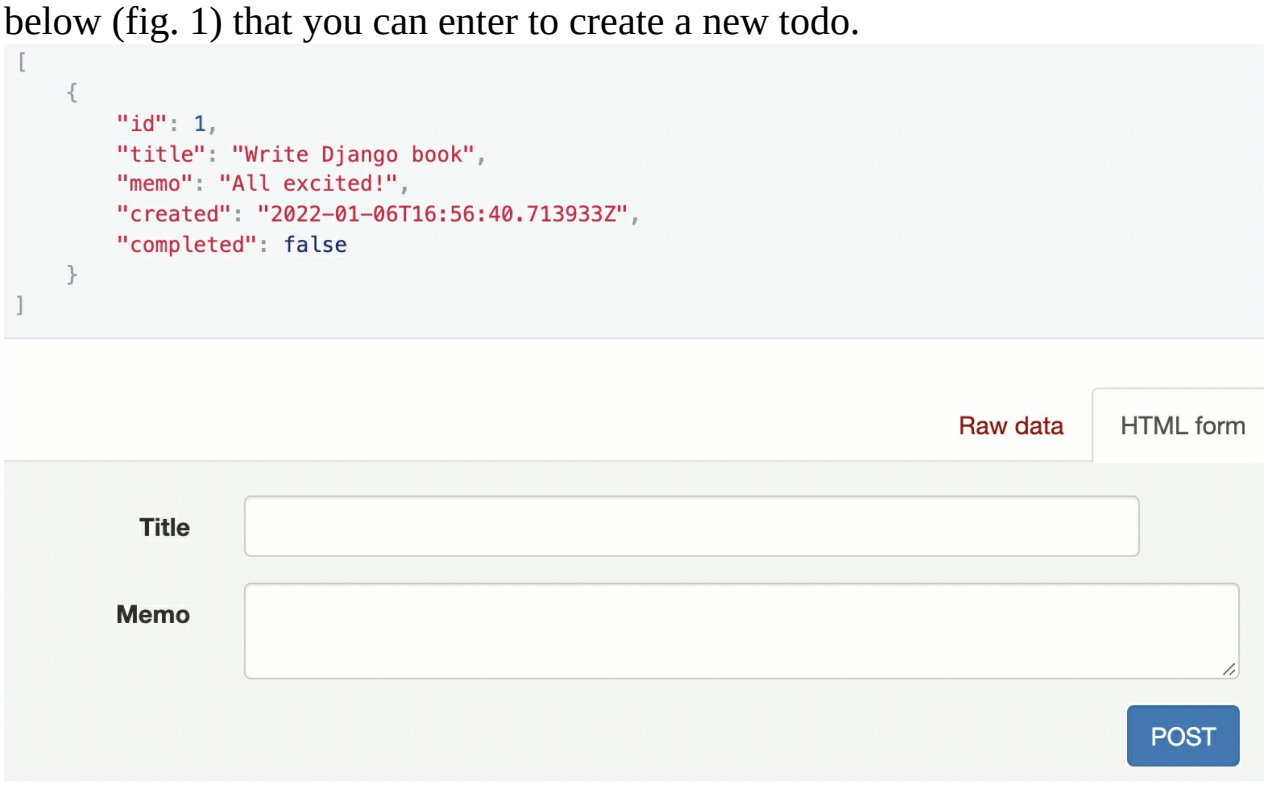


Figure 1

However, when you attempt to add a todo via the form, you get an error like (fig. 2):

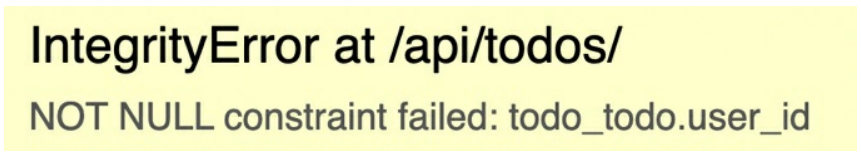


Figure 2

This is because we have not specified a user who created this todo. To do so, we have to customize the creation process using *perform_create*.

*perform_create*

What is missing currently is when we create a todo, we should automatically assign the user who created it. So how can we customize the creation process in our API?

In *todobackend/api/views.py*, add the below *perform_create* method:

**Modify Bold Code**

```
...


class TodoListCreate(generics.ListCreateAPIView):
     # ListAPIView requires two mandatory attributes, serializer_class and
    # queryset.
    # We specify TodoSerializer which we have earlier implemented
    serializer_class = TodoSerializer


    def get_queryset(self):
          user = self.request.user
        return Todo.objects.filter(user=user).order_by('-created')


    def perform_create(self, serializer):
        #serializer holds a django model
        serializer.save(user=self.request.user)
```

*perform_create* acts as a *hook* which is called before the instance is created in the database. Thus, we can specify that we set the user of the todo as the request's user before creation in the database.

These hooks are particularly useful for setting attributes that are implicit in the request, but are not part of the request data. In our case, we set the todo's user based on the request user.

See how Django allows many auto-generated features yet also allow these customizations?

### *Running your App*

When you attempt to add a todo via the form now, your todo should be successfully added. If you reload the endpoint at *localhost:8000/api/todos/*, the new todo should be in the JSON results.

# Chapter 8: Permissions

Currently, we allow anyone to access the API endpoint and list or create a todo. But this obviously shouldn't be the case as we only want registered users to call the API to read/create their own todos (which others don't have access to). So how do we ensure that only authenticated users can call the API and deny access to unauthenticated users?

Just add two lines in *todobackend/api/views.py*:

**Modify Bold Code**

```
from rest_framework import generics, permissions
from .serializers import TodoSerializer
from todo.models import Todo


class TodoListCreate(generics.ListCreateAPIView):
    ...
    serializer_class = TodoSerializer
    permission_classes = [permissions.IsAuthenticated]


    ...
```

With this, we specify that only authenticated and registered users have permission to call this API. Unauthenticated users are not allowed to access it.

A slightly less strict style of permission would be to allow full access to authenticated users, but allow read-only access to unauthenticated users. To do so, we specify the IsAuthenticatedOrReadOnly class. i.e.

**Analyze Code**

```
permission_classes = [permissions.IsAuthenticated]
```

There are other permissions like:

- *IsAdminUser* – only admins/superusers have access
- *AllowAny* – any user, authenticated or not, has full access

With permissions, we can grant or deny access for different classes of users to different parts of the API.

## *Test Your App*

Now, go to *localhost:8000/admin* in your browser and log out of your account. Revisit *localhost:8000/api/todos* and will see a message saying (fig. 1):


Figure 1

If log back in through the admin page and visit *localhost:8000/api/todos*, you will be able to see your todos now.

# Chapter 9: Other C.R.U.D. Operations

So far, we have implemented list and create todo API endpoints. A very common occurring pattern is retrieving, editing and deleting a specific model instance. To achieve this, we implement a *get*(), *put*() and *delete*() endpoint.

Django REST Framework provides the built-in *RetrieveUpdateDestroyAPIView* to automatically implement the *get*(), *put*() and *delete*() endpoint.

Let's first create an API path for it by adding in *todobackend/api/urls.py*:

**Modify Bold Code**

```
...
urlpatterns = [
    path('todos/', views.TodoListCreate.as_view()),
    path('todos/<int:pk>', views.TodoRetrieveUpdateDestroy.as_view()),
]
```

That is, to retrieve, update or delete an individual todo, the endpoint will be something like:

*localhost:8000/api/todos/123* ('123' being the todo's id)

*TodoRetrieveUpdateDestroy*

Next in *todobackend/api/views.py*, we create the TodoRetrieveUpdateDestroy generic view by adding:

**Modify Bold Code**

```
...
class TodoListCreate(generics.ListCreateAPIView):
    ...


class TodoRetrieveUpdateDestroy(generics.RetrieveUpdateDestroyAPIView):
    serializer_class = TodoSerializer
    permission_classes = [permissions.IsAuthenticated]


    def get_queryset(self):
        user = self.request.user
        # user can only update, delete own posts
        return Todo.objects.filter(user=user)
```

Like the *ListCreateAPIView*, *RetrieveUpdateDestroyAPIView* requires two mandatory attributes which are serializer_class and queryset.

Queryset here automatically retrieves just the todo instance for the id specified in the endpoint e.g. *localhost:8000/api/todos/123*

## *Testing our App*

In your browser, navigate to http://localhost:8000/api/todos/<todo_id> e.g. http://localhost:8000/api/todos/1, you will be able to execute a DELETE request by clicking on the 'DELETE' button (fig. 1).


Figure 1

In the same endpoint via the browser, you will be able to execute a PUT request through a form to update a todo (fig. 2).


Figure 2

I hope you begin to see how amazing it is that all we have to do to implement a particular API endpoint is to use the appropriate generic view. This is the benefit of using a ‘batteries included’ framework like DRF. The functionality is available and just works. Developers don’t have to write much code to reinvent the wheel. We should appreciate that this already performs the basic list and CRUD functionality we want.

# Chapter 10: Completing a Todo

Other than the common CRUD related kinds of operations, how do we implement views for customised logic? In this chapter, we will see how to implement a view that completes a todo.

We first add a path to complete a todo by adding the below code:

*todobackend/api/urls.py*

**Modify Bold Code**

```
...
urlpatterns = [
    path('todos/', views.TodoListCreate.as_view()),
    path('todos/<int:pk>', views.TodoRetrieveUpdateDestroy.as_view()),
    path('todos/<int:pk>/complete', views.TodoToggleComplete.as_view()),
]
```

That is, if a request is sent to *localhost:8000/api/todos/123/complete*, the todo with id '123' will be marked as complete.

Let's implement the *TodoToggleComplete* view in *todobackend/api/views.py*:

**Modify Bold Code**

```
from rest_framework import generics, permissions
from .serializers import TodoSerializer, TodoToggleCompleteSerializer
from todo.models import Todo


class TodoListCreate(generics.ListCreateAPIView):
    ...

class TodoRetrieveUpdateDestroy(generics.RetrieveUpdateDestroyAPIView):
    ...

class TodoToggleComplete(generics.UpdateAPIView):
    serializer_class = TodoToggleCompleteSerializer
    permission_classes = [permissions.IsAuthenticated]

    def get_queryset(self):
        user = self.request.user
        return Todo.objects.filter(user=user)

    def perform_update(self,serializer):
        serializer.instance.completed=not(serializer.instance.completed)
        serializer.save()
```

## *Code Explanation*

**Analyze Code**

```
class TodoToggleComplete(generics.UpdateAPIView):
```

As its name suggests, *TodoToggleComplete* toggles a todo from *incomplete* to *complete* and vice-versa. *TodoToggleComplete* extends the *UpdateAPIView* used for update-only endpoints for a single model instance.

**Analyze Code**

```
    serializer_class = TodoToggleCompleteSerializer
    permission_classes = [permissions.IsAuthenticated]
```

We will implement the TodoToggleCompleteSerializer later on. Only authenticated users can mark a todo as complete.

**Analyze Code**

```
    def perform_update(self,serializer):
        serializer.instance.completed=not(serializer.instance.completed)
        serializer.save()
```

Similar to perform_create, perform_update is called before the update happens. In it, we 'invert' the todo's *completed* boolean value. i.e. if its true, set to false, if false, set to true.

## *todobackend/api/serializers.py*

Let's now implement the *TodoToggleCompleteSerializer* serializer. Add the below code to *todobackend/api/serializers.py*:

**Modify Bold Code**

```
from rest_framework import serializers
from todo.models import Todo


class TodoSerializer(serializers.ModelSerializer):
    ...


class TodoToggleCompleteSerializer(serializers.ModelSerializer):
    class Meta:
        model = Todo
        fields = ['id'] # why need to show id?
            read_only_fields = ['title','memo','created','completed']
```

Because the TodoToggleCompleteSerializer doesn't receive and update any of the fields values from the endpoint (it just toggles *completed*), we set the fields to read only by specifying them in the Meta shortcut option *read_only_fields*. *read_only_fields* allows us to specify multiple fields as read-only.

## *Testing Your App*

To test the toggle complete endpoint in the browser, go to *localhost:8000/api/todos/<todo_id>/complete* e.g.:

*localhost:8000/api/todos/2/complete* and click on 'PUT' (fig. 1).

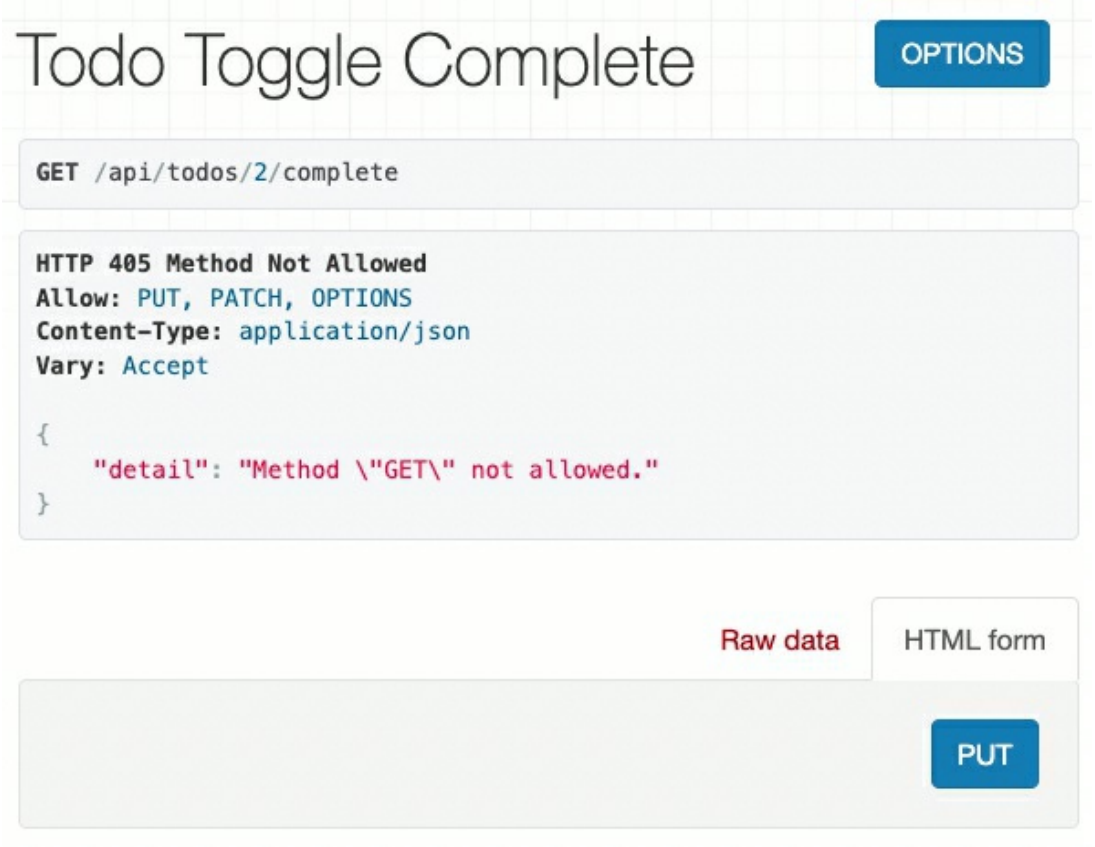


Figure 1

And if you list the todos again, that todo will be marked as complete i.e. *completed: true* (fig. 2)

# Todo List Create

```
GET /api/todos/
```

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

[
    {
        "id": 3,
        "title": "Buy gaming mouse",
        "memo": "",
        "created": "2022-01-10T00:32:18.331270Z",
        "completed": true
    },
    {
        "id": 2,
        "title": "Go to sleep",
        "memo": "7-8 hours",
        "created": "2022-01-08T07:07:41.485041Z",
        "completed": false
    }
]
```

Figure 2

# Chapter 11: Authentication – Sign Up

Now that we have implemented all the endpoints for todos-related functionality, let's proceed to implement authentication where a user can sign up for a new account, log in and log out.

Currently, we can log in and out via the Admin site. But how can we sign up or log in via the JSON API itself?

Because HTTP is a stateless protocol, we can't remember if a user is authenticated from one API request to the next. So how does a user verify himself when accessing a restricted resource via HTTP?

## *Basic Authentication*

The most common form of HTTP authentication is known as 'Basic authentication'. That is, we pass the *username:password* in the request. Using a popular HTTP client, Insomnia (https://insomnia.rest/), we can make requests to the API.

In Insomnia, enter the URL for a GET request. Under 'Auth', select 'Basic Auth' and specify the username and password to get the list of todos (fig. 1).

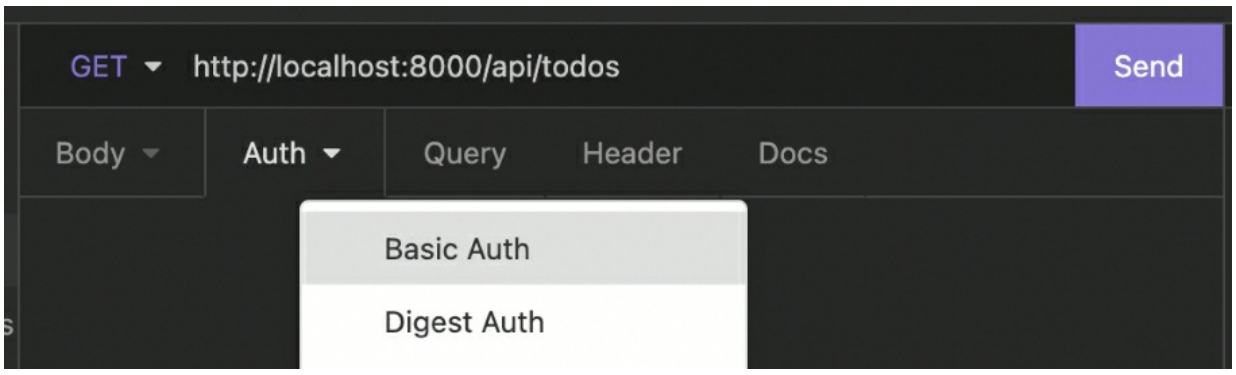

Figure 1

The primary advantage to this approach is its simplicity. The downside is that on every request, the server must look up and verify the username and password which is inefficient.

Secondly, passwords are passed in clear text all over the internet which is a big security flaw and vulnerability. Any non-encrypted data can easily be captured.

*Token Authentication*

The better approach is Token authentication which has become the most popular approach in recent years due to the popularity of single page applications.

Because token authentication is stateless, a client first sends the initial username and password to the server. The server generates a unique token which is stored by the client (in the frontend portion of the book, we will later illustrate how the client stores the token using *local storage*).

This token is then passed in the header of each HTTP request and the server uses it to verify that a user is authenticated and then responds to that user. The server doesn't keep a record of the user, but keeps just a record of valid tokens.

Tokens are a better alternative as we avoid having to re-check user passwords and pass clear text passwords around.

## *Set Up*

Before we proceed, in *todobackend/backend/settings.py*, we need to add the *authtoken* app to the *INSTALLED_APPS* setting:

**Modify Bold Code**

```
...
INSTALLED_APPS = [
    ...
    'todo',
    'rest_framework',
    'api',
    'rest_framework.authtoken',
]
...
```

*authtoken* generates the tokens on the server.

Remember that since we have made changes to INSTALLED_APPS, we need to sync our database by executing

the migrate command:

**Execute in Terminal**

```
python3 manage.py migrate
```

Next, to specify that we want to use Token authentication (instead of Basic authentication or Session authentication etc.), in *todobackend/backend/settings.py*, we add:

**Modify Bold Code**

```
...

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES':[
        'rest_framework.authentication.TokenAuthentication',
    ]
}
```

## *User Sign Up*

Let’s first implement a new user sign up. We first create a path in *todobackend/api/urls.py* for the signup endpoint:

**Modify Bold Code**

```
from django.urls import path
from . import views


urlpatterns = [
    path('todos/', views.TodoListCreate.as_view()),
    path('todos/<int:pk>', views.TodoRetrieveUpdateDestroy.as_view()),
    path('todos/<int:pk>/complete', views.TodoToggleComplete.as_view()),
    path('signup/', views.signup),
]
```

We then implement the signup view in *todobackend/api/views.py* by adding the following:

**Modify Bold Code**

```
...
from todo.models import Todo
from django.db import IntegrityError
from django.contrib.auth.models import User
from rest_framework.parsers import JSONParser
from rest_framework.authtoken.models import Token
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt


...

class TodoToggleComplete(generics.UpdateAPIView):
    ...


@csrf_exempt
def signup(request):
    if request.method == 'POST':
        try:
            data = JSONParser().parse(request) # data is a dictionary
            user = User.objects.create_user(
                                username=data['username'],
                password=data['password'])
            user.save()


            token = Token.objects.create(user=user)
                        return JsonResponse({'token':str(token)},status=201)
        except IntegrityError:
            return JsonResponse(
                                {'error':'username taken. choose another username'},
                status=400)
```

## *Code Explanation*

**Analyze Code**

```
@csrf_exempt
```

Because the POST request is coming from a different domain (the frontend domain) and will not have the token required to pass the CSRF checks (cross site request forgery), we use *csrf_exempt* for this view.

**Analyze Code**

```
def signup(request):
    if request.method == 'POST':
```

We first check if a request was performed using the HTTP ‘POST’ method. This is because the sign up form in the front end will use POST request for form submissions.

**Analyze Code**

```
try:
    data = JSONParser().parse(request) # data is a dictionary
    user = User.objects.create_user(
                        username=data['username'],
        password=data['password'])
    user.save()
```

We then call *JSONParse().parse* to parse the JSON request content and return a *dictionary* of data.

We extract the user filled-in values from the dictionary with data[‘username’] and data[‘password’] and create a user with User.objects.create_user. User.objects.create_user creates and returns a user.

With *user.save()*, we save the User object to the database.

**Analyze Code**

```
    token = Token.objects.create(user=user)
            # dictionary with some info. 201 means success creation
                return JsonResponse({'token':str(token)}, status=201)
```

After saving the user to the database, we create the token object with *Token.objects.create(user=user)*.

If all goes well, we return a JsonResponse object with a dictionary containing the token, and status code of 201 indicating successful creation.

**Analyze Code**

```
except IntegrityError:
    return JsonResponse(
                        {'error':'username taken. choose another username'},
        status=400)
```

If there is an error, we return a *JsonResponse* object with a dictionary containing the error and status code of 400 indicating that the request cannot be fulfilled.

Note: when we send the JSON Response, we attach a status code to accompany it. The status code reflects the following:

- 2xx ‘success’ - the request was successfully received, understood and accepted
- 3xx ‘redirection’ – further action needs to be taken to complete the request
- 4xx ‘client error’ – the request contains bad syntax or cannot be fulfilled, typically a bad URL request by the client
- 5xx ‘server error’ – the server failed to fulfill a valid request

You don’t need to memorize all the status codes. The most common ones are 200 ‘Ok’, 201 ‘created’, 404 ‘not found’, and 500 ‘server error’.

## *Testing Our App*

Now let’s test our app by signing up for a new user. In Insomnia, change the request to POST with the URL *http://localhost:8000/api/signup/* (fig. 2).

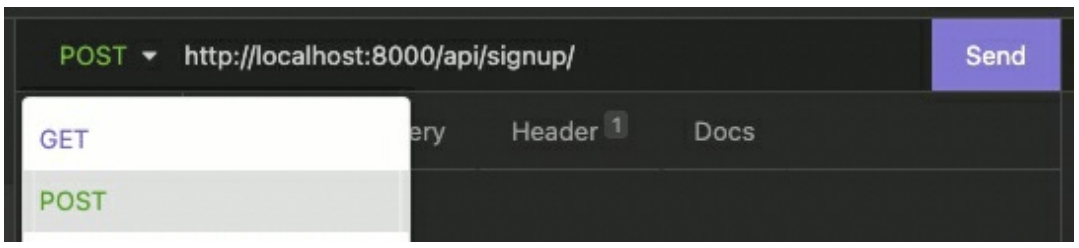


Figure 2

Specify the JSON object with the username and password we want to pass in with the POST request (fig. 3).

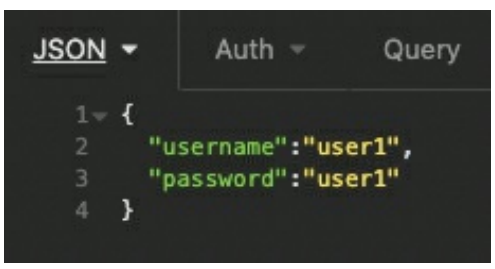


Figure 3

When you click ‘Send’, there should be a ‘201 Created’ response with the authentication token (fig. 4).

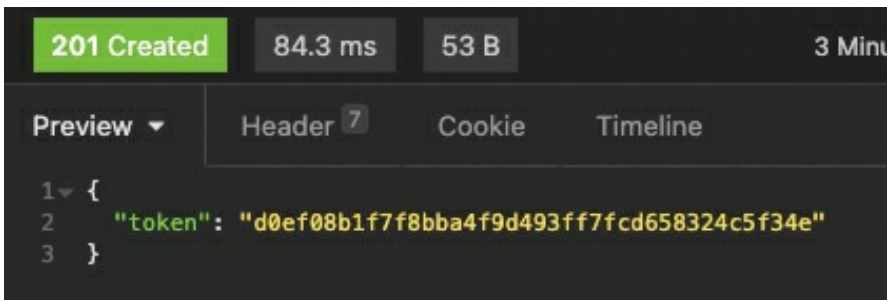

Figure 4

That means we have successfully created our user!

If we submit the same request with the same username and password, we should get a '400 Bad Request' with an error message that the username is already taken (fig. 5):

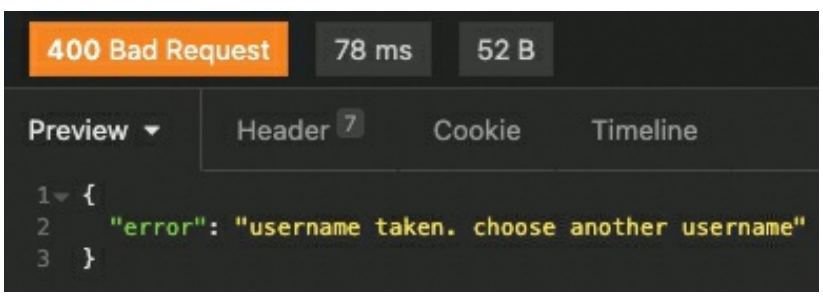

Figure 5

If you navigate to the Django Admin, you will see a Tokens section at the top (fig. 6 - make sure you are logged in with your superuser account to have access).

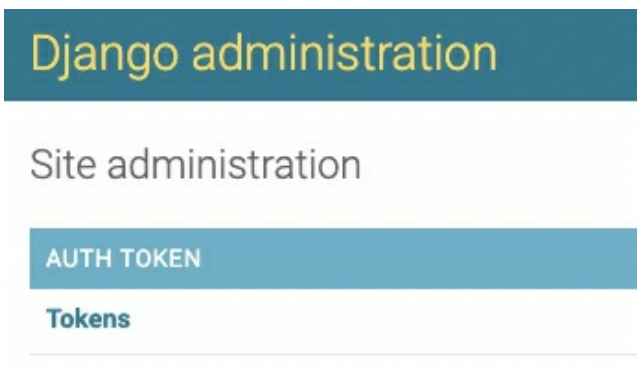

Figure 6

If you click on 'Tokens', you can see which token belongs to which user (fig. 7).

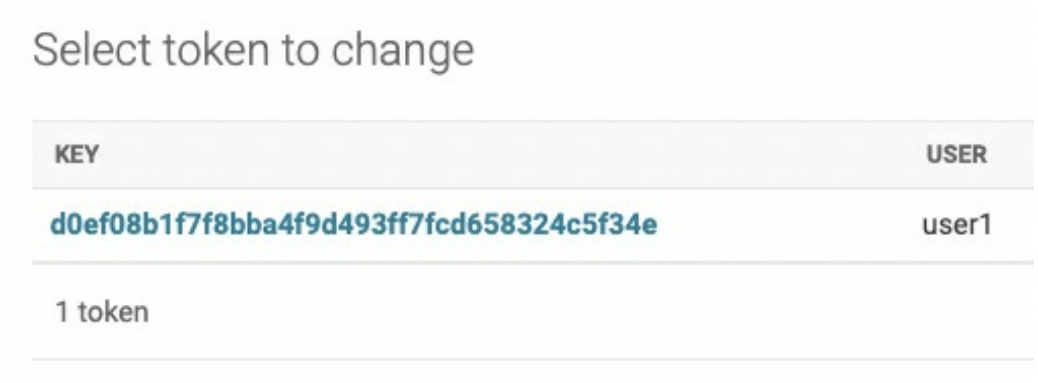

Figure 7

So if a request has a token, our server can connect it with a particular user. With the token, we can execute create, list, update and delete todo operations.

## *Testing with cURL*

To illustrate this, we will use the cURL command line utility to execute HTTP requests with an authorization token. cURL is usually available by default on Macs but requires some installation on Windows. Follow the instructions in https://curl.se/ for installing curl.

In the Terminal, run:

**Execute in Terminal**

```
curl http://localhost:8000/api/todos/ -H 'Authorization:Token <token>'
```

E.g.:

**Analyze Code**

```
curl http://localhost:8000/api/todos/ -H 'Authorization:Token d0ef08b1f7f8bba4f9d493ff7fcd658324c5f34e'
```

And you should be able to request the list of todos for that user through token authorization.

# Chapter 12: Authentication – Log In Tokens

We have implemented signing up of a new user, and generating a new token for them. But how do we retrieve a token for an existing user who logs in?

Let's first create a url for them to login and retrieve a token. In *todobackend/api/urls.py*, create the path:

**Modify Bold Code**

```
from django.urls import path
from . import views


urlpatterns = [
    path('todos/', views.TodoListCreate.as_view()),
    path('todos/<int:pk>', views.TodoRetrieveUpdateDestroy.as_view()),
    path('todos/<int:pk>/complete', views.TodoToggleComplete.as_view()),
    path('signup/', views.signup),
    path('login/', views.login),
]
```

We next implement the login view in *todobackend/api/views.py*:

**Modify Bold Code**

```
...
from django.views.decorators.csrf import csrf_exempt
from django.contrib.auth import authenticate


...


@csrf_exempt
def signup(request):
    ...


@csrf_exempt
def login(request):
    if request.method == 'POST':
        data = JSONParser().parse(request)
        user = authenticate(
            request,
            username=data['username'],
            password=data['password'])
        if user is None:
            return JsonResponse(
                                {'error':'unable to login. check username and password'},
                status=400)
        else: # return user token
            try:
                token = Token.objects.get(user=user)
            except: # if token not in db, create a new one
                token = Token.objects.create(user=user)
            return JsonResponse({'token':str(token)}, status=201)
```

## *Code Explanation*

The code in general is similar to sign up.

**Analyze Code**

```
def login(request):
    if request.method == 'POST':
```

We check if a request was performed using the HTTP 'POST' method because the login form in the front end will use POST requests for form submissions.

**Analyze Code**

```
        data = JSONParser().parse(request)
        user = authenticate(
            request,
            username=data['username'],
                        password=data['password'])
```

We then call *JSONParse().parse* to parse the JSON request content and return a *dictionary* of data.

We extract the user filled-in values from the dictionary with *data['username']* and *data['password']* and pass them into the *authenticate* method.

**Analyze Code**

```
        if user is None:
```

```
        return JsonResponse(
                        {'error':'unable to login. check username and password'},
            status=400)
```

If no user is found, we return a *JsonResponse* object with a dictionary containing an error message and status code of 400 indicating that the request cannot be fulfilled.

**Analyze Code**

```
    else: # return user token
        try:
            token = Token.objects.get(user=user)
        except: # if token not in db, create a new one
            token = Token.objects.create(user=user)
        return JsonResponse({'token':str(token)}, status=201)
```

Else, it means that the authentication is successful and we found a user. We then retrieve an existing user token with *Token.objects.get(user=user)* or create a new one if the token is not in the database.

## *Testing Our App*

Let's now test our login endpoint. In Insomnia, send a POST request to the login URL http://localhost:8000/api/login/ with the JSON object containing *username* and *password* (fig. 1).

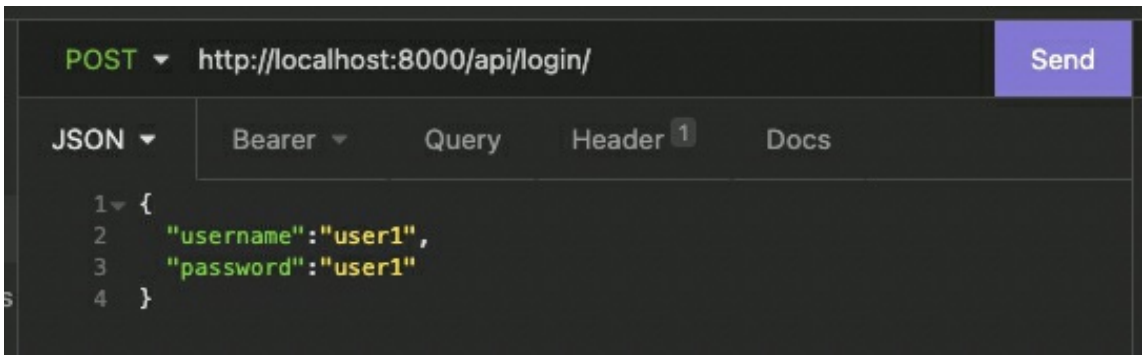


Figure 1

After sending the request, the response should return an authorization token. With it, you can then run in the Terminal:

**Execute in Terminal**

```
curl http://localhost:8000/api/todos/ -H 'Authorization:Token <token>'
```

to retrieve todos.

You can also use cURL to complete todos e.g.

**Execute in Terminal**

```
curl -X "PUT" http://localhost:8000/api/todos/2/complete -H 'Authorization:Token <token>'
```

Now that we have implemented and tested our backend API endpoints, let's proceed on to implement the React frontend in the next chapter!

# React Frontend

# Chapter 13: Introduction to React

For those who have some experience with React, this section will be familiar to you. But even if you are new to React, you should still be able to follow along. If you are interested in digging into React details, you can check out my React book.

Before we go on further, let ' s explain briefly what is React. React is a framework released by Facebook for creating user interfaces with components. For example, if we want to build a storefront module like what we see on Amazon, we can divide it into three components. The search bar component, sidebar component and products component (fig. 1).

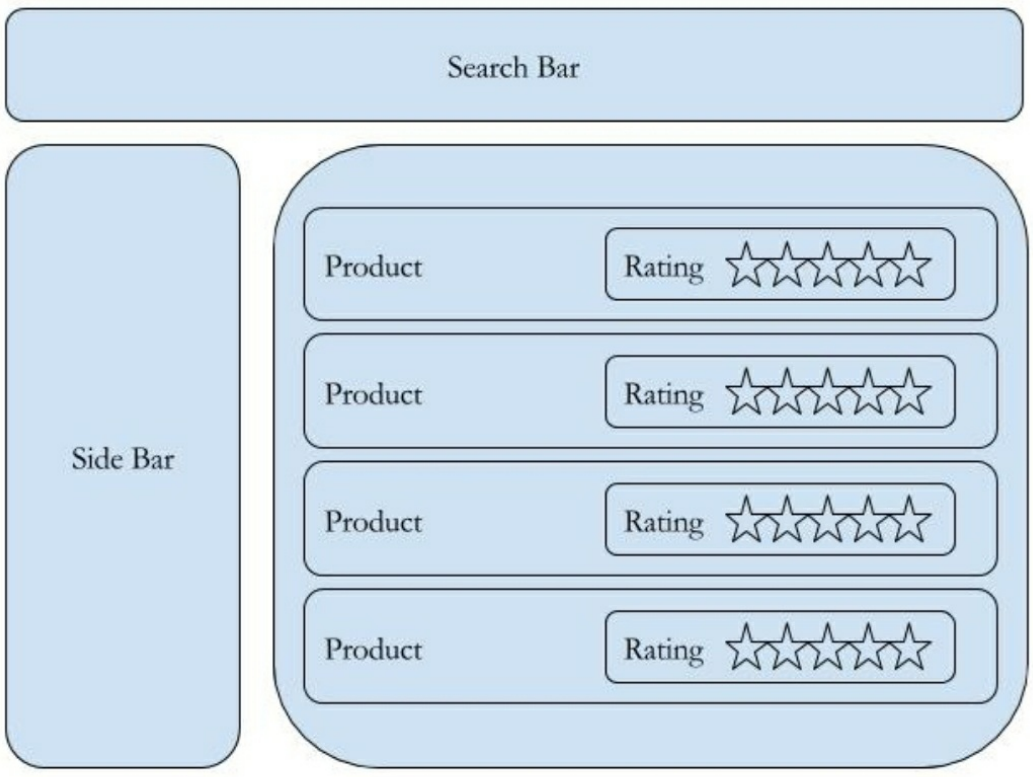


Figure 1

Components can also contain other components. For example, in *products* component where we display a list of products, we do so using multiple *product* components. Also, in each *product* component, we can have a *rating* component.

The benefit of such an architecture helps us to break up a large application into smaller manageable components. Plus, we can reuse components within the application or even in a different application. For example, we can re-use the rating component in a different application.

A React component contains a JSX template that ultimately outputs HTML elements. It has its own data and logic to control the JSX template. When the values of its data changes, React will update the concerned UI component.

Below is an example of a component that displays a simple string ' Products ' .

**Analyze Code**

```
import React from 'react';

function Products() {
   return (
     <div>
       <h2>
          Products
       </h2>
     </div>
   );
}

export default Products;
```

The function returns a React element in JSX syntax which determines what is displayed in the UI.
JSX is a syntax extension to Javascript. JSX converts to HTML when processed.

## Creating the React Project folder

We will create our initial React project by using ‘ create-react-app ’ . ‘ *create-react-app* ’ (CRA) is the best way to start building a new React single page application. It sets up our development environment so that we can use the latest Javascript features and optimization for our app. It is a Command Line Interface tool that makes creating a new React project, adding files and other on-going development tasks like testing, bundling and deployment easier. It uses build tools like Babel and Webpack under the hood and provides a pleasant developer experience for us that we don ’ t have to do any manual configurations for it.

First, let ’ s go to our *todoapp* directory and in it, we will use *create-react-app* to create our React app with:

**Analyze Code**

```
npx create-react-app <project name>
```

In our case, our project will be called *frontend*. So run:

**Execute in Terminal**

```
npx create-react-app frontend
```

Note: The reason we are using *npx* is because *create-react-app* is a package expected to be run only once in our project. So it is more convenient to run it only once rather than downloading it on to our machine and then use it.

*create-react-app* will create a directory ‘ frontend ’ containing the default React project template with all the dependencies installed. At this point, *todoapp* will have two folders:

- *todobackend* (containing backend related files) and
- *frontend* (containing frontend related files)

When the ‘ frontend ’ folder is created, navigate to it by typing in the Terminal:

**Execute in Terminal**

```
cd frontend
```

and then run:

**Execute in Terminal**

```
npm start
```

Your browser will then show a moving React icon (fig. 2) which shows that React is loaded successfully.

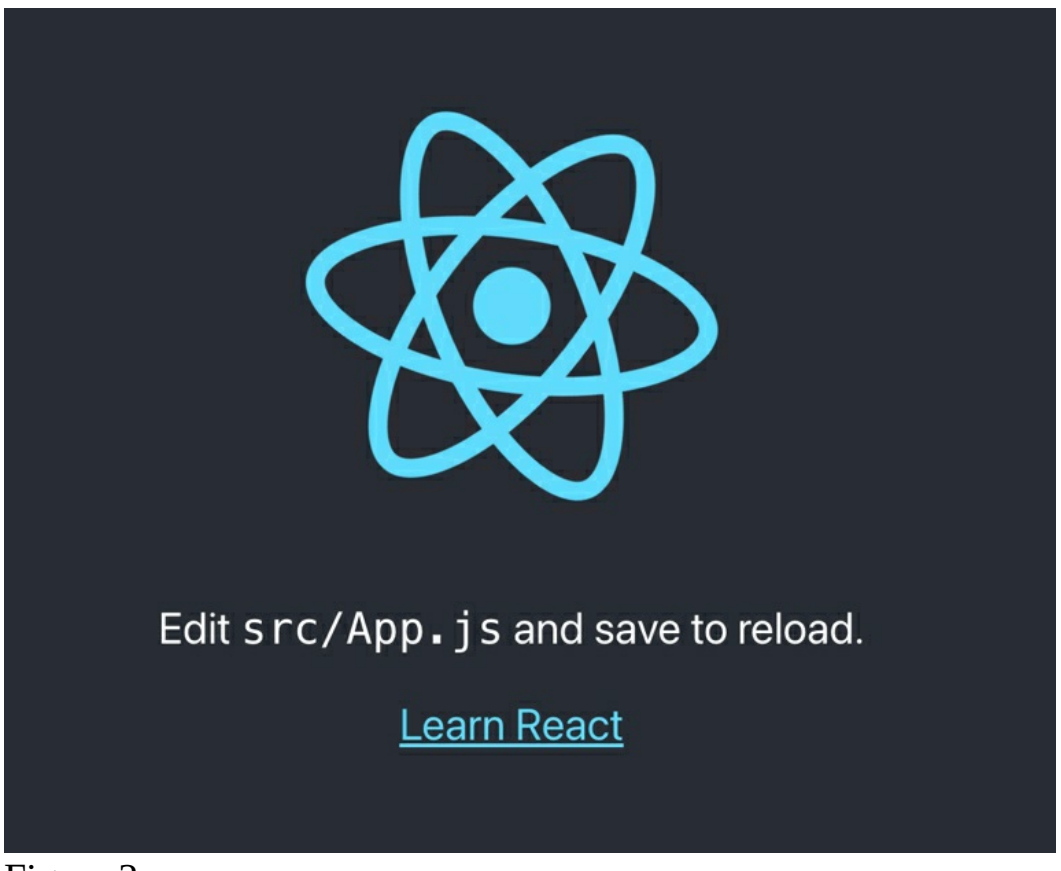


Figure 2

## Project File Review

Now let’s look at the project files that have been created for us. When you open the *todoapp/frontend* project folder in VScode editor, you will find a couple of files (fig. 3).

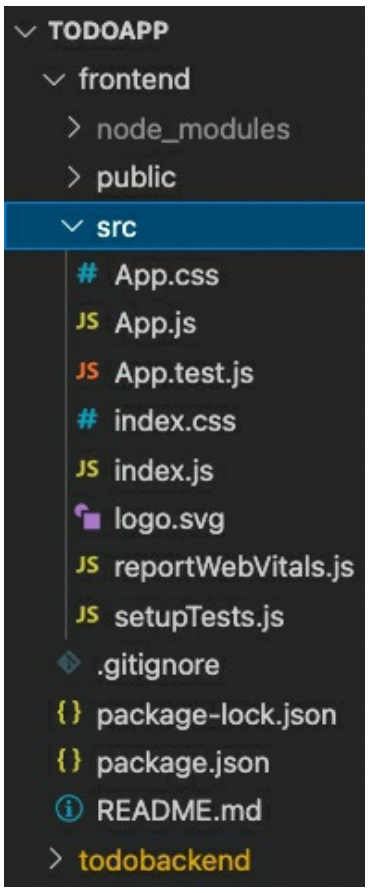


fig. 3

We will not go through all the files as our focus is to get started with our React app quickly, but we will briefly go through some of the more important files and folders.

Our app lives in the *src* folder. All React components, CSS styles, images (e.g. logo.svg) and anything else our app needs go here. Any other files outside of this folder are meant to support building your app (the app folder is where we will work 99% of the time!). In the course of this book, you will come to appreciate the uses for the rest of the library files and folders.

In the *src* folder, we have *index.js* which is the main entry point for our app. In *index.js*, we render the *App* React element into the root DOM node. Applications built with just React usually have a single root DOM node.

## *index.js*

**Analyze Code**

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';


ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);


...
reportWebVitals();
```

In *index.js*, we import both React and ReactDOM which we need to work with React in the browser. React is the library for creating views. ReactDOM is the library used to render the UI in the browser. The two libraries were split into two packages for version 0.14 and the purpose for splitting is to allow for components to be shared between the web version of React and React Native, thus supporting rendering for a variety of platforms.

*index.js* imports *index.css*, App component and *reportWebVitals* with the following lines.

**Analyze Code**

```
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
```

It then renders App with:

**Analyze Code**

```
ReactDOM.render(
  <React.StrictMode>
    <App />
```

```
  </React.StrictMode>,
  document.getElementById('root')
);
```

The last line `reportWebVitals()` has comments:

**Analyze Code**

```
// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
```

This however is out of the scope of this book and we can safely leave the code as it is.[DCB1][JL2]

*App.js* is the main React code that we display on the page.

## *App.js*

**Analyze Code**

```
import logo from './logo.svg';
import './App.css';


function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}


export default App;
```

Note: any element that has an HTML class attribute is using *className* for that property instead of *class*. Since *class* is a reserved word in Javascript, we have to use *className* to define the class attribute of an HTML element.

In the above, we have a functional-based component called *App*. Every React application has at least one component: the root component, named *App* in *App.js*. The App component controls the view through the JSX template it returns:

**Analyze Code**

```
 return (
    <div className="App">
      …
    </div>
);
```

A component has to return a **single** React element. In our case, *App* returns a single * *. The element can be a representation of a native DOM component, such as * *, or another composite component that you've defined yourself.

Components can either be *functional* based or *class* based. We will talk more on this later, but as a starter, what we have in *App* is a functional-based component as seen from its header *function App()*.

# Add React bootstrap framework:

We will use React *bootstrap* to make our UI look more professional. React Bootstrap (https://react-bootstrap.github.io) is a library of reusable frontend components that contain JSX based templates to help build user interface components (like forms, buttons, icons) for web applications.

To install React bootstrap, go to the ‘frontend’ folder, and in the Terminal, run:

**Execute in Terminal**

```
npm install react-bootstrap bootstrap
```

# React-Router-DOM

We will next install *react-router-dom* to route different URLs to different pages in our React app. The *React Router* library interprets a browser URL as an instruction to navigate to various components.

We can bind the router to links on a page and it will navigate to the appropriate application view when the user clicks a link.

Install the *react-router-dom* library by executing the below in the Terminal:

**Execute in Terminal**

```
npm install --save react-router-dom@5.2.0
```

(Note that we are installing React router version 5.2)

## *Test our App*

Now, let’s make sure that everything is working so far. Fill in *App.js* with the below code.

**Replace Entire Code**

```
import React from 'react';
import { Switch, Route, Link } from 'react-router-dom';
import 'bootstrap/dist/css/bootstrap.min.css';

function App() {
  return (
    <div className="App">
      Hello World
    </div>
  );
}

export default App;
```

## *Code Explanation*

*Switch*, Route and *Link* are imported from the ‘react-router-dom’ library which helps us create different URL routes to different components we will later use.
Bootstrap as mentioned earlier provides styling to our whole app.
And in the *return* method, we have a single and simple component with the message ‘Hello World’.

## *Test Run*

To test run our app, go to the *frontend* directory in the Terminal and run:

**Execute in Terminal**

```
npm start
```

It will then open up *localhost:3000* in your browser and print out the message ‘  Hello World ’ (fig. 4).

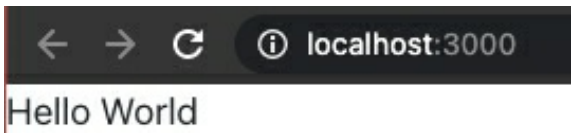

Figure 4

So our React app ’ s running well. In the next chapter, we will create a navigation header bar in our app.

# Chapter 14: Create Navigation Header Bar

Let ’ s add a navigation header bar which allows a user to select different routes to access different components in the main part of the page. We will start by creating some simple components and our router will load the different components depending on the URL route a user selects.

Let ’ s first create a *components* folder in *src* (fig. 1).

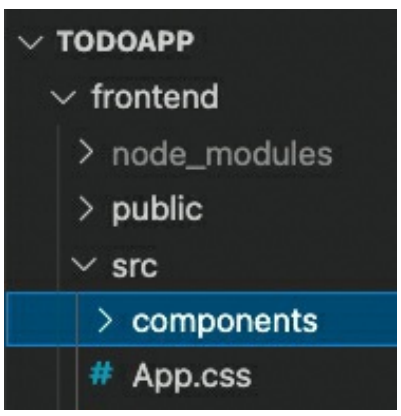

Figure 1

In the *components* folder, we will create four new component files:

*todos-list.js* – a component to list todos
*add-todo.js* – component to add a todo
*login.js* – login component
*signup.js* – sign up component

Let ’ s first have a simple boilerplate code for each component:

## *todos-list.js*

**Add Code**

```
import React from 'react';


function TodosList() {
  return (
    <div className="App">
      Todos List
    </div>
  );
}


export default TodosList;
```

## *add-todo.js*

**Add Code**

```
import React from 'react';


function AddTodo() {
  return (
     <div className="App">
      Add Todo
    </div>
  );
}


export default AddTodo;
```

## *login.js*

**Add Code**

```
import React from 'react';


function Login() {
  return (
     <div className="App">
      Login
    </div>
  );
}
```

```
export default Login;
```

*signup.js*

**Add Code**

```
import React from 'react';


function Signup() {
  return (
     <div className="App">
      Sign up
    </div>
  );
}


export default Signup;
```

We will later revisit the above components and implement them in greater detail.

Next in *App.js*, import the newly created components:

**Modify Bold Code**

```
import React from 'react';
import { Switch, Route, Link } from 'react-router-dom';
import 'bootstrap/dist/css/bootstrap.min.css';

import AddTodo from './components/add-todo';
import TodosList from './components/todos-list';
import Login from './components/login';
import Signup from './components/signup';


function App() {
  return (
    <div className="App">
      Hello World
    </div>
  );
}


export default App;
```

*React-Bootstrap Navbar Component*

Next, we will grab a navbar component from React-Bootstrap (https://react-bootstrap.github.io/components/navbar/ fig. 2)

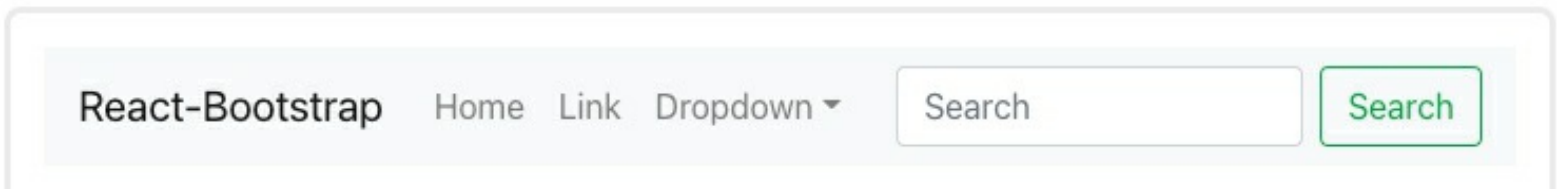


Figure 2

Paste the markup into *App.js* by adding the following codes:

**Modify Bold Code**

```
…
import Signup from './components/signup';


import Nav from 'react-bootstrap/Nav';
import Navbar from 'react-bootstrap/Navbar';


function App() {
  return (
    <div className="App">
      <Navbar bg="primary" variant="dark">
        <div className="container-fluid">
          <Navbar.Brand>React-bootstrap</Navbar.Brand>
          <Nav className="me-auto">
            <Nav.Link href="#home">Home</Nav.Link>
            <Nav.Link href="#link">Link</Nav.Link>
          </Nav>
        </div>
      </Navbar>
    </div>
  );
```

```
}

export default App;
```

Bootstrap has different components that you can use. To use a component, go to the Bootstrap documentation (*https://react-bootstrap.github.io/*), copy the component ' s markup and update it for your own purposes.

Note that we have dropped the *NavDropdown* and *Search* form elements from the *Navbar* for simplicity. So we just have a basic bootstrap navbar. If you run the app now, it should give you something like in figure 3:

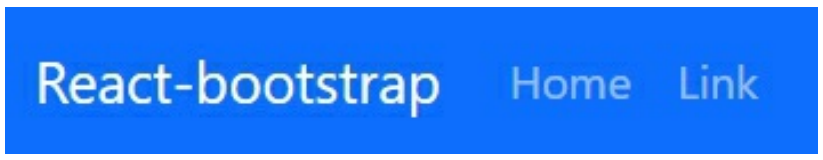


Figure 3

In the current navbar, we have three links. The first is ' React-bootstrap ' which is like the brand of the website. Sometimes, this would be a logo, image, or just some text. We will leave it as a text.

The other two are links to ' Home ' and ' Link ' . We will change ' Home ' to ' Todos ' and link it to ' /todos. We will remove ' Link ' and replace it with ' Login ' or ' Logout ' depending on the user ' s login state. We also add a ' Sign Up ' link.

So make the following changes in **bold**:

**Modify Bold Code**

```
…
import Navbar from 'react-bootstrap/Navbar';
import Container from 'react-bootstrap/Navbar';


function App() {
 const user = null;


 return (
  <div className="App">
   <Navbar bg="primary" variant="dark">
    <div className="container-fluid">
     <Navbar.Brand>TodosApp</Navbar.Brand>
     <Nav className="me-auto">
      <Container>
       <Link class="nav-link" to={"/todos"}>Todos</Link>
       { user ? (
        <Link class="nav-link">Logout ({user})</Link>
       ) : (
        <>[DCB3]
         <Link class="nav-link" to={"/login"}>Login</Link>
         <Link class="nav-link" to={"/signup"}>Sign Up</Link>
        </>
       )}
      </Container>
     </Nav>
    </div>
   </Navbar>
  </div>
 );
}

export default App;
```

## *Code Explanation*

**Analyze Code**

```
        <Link class="nav-link" to={"/todos"}>Todos</Link>
```

We use the *Link* component imported from *react-router-dom*. *Link* allows us to route to a different component. So when a user clicks on ' Todos ' , it will route to the *TodosList* component. The actual route definition will be implemented and explained in the next chapter.

**Analyze Code**

```
        { user ? (
          <Link class="nav-link">Logout ({user})</Link>
        ) : (
          <>[DCB4]
            <Link class="nav-link" to={"/login"}>Login</Link>
            <Link class="nav-link" to={"/signup"}>Sign Up</Link>
```

```
      </>
    )}
```

*Note: We had to use wrap the links in <> instead of  to avoid the two links merging in one column.

For the second link, if the user is not logged in, we show ‘ Login ’ which links to the login component and ‘ Sign Up ’ which links to the sign up component.
If the user is logged in, it will show ‘ Logout User ’ which links to the logout component.

How do we achieve this conditional rendering? In React, we can use curly braces ‘ {} ’ to put in code. The code is a ternary statement where if it is true, execute the section after the '?'. If false, execute the section after the colon ':'. For e.g. if you hardcode *user* to true, it will always show ‘ Logout User ’ : e.g.

**Modify Bold Code**

```
{ ~~user~~ true ? (
  <Link class="nav-link">Logout ({user})</Link>
) : (
  <>
    <Link class="nav-link" to={"/login"}>Login</Link>
    <Link class="nav-link" to={"/signup"}>Sign Up</Link>
  </>
)}
```

Let ’ s test our app now to see how it looks like. But before that, we need to enclose our *Link*s in a *BrowserRouter*. To do so, in *index.js*, add:

**Modify Bold Code**

```
…
import App from './App';
import { BrowserRouter } from 'react-router-dom';
…

ReactDOM.render(
  <React.StrictMode>[DCB5][JL6]
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>,
  document.getElementById('root')
);

…
```

And if you run your app, it should look like figure 4:


Figure 4

If you change *user* to *false*:

**Modify Bold Code**

```
{ ~~user~~ false ? (
  <Link class="nav-link">Logout ({user})</Link>
) : (
  <>
    <Link class="nav-link" to={"/login"}>Login</Link>
    <Link class="nav-link" to={"/signup"}>Sign Up</Link>
  </>
)}
```

it will show the *Login* and *Sign Up* links (fig. 5).


Figure 5

We should of course not leave it hard-coded as true or false. Make sure you change it back to *user*:

**Modify Bold Code**

```
{ user ? (
  <Link class="nav-link">Logout ({user})</Link>
) : (
  <>
    <Link class="nav-link" to={"/login"}>Login</Link>
    <Link class="nav-link" to={"/signup"}>Sign Up</Link>
  </>
)}
```

## *Login Logout*

In this section, we will implement a preliminary login, logout and signup function and reflect the login state of a user. Let ’ s first declare a *user* and *token* state variable using React hooks by adding the below in *App.js*:

**Modify Bold Code**

```
…
function App() {
  const user = null;
  const [user, setUser] = React.useState(null);
  const [token, setToken] = React.useState(null);
  const [error, setError] = React.useState('');

  async function login(user = null){ // default user to null
    setUser(user);
  }

  async function logout(){
    setUser(null);
  }

  async function signup(user = null){ // default user to null
    setUser(user);
  }
…
```

## *Code Explanation*

**Analyze Code**

```
const [user, setUser] = React.useState(null);
```

*React.useState* is a ‘ hook ’ that lets us add some local state to functional components. *useState* declares a ‘ state variable ’ . React preserves this state between re-renders of the component. In our case, our state consists of a *user* variable to represent either a logged in or logged out state. When we pass null to *useState*, i.e. *useState(null)*, we specify *null* to be the initial value for *user*.

*useState* returns an array with two values: the current state value and a function that lets you update it. In our case, we assign the current state user value to *user*, and the function to update it to *setUser*.

**Analyze Code**

```
const [token, setToken] = React.useState(null);
const [error, setError] = React.useState('');
```

We also have a *token* state variable to represent the authorization token for the user and an *error* state variable to store any error messages.

**Analyze Code**

```
async function login(user = null){ // default user to null
  setUser(user);
}

async function logout(){
  setUser(null);
}

async function signup(user = null){ // default user to null
  setUser(user);
}
```

With *login*, we set the *user* state. The *login* function will be called from the Login component which we will implement and re-visit later. *signup* for now works in a similar fashion. *logout()* simply sets *user* to null.

Later on, we will implement a full login system.

# Chapter 15: Defining Our Routes

After the navbar section in *App.js*, add the *route* section by adding the below codes in **bold**:

**Modify Bold Code**

```
…
 return (
   <div className="App">
     <Navbar bg="primary" variant="dark">
     ...
     </Navbar>



     <div className="container mt-4">
       <Switch>
         <Route exact path={["/", "/todos"]} render={(props) =>
           <TodosList {...props} token={token} />
         }>
         </Route>
            <Route path="/todos/create" render={(props)=>
          <AddTodo {...props} token={token} />
         }>
         </Route>
            <Route path="/todos/:id/" render={(props)=>
          <AddTodo {...props} token={token} />
        }>
        </Route>
           <Route path="/login" render={(props)=>
          <Login {...props} login={login} />
        }>
        </Route>
           <Route path="/signup" render={(props)=>
          <Signup {...props} signup={signup} />
        }>
        </Route>
       </Switch>
     </div>
   </div>
…
```

## *Code Explanation*

We use a *Switch* component to switch between different routes. The *Switch* component renders the first route that matches.

**Analyze Code**

```
        <Route exact path={["/", "/todos"]} render={(props) =>
          <TodosList {...props} token={token} />
        }>
        </Route>
```

We first have the exact path route. If the path is “/” or “/todos”, show the *TodosList* component. We use *render* to allows us to pass data into a component by passing in an object called *props*. Specifically, we pass in the token for the user to authenticate itself with the backend API. We will see later how props work when we implement the *AddTodo* component.

**Analyze Code**

```
           <Route path="/todos/create" render={(props)=>
          <AddTodo {...props} token={token} />
        }>
        </Route>
```

We next have the route for `"/todos/create"` to render the *AddTodo* component. We pass in *token* because only authenticated users can create todos.

**Analyze Code**

```
           <Route path="/todos/:id/" render={(props)=>
          <AddTodo {...props} token={token} />
        }>
        </Route>
```

We then have the route for `"/todos/:id/"` which also renders the *AddTodo* component but this time, it’s to update the todo item. Again, we pass in *token* because only authenticated users can update todos.

We next have the routes for `"/login"` and “/signup” to render the *Login* and *Signup* component respectively. We do

not need to pass in the token because these two operations don’t require authentication.

**Analyze Code**

```
        <Route path="/login" render={(props)=>
      <Login {...props} login={login} />
    }>
    </Route>
        <Route path="/signup" render={(props)=>
      <Signup {...props} signup={signup} />
    }>
    </Route>
```

Note that the login route passes in the *login* function as a prop:

**Analyze Code**

```
async function login(user = null){ // default user to null
  setUser(user);
}
```

This allows the *login* function to be called from the *Login* component and thus populate the *user* state variable as we will see later. The same applies for *signup*.

## *Testing our Routes*

If you run your React frontend now and click on the different links in the navbar, you will see the different components being rendered (fig. 1).

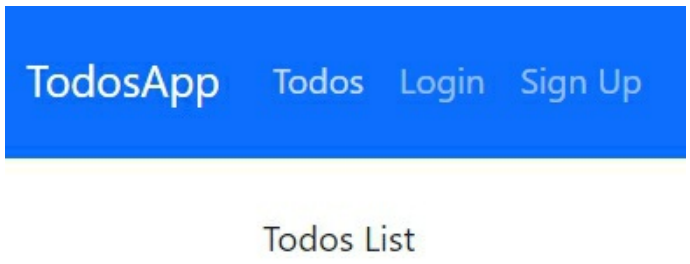


Figure 1

## *Creating Footer Section*

In *App.js*, we make the following changes in **bold** to create the footer section.

**Modify Bold Code**

```
…
 return (
   <div className="App">
     <Navbar bg="primary" variant="dark">
     ...
     </Navbar>



     <div className="container mt-4">
       …
     </div>

     <footer className="text-center text-lg-start
      bg-light text-muted mt-4">
      <div className="text-center p-4">
          © Copyright - <a
                    target="_blank"
                    className="text-reset fw-bold text-decoration-none"
        href="https://twitter.com/greglim81"
       >
        Greg Lim
          </a> - <a
                    target="_blank"
                    className="text-reset fw-bold text-decoration-none"
        href="https://twitter.com/danielgarax"
       >
        Daniel Correa
       </a>
      </div>
     </footer>
   </div>
…
```

The footer section displays a grey div in where we place the book authors ’ names, with a link to their respective

Twitter account links. If you run the app now, it should give you something like in figure 2:

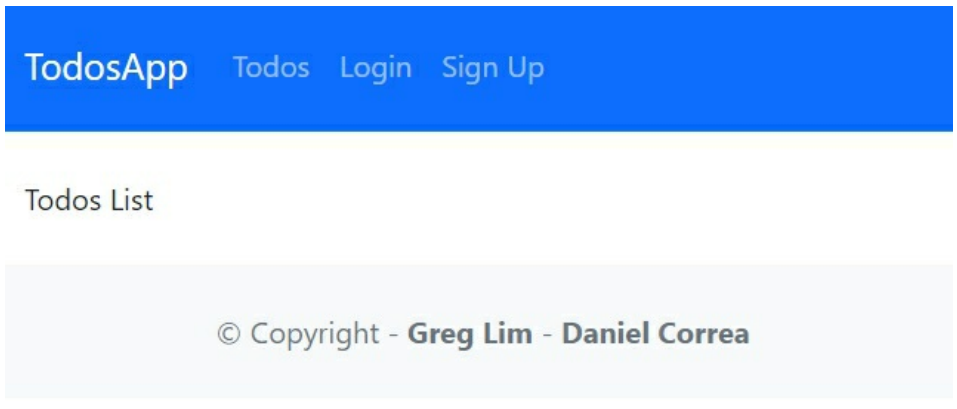


Figure 2

# Chapter 16: TodoDataService - Connecting to the Backend

To retrieve the list of todos from the database, we will need to connect to our backend server. We will create a *service* class for that. A service is a class with a well-defined specific function your app needs. In our case, our service is responsible for talking to the backend to get and save todo data. Service classes provide their functionality to be consumed by components. We will cover components in the next few chapters.

Under *src*, create a new folder called *services*. In it, create a new file *todos.js* with the following code:

**Add Code**

```
import axios from 'axios';

class TodoDataService{

 getAll(token){
   axios.defaults.headers.common["Authorization"] = "Token " + token;
   return axios.get('http://localhost:8000/api/todos/');[DCB7]
 }

 createTodo(data, token){
   axios.defaults.headers.common["Authorization"] = "Token " + token;
   return axios.post("http://localhost:8000/api/todos/", data);
 }

 updateTodo(id, data, token){
   axios.defaults.headers.common["Authorization"] = "Token " + token;
   return axios.put(`http://localhost:8000/api/todos/${id}`, data);
 }

 deleteTodo(id, token){
   axios.defaults.headers.common["Authorization"] = "Token " + token;
   return axios.delete(`http://localhost:8000/api/todos/${id}`);
  }

 completeTodo(id, token){
   axios.defaults.headers.common["Authorization"] = "Token " + token;
     return axios.put(`http://localhost:8000/api/todos/${id}/complete`);
  }

 login(data){
   return axios.post("http://localhost:8000/api/login/", data);
  }

 signup(data){
   return axios.post("http://localhost:8000/api/signup/", data);
  }
}

export default new TodoDataService();
```

## *Code Explanation*

**Analyze Code**

```
import axios from 'axios';
```

We use a library called *axios* for sending *get*, *post*, *put* and *delete* requests. Let’s first install *axios*, go to the ‘frontend’ folder, and in Terminal, run:

**Execute in Terminal**

```
npm install axios
```

The class *TodoDataService* contains functions which make the API calls to the backend endpoints we implemented earlier and return the results.

**Analyze Code**

```
 getAll(token){
   axios.defaults.headers.common["Authorization"] = "Token " + token;
   return axios.get('http://localhost:8000/api/todos/');
 }
```

*getAll* returns all the todos for a given user. We attach the token as our authorization header for our axios request. We then put the API URL into the *axios.get* method. Remember back in *todobackend/api/urls.py* we declared the following endpoint for updating of todos:

**Analyze Code**

```
…
urlpatterns = [
   path('todos/', views.TodoListCreate.as_view()),
   …
```

This endpoint is served by the *TodoListCreate* view in *todobackend/api/views.py* (refer to chapter 6).

**Analyze Code**

```
createTodo(data, token){
  axios.defaults.headers.common["Authorization"] = "Token " + token;
  return axios.post("http://localhost:8000/api/todos/", data);
}
```

*createTodo* is similar to *getAll*. We attach the token and use the same API URL but because we are creating a todo, we use the *axios.post* method. Note how with the same *TodoListCreate* API endpoint, by sending different HTTP types (e.g. get, post), we can perform different operations.

**Analyze Code**

```
updateTodo(id, data, token){
  axios.defaults.headers.common["Authorization"] = "Token " + token;
  return axios.put(`http://localhost:8000/api/todos/${id}`, data);
}
```

*updateTodo* is similar except we use the *axios.put* method and append the todo's id at the end of the API URL endpoint. Remember back in *todobackend/api/urls.py* we declared the following endpoint for updating of todos:

**Analyze Code**

```
…
urlpatterns = [
   path('todos/', views.TodoListCreate.as_view()),
   path('todos/<int:pk>', views.TodoRetrieveUpdateDestroy.as_view()),
   …
```

This endpoint is served by the *TodoRetrieveUpdateDestroy* view in *todobackend/api/views.py* (refer to chapter 8).

**Analyze Code**

```
deleteTodo(id, token){
  axios.defaults.headers.common["Authorization"] = "Token " + token;
  return axios.delete(`http://localhost:8000/api/todos/${id}`);
 }
```

*deleteTodo* is similar to *updateTodo* except we use *axios.delete*. The same *TodoRetrieveUpdateDestroy* endpoint performs different operations with different HTTP types (i.e. put, delete).

**Analyze Code**

```
completeTodo(id, token){
  axios.defaults.headers.common["Authorization"] = "Token " + token;
  return axios.put(`http://localhost:8000/api/todos/${id}/complete`);
 }
```

completeTodo is similar too except we use the *axios.put* method (we update the *completed* Boolean variable) and append the todo's id with 'complete' at the end of the API URL endpoint. Remember back in *todobackend/api/urls.py* we declare the following endpoint for completing of todos:

**Analyze Code**

```
…
urlpatterns = [
   path('todos/', views.TodoListCreate.as_view()),
   path('todos/<int:pk>', views.TodoRetrieveUpdateDestroy.as_view()),
   path('todos/<int:pk>/complete', views.TodoToggleComplete.as_view()),
   …
```

This endpoint is served by the *TodoToggleComplete* view in *todoapproj/api/views.py* (refer to chapter 9).

**Analyze Code**

```
login(data){
  return axios.post("http://localhost:8000/api/login/", data);
 }


signup(data){
  return axios.post("http://localhost:8000/api/signup/", data);
 }
```

The remaining *login* and *signup* methods connect to the login and signup views in the backend. We will later revisit the entire flow from the frontend to the backend after implementing the React frontend.

# Chapter 17: TodoDataService - Login Component

We will first implement a login authentication system in this chapter, as without it, we can't allow users to log in and show their todos. In the next chapter, we will list the todos for each user.

Fill in *login.js* with the following code:

**Replace Entire Code**

```
import React, {useState} from 'react';
import Form from 'react-bootstrap/Form';
import Container from 'react-bootstrap/Container';
import Button from 'react-bootstrap/Button';


const Login = props => {


 const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");


 const onChangeUsername = e => {
   const username = e.target.value;
   setUsername(username);
  }


const onChangePassword = e => {
  const password = e.target.value;
  setPassword(password);
 }


const login = () => {
  props.login({username: username, password: password});
  props.history.push('/');
 }


return(
  <Container>
    <Form>
      <Form.Group className="mb-3">
        <Form.Label>Username</Form.Label>
          <Form.Control
                  type="text"
                  placeholder="Enter username"
                  value={username}
        onChange={onChangeUsername}
       />
      </Form.Group>
      <Form.Group className="mb-3">
        <Form.Label>Password</Form.Label>
          <Form.Control
                  type="password"
                  placeholder="Enter password"
                  value={password}
        onChange={onChangePassword}
        />
         </Form.Group>
       <Button variant="primary" onClick={login}>
         Login
       </Button>
       </Form>
   </Container>
 )
}

export default Login;
```

And in *App.js*, fill in the below codes:

**Modify Bold Code**

```
…
import Nav from 'react-bootstrap/Nav';
import Navbar from 'react-bootstrap/Navbar';
import Container from 'react-bootstrap/Navbar';

import TodoDataService from './services/todos';

function App() {
```

```
const [user, setUser] = React.useState(null);
const [token, setToken] = React.useState(null);
const [error, setError] = React.useState('');


async function login(user = null){ // default user to null
  TodoDataService.login(user)
      .then(response =>{
        setToken(response.data.token);
      setUser(user.username);
      localStorage.setItem('token', response.data.token);
      localStorage.setItem('user', user.username);
      setError('');
    })
    .catch( e =>{
      console.log('login', e);
        setError(e.toString());
    });
}
```

* Contact support@i-ducate.com for the source code if you prefer to copy and paste

## *Code Explanation*

**Analyze Code**

```
const [username, setUsername] = useState("");
 const [password, setPassword] = useState("");
```

Our simple login form consists of a username and password fields. We use the React *useState* hook to create the *username* and *password* state variables. The *username* and *password* state variables keep track of what a user has entered into the login form fields. *username* and *password* are default set to empty strings with *useState(“”)*.

**Analyze Code**

```
      <Form.Group className="mb-3">
        <Form.Label>Username</Form.Label>
          <Form.Control
                    type="text"
                    placeholder="Enter username"
          value={username}
          onChange={onChangeUsername}
        />
      </Form.Group>
```

In the *username* FormControl (in the Form JSX Markup), we set the field’s value to the *username* state variable. So this field will always reflect the value in *username*. We set *onChange* to *onChangeUsername*, so any value changes done by the user entering in the field will call *onChangeUsername* which will in turn update *username* state variable. In essence, we have double binded *username* field to the *username* state. The *password* FormControl works similarly.

**Analyze Code**

```
const login = () => {
  props.login({username: username, password: password});
  props.history.push('/');
}
```

When we click on the Login button, it calls *login*. Notice that we call *props.login*. But who passes this *login* function into the Login component? If you recall in *App.js*, we have the following route:

**Analyze Code**

```
          <Route path="/login" render={(props)=>
          <Login {...props} login={login} />
        }>
        </Route>
```

And *login* is defined in *App.js* as:

**Analyze Code**

```
async function login(user = null){ // default user to null
  TodoDataService.login(user)
      .then(response =>{
        setToken(response.data.token);
      setUser(user.username);
      localStorage.setItem('token', response.data.token);
      localStorage.setItem('user', user.username);
      setError('');
    })
    .catch( e =>{
      console.log('login', e);
```

```
        setError(e.toString());
    });
}
```

In *login*, we call *TodoDataService*'s *login* method which calls the Django API login endpoint:

**Analyze Code**

```
login(data){
  return axios.post("http://localhost:8000/api/login/", data);
 }
```

The login API returns an authorization token which we set to the *token* state variable, setToken(response.data.token) .

**Analyze Code**

```
      localStorage.setItem('token', response.data.token);
      localStorage.setItem('user', user.username);
```

Additionally, we store the token and username in local storage so that a user will not have to constantly re-login as she navigates away and returns to our application. Appreciate that with tokens, we do not need to store or pass the user password around.

**Analyze Code**

```
    .catch( e =>{
      console.log('login', e);
        setError(e.toString());
    });
```

If there are any errors, we store them in the *error* state variable.

So, from the *Login* component, we call the *login* function in *App.js* and set *App*'s *user* state. If necessary, we can pass on the logged-in *user* to other components e.g. *AddTodo*, *TodosList*.

### */component/login.js*

**Analyze Code**

```
const login = () => {
  props.login({username: username, password: password});
  props.history.push('/');
 }
```

After login, we redirect to the main page with props.history.push('/').

## CORS

Before running and testing our app, we have to deal with Cross-Origin Resource Sharing (CORS). there are potential security issues when a frontend interacts with a backend API hosted on a different domain or port, e.g. localhost:3000 and localhost:8000. The server thus has to use the *django-cors-headers* middleware to include specific HTTP headers to allow the client to determine if cross-domain requests are permitted.

To install *django-cors-headers*, run the command:

**Execute in Terminal**

```
pip3 install django-cors-headers[DCB8]
```

To add *django-cors-headers* to our project, in *todobackend/backend/settings.py*, add the following codes in **bold**:

**Modify Bold Code**

```
…

INSTALLED_APPS = [
    …
    'api',
    'rest_framework.authtoken',
    'corsheaders',
]

MIDDLEWARE = [
    …
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'corsheaders.middleware.CorsMiddleware',
]

…
```

```
CORS_ORIGIN_WHITELIST = [
  'http://localhost:3000',
]
```

We whitelisted *localhost:3000* which is our frontend.

### *Testing your App*

In your app, try logging in with an existing user account (fig. 1),

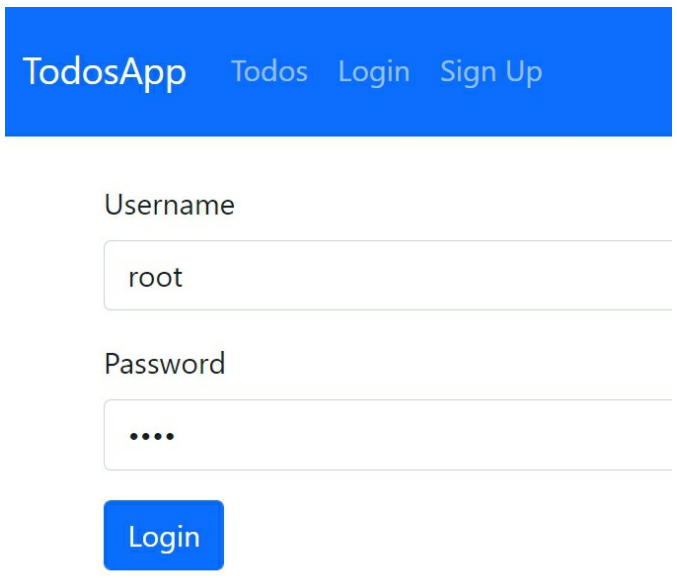


Figure 1

and you should be able to navigate to the todos home page without any errors. There are no todos listed at the moment. We will implement listing of todos in the next chapter.

## Logout

Having implemented login, let's now implement logout which is much simpler. In *App.js*, fill in the below codes into *logout*:

**Modify Bold Code**

```
…
 async function logout(){
   setToken('');
   setUser('');
   localStorage.setItem('token', '');
    localStorage.setItem('user', '');
 }
 …
```

We simply flush away the current token and logged in user. And in the logout button, add:

**Modify Bold Code**

```
…
          { user ? (
            <Link className="nav-link" onClick={logout}>
            Logout ({user})</Link>
          ) : (
```

Now, when you run your app and click on 'Logout' in the navbar, you will be logged out and see the login link instead. We should of course be displaying a message and login link to users who are not logged in. We will implement this in the next chapter.

Before we finish this chapter, let's implement signing up for an account which will be similar to login. In *App.js*, fill in the below codes into *signup*:

**Modify Bold Code**

```
…

 async function signup(user = null){ // default user to null
   TodoDataService.signup(user)
     .then(response =>{
       setToken(response.data.token);
       setUser(user.username);
       localStorage.setItem('token', response.data.token);
       localStorage.setItem('user', user.username);
     })
     .catch( e =>{
```

```
      console.log(e);
      setError(e.toString());
    })
}
```

Similar to *login*, *signup*s call *TodoDataService*'s *signup* method which calls the Django API login endpoint:

**Analyze Code**

```
signup(data){
  return axios.post("http://localhost:8000/api/signup/", data);
 }
```

The signup API returns an authorization token which we set to the *token* state variable, `setToken(response.data.token)`. Like *login*, we store the token and username in local storage.

Next in */components/signup.js*, fill it with the following code:

**Replace Entire Code**

```
import React, {useState} from 'react';
import Form from 'react-bootstrap/Form';
import Container from 'react-bootstrap/Container';
import Button from 'react-bootstrap/Button';


const Signup = props => {


const [username, setUsername] = useState("");
 const [password, setPassword] = useState("");


const onChangeUsername = e => {
  const username = e.target.value;
  setUsername(username);
 }


const onChangePassword = e => {
  const password = e.target.value;
  setPassword(password);
 }


const signup = () => {
  props.signup({username: username, password: password});
  props.history.push('/');
 }


return(
  <Container>
    <Form>
      <Form.Group className="mb-3">
        <Form.Label>Username</Form.Label>
          <Form.Control
                    type="text"
                    placeholder="Enter username"
                    value={username}
         onChange={onChangeUsername}
        />
      </Form.Group>
      <Form.Group className="mb-3">
        <Form.Label>Password</Form.Label>
          <Form.Control
                    type="password"
                    placeholder="Enter password"
                    value={password}
          onChange={onChangePassword}
        />
         </Form.Group>
      <Button variant="primary" onClick={signup}>
        Sign Up
      </Button>
      </Form>
   </Container>
 )
}


export default Signup;
```

* Contact support@i-ducate.com for the source code if you prefer to copy and paste

The signup form looks and work similar to the login form, so we won't explain it.

*Running your App*

If you run your app now, you will be able to sign up for new accounts and log in with it!

# Chapter 18: TodosList Component

Let’s now implement the *TodosList* component to consume the functionality in *TodoDataService*. Components are meant to be responsible for mainly rendering views supported by application logic for better user experience. They don’t fetch data from the backend but rather delegate such tasks to services.

We will carry on implementing our *TodosList* component. Fill in the below code into *todos-list.js*:

**Replace Entire Code**

```
import React, {useState, useEffect} from 'react';
import TodoDataService from '../services/todos';
import { Link } from 'react-router-dom';


const TodosList = props => {
  const [todos, setTodos] = useState([]);
}


export default TodosList;
```

We import *useState* to create a *todos* state variable. We import *useEffect* (which we will describe later) and also import *TodoDataService* and *Link.*

**Analyze Code**

```
const TodosList = props => {
  const [todos, setTodos] = useState([]);
}
```

*TodosList* is a functional component that receives and uses props. We use the React *useState* hook to create the *todos* state variables. Note that *todos* is default set to an empty array `useState([])` .

## *useEffect to retrieveTodos*

Next, we add the *useEffect* hook and the *retrieveTodos* as shown:

**Modify Bold Code**

```
…


const TodosList = props => {
  const [todos, setTodos] = useState([]);


  useEffect(() =>{
    retrieveTodos();
  }, [props.token]);


   const retrieveTodos = () => {
    TodoDataService.getAll(props.token)
        .then(response => {
        setTodos(response.data);
      })
      .catch( e => {
        console.log(e);
      });
   }
}


export default TodosList;
```

## *Code Explanation*

**Analyze Code**

```
  useEffect(() =>{
    retrieveTodos();
  }, [props.token]);
```

The *useEffect* hook is first called after the component renders. So if we want to tell the component to perform some code after rendering, we include it here. In our case, after the component renders, we call *retrieveTodos*(). We also specify *props.token* in the second argument array. This means that whenever *props.token* changes value, *useEffect* will be triggered as well. Thus, whenever the token value changes, (meaning a user logs out, or a new user logs in) we retrieve a fresh set of todos.

To summarize:

- if the 2nd argument of *useEffect* contains an array of variables and any of these variables change, *useEffect*

will be called.

- *useEffect* with an empty array in its second argument gets called only the first time the component renders.

**Analyze Code**

```
const retrieveTodos = () => {
  TodoDataService.getAll(props.token)
      .then(response => {
      setTodos(response.data);
    })
    .catch( e => {
      console.log(e);
    });
}
```

*retrieveTodos* calls *TodoDataService.getAll()* which if you remember, has the following implementation:

**Analyze Code**

```
getAll(token){
  axios.defaults.headers.common["Authorization"] = "Token " + token;
  return axios.get('http://localhost:8000/api/todos/');
}
```

*getAll* returns a promise with the todos retrieved from the database and we set it to the *todos* state variable with *setTodos(response.data)*.

### *JSX Markup for Displaying Todos*

Now, let's display the list of todos like in figure 1.

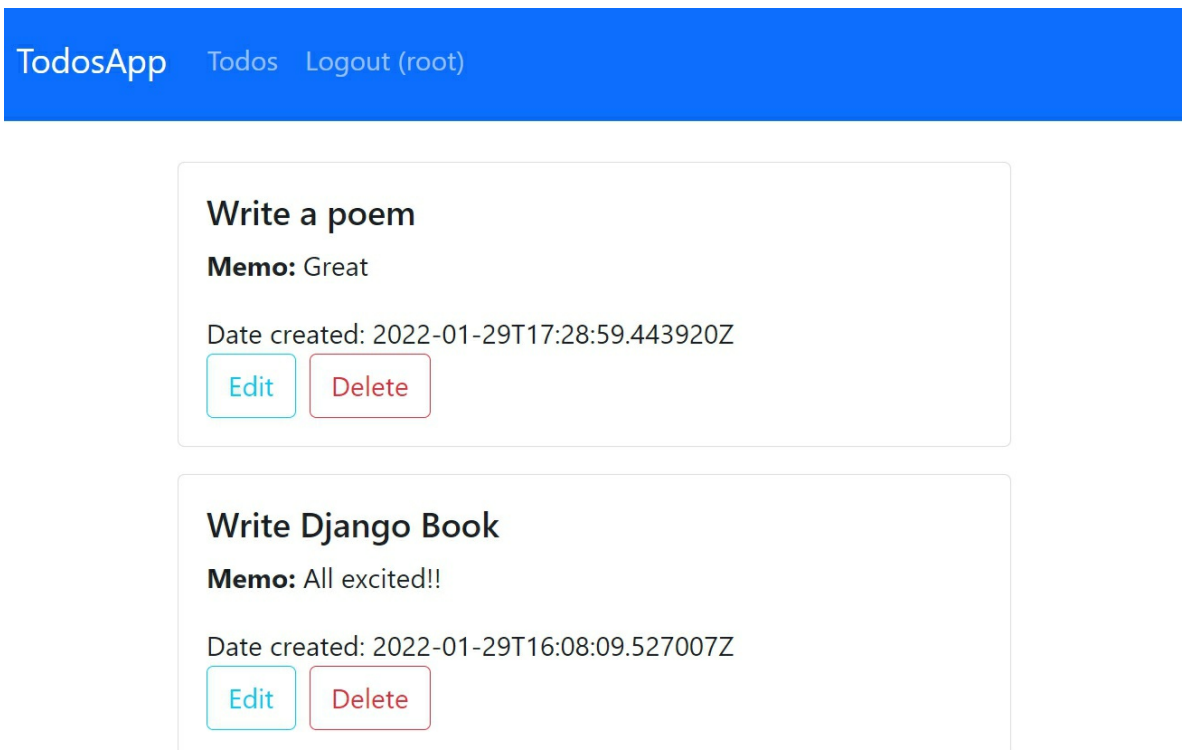


Figure 1

### *todos-list.js*

Add the code in **bold** below to *return*:

**Modify Bold Code**

```
…
import { Link } from 'react-router-dom';
import Card from 'react-bootstrap/Card';
import Container from 'react-bootstrap/Container';
import Button from 'react-bootstrap/Button';


const TodosList = props => {
  const [todos, setTodos] = useState([]);
  …



  return (
    <Container>
      {todos.map((todo) => {
        return (
          <Card key={todo.id} className="mb-3">
            <Card.Body>
              <div>
                <Card.Title>{todo.title}</Card.Title>
```

```
            <Card.Text><b>Memo:</b> {todo.memo}</Card.Text>
            <Card.Text>Date created: {todo.created}</Card.Text>
          </div>
          <Link to={{
            pathname: "/todos/" + todo.id,
            state: {
              currentTodo: todo
            }
          }}>
            <Button variant="outline-info" className="me-2">
              Edit
            </Button>
          </Link>
          <Button variant="outline-danger">
            Delete
          </Button>
        </Card.Body>
      </Card>
    )
  })}
 </Container>
 );
}

export default TodosList;
```

*Code Explanation*

We use the *map* function again where for each todo in the *todos* array, we render a *Card* component from React-bootstrap (https://react-bootstrap.github.io/components/cards/ - fig. 2).

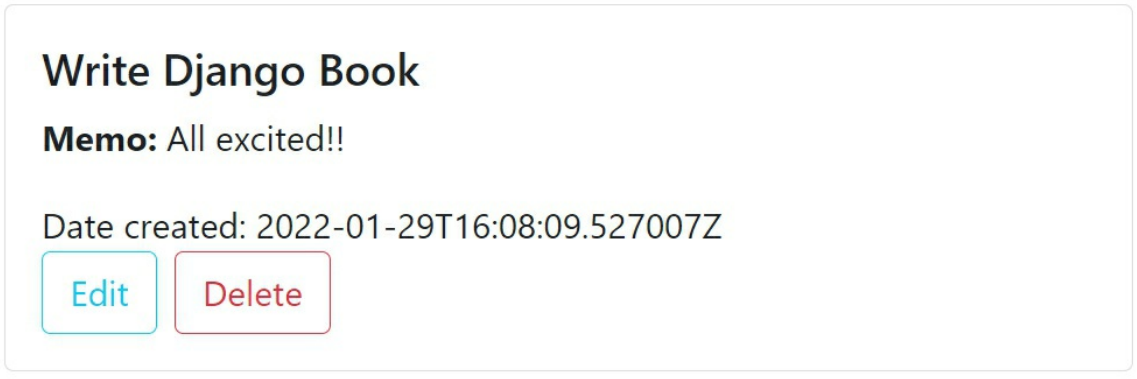


Figure 2

Each Card contains a todo with its:

- title: `<Card.Title>{todo.title}</Card.Title>`
- memo: `<Card.Text><b>Memo:</b> {todo.memo}</Card.Text>`
- created date: `<Card.Text>Date created: {todo.created}</Card.Text>`
- Edit Link:

```
<Link to={{
    pathname: "/todos/" + todo.id,
    state: {
        currentTodo: todo
    }
}}>
        <Button variant="outline-info"  className="me-2">
            Edit
        </Button>
    </Link>
```

- Delete Link:

```
<Button variant="outline-danger">
    Delete
</Button>
```

You can view all of the todo's properties back in the model file *todobackend/todo/models.py*:

**Analyze Code**

```
…


class Todo(models.Model):
    title = models.CharField(max_length=100)
    memo = models.TextField(blank=True)


    #set to current time
     created = models.DateTimeField(auto_now_add=True)
    completed = models.BooleanField(default=False)
```

```
#user who posted this
  user = models.ForeignKey(User, on_delete=models.CASCADE)


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

*Testing your App*

When you run your app now, it should return a list of todos (fig. 3):

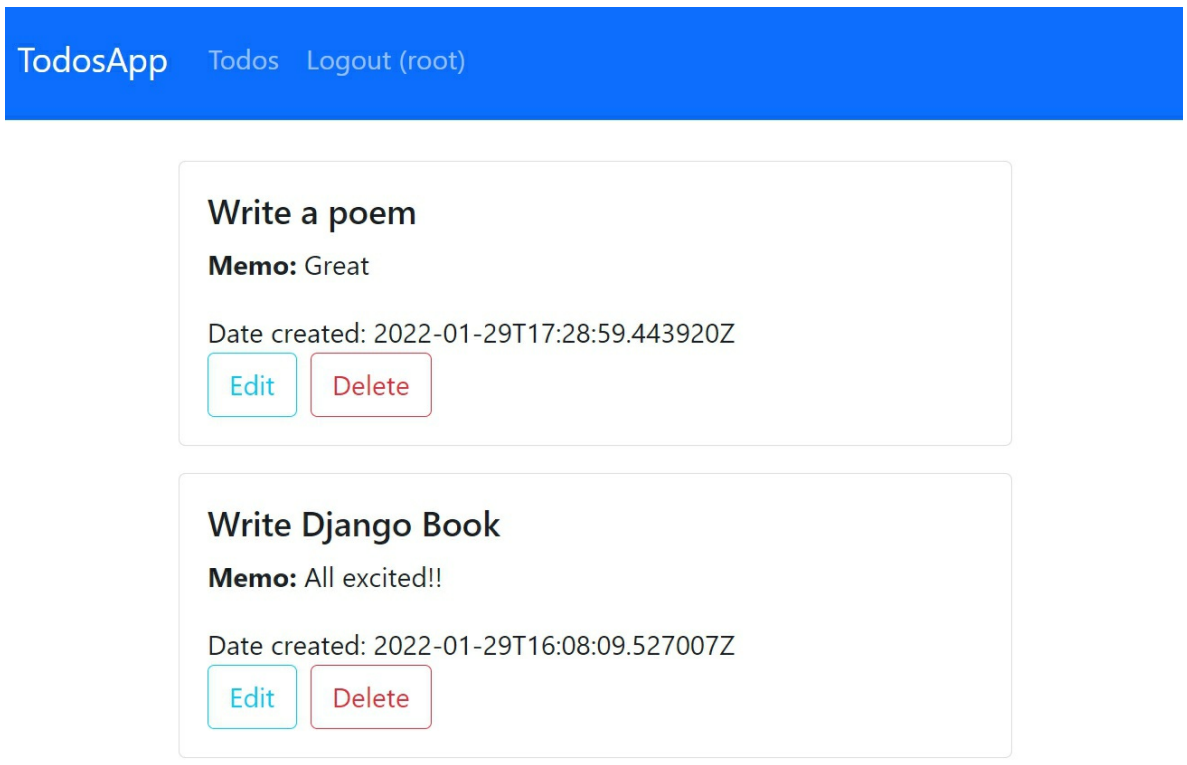


Figure 3

# Checking If a User is Logged In

Now we should only display todos if a user is logged in. To do so, in *return*, add:

**Modify Bold Code**

```
…
import Container from 'react-bootstrap/Container';
import Button from 'react-bootstrap/Button';
import Alert from 'react-bootstrap/Alert';[DCB9][JL10]


const TodosList = props => {
  …

  return (
    <Container>
      {props.token == null || props.token === "" ? (
        <Alert variant='warning'>
          You are not logged in. Please <Link to={"/login"}>login</Link> to see your todos.
        </Alert>
      ) : (
        <div>
          {todos.map((todo) => {
            …
          })}
        </div>
      )}
    </Container>
  );
}

export default TodosList;
```

If the user is logged in, i.e. *props.token* will contain a value and only then do we proceed to list the todos. Else, the user is not logged in and we log an alert warning message “You are not logged in.” and display the login link (fig. 4).

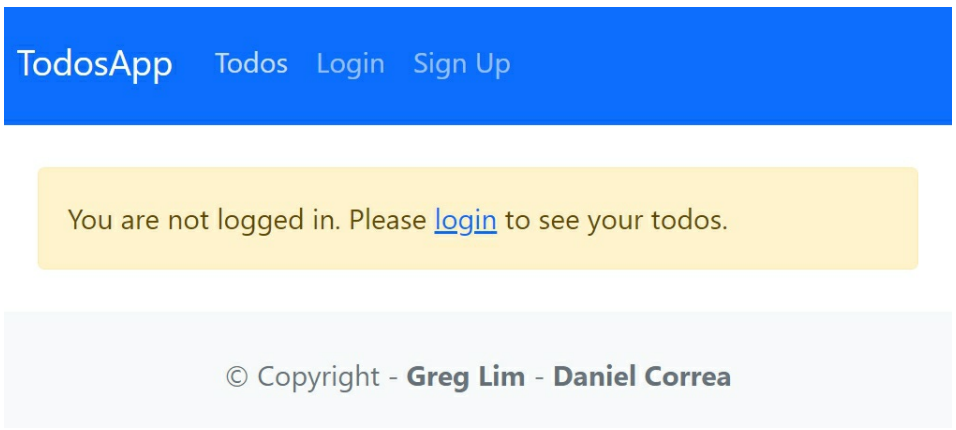


Figure 4

## *Testing our App*

If you test your app now, you will be able to see your todos when you log in.

## *Formatting the Date*

Before we go on to the next chapter, our current date is in timestamp format e.g. *2021-05-10T00:08:50.082Z*. Let's format the todo date(s) into a presentable manner. We will be using a library called *moment js*, a lightweight JavaScript library for parsing, validating and formatting dates.

In Terminal, in your 'frontend' project directory, install *moment js* with:

**Execute in Terminal**

```
npm i moment --save
```

In *todos-list.js*, import *moment* with:

**Modify Bold Code**

```
…
import Container from 'react-bootstrap/Container';
import Button from 'react-bootstrap/Button';
import Alert from 'react-bootstrap/Alert';
import moment from 'moment';

const TodosList = props => {
  …

  return (
    …
        <Card.Text>
          Date created: {moment(todo.created).format("Do MMMM YYYY")}
        </Card.Text>
    …
```

And when you run your app, the todo dates should be nicely formatted (fig. 5).

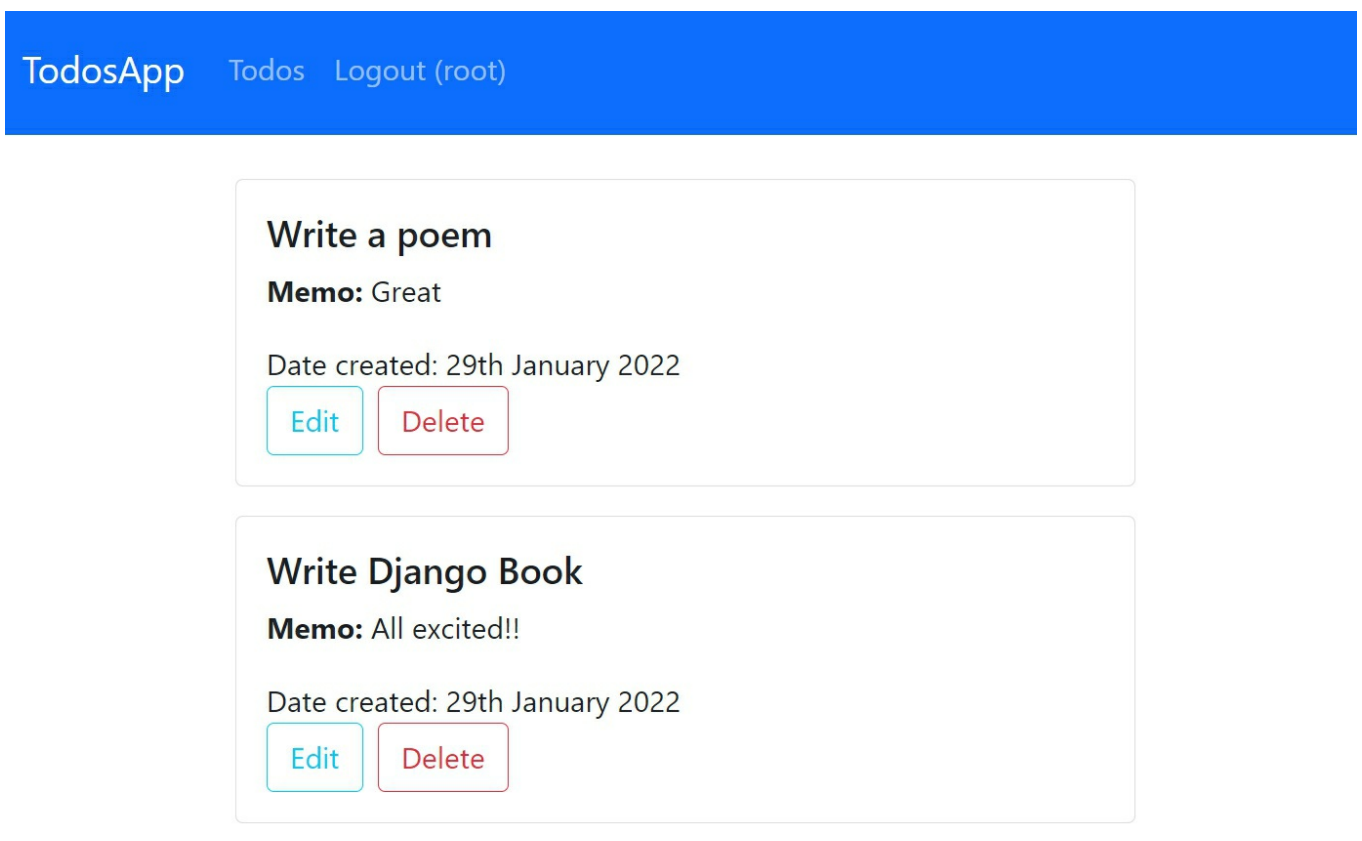


Figure 5

# Chapter 19: Adding and Editing Todos

Now, let’s go on to implement adding a todo. When a user logs in and clicks ‘Add Todo’ (fig. 1), we will render the *AddTodo* component for the user to submit a todo (fig. 2).

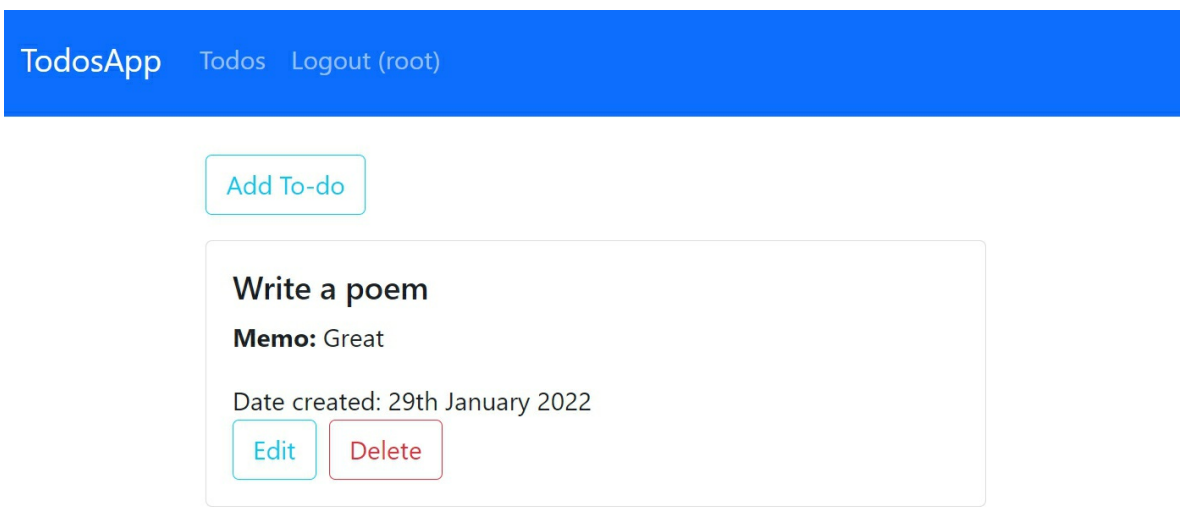


Figure 1

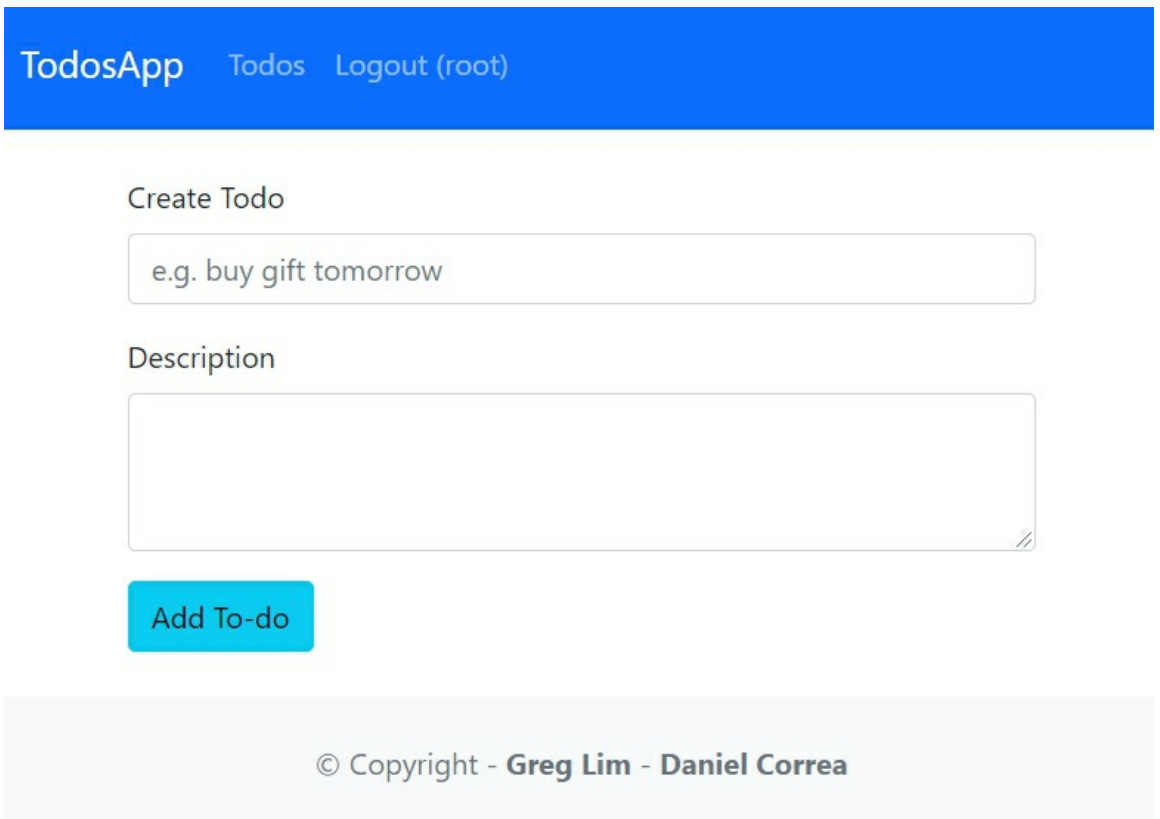


Figure 2

We will also use the *AddTodo* component to edit a todo. That is, when a user clicks on the ‘Edit’ link on an existing todo.

When editing, we will render the *AddTodo* component but with the header ‘Edit Todo’ (fig. 3). The existing todo title and memo will be shown where users can then edit and submit.

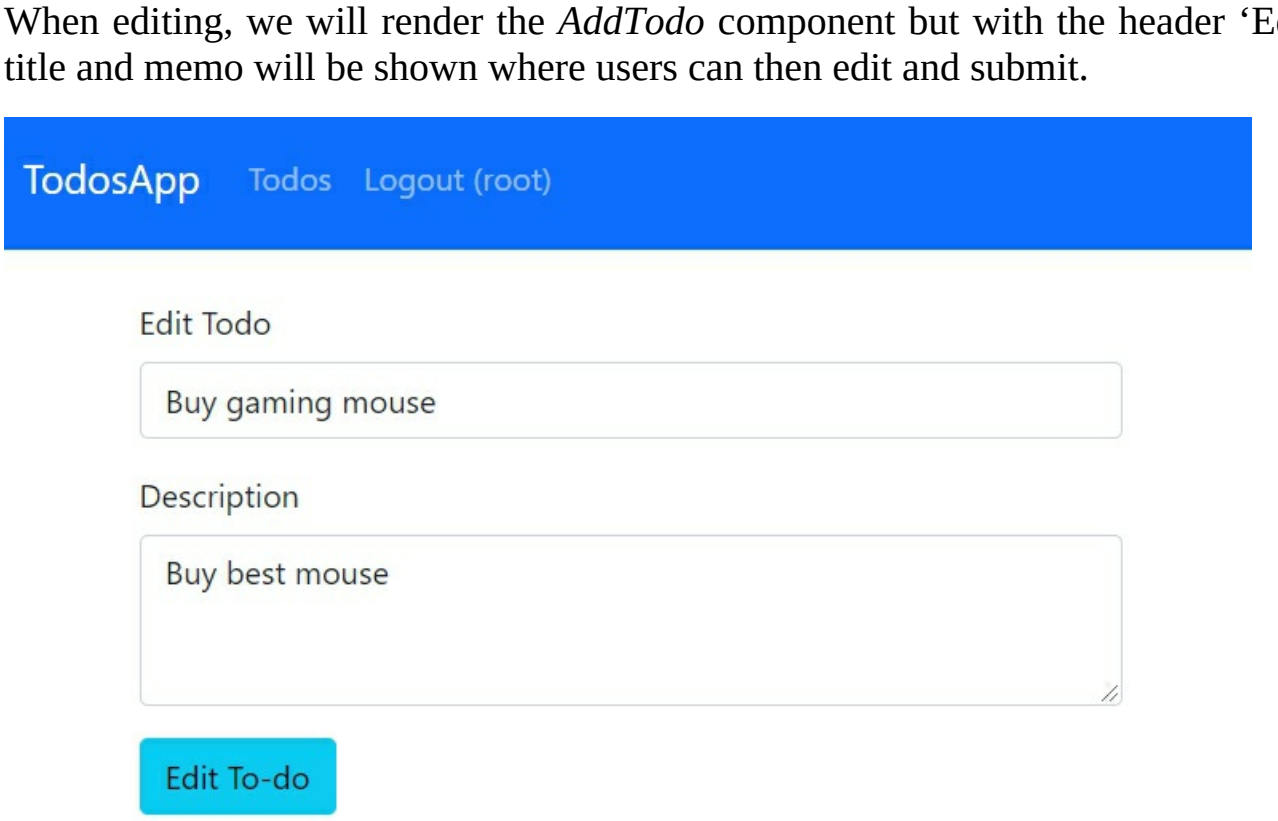


Figure 3

So, our *AddTodo* component will allow us to both add and edit todos. Let's first go through the code to add a todo.

## *Adding a Todo*

In *add-todo.js*, fill in the following code:

**Replace Entire Code**

```
import React, { useState } from 'react';
import TodoDataService from '../services/todos';
import { Link } from 'react-router-dom';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';
import Container from 'react-bootstrap/Container';


const AddTodo = props => {

 let editing = false;
 let initialTodoTitle = "";
 let initialTodoMemo = "";


 const [title, setTitle] = useState(initialTodoTitle);
 const [memo, setMemo] = useState(initialTodoMemo);
 // keeps track if todo is submitted
 const [submitted, setSubmitted] = useState(false);

const onChangeTitle = e => {
  const title = e.target.value;
  setTitle(title);
}

const onChangeMemo = e => {
  const memo = e.target.value;
  setMemo(memo);
}

const saveTodo = () => {
  var data = {
    title: title,
    memo: memo,
    completed: false,
  }


  TodoDataService.createTodo(data, props.token)
    .then(response => {
      setSubmitted(true);
    })
    .catch(e => {
      console.log(e);
    });
}

return (
<Container>
  {submitted ? (
    <div>
      <h4>Todo submitted successfully</h4>
      <Link to={"/todos/"}>
        Back to Todos
      </Link>
    </div>
  ) : (
    <Form>
      <Form.Group className="mb-3">
        <Form.Label>{editing ? "Edit" : "Create"} Todo</Form.Label>
        <Form.Control
          type="text"
          required
          placeholder="e.g. buy gift tomorrow"
          value={title}
          onChange={onChangeTitle}
        />
      </Form.Group>
      <Form.Group className="mb-3">
        <Form.Label>Description</Form.Label>
        <Form.Control
          as="textarea"
          rows={3}
```

```
            value={memo}
            onChange={onChangeMemo}
          />
        </Form.Group>
        <Button variant="info" onClick={saveTodo}>
          {editing ? "Edit" : "Add"} To-do
        </Button>
      </Form>
    )}
  </Container>
 )
}

export default AddTodo;
```

* Contact support@i-ducate.com for the source code if you prefer to copy and paste

## *Code Explanation*

**Analyze Code**

```
let editing = false;
let initialTodoTitle = "";
let initialTodoMemo = "";

const [title, setTitle] = useState(initialTodoTitle);
const [memo, setMemo] = useState(initialTodoMemo);
// keeps track if todo is submitted
const [submitted, setSubmitted] = useState(false);
```

The *editing* Boolean variable will be set to *true* if the component is in 'Editing' mode. *False* means we are adding a todo.

We have a *title* state variable set to `initialTodoTitle`. In edit mode, `initialTodoTitle` will be set to the existing title text. The same applies for memo and `initialTodoMemo`.

We also have a *submitted* state variable to keep track if the todo is submitted.

**Analyze Code**

```
const onChangeTitle = e => {
  const title = e.target.value;
  setTitle(title);
}
```

The *onChangeTitle* keeps track of the user-entered title value in the field:

**Analyze Code**

```
        <Form.Control
          type="text"
          required
          placeholder="e.g. buy gift tomorrow"
          value={title}
          onChange={onChangeTitle}
        />
```

The same applies to *onChangeMemo*. This should be familiar to you as we have used this before in the login form fields.

**Analyze Code**

```
const saveTodo = () => {
  var data = {
    title: title,
    memo: memo,
    completed: false,
  }


  TodoDataService.createTodo(data, props.token)
    .then(response => {
      setSubmitted(true);
    })
    .catch(e => {
      console.log(e);
    });
}
```

*saveTodo* is called by the submit button's `onClick={saveTodo}`. In *saveTodo*, we first create a *data* object containing the todo's properties, e.g. the todo title, memo, etc.

We then call TodoDataService.createTodo(data,props.token) we implemented earlier in /services/*todo.js*:

**Analyze Code**

```
createTodo(data, token){
  axios.defaults.headers.common["Authorization"] = "Token " + token;
  return axios.post("http://localhost:8000/api/todos/", data);
}
```

This then routes to *todobackend/api/views.py* in our backend and calls *TodoListCreate* view to create the todo instance in the backend database.

**Analyze Code**

```
class TodoListCreate(generics.ListCreateAPIView):
    # ListAPIView requires two mandatory attributes, serializer_class and
    # queryset.
    # We specify TodoSerializer which we have earlier implemented
    serializer_class = TodoSerializer
    permission_classes = [permissions.IsAuthenticated]


    def get_queryset(self):
        user = self.request.user
        return Todo.objects.filter(user=user).order_by('-created')


    def perform_create(self, serializer):
        #serializer holds a django model
        serializer.save(user=self.request.user)
```

Hopefully, you can see better how the whole flow in a Django React stack works now. Let ’ s go on to implement editing a todo.

## *Add Todo Link*

In our TodosList component, we will display a ‘ Add Todo ’ link for users to navigate to the *AddTodo* component. In *todos-list.js*, add the below codes in **bold**:

**Modify Bold Code**

```
…
const TodosList = props => {
 …


 return (
   <Container>
     {props.token == null || props.token === "" ? (
       <Alert variant='warning'>
         You are not logged in. Please <Link to={"/login"}>login</Link> to see your todos.
       </Alert>
     ) : (
       <div>
         <Link to={"/todos/create"}>
           <Button variant="outline-info" className="mb-3">
             Add To-do
           </Button>
         </Link>
     {todos.map((todo) => {
     …
```

This renders a ‘ Add To-do ’ button (fig. 4).

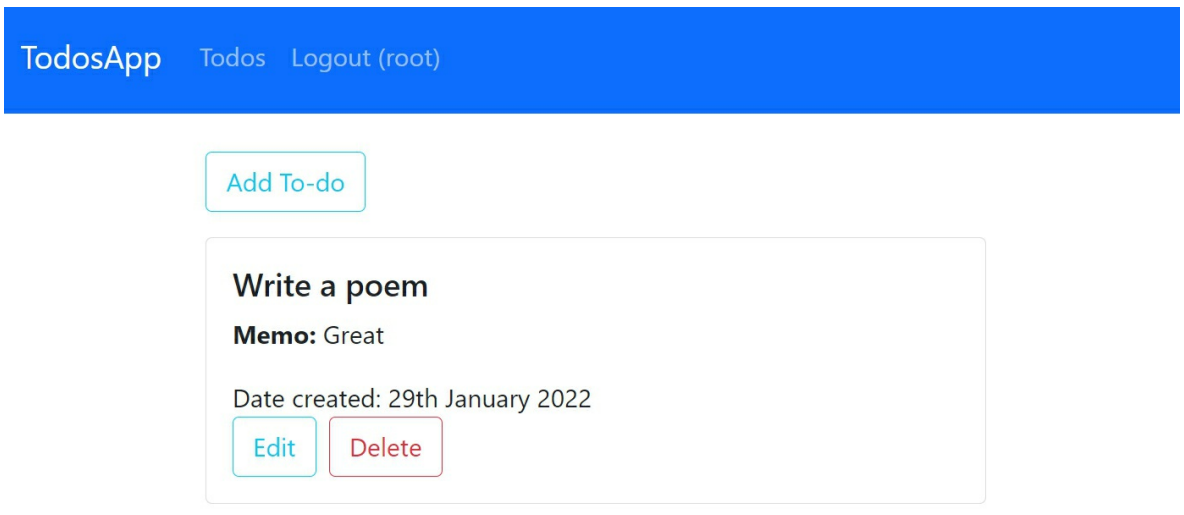


Figure 4

## *Running our App*

Now, let ’ s run our app. Login, click on the ‘  Add Todo ’ link and you should be able to add a todo. The new todo should appear in the todos page (fig. 5).

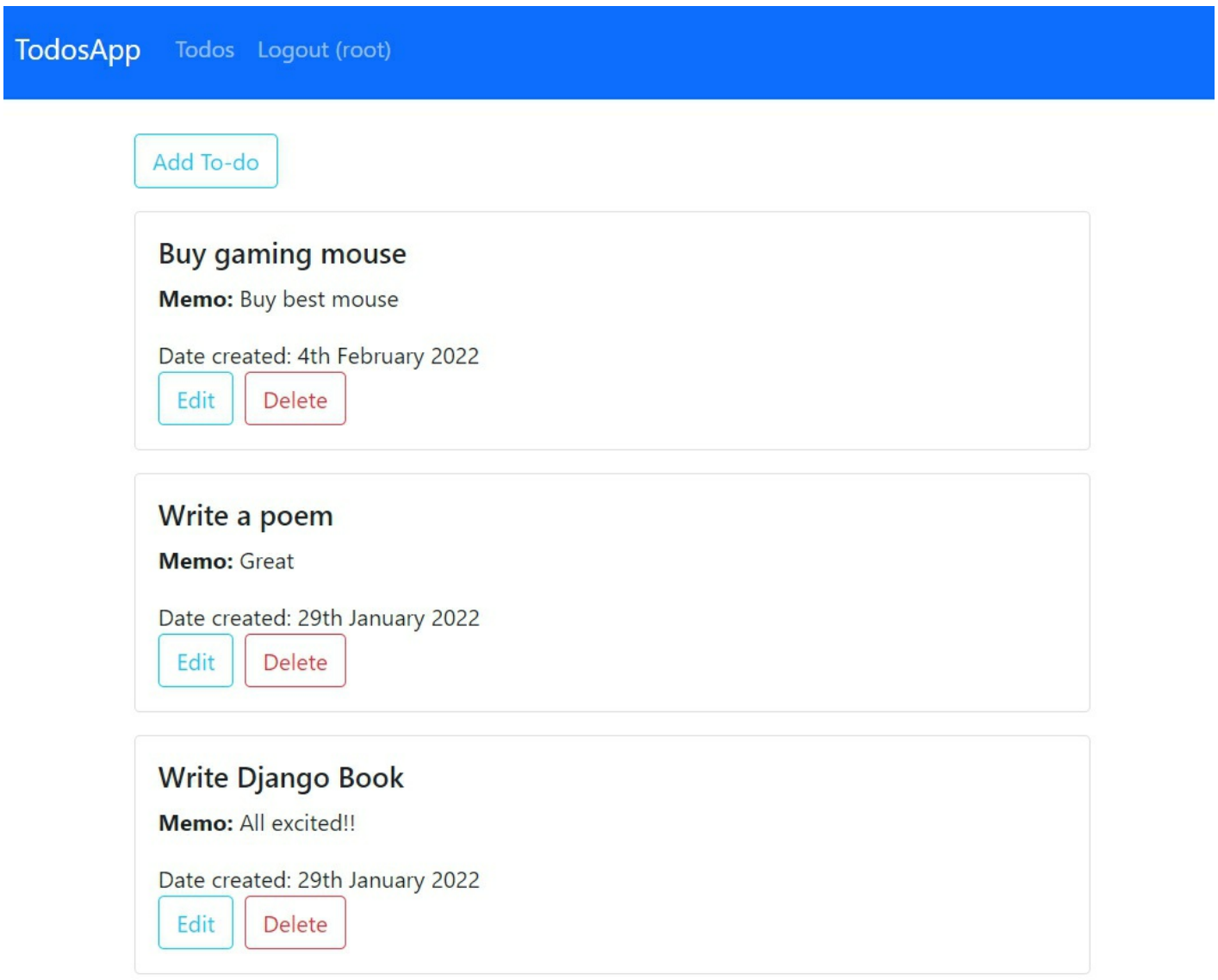


Figure 5

## *Editing a Todo*

To have our *AddTodo* component edit a todo, add in two sections of code in *add-todo.js* as shown below:

**Modify Bold Code**

```
…
const AddTodo = props => {

 let editing = false;
 let initialTodoTitle = "";
 let initialTodoMemo = "";

 if (props.location.state && props.location.state.currentTodo) {
   editing = true;
   initialTodoTitle = props.location.state.currentTodo.title;
   initialTodoMemo = props.location.state.currentTodo.memo;
 }

…

const saveTodo = () => {
  var data = {
    title: title,
    memo: memo,
    completed: false,
  }

  if (editing) {
    TodoDataService.updateTodo(
      props.location.state.currentTodo.id,
      data, props.token)
      .then(response => {
        setSubmitted(true);
        console.log(response.data)
      })
      .catch(e => {
        console.log(e);
      })
  }
  else {
```

```
    TodoDataService.createTodo(data, props.token)
      .then(response => {
        setSubmitted(true);
      })
      .catch(e => {
        console.log(e);
      });
  }
}


…
```

## *Code Explanation*

**Analyze Code**

```
if (props.location.state && props.location.state.currentTodo) {
  editing = true;
  initialTodoTitle = props.location.state.currentTodo.title;
  initialTodoMemo = props.location.state.currentTodo.memo;
}
```

We first check if a state is passed into *AddTodo*. If you recall in *todos-list.js*, we pass in a state in the link to edit:

**Analyze Code**

```
              <Link to={{
                pathname: "/todos/" + todo.id,
                state: {
                  currentTodo: todo
                }
              }}>
                <Button variant="outline-info" className="me-2">
                  Edit
                </Button>
              </Link>
```

Thus, in *AddTodo*, we check if a state is passed in and contains a *currentTodo* property. If so, set *editing* to true and the *initialTodoTitle* and **initialTodoMemo** to *currentTodo*'s title and memo.

**Analyze Code**

```
  if (editing) {
    TodoDataService.updateTodo(
      props.location.state.currentTodo.id,
      data, props.token)
      .then(response => {
        setSubmitted(true);
        console.log(response.data)
      })
      .catch(e => {
        console.log(e);
      })
  }
```

If *editing* is true, get the existing todo id and call *updateTodo* in *TodoDataService*:

**Analyze Code**

```
 updateTodo(id, data, token){
  axios.defaults.headers.common["Authorization"] = "Token " + token;
  return axios.put(`http://localhost:8000/api/todos/${id}`, data);
 }
```

The above calls the *TodoRetrieveUpdateDestroy* view in */api/views.py* in the backend similar to how we call *TodoListCreate* for adding a todo:

**Analyze Code**

```
class TodoRetrieveUpdateDestroy(generics.RetrieveUpdateDestroyAPIView):
    serializer_class = TodoSerializer
    permission_classes = [permissions.IsAuthenticated]


    def get_queryset(self):
        user = self.request.user
        # user can only update, delete own posts
        return Todo.objects.filter(user=user)
```

## *Running our App*

If you are logged in, you will be able to edit a todo by clicking on the ‘ Edit ’ todo (fig. 6). And if you check your Admin, the data is updated there.

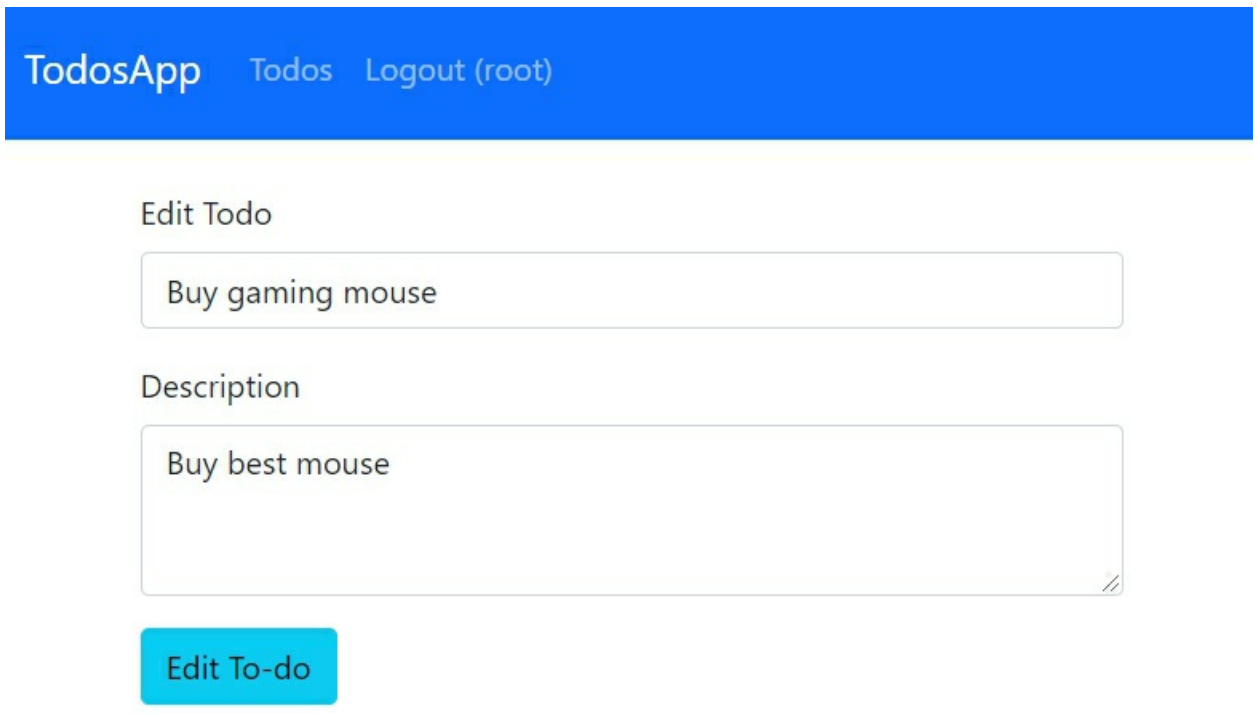


Figure 6

In the following two chapters, we will finish up implementing deleting and completing a todo.

# Chapter 20: Deleting a Todo

We are currently just rendering a delete button for each todo in *todos-list.js*:

**Analyze Code**

```
<Button variant="outline-danger">
  Delete
</Button>
```

Let ’ s now implement its functionality by adding the below in **bold**:

**Modify Bold Code**

```
…
 <Button variant="outline-danger" onClick={() => deleteTodo(todo.id)}>
   Delete
 </Button>
…
```

Next, add in the codes for *deleteTodo* just above *return*:

**Modify Bold Code**

```
…

const deleteTodo = (todoId) =>{
  TodoDataService.deleteTodo(todoId, props.token)
    .then(response => {
      retrieveTodos();
    })
    .catch(e =>{
      console.log(e);
    });
}


return (
  <Container>
    …
```

## *Code Explanation*

**Analyze Code**

```
<Button variant="outline-danger" onClick={() => deleteTodo(todo.id)}>
```

In the delete button, we pass in todo id into *deleteTodo*.

In *deleteTodo*, we then call *deleteTodo* in *TodoDataService* which calls the delete API endpoint we implemented earlier:

**Analyze Code**

```
deleteTodo(id, token){
  axios.defaults.headers.common["Authorization"] = "Token " + token;
  return axios.delete(`http://localhost:8000/api/todos/${id}`);
}
```

Remember that the *delete* endpoint is supported by *TodoRetrieveUpdateDestroy* view in *todobackend/api/views.py*:

**Analyze Code**

```
class TodoRetrieveUpdateDestroy(generics.RetrieveUpdateDestroyAPIView):
    serializer_class = TodoSerializer
    permission_classes = [permissions.IsAuthenticated]


    def get_queryset(self):
        user = self.request.user
        # user can only update, delete own posts
        return Todo.objects.filter(user=user)
```

Back in *todos-list.js*, we then add a callback function that is called when *deleteTodo* completes:

**Analyze Code**

```
 const deleteTodo = (todoId) =>{
   TodoDataService.deleteTodo(todoId, props.token)
     .then(response => {
       retrieveTodos();
     })
     .catch(e =>{
       console.log(e);
     });
 }
```

In the callback, we simply call *retrieveTodos()* which sets the updated *todos* array as the state.

## *Running your App*

When you run your app now and log in, a user will be able to delete todos they have posted. We have almost completed the entire functionality of our app using the Django API React stack. What ’ s left is marking a todo as complete. Let ’ s first see how to do so in the next chapter.

# Chapter 21: Completing a Todo

Let ’ s add a mark ‘  Complete ’ button for each Todo in *todos-list.js* by adding the below in **bold**:

**Modify Bold Code**

```
…
  <Button variant="outline-info" className="me-2">
   Edit
  </Button>
</Link>
<Button variant="outline-danger" onClick={() => deleteTodo(todo.id)} className="me-2">
  Delete
</Button>
<Button variant="outline-success" onClick={() => completeTodo(todo.id)}>
  Complete
</Button>
…
```

Next, add in the codes for *completeTodo* just above *return*:

**Modify Bold Code**

```
…
const completeTodo = (todoId) => {
  TodoDataService.completeTodo(todoId, props.token)
    .then(response => {
      retrieveTodos();
      console.log("completeTodo", todoId);
    })
    .catch(e => {
      console.log(e);
    })
}

return (
  <Container>
    …
```

## *Code Explanation*

**Analyze Code**

```
<Button variant="outline-success" onClick={() => completeTodo(todo.id)}>
```

In the complete button, we pass in the todo id into *completeTodo*.

In *completeTodo*, we then call *completeTodo* in *TodoDataService* which calls the complete API endpoint we implemented earlier:

**Analyze Code**

```
completeTodo(id, token){
  axios.defaults.headers.common["Authorization"] = "Token " + token;
    return axios.put(`http://localhost:8000/api/todos/${id}/complete`);
 }
```

Remember that the *complete* endpoint is supported by *TodoToggleComplete* view in *todobackend/api/views.py*:

**Analyze Code**

```
class TodoToggleComplete(generics.UpdateAPIView):
    serializer_class = TodoToggleCompleteSerializer
    permission_classes = [permissions.IsAuthenticated]


    def get_queryset(self):
        user = self.request.user
        return Todo.objects.filter(user=user)


    def perform_update(self,serializer):
        serializer.instance.completed=not(serializer.instance.completed)
        serializer.save()
```

Back in *todos-list.js*, we then add a callback function that is called when *deleteTodo* completes:

**Analyze Code**

```
 const completeTodo = (todoId) => {
   TodoDataService.completeTodo(todoId, props.token)
     .then(response => {
```

```
      retrieveTodos();
      console.log("completeTodo", todoId);
    })
    .catch(e => {
      console.log(e);
    })
}
```

In the callback, we simply call *retrieveTodos*() which sets the updated *todos* array as the state.

## *Rendering a Completed Todo*

We want to render a completed todo by having its details striked through (fig. 1).

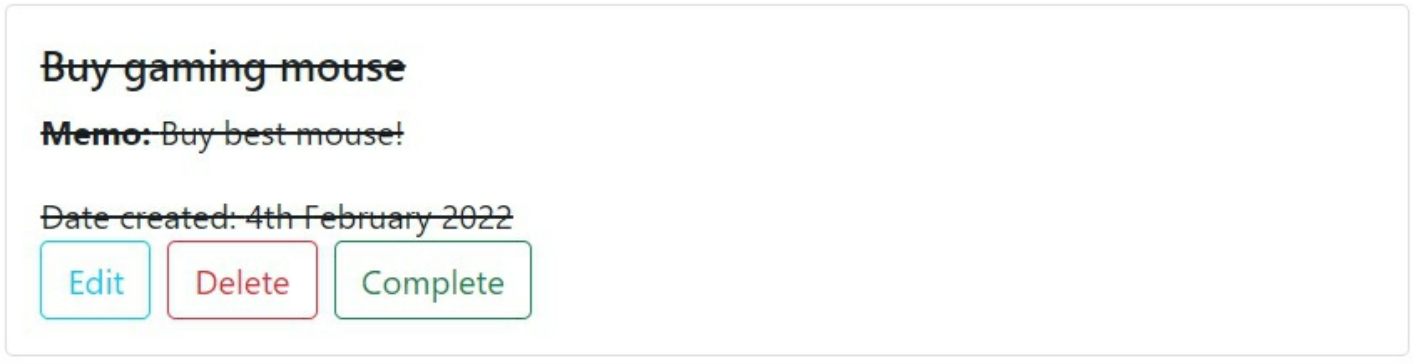


Figure 1

To do so, we add the below codes into *todos-list.js*:

**Modify Bold Code**

```
…
 {todos.map((todo) => {
  return (
    <Card key={todo.id} className="mb-3">
      <Card.Body>
        <div className={`${todo.completed ? "text-decoration-line-through" : ""}`}>
          <Card.Title>{todo.title}</Card.Title>
          <Card.Text><b>Memo:</b> {todo.memo}</Card.Text>
          <Card.Text>
            Date created: {moment(todo.created).format("Do MMMM YYYY")}
          </Card.Text>
        </div>
        …
```

If *todo.completed* is true, we add the line through text decoration to the div that contains the title, memo and date created. For uncompleted todos, render them normally (fig. 2).

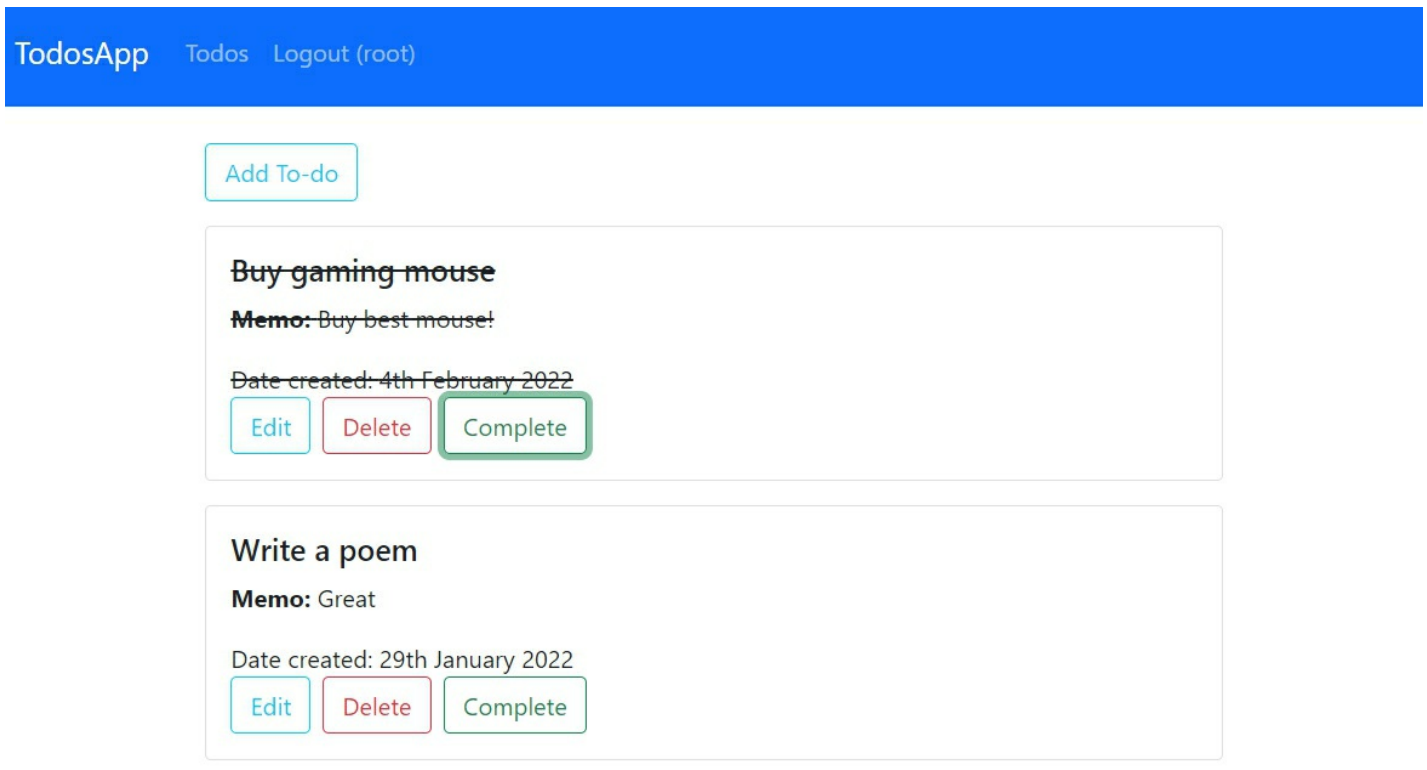


Figure 2

Next, we also want to disable editing of a completed todo. Only uncompleted todos can be edited. We will still allow deletion of completed todos. To disable editing of a completed todo, add the codes to the Edit link:

**Modify Bold Code**

```
…
        {todos.map((todo) => {
```

```
return (
  <Card key={todo.id} className="mb-3">
    …
    {!todo.completed &&
      <Link to={{
        pathname: "/todos/" + todo.id,
        state: {
          currentTodo: todo
        }
      }}>
        <Button variant="outline-info" className="me-2">
          Edit
        </Button>
      </Link>
    }
    …
```

If *todo.completed* is not true, we render the Edit button. For completed todos, the Edit button will not be displayed (fig. 3).

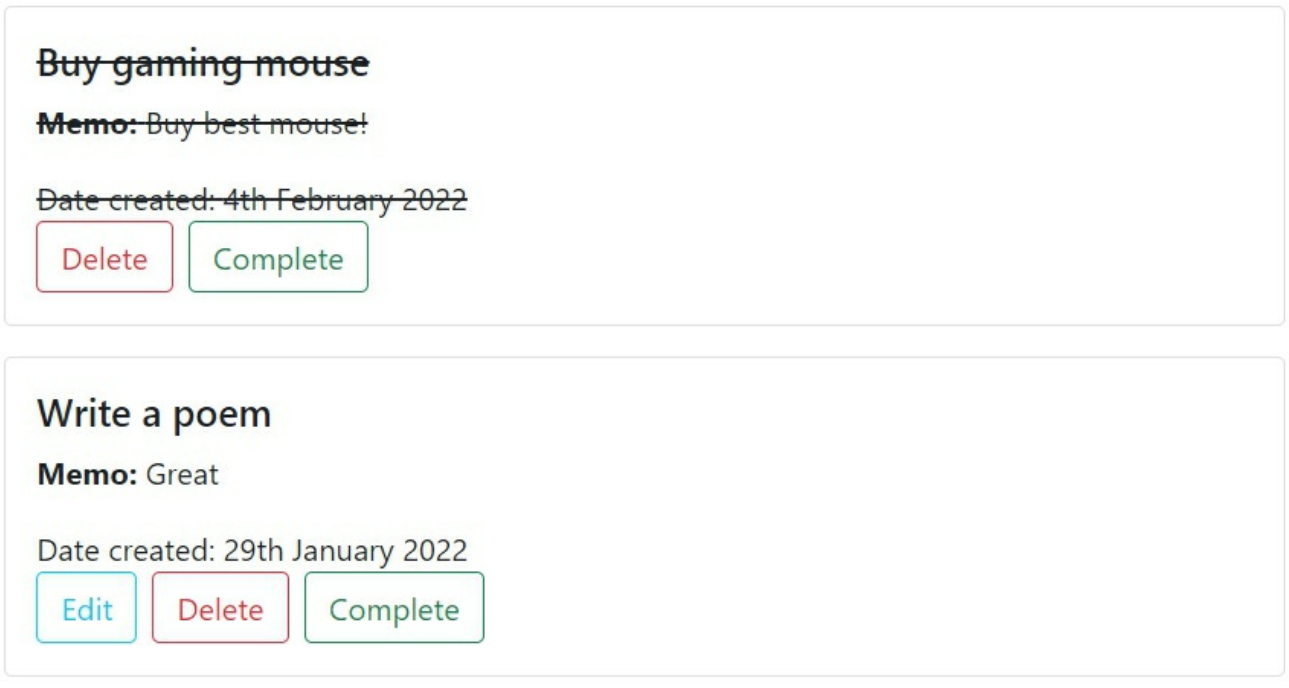


Figure 3

### *Running your App*

When you run your app now and log in, a user will be able to complete todos they have posted.

# Chapter 22: Deployment

Both our backend and frontend are currently running on our local machine. To have our site live on the real, public internet for the world to use, we need to deploy them onto an actual server. A popular way to do so is to deploy our Django backend on PythonAnywhere. In the next chapter, we will deploy our React frontend on Netlify.

To get our code on to PythonAnywhere, we need our code to be on a code sharing site like GitHub or Bitbucket. In this chapter, we will use GitHub. If you are already familiar with uploading your code to GitHub, please skip the following section. Else, you can follow along.

## *GitHub and Git*

In this section, we cover only the necessary steps to move your code onto GitHub. We will not cover all details about Git and GitHub (entire books have been written on these alone!).

Go to *github.com* and sign up for an account if you don ’ t have one. To put your project on GitHub, you will need to create a repository for it to live in. Create a new repository by clicking on ‘ + ’ on the top-right and select ‘ New repostiory ’ (fig. 29.1).

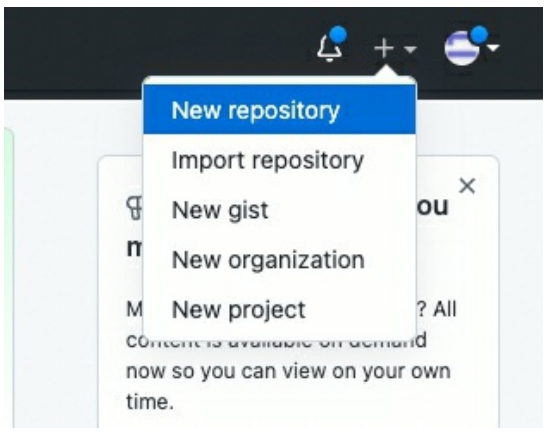

Figure 29.1

Give your repository a name e.g. *todoapp*. Select the ‘ public ’ radio box and hit ‘ Create repository ’ .

We will begin to move our code onto GitHub. In your local machine ’ s Terminal, ensure you have *git* installed by running: *git*

Now, what is Git? Git is a version control for projects and is very popular in the development world. It allows us to have save points (Git calls them *commits*) in our code. If we make mistakes in our project at any point in time, we can go back to previous save points when the project is working. Git also allows multiple developers to work on the project together without worrying about one overwriting the code of another.

When you run *git* in the Terminal, if you see Git commands listed down, you have Git installed. If you don ’ t see them, you will need to install Git. Visit the Git site (https://git-scm.com/) and follow the instructions to install Git. When Git is installed, you might need to close and re-open the Terminal and in it. Type *git* to ensure that it is installed.

In the *todobackend* folder, enter:

**Execute in Terminal**

```
git init
```

*git init* marks your folder as a Git project where you can begin to track changes. A hidden folder *.git* is added to the project folder. The *.git* folder stores all the objects and refs that Git uses and creates as part of your project ’ s history.

Next, run:

**Execute in Terminal**

```
git add -A
```

This adds everything in your project into the staging environment to prepare a snapshot before committing it to the official history. Go ahead to commit them by running:

**Execute in Terminal**

```
git commit -m "first message"
```

This creates a save point in your code. You can identify different commits by the descriptive messages you provide. Next, we want to save our Git project on to GitHub.

In the repository page in GitHub, copy the *git remote add origin <origin path>* command and run it the Terminal. Eg:

**Execute in Terminal**

```
git remote add origin https://github.com/greglim81/todoapp.git
```

This creates a new connection record to the remote repository. To move the code from your local computer to GitHub, run:

**Execute in Terminal**

```
git push -u origin main
```

This pushes the code to GitHub. When you reload the GitHub repository page, your project ' s code will be reflected there.

*Do note that there is much more to Git and GitHub. We have just covered the necessary steps to upload our code on to GitHub. With this, we have now gotten our code on GitHub. Next, we will clone our code on to PythonAnywhere.

## *Cloning our Code on to PythonAnywhere*

The steps to deploy an existing Django project on PythonAnywhere can be found at https://help.pythonanywhere.com/pages/DeployExistingDjangoProject/. But I will go through it with you here.

Now that we have our code on GitHub, we will have PythonAnywhere retrieve our code from there. First, create a beginner free account in PythonAnywhere here: https://www.pythonanywhere.com/registration/register/beginner/

In PythonAnywhere, click on ' New console ' -> ' Bash ' to access its Linux Terminal (fig. 29.2).

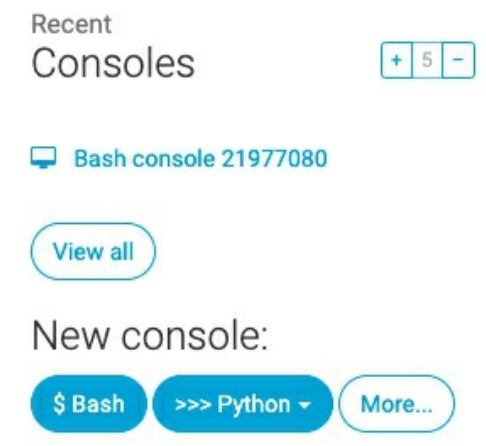


Figure 29.2

Back in your GitHub repository, click on ' Code ' and copy the URL to clone (fig. 29.3).

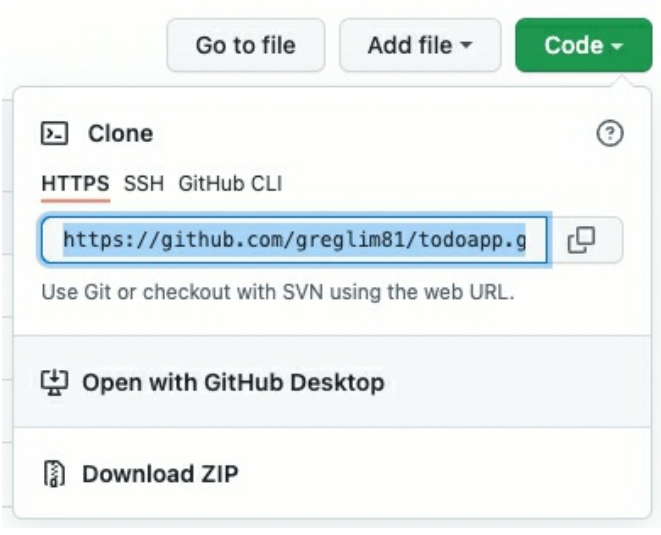


Figure 29.3

To clone, back in the PythonAnywhere Bash, run: *git clone <paste url>*, e.g.

**Execute in Terminal**

```
git clone https://github.com/greglim81/todoapp.git
```

This will take all your code from the GitHub repository and clone it in PythonAnywhere. When the clone completes, you can do a ' ls ' in bash and you will see the folder as what you see on your machine.

## *Virtual Environments*

Virtual environments are used to create separate development environments for different projects with different

requirements. For example, you can specify which version of Python and which libraries/modules you want installed in a particular virtual environment.

As an example, to create a virtual environment in the PythonAnywhere bash, we run:

**Analyze Command**

```
mkvirtualenv -p python3.8 <environment name>
```

In the above, we have specified that we use Python 3.8 in the virtualenv. Whatever packages we install will always be there and independent of other virtualenvs.

If I ran:

**Execute in Terminal**

```
mkvirtualenv -p python3.8 todoappenv
```

I will see the name of the virtualenv in the Bash which means we are in the VE eg:

(**todoappenv**) 00:08 ~ $

Back in our virtualenv, we need to install django, djangorestframework, django-cors-headers (as what we have done in development). So run:

**Execute in Terminal**

```
pip install django djangorestframework django-cors-headers
```

# Setting up your Web app

At this point, you need to be ready with 3 pieces of information:

1. The path to your Django project's top folder -- the folder that contains "manage.py". A simple way to get this is, in your project folder in bash, type *pwd*. Eg: */home/greglim/todoapp*

2. The name of your project (the name of the folder that contains your settings.py), e.g. *backend*

3. The name of your virtualenv, eg *todoappenv*

## *Create a Web app with Manual Config*

In your browser, open a new tab and go to the PythonAnywhere dashboard. Click on the ‘ Web ’ tab (fig. 29.4).


Figure 29.4

Click, ‘ Add a new web app ’ . Under ‘ Select a Python Web framework ’ , choose the ‘ Manual Configuration ’ (fig. 29.5).


Figure 29.5

NOTE: Make sure you choose ‘Manual Configuration’, not the ‘Django’ option, that's for new projects only.

Select the right version of Python (the same one you used to create your virtualenv).

## *Enter your virtualenv name*

Enter the name of your virtualenv in the *Virtualenv* section (fig. 29.6).

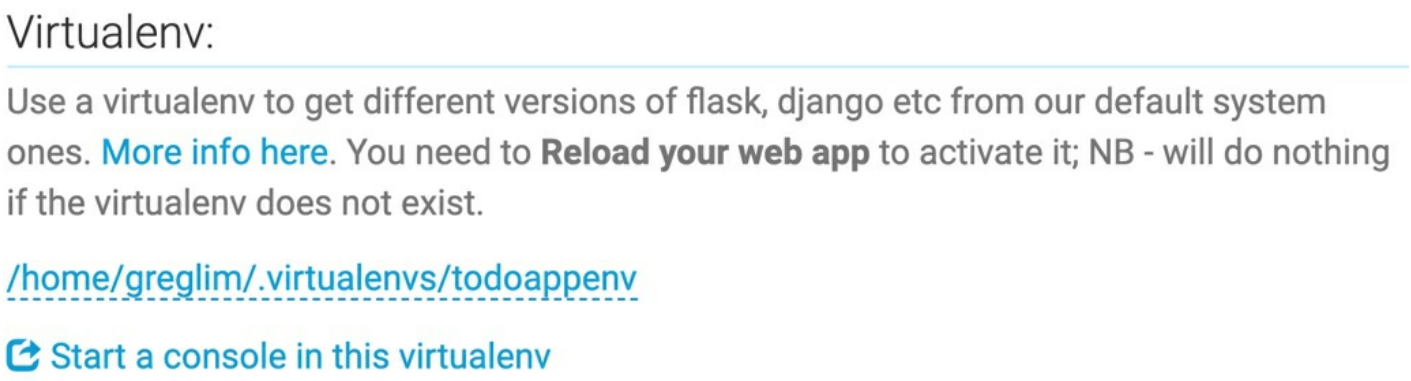


Figure 29.6

You can just use its short name "todoappenv", and it will automatically complete to its full path in */home/username/.virtualenvs*.

## *Enter path to your code*

Next, enter the path to your project folder in the *Code* section, both for ‘ Source code ’ and ‘ Working directory ’ (fig. 29.7):

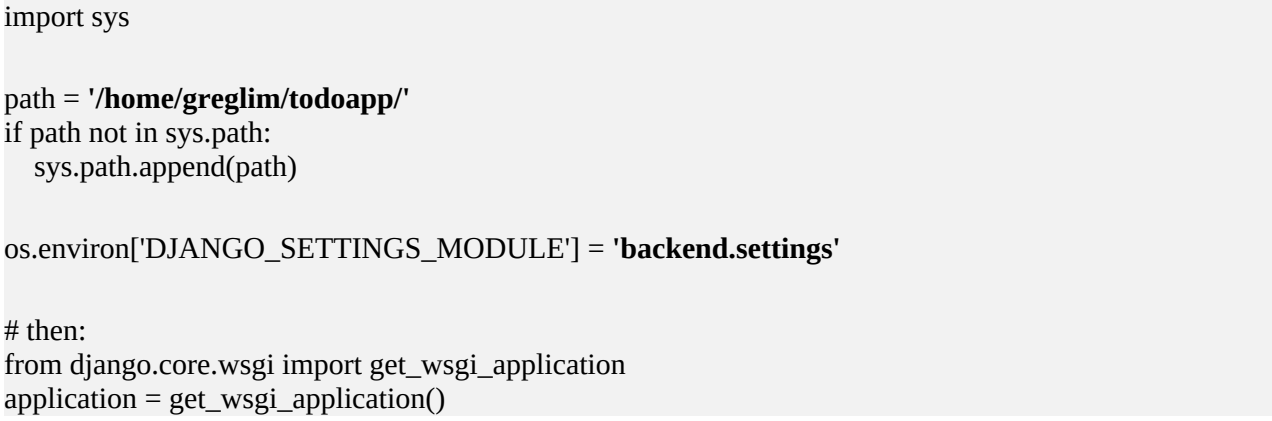


Figure 29.7

## *Edit your WSGI file*

In the *wsgi.py* file inside the ‘ Code ’ section (fig. 29.8 - not the one in your local Django project folder),

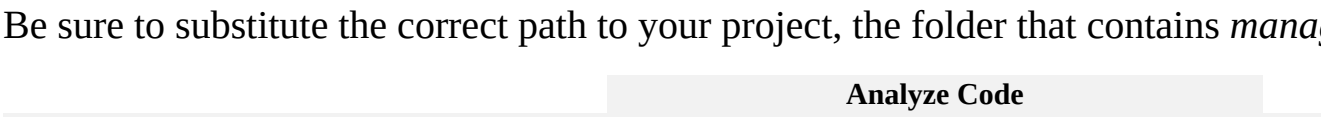


Figure 29.8

Click on the WSGI file link and it will take you to an editor where you can change it. Delete everything except the Django section and uncomment that section. Your WSGI file will look something like:

**Modify Bold Code**

```
# ++++++++++ DJANGO ++++++++++
# To use your own Django app use code like this:
import os
import sys

path = '/home/greglim/todoapp/'
if path not in sys.path:
    sys.path.append(path)

os.environ['DJANGO_SETTINGS_MODULE'] = 'backend.settings'

# then:
from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

## *Code Explanation*

**Analyze Code**

```
path = '/home/greglim/todoapp/'
```

Be sure to substitute the correct path to your project, the folder that contains *manage.py*,

**Analyze Code**

```
os.environ['DJANGO_SETTINGS_MODULE'] = 'backend.settings'
```

Make sure you put the correct value for DJANGO_SETTINGS_MODULE and save the file.

## *Exiting and Entering a Virtualenv*

At any point you want to get out of the virtualenv, run ‘ deactivate ’ in the Bash.
To get back in to a virtualenv, run: workon <virtualenv name>

If you forgot the name of a virtualenv, you can see the list of virtualenvs you have by going to the *.virtualenvs* folder:

**Execute in Terminal**

```
cd .virtualenvs
ls
```

You can then see the list of virtualenvs.

### *Allowed Hosts*

Next, we need to add to the allowed hosts in *settings.py*. Go to the ‘ Files ’ tab, navigate through the source code directory, and in *settings.py*, under ALLOWED_HOSTS, add the URL which PythonAnywhere generates for you. In my case, it is:

**Modify Bold Code**

```
ALLOWED_HOSTS = ['greglim.pythonanywhere.com']
```

The ALLOWED_HOSTS settings represent which host/domain names our Django site can serve. This is a security measure to prevent HTTP Host header attacks.

### *Set Debug to False*

For local development, we have set DEBUG=True. This shows us the detailed error description and from which line of code it is resulting from. We should however hide these in production (we don ’ t want to expose all our secrets out in the open) by setting DEBUG=False.

Thus, in the PythonAnywhere dashboard, go your project ’ s *settings.py* and set DEBUG = False.

### *createsuperuser*

Remember to create a superuser in your production environment (just as we did in our development environment) and keep the password secure.

### *.gitignore*

We can ignore some files when uploading to GitHub. For e.g. the *__pycache__* directory which are created automatically when Django runs *.py* files. We should also ignore the *db.sqlite3* file as it might be accidentally pushed to production. And if you are using a Mac, we can ignore *.DS_Store* which stores information about folder settings on MacOS.

So the final *.gitignore* will look like:

**Add Code**

```
__pycache__/
db.sqlite3
.DS_Store
```

## Change *db.sqlite3* to MySQL or PostgresSQL

Our current database is set to SQLite which works fine for small projects. But when the project or the data size grows, we want to switch to other databases like MySQL or PostgreSQL.

PythonAnywhere allows usage of MySQL free. But for PostgreSQL, you will need to have a paid account.

To start using MySQL, you can refer to PythonAnywhere ’ s brief and useful documentation (*https://help.pythonanywhere.com/pages/UsingMySQL/*).

Note: After you have setup MySQL or any other database, because it ’ s a brand new database, you will have to create a new superuser (*python manage.py createsuperuser*) and run the makemigrations/migrate command:

**Execute in Terminal**

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

We are almost done with deploying our Django backend. One remaining step is to specify our frontend URL in CORS_ORIGIN_WHITELIST in *todoapp/backend/settings.py*. We thus first deploy our frontend to get the URL

and come back to set CORS_ORIGIN_WHITELIST in the next chapter.

# Chapter 23: Hosting and Deploying our React Frontend

In this section, we will deploy our React frontend to the Internet to share it with the world. We are going to use Netlify (netlify.com – fig.1) for our deployment.

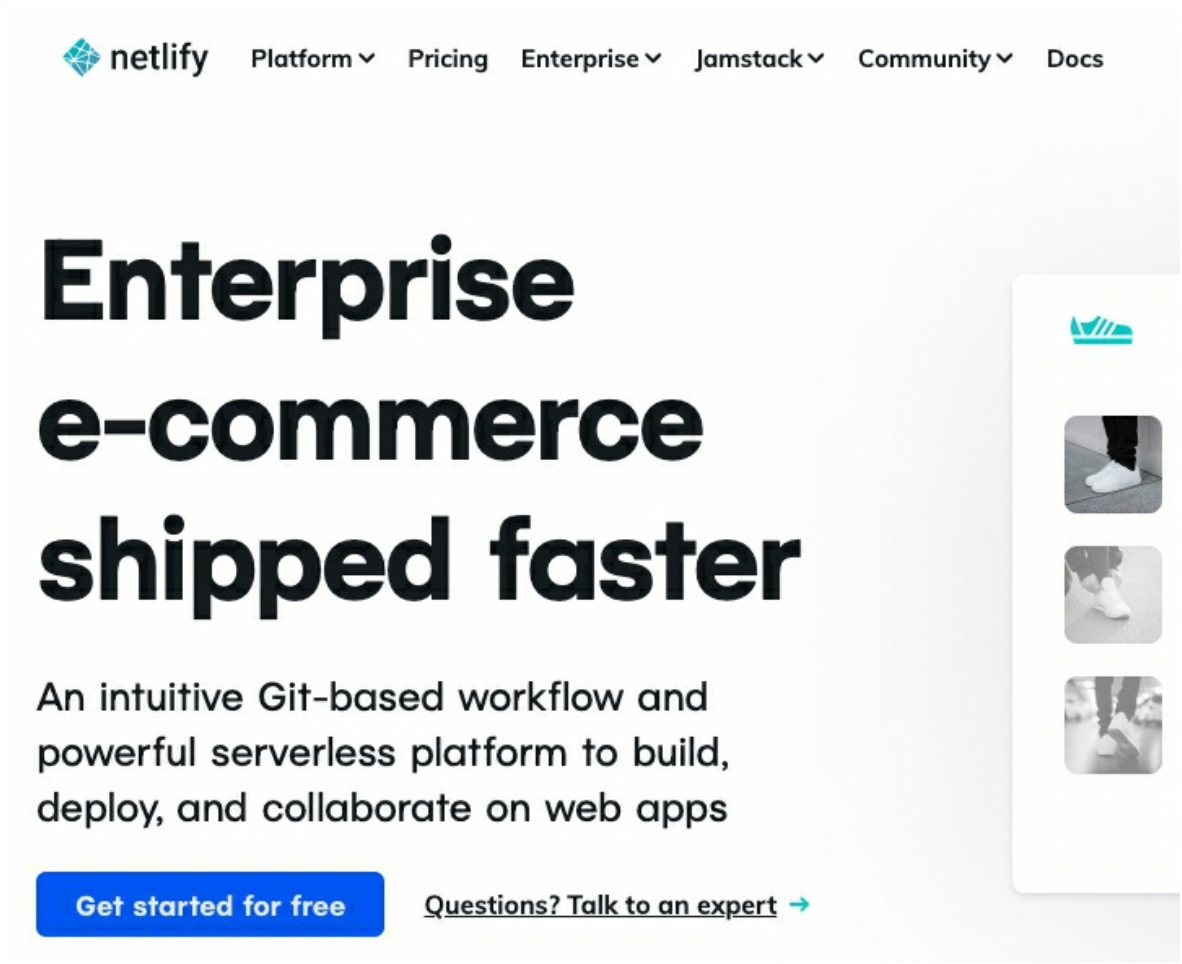


Figure 1

Before deploying it to Netlify, we have to replace the localhost:8000 URL in our React project *services/todos.js* (since we have already deployed our backend on PythonAnywhere). For example:

**Modify Bold Code**

```
getAll(token){
   axios.defaults.headers.common["Authorization"] = "Token " + token;
 return axios.get('https://greglim.pythonanywhere.com/api/todos/');
}
```

Do this for all the API calls.

Now, back in netlify.com, create a free account or log in if you already have one.

When you log in, a list will show any deployed apps that you have. In your case, it will be empty since this is probably your first time deploying on Netlify. At the bottom, there will be a box with the message, “Want to deploy a new site without connecting to Git? Drag and drop your site output folder here” (fig. 2).

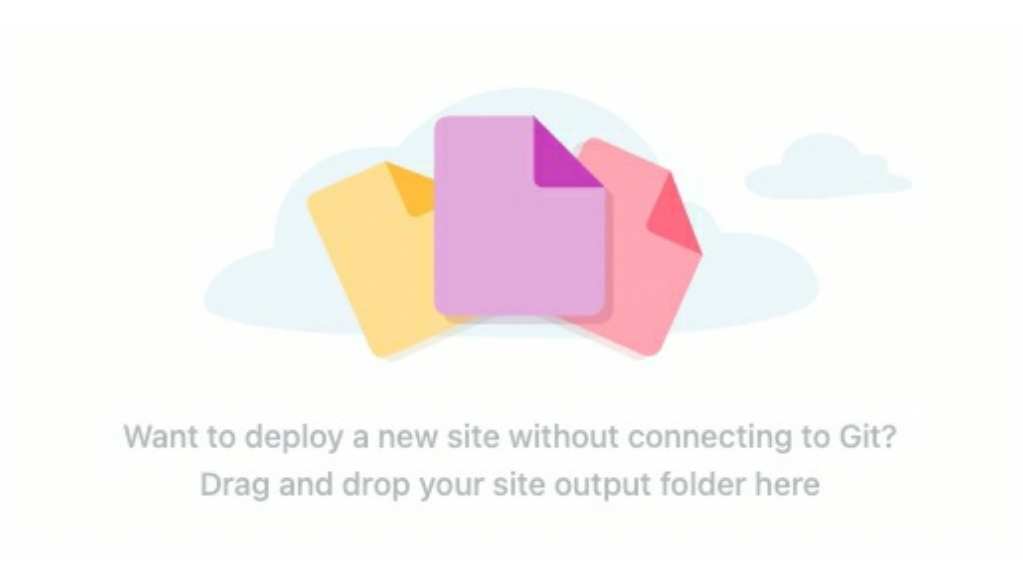


Figure 2

Go back to the Terminal and navigate to the *todoapp/frontend* folder. Build your React application with the command:

**Execute in Terminal**

```
npm run build
```

This will create a build version of React that we can deploy on the web. When the build is finished, you will be able

to see a *build* folder in the directory.

Select the *build* folder and drag and drop it into the box we saw earlier in Netlify. Netlify will take a few seconds and then generate a url where you can access the page (fig. 3).

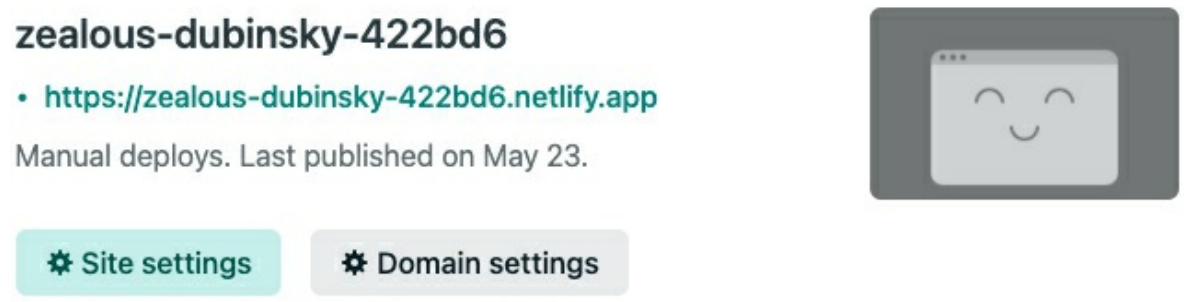


Figure 3

If you wish to have your own custom domain, you can go to ‘Add custom domain’ to purchase one.

## *PythonAnywhere WhiteList*

Now that we have the Netlify URL, back in our backend on PythonAnywhere, we need to replace *localhost* in CORS_ORIGIN_WHITELIST in *todoapp/backend/settings.py*. Go to the ‘Files’ tab, navigate through the source code directory, and in *settings.py*, under CORS_ORIGIN_WHITELIST , replace *localhost* with the Netlify URL. In my case, it is:

**Modify Bold Code**

```
CORS_ORIGIN_WHITELIST = [
    'https://nifty-ride-711a17.netlify.app/'
]
```

This whitelists our Netlify frontend so that it can connect to the Django API backend. Save the file. Then go to the ‘ Web ’ tab and hit the Reload button for your domain (fig. 29.9).

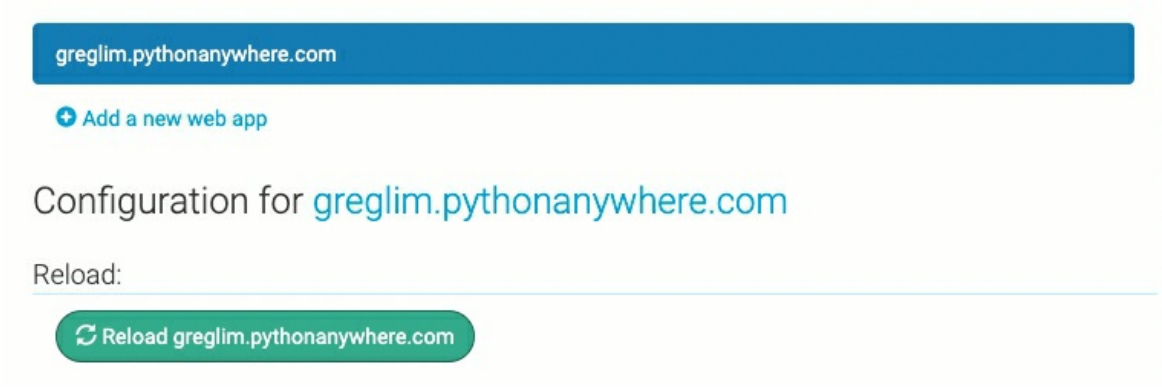


Figure 29.9

And there you have it! Both your React frontend and Django API backend are deployed to the app, meaning that your fully functioning Django API + React app is live and running.

# Final Words

We have gone through quite a lot of content to equip you with the skills to create a Django API React stack app.

Hopefully, you have enjoyed this book and would like to learn more from me. I would love to get your feedback, learning what you liked and didn't for us to improve.

Please feel free to email me at support@i-ducate.com to get updated versions of this book.

If you didn't like the book, or if you feel that I should have covered certain additional topics, please email us to let us know. This book can only get better thanks to readers like you.

If you like the book, I would appreciate if you could leave us a review too. Thank you and all the best for your learning journey in Django API React stack development!

# About the Author

Greg Lim is a technologist and author of several programming books. Greg has many years in teaching programming in tertiary institutions and he places special emphasis on learning by doing.

Contact Greg at support@i-ducate.com or on Twitter at @greglim81

# About the Co-Author

Daniel Correa is a researcher and has been a software developer for several years. Daniel has a Ph.D. in Computer Science; currently he is a professor at Universidad EAFIT in Colombia. He is interested in software architectures, frameworks (such as Laravel, Django, Express, Nest, Vue, React, Angular, and many more), web development, and clean code.

Daniel is very active on Twitter; he shares tips about software development and reviews software engineering books. Contact Daniel on Twitter at @danielgarax

"Hecho en Medell í n"

---

[DCB1]Remember to update this. Seems to be that reportWebVitals is the new one.
[JL2]
[DCB3]I had to use this, to avoid the definition of a parent (such as div), because if I wrapped this in a div, the menu looked strange (it merged these two links in one column – one above the another).

This is a short snippet for a React Fragment. Check the second answer: https://stackoverflow.com/questions/28371370/how-do-i-render-sibling-elements-without-wrapping-them-in-a-parent-tag
[DCB4]Maybe a short explanation about this, will be helpful.
[DCB5]Be careful, the original code didn’t show this tag. But it was presented before. Do you remove it? Should we remove it?
[JL6]Can keep it
[DCB7]Always include the final semicolon, to make the code look more consistent and professional.
[DCB8]Without this, my code didn’t work!
[DCB9]This one was missing.
[JL10]