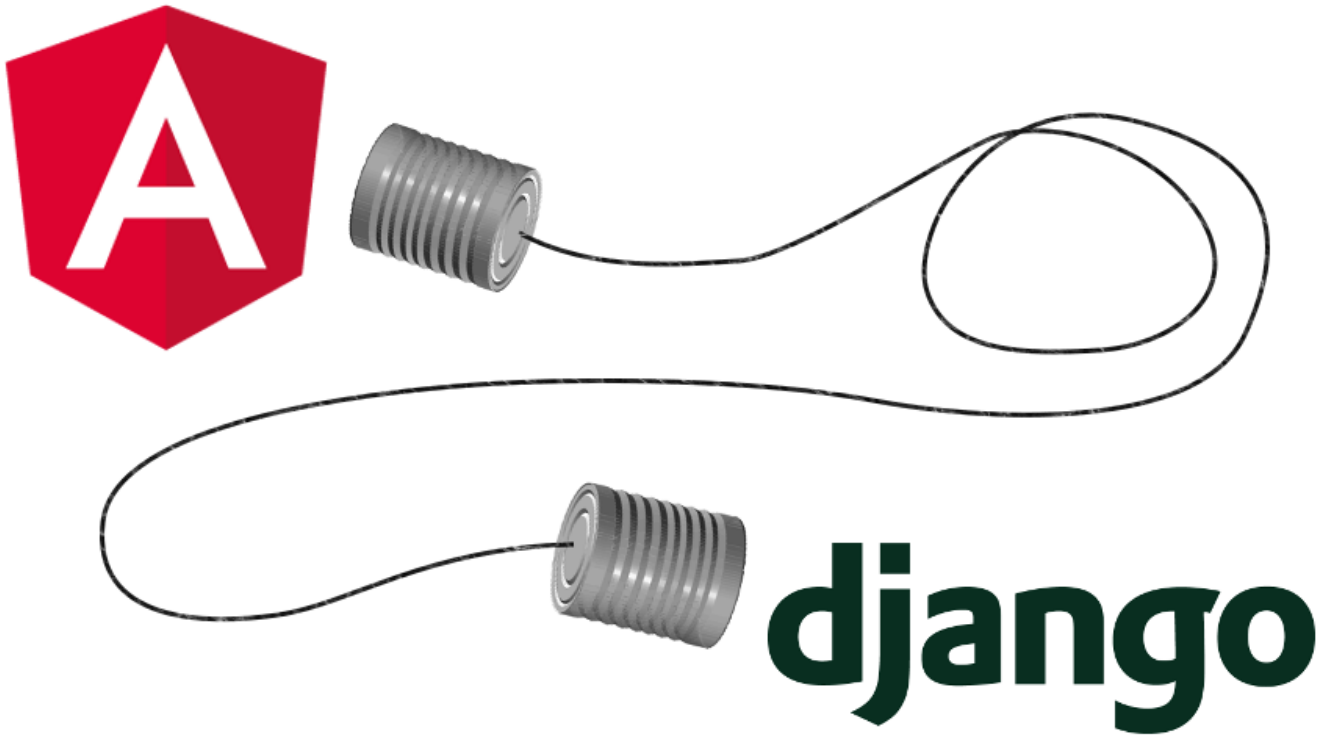


Angular API Calls with Django, Part 2: Building the Micro-Blog App

by Keith Dechant , Software Architect

Filed under: AngularJS (/help/angularjs) | Django (/help/django) | Python (/help/python)

This is the second part of a two-part series exploring the use of the Angular 6 HttpClient to make API calls against a Django back-end using Django Rest Framework. In the first part (/blog/angular-api-calls-django-authentication-jwt), we learned how to authenticate

against Django using the Django Rest Framework JWT package. This post demonstrates how to set up the Django models and views for the micro-blogging app, as well as the Angular Service and templates.

Try this at home

The source code for this tutorial can be found on GitHub at https://github.com/kdechant/angular-django-example (https://github.com/kdechant/angular-django-example).

The example code requires Python 3.5 or higher. It uses a SQLite database, so install should be easy and painless.

Overview

This is a decoupled application using two different frameworks. Thus, the data has to flow through several steps to get from the database to the user's screen. Here is a simple diagram of the data flow:

SQLite Database → Django Model → Django Serializer → Django ViewSet → Angular Service → Angular Component

This tutorial will explore how to create each of these pieces for our micro-blogging application.

The Django App

The Django models

Our micro-blog application contains two relevant data models: the User and the BlogPost. The User model is provided out-of-the-box by Django, so in our app's models, we only need to define the model class for our BlogPost. This is quite simple and defines a few fields, plus the magic Python "to-string" method called `__str__()`.

microblog/models.py:

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

class BlogPost(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    date = models.DateTimeField(
        default=timezone.now
    )
    body = models.CharField(default='', max_length=200)

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

Once the model class is created, you can create and run Django migrations (https://docs.djangoproject.com/en/2.1/topics/migrations/) to persist the schema to the database. However, if you use the working example code on GitHub (https://github.com/kdechant/angular-django-example), the SQLite database already contains the necessary schema, and some test data as well.

The Serializer

Django Rest Framework works on a concept of Models, Serializers, and ViewSets. A Serializer describes how to convert the model objects to JSON and back. The Django ORM uses model classes, and the API endpoints use JSON. The Serializer is how one gets translated to the other.

microblog/serializers.py:

```
from django.contrib.auth.models import User
from rest_framework import serializers
from .models import BlogPost

class UserSerializer(serializers.ModelSerializer):

    class Meta:
        model = User
        fields = ('id', 'username', 'first_name', 'last_name')

class BlogPostSerializer(serializers.ModelSerializer):
    user = serializers.StringRelatedField(many=False)

    class Meta:
        model = BlogPost
        fields = ('id', 'user', 'date', 'body')
```

Unlike our model classes above, we do need to provide a serializer for the User model. Django Rest Framework does not proivde this for us, even though the User model itself comes directly from Django.

In addition, we create a BlogPostSerializer class, which defines which fields on the BlogPost model should be converted to JSON. Notice that we also define a relationship to the User class. This means that, when we serialize a BlogPost object, the resulting JSON will contain a nested object which is a serialized User model. The JSON representation of a BlogPost might look like this:

```
{
    "id": 1
    "user": {
        "id": 10,
        "username": "alice123",
        "first_name": "Alice",
        "last_name": "Smith"
    },
    "date": 1528071221,
    "body": "The quick brown fox jumped over the lazy dog"
}
```

The Viewset

The next thing we need for Django Rest Framework is the ViewSet. This is a collection of Django Views (a.k.a., controllers) which are contained within a class.

microblog/views.py:

```
from django.shortcuts import render
from django.contrib.auth.models import User
from rest_framework import viewsets, permissions
from .models import BlogPost
from . import serializers
from .permissions import ReadOnly

def index(request, path=''):
    return render(request, 'index.html')

class UserViewSet(viewsets.ModelViewSet):
    """
    Provides basic CRUD functions for the User model
    """
    queryset = User.objects.all()
    serializer_class = serializers.UserSerializer
    permission_classes = (ReadOnly, )

class BlogPostViewSet(viewsets.ModelViewSet):
    """
    Provides basic CRUD functions for the Blog Post model
    """
    queryset = BlogPost.objects.all()
    serializer_class = serializers.BlogPostSerializer
    permission_classes = (permissions.IsAuthenticatedOrReadOnly, )

    def perform_create(self, serializer):
        serializer.save(user=self.request.user)
```

Like the Serializers above, we have a ViewSet for both the User model and the BlogPost model.

In this example, the app does not allow editing the User models through the API, so the UserViewSet is configured with `permission_classes = (ReadOnly, )` preventing the API from writing to it. How do you edit the user profiles, you ask? That's out of scope for this example, but the built-in Django admin app provides a way to do this should you need to. When building your own app, you might want to create an Angular component to allow users to edit their profiles, and change the ReadOnly permission to a different permission, allowing the users to edit their own profiles and no one else's.

The BlogPostViewSet class has a different permission class called IsAuthenticatedOrReadOnly. This allows the public to view any blog posts in our app, but only users who are logged in can post.

The Django URL configuration

We also need to add the ViewSet's URLs to our app's URL configuration.

First, we add the ViewSet URLs to our microblog app:

microblog/urls.py

```
from django.urls import path, include
from rest_framework import routers

from . import views

router = routers.DefaultRouter(trailing_slash=False)
router.register(r'posts', views.BlogPostViewSet)
router.register(r'users', views.UserViewSet)

urlpatterns = [
    path(r'api/', include(router.urls)),
    path(r'', views.index, name='index')
]
```

Next, add the app's URLs to the project's URLs:

angular_django_example/urls.py

```
...
urlpatterns = [
    path('admin/', admin.site.urls),
    path(r'', include('microblog.urls')),   # add this
    path(r'api-token-auth/', obtain_jwt_token),
    path(r'api-token-refresh/', refresh_jwt_token),
]
```

Trying out the API

Django Rest Framework comes with the Swagger API viewer already built in. To see what's available in the API, all you need to do is run `python manage.py runserver` and visit localhost:8000/api to see the API endpoints in action.

Calling the API from Angular

We have completed the setup of the back end of the application. Now it's time to create our Angular components and services. These will call the API endpoints and also provide the user interface for the app.

In the previous post, we already set up the Django templates needed to serve the Angular app, so we can dive right in to the Angular code itself.

The Component and Template

In Part 1, we created the basic Angular component. Now, all we need is to add some additional properties and methods for working with the Blog Posts.

microblog/front-end/app.component.ts

```
import {Component, OnInit} from '@angular/core';
import {BlogPostService} from './blog_post.service';   # add this
import {UserService} from './user.service';
import {throwError} from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {

  /**
   * An array of all the BlogPost objects from the API
   */
  public posts;

  /**
   * An object representing the data in the "add" form
   */
  public new_post: any;

  constructor(private _blogPostService: BlogPostService, private _userS

  ngOnInit() {
    this.getPosts();
    this.new_post = {};
    this.user = {
      username: '',
      password: ''
    };
  }

  ... user login/logout methods are unchanged from part 1. omitted for

  getPosts() {
    this._blogPostService.list().subscribe(
      // the first argument is a function which runs on success
      data => {
        this.posts = data;
        // convert the dates to a nice format
        for (let post of this.posts) {
          post.date = new Date(post.date);
        }
      },
      // the second argument is a function which runs on error
      err => console.error(err),
      // the third argument is a function which runs on completion
      () => console.log('done loading posts')
    );
  }
```

```
  createPost() {
    this._blogPostService.create(this.new_post, this.user.token).subscr
       data => {
         // refresh the list
         this.getPosts();
         return true;
       },
       error => {
         console.error('Error saving!');
         return throwError(error);
       }
    );
  }

}
```

Our component has two new properties: 'posts' represents an array of blog posts we received from the API. The 'new_post' property contains a simple JavaScript object. This represents the data in the form field used to create a new post.

Also, there are two methods: getBlogPosts() will query the API via the BlogPostService, and populate the 'posts' object. createPost() takes the data in 'new_post' and sends it off to the API.

In our Angular template file, we add the list of blog posts and the form to create a new post.

microblog/front-end/app.component.html

```
...
<h2 class="mt-3">Micro Blog Posts</h2>
<div *ngFor="let post of posts">
  <div class="row mb-3">
    <label class="col-md-2">By:</label>
    <div class="col-md-2 mb-1">{{ post.user }}</div>
    <label class="col-md-2">Date:</label>
    <div class="col-md-6">{{ post.date }}</div>
    <div class="col-md-12">{{ post.body }}</div>
  </div>
</div>

<h3>Create a new post:</h3>

<div class="row mb-1">
  <label class="col-md-3">Enter your post:</label>
  <div class="col-md-9 mb-1"><input type="text" name="body" [(ngModel)]
  <div class="col-md-2 offset-3">
    <button (click)="createPost()" class="btn btn-primary">Save</button
  </div>
</div>
```

The Angular Blog Post Service

In Part 1, we created a UserService class in Angular to handle API calls related to user login and token management. Now, we will create a second service, the BlogPostService, which handles connections to the BlogPost API endpoints we created above.

Here is the final piece of our puzzle, the Blog Post Service:

microblog/front-end/blog_post.service.ts

```
import {Injectable} from '@angular/core';
import {HttpClient, HttpHeaders} from '@angular/common/http';
import {UserService} from './user.service';

@Injectable()
export class BlogPostService {

  constructor(private http: HttpClient, private _userService: UserServi

  // Uses http.get() to load data from a single API endpoint
  list() {
    return this.http.get('/api/posts');
  }

  // send a POST request to the API to create a new data object
  create(post, token) {
    httpOptions = {
      headers: new HttpHeaders({
        'Content-Type': 'application/json',
        'Authorization': 'JWT ' + this._userService.token   // this is
      })
    };
    return this.http.post('/api/posts', JSON.stringify(post), httpOptio
  }
}
```

Notice the list() and create() methods of our service. These create an Observable using Angular's HttpClient and return it to the caller. This is how the component gets its data.

Using CSRF tokens

Remember in the ViewSet code above that the Blog Post endpoints had the "is authenticated or read only" permission applied to them? When we POST data to the API, we also need a way to tell the API that the user is authenticated.

So, let's look back to Part 1 (/blog/angular-api-calls-django-authentication-jwt), where we called the authentication API endpoint to get the JSON Web Token. We can use this token to indicate to the API that the user is logged in, and also the user's identity. All we need to do is add the 'Authorization' header to the HTTP headers we send with the API request. When using JWT, the value should be "JWT" followed by a space, then by the entire token.

Here's a full example of an Authorization header:

```
Authorization: JWT eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI
```

Note that, in this example, we only need to send the header when *writing* to the API. Since anonymous users can see blog posts, and they won't have a JWT token, we don't send the Authorization header with the GET requests made by the list() method. However, in a different application where the content isn't visible to anonymous users, we would need to send the Authorization header to the GET request as well.

Further Reading

Looking for more examples of Angular API calls using HttpClient and ForkJoin? See how I used Angular, Django Rest Framework, HttpClient, and ForkJoin to rebuild a classic text adventure game (https://www.kdechant.com/blog/angular-text-adventure-part-3-the-game-data-models).

For the front-end developers, you may be interested in some examples of Angular animations (https://www.kdechant.com/blog/angular-animations-fade-in-and-fade-out).

Happy coding!

Date posted: September 9, 2018

Comments

> Fri, 09/21/2018 - 17:02
>
> First at all I would like to say thank you for a great post that you did.
> How can I use data from my token inside a navbar component that I created.

> Sun, 09/23/2018 - 12:09
>
> If I log in and hit "ctrl + R" then my user sees login screen again. How to have a persistent session?

> Sun, 01/20/2019 - 17:31
>
> which url should i access, the django or angular's ? The angular URL is not loading any of the UI or html part.

Add new comment

Your name

Subject

Comment *

I'm not a robot

reCAPTCHA

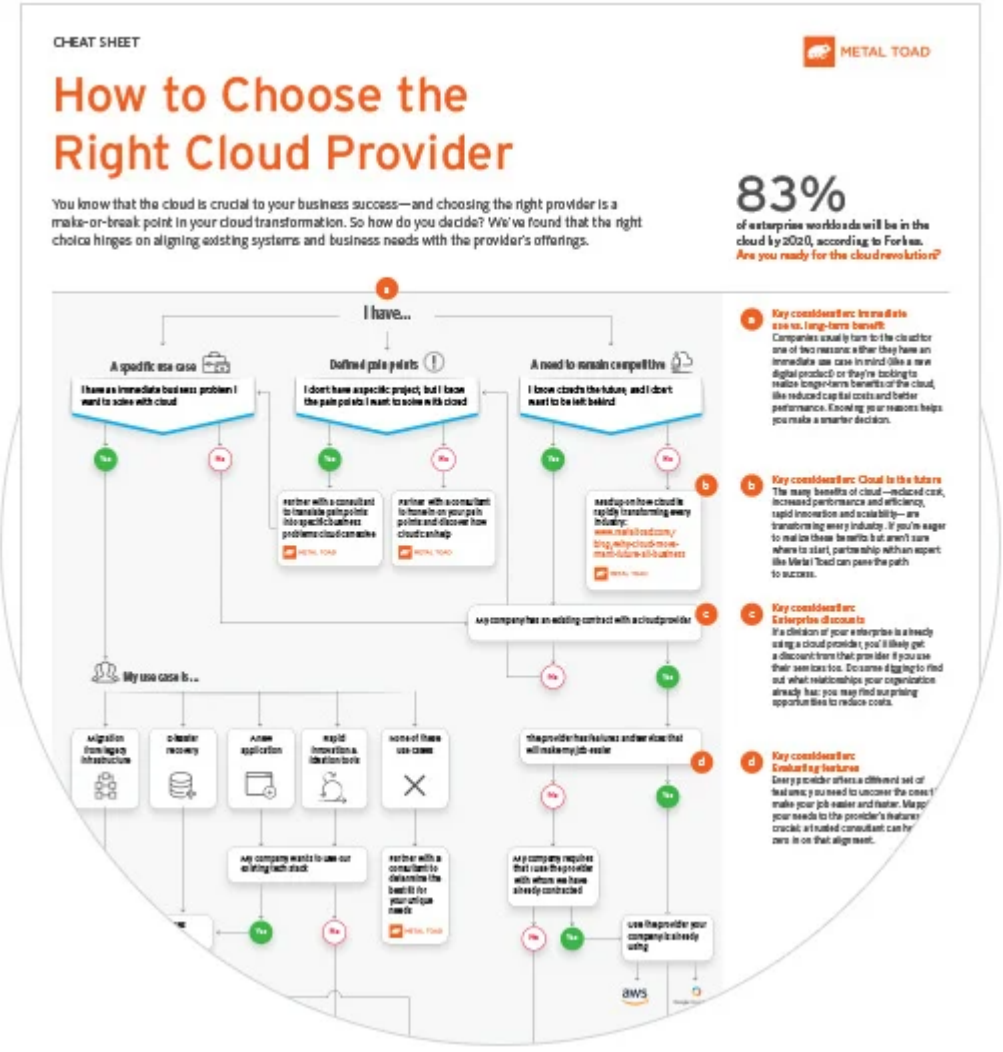
Privacy - Terms

Save Preview

Subscribe to our newsletter*

Sign up

FREE Whitepaper Download



(https://metaltoad-6104926.hs-sites.com/landing-page/choose-cloud-cheat-sheet)

Have questions?

GET IN TOUCH (/CONTACT)

- Values (/blog/metal-toad-corporate-values) • Privacy Policy (/privacy-policy)
- Careers (/careers-metal-toad) • AWS Lambda (/services/aws-lambda)
- AWS Los Angeles (/los-angeles-aws-consultant)
- AWS Media & Entertainment (/industries/media-and-entertainment)
- AWS Oregon (/oregon-aws-consultant) • Cloud Hosting (/cloud-hosting-solutions)
- Technical Help (/help) • Top 10 React JS Tips (/help/top-10-reactjs-tips)
- Top 20 Drupal Tips (/help/top-20-drupal-tips)