Learning Django and AngularJS

In this tutorial you will build a simplified Google+ clone called “Not Google Plus” with Django and AngularJS.

Before we hit the proverbial books and learn to build a rich, modern web application with Django and Angular, let's take a moment to explore the motivations behind this tutorial and how you can get the most out of it.

What is the goal of this tutorial?

Here at Thinkster, we strive to create high value, in depth content while maintaining a low barrier to entry. We release this content for free with the hope that you find it both exciting as well as informative.

Each tutorial we release has a specific goal. In this tutorial, that goal is to give you a brief overview of how Django and AngularJS play together and how these technologies can be combined to build amazing web applications. Furthermore, we place a heavy emphasis on building good engineering habits. This includes everything from considering the tradeoffs that come from making architectural decisions, to maintaining high quality code throughout your project. While these things may not sound like fun, they are key to becoming a well-rounded software developer.

Who is this tutorial for?

Every author must answer this difficult question. Our goal is to make this tutorial useful for novices as well as experienced developers.

For those of your who are in the early days of your software development careers, we have tried to be thorough and logical in our explanations as possible, while still making the text flow fluidly; we try to avoid making intuitive leaps where doing so makes sense.

For those of you who have been around the block a few times, and perhaps are just interested in learning more about Django or AngularJS, we know you don't need the basics explained to you. One of our goals when writing this tutorial was to make it easy to skim. This allows you to use your existing knowledge to speed up the reading process and identify where unfamiliar concepts are presented so you can grok them quickly and move on.

We want to make this tutorial accessible to anyone with enough interest to take the time necessary to learn and understand the concepts presented.

A brief interlude about formatting

Throughout this tutorial, we strive to maintain consistent formatting. This section details what that formatting looks like and what it means.

- When presenting a new code snippet, we will present the snippet in it's entirety and then walk through it line-by-line as necessary to cover new concepts.
- Variable names and file names appear in-line with special formatting: `thinkster_django_angular/settings.py` .
- Longer code snippets will appear on their own lines:

```
def is_this_google_plus():
    return False
```

- Terminal commands also appear on their own line, prefixed by a `$` :

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

- Unless otherwise specified, you should assume that all terminal commands are run from the root directory of your project.

A word on code style

Where possible, we opt to follow style guides created by the Django and Angular communities.

For Django, we follow PEP8 (http://legacy.python.org/dev/peps/pep-0008/) strictly and try our best to adhere to Django Coding style (https://docs.djangoproject.com/en/1.7/internals/contributing/writing-code/coding-style/).

For AngularJS, we have adopted John Papa's AngularJS Style Guide (https://github.com/johnpapa/angularjs-styleguide). We also adhere to Google's JavaScript Style Guide (https://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml) where it makes sense to do so.

A humble request for feedback

At the risk of sounding cliche, we would not have a reason to make this tutorial if not for you. Because we believe that your success is our success, we invite you to contact us with any thoughts you have about the tutorial. You can reach us via Twitter at @jamesbrwr (http://twitter.com/jamesbrwr) or @GoThinkster (http://twitter.com/gothinkster), or by emailing support@thinkster.io (mailto:support@thinkster.io).

We welcome criticism openly and accept praise if you believe it is warranted. We're interested in knowing what you like, what you don't like, what you want to know more about, and anything else you feel is relevant.

If you are too busy to reach out to us, that's OK. We know that learning takes a lot of work. If, on the other hand, you want to help us build something amazing, we await your mail.

amazing, we await your mail.

A final word before we begin

It is our experience that the developers who gain the most from our tutorials are the ones who take an active approach to their learning.

We **strongly** recommend you type out the code for yourself. When you copy and paste code, you don't interact with it and that interaction is in turn what makes you a better developer.

In addition to typing the code yourself, do not be afraid to get your hands dirty; jump in and play around, break things and build missing features. If you encounter a bug, explore and figure out what is causing it. These are the obstacles we as engineers must tackle multiple times a day, and have thus learned to embrace these explorations as the best source of learning.

Let's build some software.

Setting up your environment

The application we will be building requires a non-trivial amount of boilerplate. Instead of spending time setting up your environment, which is not the purpose of this tutorial, we have created a boilerplate project to get you started.

You can find the boilerplate project on Github at brwr/thinkster-django-angular-boilerplate (https://github.com/brwr/thinkster-django-angular-boilerplate). The repository includes a list of commands you need to run to get everything running.

Go ahead and follow the setup instructions now.

Follow the instructions to set up your environment

Checkpoint

If all went well running the server with `python manage.py runserver` should allow you to visit `http://localhost:8000/` in your browser. The page will be blank except for the navigation bar at the top. The links in the navigation bar currently do nothing.

Make sure your environment is running by visiting `http://localhost:8000/`

Extending Django's built-in User model

Django has a built-in `User` model that offers a lot of functionality. The problem with `User` is that the model cannot be extended to include more information. For example, we will be giving our users a tagline to display on their profile. `User` does not have a tagline attribute and we cannot add one ourselves.

`User` inherits from `AbstractBaseUser`. That is where `User` gets most of it's functionality. By creating a new model called `Account` and inheriting from `AbstractBaseUser`, we will get the necessary functionality of `User` (password hashing, session management, etc) and be able to extend `Account` to include extra information, such as a tagline.

In Django, the concept of an "app" is used to organize code in a meaningful way. An app is a module that houses code for models, view, serializers, etc that are all related in some way. By way of example, our first step in building our Django and AngularJS web application will be to create an app called `authentication`. The `authentication` app will contain the code relevent to the `Account` model we just talked about as well as views for logging in, logging out and register.

Make a new app called `authentication` by running the following command:

```
$ python manage.py startapp authentication
```

Make a Django app named `authentication`

Creating the Account model

To get started, we will create the `Account` model we just talked about.

Open `authentication/models.py` in your favorite text editor and edit it to reflect the following:

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

class Account(AbstractBaseUser):
    email = models.EmailField(unique=True)
    username = models.CharField(max_length=40, unique=True)

    first_name = models.CharField(max_length=40, blank=True)
```

```
    first_name = models.CharField(max_length=40, blank=True)
    last_name = models.CharField(max_length=40, blank=True)
    tagline = models.CharField(max_length=140, blank=True)

    is_admin = models.BooleanField(default=False)

    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    objects = AccountManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['username']

    def __unicode__(self):
        return self.email

    def get_full_name(self):
        return ' '.join([self.first_name, self.last_name])

    def get_short_name(self):
        return self.first_name
```

Make a new model in `authentication/models.py` called `Account`
Let's take a closer look at each attribute and method in turn.

```
email = models.EmailField(unique=True)

# ...

USERNAME_FIELD = 'email'
```

Django's built-in `User` requires a username. That username is used for logging the user in. By contrast, our application will use the user's email address for this purpose.

To tell Django that we want to treat the email field as the username for this model, we set the `USERNAME_FIELD` attribute to `email`. The field specified by `USERNAME_FIELD` must be unique, so we pass the `unique=True` argument in the email field.

```
username = models.CharField(max_length=40, unique=True)
```

Even though we will log in with our email address, we still want the user to have a username. We need some to display on their posts and profile page. We will also use the username in our URLs, so the username must be unique. To this end, we pass the `unique=True` argument in the username field.

```
first_name = models.CharField(max_length=40, blank=True)
last_name = models.CharField(max_length=40, blank=True)
```

Ideally we should have a more personal way to reference our users. Because we understand that not all users are comfortable giving out their personal details, we make the `first_name` and `last_name` fields optional by passing the `blank=True` argument.

```
tagline = models.CharField(max_length=140, blank=True)
```

As mentioned before, the `tagline` attribute will be displayed on the user's profile. This gives the profile a hint of the user's personally, so it is worth including.

```
created_at = models.DateTimeField(auto_now_add=True)
updated_at = models.DateTimeField(auto_now=True)
```

The `created_at` field records the time that the `Account` object was created. By passing `auto_now_add=True` to `models.DateTimeField`, we are telling Django that this field should be automatically set when the object is created and non-editable after that.

Similar to `created_at`, `updated_at` is automatically set by Django. The difference between `auto_now_add=True` and `auto_now=True` is that `auto_now=True` causes the field to update each time the object is saved.

```
objects = AccountManager()
```

When you want to get a model instance in Django, you use an expression of the form `Model.objects.get(**kwargs)`. The `objects` attribute here is a `Manager` class whose name typically follows the `<model name>Manager` convention. In our case, we will create an `AccountManager` class. We will do this momentarily.

```
REQUIRED_FIELDS = ['username']
```

We will be displaying the username in multiple places. To this end, having a username is not optional, so we include it in the `REQUIRED_FIELDS` list. Normally the `required=True` argument would accomplish this goal, but because this model is replacing the `User` model, Django requires us to specify required fields in this way.

```
def __unicode__(self):
    return self.email
```

When working in the shell, as we will see shortly, the default string representation of an `Account` object looks something like `<Account: Account>`. Because we will have many different accounts, this is not very helpful. Overwriting `__unicode__()` will change this default behavior. Here we choose to show the user's email instead. The string representation of an account with the email `james@notgoogleplus.com` will now look like `<Account: james@notgoogleplus.com>`.

```
    def get_full_name(self):
        return ' '.join([self.first_name, self.last_name])

    def get_short_name(self):
        return self.first_name
```

`get_full_name()` and `get_short_name()` are Django conventions. We won't be using either of these methods, but it is still a good idea to include them to comply with Django conventions.

Making a Manager class for Account

When substituting a customer user model, it is required that you also define a related `Manager` class the overrides the `create_user()` and `create_superuser()` methods.

With `authentication/models.py` still open, add the following class above the `Account` class:

```
from django.contrib.auth.models import BaseUserManager


class AccountManager(BaseUserManager):
    def create_user(self, email, password=None, **kwargs):
        if not email:
            raise ValueError('Users must have a valid email address.')

        if not kwargs.get('username'):
            raise ValueError('Users must have a valid username.')

        account = self.model(
            email=self.normalize_email(email), username=kwargs.get('username')
        )

        account.set_password(password)
        account.save()

        return account

    def create_superuser(self, email, password, **kwargs):
        account = self.create_user(email, password, **kwargs)

        account.is_admin = True
        account.save()

        return account
```

Like we did with `Account`, let's step through this file line-by-line. We will only cover new information.

```
if not email:
    raise ValueError('Users must have a valid email address.')

if not kwargs.get('username'):
    raise ValueError('Users must have a valid username.')
```

Because users are required to have both an email and a username, we should raise an error if either of these attributes are missing.

```
account = self.model(
    email=self.normalize_email(email), username=kwargs.get('username')
)
```

Since we haven't defined a `model` attribute on the `AccountManager` class, `self.model` refers to the `model` attribute of `BaseUserManager`. This defaults to `settings.AUTH_USER_MODEL`, which we will change in just a moment to point to the `Account` class.

```
account = self.create_account(email, password, **kwargs)

account.is_admin = True
account.save()
```

Writing the same thing more than once sucks. Instead of copying all of the code from `create_account` and pasting it in `create_superuser`, we simply let `create_user` handle the actual creation. This frees up `create_superuser` to only worry about turning an `Account` into a superuser.

Changing the Django AUTH_USER_MODEL setting

Even though we have created this `Account` model, the command `python manage.py createsuperuser` (which we will talk more about shortly) still creates `User` objects. This is because, at this point, Django still believes that `User` is the model we want to use for authentication.

To set things straight and start using `Account` as our authentication model, we have to update `settings.AUTH_USER_MODEL`.

Open `thinkster_django_angular_tutorial/settings.py` and add the following to the end of the file:

```
AUTH_USER_MODEL = 'authentication.Account'
```

Change `settings.AUTH_USER_MODEL` to use `Account` instead of `User`
This line tells Django to look in the `authentication` app and find a model named `Account`.

Installing your first app

In Django, you must explicitly declare which apps are being used. Since we haven't added our `authentication` app to the list of installed apps yet, we will do that now.

Open `thinkster_django_angular_boilerplate/settings.py` and append `'authentication',` to `INSTALLED_APPS` like so:

```
INSTALLED_APPS = (
    ...,
    'authentication',
)
```

Install the `authentication` app

Migrating your first app

When Django 1.7 was released, it was like Christmas in September! Migrations had finally arrived!

Anyone with a background in Rails will find the concept of migrations familiar. In short, migrations handle the SQL needed to update the schema of our database so you don't have to. By way of example, consider the `Account` model we just created. These models need to be stored in the database, but our database doesn't have a table for `Account` objects yet. What do we do? We create our first migration! The migration will handle adding the tables to the database and offer us a way to rollback the changes if we make a mistake.

When you're ready, generate the migrations for the `authentication` app and apply them:

```
$ python manage.py makemigrations
Migrations for 'authentication':
    0001_initial.py:
        - Create model Account
$ python manage.py migrate
Operations to perform:
    Synchronize unmigrated apps: rest_framework
    Apply all migrations: admin, authentication, contenttypes, auth, sessions
Synchronizing apps without migrations:
    Creating tables...
    Installing custom SQL...
    Installing indexes...
Running migrations:
    Applying authentication.0001_initial... OK
```

> From now on, the output from migration commands will not be included for brevity.

Generate the migrations for the `authentication` app and apply them

Making yourself a superuser

Let's talk more about the `python manage.py createsuperuser` command from a few minutes ago.

Different users have different levels of access in any given application. Some users are admins and can do anywhere, while some are just regular users whose actions should be limited. In Django, a super user is the highest level of access you can have. Because we want the ability to work will all facets of our application, we will create a super user. That is what `python manage.py createsuperuser` does.

After running the command, Django will prompt you for some information and create an `Account` with superuser access. Go ahead and give it a try.

```
$ python manage.py createsuperuser
```

Make a new super user `Account`

Checkpoint

To make sure everything is properly configured, let's take a quick break and open Django's shell:

```
$ python manage.py shell
```

You should see a new prompt: `>>>` . Inside the shell, we can get the `Account` we just created like so:

```
>>> from authentication.models import Account
>>> a = Account.objects.latest('created_at')
```

If everything went well, you should be able to access the various attributes of your `Account` object:

```
>>> a
>>> a.email
>>> a.username
```

Access the `Account` object you just created

Serializing the Account Model

The AngularJS application we are going to build will make AJAX requests to the server to get the data it intends to display. Before we can send that data back to the client, we need to format it in a way that the client can understand; in this case, we choose JSON. The process of transforming Django models to JSON is called serialization and that is what we will talk about now.

As the model we want to serialize is called `Account` , the serializer we will create is going to be called `AccountSerializer` .

Django REST Framework

As part of the boilerplate project you cloned earlier, we have included a project called Django REST Framework. Django REST Framework is a toolkit that provides a number of features common to most web applications, including serializers. We will make use of these features throughout the tutorial to save us both time and frustration. Our first look at Django REST Framework starts here.

AccountSerializer

Before we write our serializers, let's create a `serializers.py` file inside our `authentication` app:

```
$ touch authentication/serializers.py
```

Create a `serializers.py` file inside the `authentication` app
Open `authentication/serializers.py` and add the following code and imports:

```
from django.contrib.auth import update_session_auth_hash

from rest_framework import serializers

from authentication.models import Account


class AccountSerializer(serializers.ModelSerializer):
    password = serializers.CharField(write_only=True, required=False)
    confirm_password = serializers.CharField(write_only=True, required=False)

    class Meta:
        model = Account
        fields = ('id', 'email', 'username', 'created_at', 'updated_at',
                  'first_name', 'last_name', 'tagline', 'password',
                  'confirm_password',)
        read_only_fields = ('created_at', 'updated_at',)

        def create(self, validated_data):
            return Account.objects.create(**validated_data)

        def update(self, instance, validated_data):
            instance.username = validated_data.get('username', instance.username)
            instance.tagline = validated_data.get('tagline', instance.tagline)

            instance.save()

            password = validated_data.get('password', None)
            confirm_password = validated_data.get('confirm_password', None)

            if password and confirm_password and password == confirm_password:
                instance.set_password(password)
                instance.save()

            update_session_auth_hash(self.context.get('request'), instance)

            return instance
```

Make a serializer called `AccountSerializer` in `authentication/serializers.py`

> From here on, we will declare imports that are used in each snippet. These may already be present in the file. If so, they do not need to be added a second time.

Let's take a closer look.

```
password = serializers.CharField(write_only=True, required=False)
confirm_password = serializers.CharField(write_only=True, required=False)
```

Instead of including `password` in the `fields` tuple, which we will talk about in a few minutes, we explicitly define the field at the top of the `AccountSerializer` class. The reason we do this is so we can pass the `required=False` argument. Each field in `fields` is required, but we don't want to update the user's password unless they provide a new one.

`confirm_pssword` is similar to `password` and is used only to make sure the user didn't make a typo on accident.

Also note the use of the `write_only=True` argument. The user's password, even in it's hashed and salted form, should not be visible to the client in the AJAX response.

```
class Meta:
```

The `Meta` sub-class defines metadata the serializer requires to operate. We have defined a few common attributes of the `Meta` class here.

```
model = Account
```

Because this serializers inherits from `serializers.ModelSerializer`, it should make sense that we must tell it which model to serialize. Specifying the model creates a guarantee that only attributes of that model or explicitly created fields can be serialized. We will cover serializing model attributes now and explicitly created fields shortly.

```
fields = ('id', 'email', 'username', 'created_at', 'updated_at',
```

```
        fields = ('id', 'email', 'username', 'created_at', 'updated_at',
                  'first_name', 'last_name', 'tagline', 'password',
                  'confirm_password',)
```

The `fields` attribute of the `Meta` class is where we specify which attributes of the `Account` model should be serialized. We must be careful when specifying which fields to serialize because some fields, like `is_superuser`, should not be available to the client for security reasons.

```
read_only_fields = ('created_at', 'updated_at',)
```

If you recall, when we created the `Account` model, we made the `created_at` and `updated_at` fields self-updating. Because of this feature, we add them to a list of fields that should be read-only.

```
def create(self, validated_data):
    # ...

def update(self, instance, validated_data):
    # ...
```

Earlier we mentioned that we sometimes want to turn JSON into a Python object. This is called deserialization and it is handled by the `.create()` and `.update()` methods. When creating a new object, such as an `Account`, `.create()` is used. When we later update that `Account`, `.update()` is used.

```
instance.username = attrs.get('username', instance.username)
instance.tagline = attrs.get('tagline', instance.tagline)
```

We will let the user update their username and tagline attributes for now. If these keys are present in the arrays dictionary, we will use the new value. Otherwise, the current value of the `instance` object is used. Here, `instance` is of type `Account`.

```
password = validated_data.get('password', None)
confirm_password = validated_data.get('confirm_password', None)

if password and confirm_password and password == confirm_password:
    instance.set_password(password)
    instance.save()
```

Before updating the user's password, we need to confirm they have provided values for both the `password` and `password_confirmation` field. We then check to make sure these two fields have equivelant values.

After we verify that the password should be updated, we much use `Account.set_password()` to perform the update. `Account.set_password()` takes care of storing passwords in a secure way. It is important to note that we must explicitly save the model after updating the password.

> This is a naive implementation of how to validate a password. I would not recommend using this in a real-world system, but for our purposes this does nicely.

```
update_session_auth_hash(self.context.get('request'), instance)
```

When a user's password is updated, their session authentication hash must be explicitly updated. If we don't do this here, the user will not be authenticated on their next request and will have to log in again.

Checkpoint

By now we should have no problem seeing the serialized JSON of an `Account` object. Open up the Django shell again by running `python manage.py shell` and try typing the following commands:

```
>>> from authentication.models import Account
>>> from authentication.serializers import AccountSerializer
>>> account = Account.objects.latest('created_at')
>>> serialized_account = AccountSerializer(account)
>>> serialized_account.data.get('email')
>>> serialized_account.data.get('username')
```

Make sure your `AccountSerializer` serializer is working

Registering new users

At this point we have the models and serializers needed to represent users. Now we need to build an authentication system. This involves creating the various views and interfaces for registering, logging in and logging out. We will also touch on an `Authentication` service with AngularJS and a few different controllers.

Because we can't log in users that don't exist, it makes sense to start with registration.

To register a user, we need an API endpoint that will create an `Account` object, an AngularJS service to make an AJAX request to the API and a registration form. Let's make the API endpoint first.

Making the account API viewset

Open `authentication/views.py` and replace it's contents with the following code:

```
from rest_framework import permissions, viewsets

from authentication.models import Account
from authentication.permissions import IsAccountOwner
```

```
from authentication.permissions import IsAccountOwner
from authentication.serializers import AccountSerializer


class AccountViewSet(viewsets.ModelViewSet):
    lookup_field = 'username'
    queryset = Account.objects.all()
    serializer_class = AccountSerializer

    def get_permissions(self):
        if self.request.method in permissions.SAFE_METHODS:
            return (permissions.AllowAny(),)

        if self.request.method == 'POST':
            return (permissions.AllowAny(),)

        return (permissions.IsAuthenticated(), IsAccountOwner(),)

    def create(self, request):
        serializer = self.serializer_class(data=request.data)

        if serializer.is_valid():
            Account.objects.create_user(**serializer.validated_data)

            return Response(serializer.validated_data, status=status.HTTP_201_CREATED)

        return Response({
            'status': 'Bad request',
            'message': 'Account could not be created with received data.'
        }, status=status.HTTP_400_BAD_REQUEST)
```

Make a viewset called `AccountViewSet` in `authentication/views.py`
Let's step through this snippet line-by-line:

```
class AccountViewSet(viewsets.ModelViewSet):
```

Django REST Framework offers a feature called viewsets. A viewset, as the name implies, is a set of views. Specifically, the `ModelViewSet` offers an interface for listing, creating, retrieving, updating and destroying objects of a given model.

```
lookup_field = 'username'
queryset = Account.objects.all()
serializer_class = AccountSerializer
```

Here we define the query set and the serialzier that the viewset will operate on. Django REST Framework uses the specified queryset and serializer to perform the actions listed above. Also note that we specify the `lookup_field` attribute. As mentioned earlier, we will use the `username` attribute of the `Account` model to look up accounts instead of the `id` attribute. Overriding `lookup_field` handles this for us.

```
def get_permissions(self):
    if self.request.method in permissions.SAFE_METHODS:
        return (permissions.AllowAny(),)

    if self.request.method == 'POST':
        return (permissions.AllowAny(),)

    return (permissions.IsAuthenticated(), IsAccountOwner(),)
```

The only user that should be able to call dangerous methods (such as `update()` and `delete()`) is the owner of the account. We first check if the user is authenticated and then call a custom permission that we will write in just a moment. This case does not hold when the HTTP method is `POST`. We want to allow any user to create an account.

If the HTTP method of the request ('GET', 'POST', etc) is "safe", then anyone can use that endpoint.

```
def create(self, request):
    serializer = self.serializer_class(data=request.data)

    if serializer.is_valid():
        Account.objects.create_user(**serializer.validated_data)

        return Response(serializer.validated_data, status=status.HTTP_201_CREATED)
    return Response({
        'status': 'Bad request',
        'message': 'Account could not be created with received data.'
    }, status=status.HTTP_400_BAD_REQUEST)
```

When you create an object using the serializer's `.save()` method, the object's attributes are set literally. This means that a user registering with the password `'password'` will have their password stored as `'password'`. This is bad for a couple of reasons: 1) Storing passwords in plain text is a massive security issue. 2) Django hashes and salts passwords before comparing them, so the user wouldn't be able to log in using `'password'` as their password.

We solve this problem by overriding the `.create()` method for this viewset and using `Account.objects.create_user()` to create the `Account` object.

Making the IsAccountOwner permission

Let's create the `IsAccountOwner()` permission from the view we just made.

Create a file called `authentication/permissions.py` with the following content:

```
from rest_framework import permissions


class IsAccountOwner(permissions.BasePermission):
    def has_object_permission(self, request, view, account):
        if request.user:
            return account == request.user
        return False
```

Make a permission called `IsAccountOwner` in `authentication/permissions.py`
This is a pretty basic permission. If there is a user associated with the current request, we check whether that user is the same object as `account`. If there is no user associated with this request, we simply return `False`.

Adding an API endpoint

Now that we have created the view, we need to add it to the URLs file. Open `thinkster_django_angular_boilerplate/urls.py` and update it to look like so:

```
# .. Imports
from rest_framework_nested import routers

from authentication.views import AccountViewSet

router = routers.SimpleRouter()
router.register(r'accounts', AccountViewSet)

urlpatterns = patterns(
     '',
    # ... URLs
    url(r'^api/v1/', include(router.urls)),

    url('^.*$', IndexView.as_view(), name='index'),
)
```

Add an API endpoint for `AccountViewSet`

It is very important that the last URL in the above snippet always be the last URL. This is known as a passthrough or catch-all route. It accepts all requests not matched by a previous rule and passes the request through to AngularJS's router for processing. The order of other URLS is normally insignificant.

An Angular service for registering new users

With the API endpoint in place, we can create an AngularJS service that will handle communication between the client and the server.

Make a file in `static/javascripts/authentication/services/` called `authentication.service.js` and add the following code:

Feel free to leave the comments out of your own code. It takes a lot of time to type them all out!

```
/**
* Authentication
* @namespace thinkster.authentication.services
*/
(function () {
  'use strict';

  angular
    .module('thinkster.authentication.services')
    .factory('Authentication', Authentication);

  Authentication.$inject = ['$cookies', '$http'];

  /**
  * @namespace Authentication
  * @returns {Factory}
  */
  function Authentication($cookies, $http) {
    /**
    * @name Authentication
    * @desc The Factory to be returned
    */
    var Authentication = {
      register: register
    };

    return Authentication;

    ////////////////////

    /**
```

```
    * @name register
    * @desc Try to register a new user
    * @param {string} username The username entered by the user
    * @param {string} password The password entered by the user
    * @param {string} email The email entered by the user
    * @returns {Promise}
    * @memberOf thinkster.authentication.services.Authentication
    */
    function register(email, password, username) {
      return $http.post('/api/v1/accounts/', {
        username: username,
        password: password,
        email: email
      });
    }
  }
})();
```

Make a factory called `Authentication` in `static/javascripts/authentication/services/authentication.service.js`
Let's step through this line-by-line:

```
angular
  .module('thinkster.authentication.services')
```

AngularJS supports the use of modules. Modularization is a great feature because it promotes encapsulation and loose coupling. We make thorough use of Angular's module system throughout the tutorial. For now, all you need to know is that this service is in the `thinkster.authentication.services` module.

```
.factory('Authentication', Authentication);
```

This line registers a factory named `Authentication` on the module from the previous line.

```
function Authentication($cookies, $http) {
```

Here we define the factory we just registered. We inject the `$cookies` and `$http` services as a dependency. We will be using `$cookies` later.

```
var Authentication = {
  register: register
};
```

This is personal preference, but I find it's more readable to define your service as a named object and then return it, leaving the details lower in the file.

```
function register (username, password, email) {
```

At this point, the `Authentication` service has only one method: `register`, which takes a `username`, `password`, and `email`. We will add more methods to the service as we move forward.

```
return $http.post('/api/v1/accounts/', {
  username: username,
  password: password,
  email: email
});
```

As mentioned before, we need to make an AJAX request to the API endpoint we made. As data, we include the `username`, `password` and `email` parameters this method received. We have no reason to do anything special with the response, so we will let the caller of `Authentication.register` handle the callback.

Making an interface for registering new users

Let's begin creating the interface users will use to register. Begin by creating a file in `static/templates/authentication/` called `register.html` with the following content:

```
<div class="row">
  <div class="col-md-4 col-md-offset-4">
    <h1>Register</h1>

    <div class="well">
      <form role="form" ng-submit="vm.register()">
        <div class="form-group">
          <label for="register__email">Email</label>
          <input type="email" class="form-control" id="register__email" ng-model="vm.email" placeholder="ex. john@notgoogle.com" />
        </div>

        <div class="form-group">
          <label for="register__username">Username</label>
          <input type="text" class="form-control" id="register__username" ng-model="vm.username" placeholder="ex. john" />
        </div>

        <div class="form-group">
          <label for="register__password">Password</label>
          <input type="password" class="form-control" id="register__password" ng-model="vm.password" placeholder="ex. thisisnotgoogleplus" />
        </div>
```

```
        <div class="form-group">
          <button type="submit" class="btn btn-primary">Submit</button>
        </div>
      </form>
    </div>
  </div>
</div>
```

Create a `register.html` template
We won't go into much detail this time because this is pretty basic HTML. A lot of the classes come from Bootstrap, which is included by the boilerplate project. There are only two lines that we are going to pay attention to:

```
<form role="form" ng-submit="vm.register()">
```

This is the line responsible for calling `$scope.register`, which we set up in our controller. `ng-submit` will call `vm.register` when the form is submitted. If you have used Angular before, you are probably used to using `$scope`. In this tutorial, we choose to avoid using `$scope` where possible in favor of `vm` for ViewModel. See the Controllers (https://github.com/johnpapa/angularjs-styleguide#controllers) section of John Papa's AngularJS Style Guide for more on this.

```
<input type="email" class="form-control" id="register__email" ng-model="vm.email" placeholder="ex. john@notgoogle.com" />
```

On each `<input />`, you will see another directive, `ng-model`. `ng-model` is responsible for storing the value of the input on the ViewModel. This is how we get the username, password, and email when `vm.register` is called.

Controlling the interface with RegisterController

With a service and interface in place, we need a controller to hook the two together. The controller we create, `RegisterController` will allow us to call the `register` method of the `Authentication` service when a user submits the form we've just built.

Create a file in `static/javascripts/authentication/controllers/` called `register.controller.js` and add the following:

```
/**
* Register controller
* @namespace thinkster.authentication.controllers
*/
(function () {
  'use strict';

  angular
    .module('thinkster.authentication.controllers')
    .controller('RegisterController', RegisterController);

  RegisterController.$inject = ['$location', '$scope', 'Authentication'];

  /**
  * @namespace RegisterController
  */
  function RegisterController($location, $scope, Authentication) {
    var vm = this;

    vm.register = register;

    /**
    * @name register
    * @desc Register a new user
    * @memberOf thinkster.authentication.controllers.RegisterController
    */
    function register() {
      Authentication.register(vm.email, vm.password, vm.username);
    }
  }
})();
```

Make a controller named `RegisterController` in `static/javascripts/authentication/controllers/register.controller.js`
As usual, we will skip over the familiar and talk about new concepts.

```
.controller('RegisterController', RegisterController);
```

This is similar to the way we registered our service. The difference is that, this time, we are registering a controller.

```
vm.register = register;
```

`vm` allows the template we just created to access the `register` method we define later in the controller.

```
Authentication.register(vm.email, vm.password, vm.username);
```

```
Authentication.register(vm.email, vm.password, vm.username);
```

Here we call the service we created a few minutes ago. We pass in a username, password and email from `vm`.

Registration Routes and Modules

Let's set up some client-side routing so users of the app can navigate to the register form.

Create a file in `static/javascripts` called `thinkster.routes.js` and add the following:

```
(function () {
  'use strict';

  angular
    .module('thinkster.routes')
    .config(config);

  config.$inject = ['$routeProvider'];

  /**
  * @name config
  * @desc Define valid application routes
  */
  function config($routeProvider) {
    $routeProvider.when('/register', {
      controller: 'RegisterController',
      controllerAs: 'vm',
      templateUrl: '/static/templates/authentication/register.html'
    }).otherwise('/');
  }
})();
```

Define a route for the registration form
There are a few points we should touch on here.

```
.config(config);
```

Angular, like just about any framework you can imagine, allows you to edit it's configuration. You do this with a `.config` block.

```
function config($routeProvider) {
```

Here, we are injecting `$routeProvider` as a dependency, which will let us add routing to the client.

```
$routeProvider.when('/register', {
```

`$routeProvider.when` takes two arguments: a path and an options object. Here we use `/register` as the path because thats where we want the registration form to show up.

```
controller: 'RegisterController',
controllerAs: 'vm',
```

One key you can include in the options object is `controller`. This will map a certain controller to this route. Here we use the `RegisterController` controller we made earlier. `controllerAs` is another option. This is required to use the `vm` variable. In short, we are saying that we want to refer to the controller as `vm` in the template.

```
templateUrl: '/static/templates/authentication/register.html'
```

The other key we will use is `templateUrl`. `templateUrl` takes a string of the URL where the template we want to use for this route can be found.

```
}).otherwise('/');
```

We will add more routes as we move forward, but it's possible a user will enter a URL that we don't support. When this happens, `$routeProvider.otherwise` will redirect the user to the path specified; in this case, '/'.

Setting up AngularJS modules

Let us quickly discuss modules in AngularJS.

In Angular, you must define modules prior to using them. So far we need to define `thinkster.authentication.services`, `thinkster.authentication.controllers`, and `thinkster.routes`. Because `thinkster.authentication.services` and `thinkster.authentication.controllers` are submodules of `thinkster.authentication`, we need to create a `thinkster.authentication` module as well.

Create a file in `static/javascripts/authentication/` called `authentication.module.js` and add the following:

```
(function () {
  'use strict';

  angular
    .module('thinkster.authentication', [
      'thinkster.authentication.controllers',
      'thinkster.authentication.services'
    ]);
```

```
  angular
    .module('thinkster.authentication.controllers', []);

  angular
    .module('thinkster.authentication.services', ['ngCookies']);
})();
```

Define the `thinkster.authentication` module and it's dependencies
There are a couple of interesting syntaxes to note here.

```
angular
  .module('thinkster.authentication', [
    'thinkster.authentication.controllers',
    'thinkster.authentication.services'
  ]);
```

This syntax defines the module `thinkster.authentication` with `thinkster.authentication.controllers` and `thinkster.authentication.services` as dependencies.

```
angular
  .module('thinkster.authentication.controllers', []);
```

This syntax defines the module `thinkster.authentication.controllers` with no dependencies.

Now we need define to include `thinkster.authentication` and `thinkster.routes` as dependencies of `thinkster`.

Open `static/javascripts/thinkster.js`, define the required modules, and include them as dependencies of the `thinkster` module. Note that `thinkster.routes` relies on `ngRoute`, which is included with the boilerplate project.

```
(function () {
  'use strict';

  angular
    .module('thinkster', [
      'thinkster.routes',
      'thinkster.authentication'
    ]);

  angular
    .module('thinkster.routes', ['ngRoute']);
})();
```

Update the `thinkster` module to include it's new dependencies

Hash routing

By default, Angular uses a feature called hash routing. If you've ever seen a URL that looks like `www.google.com/#/search` then you know what I'm talking about. Again, this is personal preference, but I think those are incredibly ugly. To get rid of hash routing, we can enabled `$locationProvider.html5Mode`. In older browsers that do not support HTML5 routing, Angular will intelligently fall back to hash routing.

Create a file in `static/javascripts/` called `thinkster.config.js` and give it the following content:

```
(function () {
  'use strict';

  angular
    .module('thinkster.config')
    .config(config);

  config.$inject = ['$locationProvider'];

  /**
  * @name config
  * @desc Enable HTML5 routing
  */
  function config($locationProvider) {
    $locationProvider.html5Mode(true);
    $locationProvider.hashPrefix('!');
  }
})();
```

Enable HTML5 routing for AngularJS
As mentioned, enabling `$locationProvider.html5Mode` gets rid of the hash sign in the URL. The other setting here, `$locationProvider.hashPrefix`, turns the `#` into a `#!`. This is mostly for the benefit of search engines.

Because we are using a new module here, we need to open up `static/javascripts/thinkster.js`, define the module, and include is as a dependency for the `thinkster` module.

```
angular
  .module('thinkster', [
    'thinkster.config',
    // ...
```

```
  ]);

angular
  .module('thinkster.config', []);
```

Define the `thinkster.config` module

Include new .js files

In this chapter so far, we have already created a number of new JavaScript files. We need to include these in the client by adding them to `templates/javascripts.html` inside the `{% compress js %}` block.

Open `templates/javascripts.html` and add the following above the `{% endcompress %}` tag:

```
<script type="text/javascript" src="{% static 'javascripts/thinkster.config.js' %}"></script>
<script type="text/javascript" src="{% static 'javascripts/thinkster.routes.js' %}"></script>
<script type="text/javascript" src="{% static 'javascripts/authentication/authentication.module.js' %}"></script>
<script type="text/javascript" src="{% static 'javascripts/authentication/services/authentication.service.js' %}"></script>
<script type="text/javascript" src="{% static 'javascripts/authentication/controllers/register.controller.js' %}"></script>
```

Add the new JavaScript files to `templates/javascripts.html`

Handling CSRF protection

Because we are using session-based authentication, we have to worry about CSRF protection. We don't go into detail on CSRF here because it's outside the scope of this tutorial, but suffice it to say that CSRF is very bad.

Django, by default, stores a CSRF token in a cookie named `csrftoken` and expects a header with the name `X-CSRFToken` for any dangerous HTTP request (`POST` , `PUT` , `PATCH` , `DELETE`). We can easily configure Angular to handle this.

Open up `static/javascripts/thinkster.js` and add the following under your module definitions:

```
angular
  .module('thinkster')
  .run(run);

run.$inject = ['$http'];

/**
* @name run
* @desc Update xsrf $http headers to align with Django's defaults
*/
function run($http) {
  $http.defaults.xsrfHeaderName = 'X-CSRFToken';
  $http.defaults.xsrfCookieName = 'csrftoken';
}
```

Configure AngularJS CSRF settings

Checkpoint

Try registering a new user by running your server (`python manage.py runserver`), visiting `http://localhost:8000/register` in your browser and filling out the form.

If the registration worked, you can view the new `Account` object created by opening the shell (`python manage.py shell`) and running the following commands:

```
>>> from authentication.models import Account
>>> Account.objects.latest('created_at')
```

The `Account` object returned should match the one you just created.

Register a new user at `http://localhost:8000/register` and confirm the `Account` object was created

Logging users in

Now that users can register, they need a way to log in. As it turns out, this is part of what we are missing from our registration system. Once a user registers, we should automatically log them in.

To get started, we will create views for logging in and logging out. Once those are done we will progress in a fashion similar to the registration systems: services, controllers, etc.

Making the login API view

Open up `authentication/views.py` and add the following:

```
import json

from django.contrib.auth import authenticate, login

from rest_framework improt status, views
from rest_framework.response import Response

class LoginView(views.APIView):
```

```
    def post(self, request, format=None):
        data = json.loads(request.body)

        email = data.get('email', None)
        password = data.get('password', None)

        account = authenticate(email=email, password=password)

        if account is not None:
            if account.is_active:
                login(request, account)

                serialized = AccountSerializer(account)

                return Response(serialized.data)
            else:
                return Response({
                    'status': 'Unauthorized',
                    'message': 'This account has been disabled.'
                }, status=status.HTTP_401_UNAUTHORIZED)
        else:
            return Response({
                'status': 'Unauthorized',
                'message': 'Username/password combination invalid.'
            }, status=status.HTTP_401_UNAUTHORIZED)
```

Make a view called `LoginView` in `authentication/views.py`
This is a longer snippet than we've seen in the past, but we will approach it the same way: by talking about what's new and ignoring what we have already encountered.

```
class LoginView(views.APIView):
```

You will notice that we are not using a generic view this time. Because this view does not perfect a generic activity like creating or updating an object, we must start with something more basic. Django REST Framework's `views.APIView` is what we use. While `APIView` does not handle everything for us, it does give us much more than standard Django views do. In particular, `views.APIView` are made specifically to handle AJAX requests. This turns out to save us a lot of time.

```
def post(self, request, format=None):
```

Unlike generic views, we must handle each HTTP verb ourselves. Logging in should typically be a `POST` request, so we override the `self.post()` method.

```
account = authenticate(email=email, password=password)
```

Django provides a nice set of utilities for authenticating users. The `authenticate()` method is the first utility we will cover. `authenticate()` takes an email and a password. Django then checks the database for an `Account` with email `email`. If one is found, Django will try to verify the given password. If the username and password are correct, the `Account` found by `authenticate()` is returned. If either of these steps fail, `authenticate()` will return `None`.

```
if account is not None:
    # ...
else:
    return Response({
        'status': 'Unauthorized',
        'message': 'Username/password combination invalid.'
    }, status=status.HTTP_401_UNAUTHORIZED)
```

In the event that `authenticate()` returns `None`, we respond with a `401` status code and tell the user that the email/password combination they provided is invalid.

```
if account.is_active:
    # ...
else:
    return Response({
        'status': 'Unauthorized',
        'message': 'This account has been disabled.'
    }, status=status.HTTP_401_UNAUTHORIZED)
```

If the user's account is for some reason inactivate, we respond with a `401` status code. Here we simply say that the account has been disabled.

```
login(request, account)
```

If `authenticate()` success and the user is active, then we use Django's `login()` utility to create a new session for this user.

```
serialized = AccountSerializer(account)

return Response(serialized.data)
```

We want to store some information about this user in the browser if the login request succeeds, so we serialize the `Account` object found by `authenticate()` and return the resulting JSON as the response.

Adding a login API endpoint

Just as we did with `AccountViewSet`, we need to add a route for `LoginView`.

Open up `thinkster_django_angular_boilerplate/urls.py` and add the following URL between `^/api/v1/` and `^`:

```
from authentication.views import LoginView

urlpatterns = patterns(
    # ...
    url(r'^api/v1/auth/login/$', LoginView.as_view(), name='login'),
    # ...
)
```

Add an API endpoint for `LoginView`

Authentication Service

Let's add some more methods to our `Authentication` service. We will do this in two stages. First we will add a `login()` method and then we will add some utility methods for storing session data in the browser.

Open `static/javascripts/authentication/services/authentication.service.js` and add the following method to the `Authentication` object we created earlier:

```
/**
 * @name login
 * @desc Try to log in with email `email` and password `password`
 * @param {string} email The email entered by the user
 * @param {string} password The password entered by the user
 * @returns {Promise}
 * @memberOf thinkster.authentication.services.Authentication
 */
function login(email, password) {
  return $http.post('/api/v1/auth/login/', {
    email: email, password: password
  });
}
```

Make sure to expose it as part of the service:

```
var Authentication = {
  login: login,
  register: register
};
```

Add a `login` method to your `Authentication` service
Much like the `register()` method from before, `login()` returns makes an AJAX request to our API and returns a promise.

Now let's talk about a few utility methods we need for managing session information on the client.

We want to display information about the currently authenticated user in the navigation bar at the top of the page. This means we will need a way to store the response returned by `login()`. We will also need a way to retrieve the authenticated user. We need need a way to unauthenticate the user in the browser. Finally, it would be nice to have an easy way to check if the current user is authenticated.

NOTE: Unauthenticating is different from logging out. When a user logs out, we need a way to remove all remaining session data from the client.

Given these requirements, I suggest three methods: `getAuthenticatedAccount`, `isAuthenticated`, `setAuthenticatedAccount`, and `unauthenticate`.

Let's implement these now. Add each of the following functions to the `Authentication` service:

```
/**
 * @name getAuthenticatedAccount
 * @desc Return the currently authenticated account
 * @returns {object|undefined} Account if authenticated, else `undefined`
 * @memberOf thinkster.authentication.services.Authentication
 */
function getAuthenticatedAccount() {
  if (!$cookies.authenticatedAccount) {
    return;
  }

  return JSON.parse($cookies.authenticatedAccount);
}
```

If there is no `authenticatedAccount` cookie (set in `setAuthenticatedAccount()`), then return; otherwise return the parsed user object from the cookie.

```
/**
 * @name isAuthenticated
 * @desc Check if the current user is authenticated
 * @returns {boolean} True is user is authenticated, else false.
 * @memberOf thinkster.authentication.services.Authentication
 */
function isAuthenticated() {
  return !!$cookies.authenticatedAccount;
}
```

Return the boolean value of the `authenticatedAccount` cookie.

```
/**
 * @name setAuthenticatedAccount
```

```
 * @desc Stringify the account object and store it in a cookie
 * @param {Object} user The account object to be stored
 * @returns {undefined}
 * @memberOf thinkster.authentication.services.Authentication
 */
function setAuthenticatedAccount(account) {
  $cookies.authenticatedAccount = JSON.stringify(account);
}
```

Set the `authenticatedAccount` cookie to a stringified version of the `account` object.

```
/**
 * @name unauthenticate
 * @desc Delete the cookie where the user object is stored
 * @returns {undefined}
 * @memberOf thinkster.authentication.services.Authentication
 */
function unauthenticate() {
  delete $cookies.authenticatedAccount;
}
```

Remove the `authenticatedAccount` cookie.

Again, don't forget to expose these methods as part of the service:

```
var Authentication = {
  getAuthenticatedAccount: getAuthenticatedAccount,
  isAuthenticated: isAuthenticated,
  login: login,
  register: register,
  setAuthenticatedAccount: setAuthenticatedAccount,
  unauthenticate: unauthenticate
};
```

Add `getAuthenticatedAccount`, `isAuthenticated`, `setAuthenticatedAccount`, and `unauthenticate` methods to your `Authentication` service
Before we move on to the login interface, let's quickly update the `login` method of the `Authentication` service to use one of these new utility methods. Replace `Authentication.login` with the following:

```
/**
 * @name login
 * @desc Try to log in with email `email` and password `password`
 * @param {string} email The email entered by the user
 * @param {string} password The password entered by the user
 * @returns {Promise}
 * @memberOf thinkster.authentication.services.Authentication
 */
function login(email, password) {
  return $http.post('/api/v1/auth/login/', {
    email: email, password: password
  }).then(loginSuccessFn, loginErrorFn);

  /**
   * @name loginSuccessFn
   * @desc Set the authenticated account and redirect to index
   */
  function loginSuccessFn(data, status, headers, config) {
    Authentication.setAuthenticatedAccount(data.data);

    window.location = '/';
  }

  /**
   * @name loginErrorFn
   * @desc Log "Epic failure!" to the console
   */
  function loginErrorFn(data, status, headers, config) {
    console.error('Epic failure!');
  }
}
```

Update `Authentication.login` to use our new utility methods

Making a login interface

We now have `Authentication.login()` to log a user in, so let's create the login form. Open up `static/templates/authentication/login.html` and add the following HTML:

```
<div class="row">
  <div class="col-md-4 col-md-offset-4">
    <h1>Login</h1>

    <div class="well">
      <form role="form" ng-submit="vm.login()">
        <div class="alert alert-danger" ng-show="error" ng-bind="error"></div>

        <div class="form-group">
          <label for="login__email">Email</label>
          <input type="text" class="form-control" id="login__email" ng-model="vm.email" placeholder="ex. john@example.com" />
        </div>

        <div class="form-group">
          <label for="login__password">Password</label>
          <input type="password" class="form-control" id="login__password" ng-model="vm.password" placeholder="ex. thisisnotgoogleplus" />
        </div>

        <div class="form-group">
          <button type="submit" class="btn btn-primary">Submit</button>
        </div>
      </form>
    </div>
  </div>
</div>
```

Create a `login.html` template

Controlling the login interface with LoginController

Create a file in `static/javascripts/authentication/controllers/` called `login.controller.js` and add the following contents:

```
/**
* LoginController
* @namespace thinkster.authentication.controllers
*/
(function () {
  'use strict';

  angular
    .module('thinkster.authentication.controllers')
    .controller('LoginController', LoginController);

  LoginController.$inject = ['$location', '$scope', 'Authentication'];

  /**
  * @namespace LoginController
  */
  function LoginController($location, $scope, Authentication) {
    var vm = this;

    vm.login = login;

    activate();

    /**
    * @name activate
    * @desc Actions to be performed when this controller is instantiated
    * @memberOf thinkster.authentication.controllers.LoginController
    */
    function activate() {
      // If the user is authenticated, they should not be here.
      if (Authentication.isAuthenticated()) {
        $location.url('/');
      }
    }

    /**
    * @name login
    * @desc Log the user in
    * @memberOf thinkster.authentication.controllers.LoginController
    */
    function login() {
      Authentication.login(vm.email, vm.password);
    }
  }
```

```
  }
})();
```

Make a controller called `LoginController` in `static/javascripts/authentication/controllers/login.controller.js`
Let's look at the `activate` function.

```
function activate() {
  // If the user is authenticated, they should not be here.
  if (Authentication.isAuthenticated()) {
    $location.url('/');
  }
}
```

You will start to notice that we use a function called `activate` a lot throughout this tutorial. There is nothing inherently special about this name; we chose a standard name for the function that will be run when any given controller is instantiated.

As the comment suggests, if a user is already authenticated, they have no business on the login page. We solve this by redirecting the user to the index page.

We should do this on the registration page too. When we wrote the registration controller, we didn't have `Authentication.isAuthenticated()`. We will update `RegisterController` shortly.

Back to RegisterController

Taking a step back, let's add a check to `RegisterController` and redirect the user if they are already authenticated.

Open `static/javascripts/authentication/controllers/register.controller.js` and add the following just inside the definition of the controller:

```
activate();

/**
 * @name activate
 * @desc Actions to be performed when this controller is instantiated
 * @memberOf thinkster.authentication.controllers.RegisterController
 */
function activate() {
  // If the user is authenticated, they should not be here.
  if (Authentication.isAuthenticated()) {
    $location.url('/');
  }
}
```

Redirect authenticated users to the index view in `RegisterController`
If you remember, we also talked about logging a user in automatically when they register. Since we are already updating registration related content, let's update the `register` method in the `Authentication` service.

Replace `Authentication.register` when the following:

```
/**
* @name register
* @desc Try to register a new user
* @param {string} email The email entered by the user
* @param {string} password The password entered by the user
* @param {string} username The username entered by the user
* @returns {Promise}
* @memberOf thinkster.authentication.services.Authentication
*/
function register(email, password, username) {
  return $http.post('/api/v1/accounts/', {
    username: username,
    password: password,
    email: email
  }).then(registerSuccessFn, registerErrorFn);

  /**
  * @name registerSuccessFn
  * @desc Log the new user in
  */
  function registerSuccessFn(data, status, headers, config) {
    Authentication.login(email, password);
  }

  /**
  * @name registerErrorFn
  * @desc Log "Epic failure!" to the console
  */
  function registerErrorFn(data, status, headers, config) {
    console.error('Epic failure!');
  }
}
```

Update `Authentication.register`

Making a route for the login interface

The next step is to create the client-side route for the login form.

Open up `static/javascripts/thinkster.routes.js` and add a route for the login form:

```
$routeProvider.when('/register', {
  controller: 'RegisterController',
  controllerAs: 'vm',
  templateUrl: '/static/templates/authentication/register.html'
}).when('/login', {
  controller: 'LoginController',
  controllerAs: 'vm',
  templateUrl: '/static/templates/authentication/login.html'
}).otherwise('/');
```

Add a route for `LoginController`

See how you can chain calls to `$routeProvider.when()`? Going forward, we will ignore old routes for brevity. Just keep in mind that these calls should be chained and that the first route matched will take control.

Include new .js files

If you can believe it, we've only created one new JavaScript file since the last time: `login.controller.js`. Let's add it to `javascripts.html` with the other JavaScript files:

```
<script type="text/javascript" src="{% static 'javascripts/authentication/controllers/login.controller.js' %}"></script>
```

Include `login.controller.js` in `javascripts.html`

Checkpoint

Open `http://localhost:8000/login` in your browser and log in with the user you created earlier. If this works, the page should redirect to `http://localhost:8000/` and the navigation bar should change.

Log in with one of the users you created earlier by visiting `http://localhost:8000/login`

Logging users out

Given that users can register and login, we can assume they will want a way to log out. People get mad when they can't log out.

Making a logout API view

Let's implement the last authentication-related API view.

Open up `authentication/views.py` and add the following imports and class:

```
from django.contrib.auth import logout

from rest_framework import permissions

class LogoutView(views.APIView):
    permission_classes = (permissions.IsAuthenticated,)

    def post(self, request, format=None):
        logout(request)

        return Response({}, status=status.HTTP_204_NO_CONTENT)
```

Make a view called `LogoutView` in `authentication/views.py`
There are only a few new things to talk about this time.

```
permission_classes = (permissions.IsAuthenticated,)
```

Only authenticated users should be able to hit this endpoint. Django REST Framework's `permissions.IsAuthenticated` handles this for us. If you user is not authenticated, they will get a `403` error.

```
logout(request)
```

If the user is authenticated, all we need to do is call Django's `logout()` method.

```
return Response({}, status=status.HTTP_204_NO_CONTENT)
```

There isn't anything reasonable to return when logging out, so we just return an empty response with a `200` status code.

Moving on to the URLs.

Open up `thinkster_django_angular_boilerplate/urls.py` again and add the following import and URL:

```
from authentication.views import LogoutView

urlpatterns = patterns(
    # ...
    url(r'^api/v1/auth/logout/$', LogoutView.as_view(), name='logout'),
    #...
)
```

Create an API endpoint for `LogoutView`

Logout: AngularJS Service

The final method you need to add to your `Authentication` service is the `logout()` method.

Add the following method to the `Authentication` service in `authentication.service.js`:

```
/**
 * @name logout
 * @desc Try to log the user out
 * @returns {Promise}
 * @memberOf thinkster.authentication.services.Authentication
 */
function logout() {
  return $http.post('/api/v1/auth/logout/')
    .then(logoutSuccessFn, logoutErrorFn);

  /**
   * @name logoutSuccessFn
   * @desc Unauthenticate and redirect to index with page reload
   */
  function logoutSuccessFn(data, status, headers, config) {
    Authentication.unauthenticate();

    window.location = '/';
  }

  /**
   * @name logoutErrorFn
   * @desc Log "Epic failure!" to the console
   */
  function logoutErrorFn(data, status, headers, config) {
    console.error('Epic failure!');
  }
}
```

As always, remember to expose `logout` as part of the `Authentication` service:

```
var Authentication = {
  getAuthenticatedUser: getAuthenticatedUser,
  isAuthenticated: isAuthenticated,
  login: login,
  logout: logout,
  register: register,
  setAuthenticatedUser: setAuthenticatedUser,
  unauthenticate: unauthenticate
};
```

Add a `logout()` method to your `Authentication` service

Controlling the navigation bar with NavbarController

There will not actually be a `LogoutController` or `logout.html`. Instead, the navigation bar already contains a logout link for authenticated users. We will create a `NavbarController` for handling the logout buttons `onclick` functionality and we will update the link itself with an `ng-click` attribute.

Create a file in `static/javascripts/layout/controllers/` called `navbar.controller.js` and add the following to it:

```
/**
* NavbarController
* @namespace thinkster.layout.controllers
```

```
 */
(function () {
  'use strict';

  angular
    .module('thinkster.layout.controllers')
    .controller('NavbarController', NavbarController);

  NavbarController.$inject = ['$scope', 'Authentication'];

  /**
  * @namespace NavbarController
  */
  function NavbarController($scope, Authentication) {
    var vm = this;

    vm.logout = logout;

    /**
    * @name logout
    * @desc Log the user out
    * @memberOf thinkster.layout.controllers.NavbarController
    */
    function logout() {
      Authentication.logout();
    }
  }
})();
```

Create a `NavbarController` in `static/javascripts/layout/controllers/navbar.controller.js`
Open `templates/navbar.html` and add an `ng-controller` directive with the value `NavbarController as vm` to the `<nav />` tag like so:

```
<nav class="navbar navbar-default" role="navigation" ng-controller="NavbarController as vm">
```

While you have `templates/navbar.html` open, go ahead and find the logout link and add `ng-click="vm.logout()"` to it like so:

```
<li><a href="javascript:void(0)" ng-click="vm.logout()">Logout</a></li>
```

Update `navbar.html` to include the `ng-controller` and `ng-click` directives where appropriate

Layout modules

We need to add a few new modules this time around.

Create a file in `static/javascripts/layout/` called `layout.module.js` and give it the following contents:

```
(function () {
  'use strict';

  angular
    .module('thinkster.layout', [
      'thinkster.layout.controllers'
    ]);

  angular
    .module('thinkster.layout.controllers', []);
})();
```

And don't forget to update `static/javascripts/thinkster.js` also:

```
angular
  .module('thinkster', [
    'thinkster.config',
    'thinkster.routes',
    'thinkster.authentication',
    'thinkster.layout'
  ]);
```

Define new `thinkster.layout` and `thinkster.layout.controllers` modules

Including new .js files

This time around there are a couple new JavaScript files to include. Open up `javascripts.html` and add the following:

```
<script type="text/javascript" src="{% static 'javascripts/layout/layout.module.js' %}"></script>
<script type="text/javascript" src="{% static 'javascripts/layout/controllers/navbar.controller.js' %}"></script>
```

Include new JavaScript files in `javascripts.html`

Checkpoint

If you visit `http://localhost:8000/` in your browser, you should still be logged in. If not, you will need to log in again.

You can confirm the logout functionality is working by clicking the logout button in the navigation bar. This should refresh the page and update the navigation bar to it's logged out view.

Log out of your account by using the logout button in the navigation bar

Making a Post model

In this section we will make a new app and create a `Post` model similar to a status on Facebook or a tweet on Twitter. After we create our model we will move on to serializing `Post` s and then we will create a few new endpoints for our API.

Making a posts app

First things first: go ahead and create a new app called `posts` .

```
$ python manage.py startapp posts
```

Make a new app named `posts`
Remember: whenever you create a new app you have to add it to the `INSTALLED_APPS` setting. Open `thinkster_django_angular_boilerplate/settings.py` and modify it like so:

```
INSTALLED_APPS = (
    # ...
    'posts',
)
```

Making the Post model

After you create the `posts` app Django made a new file called `posts/models.py` . Go ahead and open it up and add the following:

from django.db import models

from authentication.models import Account

```
class Post(models.Model):
    author = models.ForeignKey(Account)
    content = models.TextField()

    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def __unicode__(self):
        return '{0}'.format(self.content)
```

Make a new model called `Post` in `posts/models.py`
Our method of walking through the code line-by-line is working well so far. Why mess with a good thing? Let's do it.

```
author = models.ForeignKey(Account)
```

Because each `Account` can have many `Post` objects, we need to set up a many-to-one relation.

The way to do this in Django is with using a `ForeignKey` field to associate each `Post` with a `Account` .

Django is smart enough to know the foreign key we've set up here should be reversible. That is to say, given a `Account` , you should be able to access that user's `Post` s. In Django these `Post` objects can be accessed through `Account.post_set` (not `Account.posts`).

Now that the model exists, don't forget to migrate.

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

Make migrations for `Post` and apply them

Serializing the Post model

Create a new file in `posts/` called `serializers.py` and add the following:

```
from rest_framework import serializers

from authentication.serializers import Account
from posts.models import Post


class PostSerializer(serializers.ModelSerializer):
    author = AccountSerializer(read_only=True, required=False)

    class Meta:
        model = Post
```

```
        fields = ('id', 'author', 'content', 'created_at', 'updated_at')
        read_only_fields = ('id', 'created_at', 'updated_at')

    def get_validation_exclusions(self, *args, **kwargs):
        exclusions = super(PostSerializer, self).get_validation_exclusions()

        return exclusions + ['author']
```

Make a new serializer called `PostSerializer` in `posts/serializers.py`
There isn't much here that's new, but there is one line in particular I want to look at.

```
author = AccountSerializer(read_only=True, required=False)
```

We explicitly defined a number of fields in our `AccountSerializer` from before, but this definition is a little different.

When serializing a `Post` object, we want to include all of the author's information. Within Django REST Framework, this is known as a nested relationship. Basically, we are serializing the `Account` related to this `Post` and including it in our JSON.

We pass `read_only=True` because we should not be updating an `Account` object with a `PostSerializer`. We also set `required=False` here because we will set the author of this post automatically.

```
def get_validation_exclusions(self, *args, **kwargs):
    exclusions = super(PostSerializer, self).get_validation_exclusions()

    return exclusions + ['author']
```

For the same reason we use `required=False`, we must also add `author` to the list of validations we wish to skip.

Making API views for Post objects

The next step in creating `Post` objects is adding an API endpoint that will handle performing actions on the `Post` model such as create or update.

Replace the contents of `posts/views.py` with the following:

```
from rest_framework import permissions, viewsets
from rest_framework.response import Response

from posts.models import Post
from posts.permissions import IsAuthorOfPost
from posts.serializers import PostSerializer


class PostViewSet(viewsets.ModelViewSet):
    queryset = Post.objects.order_by('-created_at')
    serializer_class = PostSerializer

    def get_permissions(self):
        if self.request.method in permissions.SAFE_METHODS:
            return (permissions.AllowAny(),)
        return (permissions.IsAuthenticated(), IsAuthorOfPost(),)

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

    return super(PostViewSet, self).perform_create(serializer)



class AccountPostsViewSet(viewsets.ViewSet):
    queryset = Post.objects.select_related('author').all()
    serializer_class = PostSerializer

    def list(self, request, account_username=None):
        queryset = self.queryset.filter(author__username=account_username)
        serializer = self.serializer_class(queryset, many=True)

        return Response(serializer.data)
```

Make a `PostViewSet` viewset Make an `AccountPostsViewSet` viewset
Do these views look similar? They aren't that different than the ones we made to create `User` objects.

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

    return super(PostViewSet, self).perform_create(serializer)
```

`perform_create` is called before the model of this view is saved.

When a `Post` object is created it has to be associated with an author. Making the author type in their own username or id when creating adding a post to the site would be a bad experience, so we handle this association for them with the `perform_create` hook. We simply grab the user associated with this request and make them the author of this `Post`.

```
def get_permissions(self):
    if self.request.method in permissions.SAFE_METHODS:
        return (permissions.AllowAny(),)
    return (permissions.IsAuthenticated(), IsAuthorOfPost(),)
```

Similar to the permissions we used for the `Account` viewset, dangerous HTTP methods require the user be authenticated and authorized to make changes to this `Post`. We will created the `IsAuthorOfPost` permission shortly. If the HTTP method is safe, we allow anyone to access this view.

```
class AccountPostsViewSet(viewsets.ViewSet):
```

This viewset will be used to list the posts associated with a specific `Account`.

```
queryset = self.queryset.filter(author__username=account_username)
```

Here we filter our queryset based on the author's username. The `account_username` argument will be supplied by the router we will create in a few minutes.

Making the IsAuthorOfPost permission

Create `permissions.py` in the `posts/` directory with the following content:

```
from rest_framework import permissions


class IsAuthorOfPost(permissions.BasePermission):
    def has_object_permission(self, request, view, post):
        if request.user:
            return post.author == request.user
        return False
```

Make a new permission called `IsAuthenticatedAndOwnsObject` in `posts/permissions.py`
We will skip the explanation for this. This permission is almost identical to the one we made previously.

Making an API endpoint for posts

With the views created, it's time to add the endpoints to our API.

Open `thinkster_django_angular_boilerplate/urls.py` and add the following import:

```
from posts.views import AccountPostsViewSet, PostViewSet
```

Now add these lines just above `urlpatterns = patterns(`:

```
router.register(r'posts', PostViewSet)

accounts_router = routers.NestedSimpleRouter(
    router, r'accounts', lookup='account'
)
accounts_router.register(r'posts', AccountPostsViewSet)
```

`accounts_router` provides the nested routing need to access the posts for a specific `Account`. You should also now add `accounts_router` to `urlpatterns` like so:

```
urlpatterns = patterns(
  # ...

  url(r'^api/v1/', include(router.urls)),
  url(r'^api/v1/', include(accounts_router.urls)),

  # ...
)
```

Make an API endpoint for the `PostViewSet` viewset Make an API endpoint for the `AccountPostsViewSet` viewset

Checkpoint

At this point, feel free to open up your shell with `python manage.py shell` and play around with creating and serializing `Post` objects.

```
>>> from authentication.models import Account
>>> from posts.models import Post
>>> from posts.serializers import PostSerializer
>>> account = Account.objects.latest('created_at')
>>> post = Post.objects.create(author=account, content='I promise this is not Google Plus!')
>>> serialized_post = PostSerializer(post)
>>> serialized_post.data
```

Play around with the `Post` model and `PostSerializer` serializer in Django's shell
We will confirm the views are working at the end of the next section.

Rendering Post objects

Until now, the index page has been empty. Now that we have handled authentication and the backend details for the `Post` model, it's time to give our users something to interact with. We will do this by creating a service that handles retrieving and creating `Post`s and some controllers and directives for handling how the data is displayed.

A module for posts

A module for posts

Let's define the posts modules.

Create a file in `static/javascripts/posts` called `posts.module.js` and add the following:

```
(function () {
  'use strict';

  angular
    .module('thinkster.posts', [
      'thinkster.posts.controllers',
      'thinkster.posts.directives',
      'thinkster.posts.services'
    ]);

  angular
    .module('thinkster.posts.controllers', []);

  angular
    .module('thinkster.posts.directives', ['ngDialog']);

  angular
    .module('thinkster.posts.services', []);
})();
```

Define the `thinkster.posts` module
Remember to add `thinkster.posts` as a dependency of `thinkster` in `thinkster.js`:

```
angular
  .module('thinkster', [
    'thinkster.config',
    'thinkster.routes',
    'thinkster.authentication',
    'thinkster.layout',
    'thinkster.posts'
  ]);
```

Add `thinkster.posts` as a dependency of the `thinkster` module
There are two things worth noting about this module.

First, we have created a module named `thinkster.posts.directives`. As you probably guessed, this means we will introduce the concept of directives to our app in this chapter.

Secondly, the `thinkster.posts.directives` module requires the `ngDialog` module. `ngDialog` is included in the boilerplate project and handles the display of modals. We will use a modal in the next chapter when we write the code for creating new posts.

Include this file in `javascripts.html`:

```
<script type="text/javascript" src="{% static 'javascripts/posts/posts.module.js' %}"></script>
```

Include `posts.module.js` in `javascripts.html`

Making a Posts service

Before we can render anything, we need to transport data from the server to the client.

Create a file at `static/javascripts/posts/services/` called `posts.service.js` and add the following:

```
/**
* Posts
* @namespace thinkster.posts.services
*/
(function () {
  'use strict';

  angular
    .module('thinkster.posts.services')
    .factory('Posts', Posts);

  Posts.$inject = ['$http'];

  /**
  * @namespace Posts
  * @returns {Factory}
  */
  function Posts($http) {
    var Posts = {
      all: all,
      create: create,
      get: get
    };

    return Posts;
```

```
    return Posts;

    ////////////////////

    /**
    * @name all
    * @desc Get all Posts
    * @returns {Promise}
    * @memberOf thinkster.posts.services.Posts
    */
    function all() {
      return $http.get('/api/v1/posts/');
    }


    /**
    * @name create
    * @desc Create a new Post
    * @param {string} content The content of the new Post
    * @returns {Promise}
    * @memberOf thinkster.posts.services.Posts
    */
    function create(content) {
      return $http.post('/api/v1/posts/', {
        content: content
      });
    }

    /**
     * @name get
     * @desc Get the Posts of a given user
     * @param {string} username The username to get Posts for
     * @returns {Promise}
     * @memberOf thinkster.posts.services.Posts
     */
    function get(username) {
      return $http.get('/api/v1/accounts/' + username + '/posts/');
    }
  }
})();
```

Make a new factory called `Posts` in `static/javascripts/posts/services/posts.service.js`
Include this file in `javascripts.html`:

```
<script type="text/javascript" src="{% static 'javascripts/posts/services/posts.service.js' %}"></script>
```

Include `posts.service.js` in `javascripts.html`
This code should look pretty familiar. It is very similar to the services we created before.

The `Posts` service only has two methods: `all` and `create`.

On the index page, we will use `Posts.all()` to get the list of objects we want to display. We will use `Posts.create()` to let users add their own posts.

Making an interface for the index page

Create `static/templates/layout/index.html` with the following contents:

```
<posts posts="vm.posts" ng-show="vm.posts && vm.posts.length"></posts>
```

Create the index template
We will add a little more later, but not much. Most of what we need will be in the template we create for the posts directive next.

Making a Snackbar service

In the boilerplate project for this tutorial, we've included SnackbarJS. SnackbarJS is a small JavaScript library that makes showing snackbars (a concept from Google's Material Design) easy. Here, we will create a service to include this functionality in our AngularJS application.

Open `static/javascripts/utils/services/snackbar.service.js` and add the following:

```
/**
* Snackbar
* @namespace thinkster.utils.services
*/
(function ($, _) {
  'use strict';

  angular
    .module('thinkster.utils.services')
    .factory('Snackbar', Snackbar);

  /**
  * @namespace Snackbar
  */
  function Snackbar() {
    /**
    * @name Snackbar
    * @desc The factory to be returned
    */
```

```
    var Snackbar = {
      error: error,
      show: show
    };

    return Snackbar;

    ////////////////////

    /**
    * @name _snackbar
    * @desc Display a snackbar
    * @param {string} content The content of the snackbar
    * @param {Object} options Options for displaying the snackbar
    */
    function _snackbar(content, options) {
      options = _.extend({ timeout: 3000 }, options);
      options.content = content;

      $.snackbar(options);
    }


    /**
    * @name error
    * @desc Display an error snackbar
    * @param {string} content The content of the snackbar
    * @param {Object} options Options for displaying the snackbar
    * @memberOf thinkster.utils.services.Snackbar
    */
    function error(content, options) {
      _snackbar('Error: ' + content, options);
    }


    /**
    * @name show
    * @desc Display a standard snackbar
    * @param {string} content The content of the snackbar
    * @param {Object} options Options for displaying the snackbar
    * @memberOf thinkster.utils.services.Snackbar
    */
    function show(content, options) {
      _snackbar(content, options);
    }
  }
})($, _);
```

Make a `Snackbar` service
Don't forget to set up your modules. Open `static/javascripts/utils/utils.module.js` and add the following:

```
(function () {
  'use strict';

  angular
    .module('thinkster.utils', [
      'thinkster.utils.services'
    ]);

  angular
    .module('thinkster.utils.services', []);
})();
```

And make `thinksters.utils` a dependency of `thinkster` in `static/javascripts/thinkster.js` :

```
angular
  .module('thinkster', [
    // ...
    'thinkster.utils',
    // ...
  ]);
```

Set up your modules for the `thinkster.utils` module Make `thinkster.utils` a dependency of `thinkster`
The last step for this service is to include the new JavaScript files in `javascripts.html` :

```
<script type="text/javascript" src="{% static 'javascripts/utils/utils.module.js' %}"></script>
<script type="text/javascript" src="{% static 'javascripts/utils/services/snackbar.service.js' %}"></script>
```

Include `utils.module.js` and `snackbar.service.js` in `javascripts.html`

Controlling the index interface with IndexController

Defining the index interface with IndexController

Create a file in `static/javascripts/layout/controllers/` called `index.controller.js` and add the following:

```
/**
* IndexController
* @namespace thinkster.layout.controllers
*/
(function () {
  'use strict';

  angular
    .module('thinkster.layout.controllers')
    .controller('IndexController', IndexController);

  IndexController.$inject = ['$scope', 'Authentication', 'Posts', 'Snackbar'];

  /**
  * @namespace IndexController
  */
  function IndexController($scope, Authentication, Posts, Snackbar) {
    var vm = this;

    vm.isAuthenticated = Authentication.isAuthenticated();
    vm.posts = [];

    activate();

    /**
    * @name activate
    * @desc Actions to be performed when this controller is instantiated
    * @memberOf thinkster.layout.controllers.IndexController
    */
    function activate() {
      Posts.all().then(postsSuccessFn, postsErrorFn);

      $scope.$on('post.created', function (event, post) {
        vm.posts.unshift(post);
      });

      $scope.$on('post.created.error', function () {
        vm.posts.shift();
```

```
      });


      /**
      * @name postsSuccessFn
      * @desc Update posts array on view
      */
      function postsSuccessFn(data, status, headers, config) {
        vm.posts = data.data;
      }


      /**
      * @name postsErrorFn
      * @desc Show snackbar with error
      */
      function postsErrorFn(data, status, headers, config) {
        Snackbar.error(data.error);
      }
    }
  }
})();
```

Make a new controller called `IndexController` in `static/javascripts/layout/controllers/index.controller.js`
Include this file in `javascripts.html`:

```
<script type="text/javascript" src="{% static 'javascripts/layout/controllers/index.controller.js' %}"></script>
```

Include `index.controller.js` in `javascripts.html`
Let's touch on a couple of things here.

```
$scope.$on('post.created', function (event, post) {
  vm.posts.unshift(post);
});
```

Later, when we get around to creating a new post, we will fire off an event called `post.created` when the user creates a post. By catching this event here, we can add this new post to the front of the `vm.posts` array. This will prevent us from having to make an extra API request to the server for updated data. We will talk about this more shortly, but for now you should know that we do this to increase the *perceived* performance of our application.

```
$scope.$on('post.created.error', function () {
  vm.posts.shift();
});
```

Analogous to the previous event listener, this one will remove the post at the front of `vm.posts` if the API request returns an error status code.

Making a route for the index page

With a controller and template in place, we need to set up a route for the index page.

Open `static/javascripts/thinkster.routes.js` and add the following route:

```
.when('/', {
  controller: 'IndexController',
  controllerAs: 'vm',
  templateUrl: '/static/templates/layout/index.html'
})
```

Add a route to `thinkster.routes.js` for the `/` path

Making a directive for displaying Posts

Create `static/javascripts/posts/directives/posts.directive.js` with the following contents:

```
/**
* Posts
* @namespace thinkster.posts.directives
*/
(function () {
  'use strict';

  angular
    .module('thinkster.posts.directives')
    .directive('posts', posts);

  /**
  * @namespace Posts
  */
  function posts() {
    /**
    * @name directive
    * @desc The directive to be returned
    * @memberOf thinkster.posts.directives.Posts
    */
    var directive = {
      controller: 'PostsController',
      controllerAs: 'vm'
```

```
      controllerAs: 'vm',
      restrict: 'E',
      scope: {
        posts: '='
      },
      templateUrl: '/static/templates/posts/posts.html'
    };

    return directive;
  }
})();
```

Make a new directive called `posts` in `static/javascripts/posts/directives/posts.directive.js`
Include this file in `javascripts.html`:

```
<script type="text/javascript" src="{% static 'javascripts/posts/directives/posts.directive.js' %}"></script>
```

Include `posts.directive.js` in `javascripts.html`
There are two parts of the directives API that I want to touch on: `scope` and `restrict`.

```
scope: {
  posts: '='
},
```

`scope` defines the scope of this directive, similar to how `$scope` works for controllers. The difference is that, in a controller, a new scope is implicitly created. For a directive, we have the option of explicitly defining our scopes and that's what we do here.

The second line, `posts: '='` simply means that we want to set `$scope.posts` to the value passed in through the `posts` attribute in the template that we made earlier.

```
restrict: 'E',
```

`restrict` tells Angular how we are allowed to use this directive. In our case, we set the value of `restrict` to `E` (for element) which means Angular should only match the name of our directive with the name of an element: `<posts></posts>`.

Another common option is `A` (for attribute), which tells Angular to only match the name of the directive with the name of an attribute. `ngDialog` uses this option, as we will see shortly.

Controller the posts directive with PostsController

The directive we just created requires a controller called `PostsController`.

Create `static/javascripts/posts/controllers/posts.controller.js` with the following content:

```
/**
* PostsController
* @namespace thinkster.posts.controllers
*/
(function () {
  'use strict';

  angular
    .module('thinkster.posts.controllers')
    .controller('PostsController', PostsController);

  PostsController.$inject = ['$scope'];

  /**
  * @namespace PostsController
  */
  function PostsController($scope) {
    var vm = this;

    vm.columns = [];

    activate();


    /**
    * @name activate
    * @desc Actions to be performed when this controller is instantiated
    * @memberOf thinkster.posts.controllers.PostsController
    */
    function activate() {
      $scope.$watchCollection(function () { return $scope.posts; }, render);
      $scope.$watch(function () { return $(window).width(); }, render);
    }


    /**
    * @name calculateNumberOfColumns
    * @desc Calculate number of columns based on screen width
    * @returns {Number} The number of columns containing Posts
    * @memberOf thinkster.posts.controllers.PostsControllers
    */
    function calculateNumberOfColumns() {
      var width = $(window).width();
```

```
      if (width >= 1200) {
        return 4;
      } else if (width >= 992) {
        return 3;
      } else if (width >= 768) {
        return 2;
      } else {
        return 1;
      }
    }


    /**
    * @name approximateShortestColumn
    * @desc An algorithm for approximating which column is shortest
    * @returns The index of the shortest column
    * @memberOf thinkster.posts.controllers.PostsController
    */
    function approximateShortestColumn() {
      var scores = vm.columns.map(columnMapFn);

      return scores.indexOf(Math.min.apply(this, scores));


      /**
      * @name columnMapFn
      * @desc A map function for scoring column heights
      * @returns The approximately normalized height of a given column
      */
      function columnMapFn(column) {
        var lengths = column.map(function (element) {
          return element.content.length;
        });

        return lengths.reduce(sum, 0) * column.length;
      }



      /**
      * @name sum
      * @desc Sums two numbers
      * @params {Number} m The first number to be summed
      * @params {Number} n The second number to be summed
      * @returns The sum of two numbers
      */
      function sum(m, n) {
        return m + n;
      }
    }


    /**
    * @name render
    * @desc Renders Posts into columns of approximately equal height
    * @param {Array} current The current value of `vm.posts`
    * @param {Array} original The value of `vm.posts` before it was updated
    * @memberOf thinkster.posts.controllers.PostsController
    */
    function render(current, original) {
      if (current !== original) {
        vm.columns = [];

        for (var i = 0; i < calculateNumberOfColumns(); ++i) {
          vm.columns.push([]);
        }

        for (var i = 0; i < current.length; ++i) {
          var column = approximateShortestColumn();

          vm.columns[column].push(current[i]);
        }
      }
    }
  }
})();
```

Make a new controller called `PostsController` in `static/javascripts/posts/controllers/posts.controller.js`
Include this file in `javascripts.html`:

```
<script type="text/javascript" src="{% static 'javascripts/posts/controllers/posts.controller.js' %}"></script>
```

Include `posts.controller.js` in `javascripts.html`
It isn't worth taking the time to step through this controller line-by-line. Suffice it to say that this controller presents an algorithm for ensuring the columns of posts are of approximately equal height.

The only thing worth mentioning here is this line:

```
$scope.$watchCollection(function () { return $scope.posts; }, render);
```

```
$scope.$watchCollection(function () { return $scope.posts; }, render);
```

Because we do not have direct access to the ViewModel that `posts` is stored on, we watch `$scope.posts` instead of `vm.posts`. Furthermore, we use `$watchCollection` here because `$scope.posts` is an array. `$watch` watches the object's reference, not it's actual value. `$watchCollection` watches the value of an array from changes. If we used `$watch` here instead of `$watchCollection`, the changes caused by `$scope.posts.shift()` and `$scope.posts.unshift()` would not trigger the watcher.

Making a template for the posts directive

In our directive we defined a `templateUrl` that doesn't match any of our existing templates. Let's go ahead and make a new one.

Create `static/templates/posts/posts.html` with the following content:

```
<div class="row" ng-cloak>
  <div ng-repeat="column in vm.columns">
    <div class="col-xs-12 col-sm-6 col-md-4 col-lg-3">
      <div ng-repeat="post in column">
        <post post="post"></post>
      </div>
    </div>
  </div>

  <div ng-hide="vm.columns && vm.columns.length">
    <div class="col-sm-12 no-posts-here">
      <em>The are no posts here.</em>
    </div>
  </div>
</div>
```

Create a template for the `posts` directive
A few things worth noting:

1. We use the `ng-cloak` directive to prevent flashing since this directive will be used on the first page loaded.
2. We will need to create a `post` directive for rendering each individual post.
3. If no posts are present, we render a message informing the user.

Making a directive for displaying a single Post

In the template for the posts directive, we use another directive called `post`. Let's create that.

Create `static/javascripts/posts/directives/post.directive.js` with the following content:

```
/**
* Post
* @namespace thinkster.posts.directives
*/
(function () {
  'use strict';

  angular
    .module('thinkster.posts.directives')
    .directive('post', post);

  /**
  * @namespace Post
  */
  function post() {
    /**
    * @name directive
    * @desc The directive to be returned
    * @memberOf thinkster.posts.directives.Post
    */
    var directive = {
      restrict: 'E',
      scope: {
        post: '='
      },
      templateUrl: '/static/templates/posts/post.html'
    };

    return directive;
  }
})();
```

Make a new directive called `post` in `static/javascripts/posts/directives/post.directive.js`
Include this file in `javascripts.html`:

```
<script type="text/javascript" src="{% static 'javascripts/posts/directives/post.directive.js' %}"></script>
```

Include `post.directive.js` in `javascripts.html`
There is nothing new worth discussing here. This directive is almost identical to the previous one. The only difference is we use a different template.

Making a template for the post directive

Like we did for the `posts` directive, we now need to make a template for the `post` directive.

Create `static/templates/posts/post.html` with the following content:

```
<div class="row">
  <div class="col-sm-12">
    <div class="well">
      <div class="post">
        <div class="post__meta">
          <a href="/+{{ post.author.username }}">
            +{{ post.author.username }}
          </a>
        </div>

        <div class="post__content">
          {{ post.content }}
        </div>
      </div>
    </div>
  </div>
</div>
```

Create a template for the `post` directive

Some quick CSS

We want to add a few simple styles to make our posts look better. Open `static/stylesheets/styles.css` and add the following:

```
.no-posts-here {
  text-align: center;
}

.post {}

.post .post__meta {
  font-weight: bold;
  text-align: right;
  padding-bottom: 19px;
}

.post .post__meta a:hover {
  text-decoration: none;
}
```

Add some CSS to `static/stylesheets/style.css` to make our posts look better

Checkpoint

Assuming all is well, you can confirm you're on the right track by loading `http://localhost:8000/` in your browser. You should see the `Post` object you created at the end of the last section!

This also confirms that `PostViewSet` from the last section is working.

Visit `http://localhost:8000/` and confirm the `Post` object you made earlier is shown.

Making new posts

Given that we already have the necessary endpoints in place, the next thing we need to let users make new posts is an interface. We accomplish this by adding a button to the bottom-right corner of the screen. When this button is clicked, a modal shows up asking the user to type in their post.

We only want this button to show up on the index page for now, so open `static/templates/layout/index.html` and add the following snippet to the bottom of the file:

```
<a class="btn btn-primary btn-fab btn-raised mdi-content-add btn-add-new-post"
  href="javascript:void(0)"
  ng-show="vm.isAuthenticated"
  ng-dialog="/static/templates/posts/new-post.html"
  ng-dialog-controller="NewPostController as vm"></a>
```

The anchor tag in this snippet uses the `ngDialog` directive we included as a dependency earlier to show a modal when the user wants to submit a new post.

Because we want the button to be fixed to the bottom-right corner of the screen, we also need to add a new CSS rule.

Open `static/stylesheets/styles.css` and add this rule to the bottom of the file:

```
.btn-add-new-post {
  position: fixed;
  bottom: 20px;
  right: 20px;
}
```

Add a button for showing the new post modal Style the new post button

An interface for submitting new posts

An interface for submitting new posts

Now we need to create the form the user will type their new post into. Open `static/templates/posts/new-post.html` and add the following to the bottom of the file:

```
<form role="form" ng-submit="vm.submit()">
  <div class="form-group">
    <label for="post__content">New Post</label>
    <textarea class="form-control"
              id="post__content"
              rows="3"
              placeholder="ex. This is my first time posting on Not Google Plus!"
              ng-model="vm.content">
    </textarea>
  </div>

  <div class="form-group">
    <button type="submit" class="btn btn-primary">
      Submit
    </button>
  </div>
</form>
```

Make a template for adding new posts in `static/templates/posts/new-post.html`

Controlling the new post interface with NewPostController

Create `static/javascripts/posts/controller/new-post.controller.js` with the following content:

```
/**
* NewPostController
* @namespace thinkster.posts.controllers
*/
(function () {
  'use strict';

  angular
    .module('thinkster.posts.controllers')
    .controller('NewPostController', NewPostController);

  NewPostController.$inject = ['$rootScope', '$scope', 'Authentication', 'Snackbar', 'Posts'];

  /**
  * @namespace NewPostController
  */
  function NewPostController($rootScope, $scope, Authentication, Snackbar, Posts) {
    var vm = this;

    vm.submit = submit;

    /**
    * @name submit
    * @desc Create a new Post
    * @memberOf thinkster.posts.controllers.NewPostController
    */
    function submit() {
      $rootScope.$broadcast('post.created', {
        content: vm.content,
        author: {
          username: Authentication.getAuthenticatedAccount().username
        }
      });

      $scope.closeThisDialog();

      Posts.create(vm.content).then(createPostSuccessFn, createPostErrorFn);


      /**
      * @name createPostSuccessFn
      * @desc Show snackbar with success message
      */
      function createPostSuccessFn(data, status, headers, config) {
        Snackbar.show('Success! Post created.');
      }


      /**
      * @name createPostErrorFn
      * @desc Propogate error event and show snackbar with error message
      */
      function createPostErrorFn(data, status, headers, config) {
        $rootScope.$broadcast('post.created.error');
        Snackbar.error(data.error);
      }
    }
  }
})();
```

Make a `NewPostController` in `static/javascripts/posts/controllers/new-post.controller.js`
There are a few things going on here that we should talk about.

```
$rootScope.$broadcast('post.created', {
  content: $scope.content,
  author: {
    username: Authentication.getAuthenticatedAccount().username
  }
});
```

Earlier we set up an event listener in `IndexController` that listened for the `post.created` event and then pushed the new post onto the front of `vm.posts`. Let's look at this a little more closely, as this turns out to be an important feature of rich web applications.

What we are doing here is being *optimistic* that the API response from `Posts.create()` will contain a 200 status code telling us everything went according to plan. This may seem like a bad idea at first. Something could go wrong during the request and then our data is stale. Why don't we just wait for the response?

When I said we are increasing the *perceived* performance of our app, this is what I was talking about. We want the user to *perceive* the response as instant.

The fact of the matter is that this call will rarely fail. There are only two cases where this will reasonably fail: either the user is not authenticated or the server is down.

In the case where the user is not authenticated, they shouldn't be submitting new posts anyways. Consider the error to be a small punishment for the user doing things they shouldn't.

If the server is down, then there is nothing we can do. Unless the user already had the page loaded before the server crashed, they wouldn't be able to see this page anyways.

Other things that could possibly go wrong make up such a small percentage that we are willing to allow a slightly worse experience to make the experience better for the 99.9% of cases where everything is working properly.

Furthermore, the object we pass as the second argument is meant to emulate the response from the server. This is not the best design pattern because it assumes we know what the response will look like. If the response changes, we have to update this code. However, given what we have, this is an acceptable cost.

So what happens when the API call returns an error?

```
$rootScope.$broadcast('post.created.error');
```

If the error callback is triggered, then we will broadcast a new event: `post.created.error`. The event listener we set up earlier will be trigger by this event and remove the post at the front of `vm.posts`. We will also show the error message to the user to let them know what happened.

```
$scope.closeThisDialog();
```

This is a method provided by `ngDialog`. All it does is close the model we have open. It's also worth nothing that `closeThisDialog()` is not stored on the ViewModel, so we must call `$scope.closeThisDialog()` instead of `vm.closeThisDialog()`.

Be sure to include `new-post.controller.js` in `javascripts.html`:

```
<script type="text/javascript" src="{% static 'javascripts/posts/controllers/new-post.controller.js' %}"></script>
```

Include `new-post.controller.js` in `javascripts.html`

Checkpoint

Visit `http://localhost:8000/` and click the + button in the bottom-right corner. Fill out this form to create a new post. You will know everything worked because the new post will be displayed at the top of the page.

Create a new `Post` object via the interface you've just created

Displaying user profiles

We already have the Django views and routes necessary to display a profile for each user. From here we can jump into making an AngularJS service and then move on to the template and controllers.

> In this section and the next, we will refer to accounts as profiles. For the purposes of our client, that is effectively what the `Account` model translates into: a user's profile.

Making the profile modules

We will be creating a service and a couple of controllers relating to user profiles, so let's go ahead and define the modules we will need.

Create `static/javascripts/profiles/profiles.module.js` with the following content:

```
(function () {
  'use strict';

  angular
    .module('thinkster.profiles', [
      'thinkster.profiles.controllers',
      'thinkster.profiles.services'
    ]);

    -
```

```
  angular
    .module('thinkster.profiles.controllers', []);

  angular
    .module('thinkster.profiles.services', []);
})();
```

Define the modules needed for profiles in `profiles.module.js`
As always, don't forget to register `thinkster.profiles` as a dependency of `thinkster` in `thinkster.js`:

```
angular
  .module('thinkster', [
    'thinkster.config',
    'thinkster.routes',
    'thinkster.authentication',
    'thinkster.layout',
    'thinkster.posts',
    'thinkster.profiles'
  ]);
```

Register `thinkster.profiles` as a dependency of the `thinkster` module
Include this file in `javascripts.html`:

```
<script type="text/javascript" src="{% static 'javascripts/profiles/profiles.module.js' %}"></script>
```

Making a Profile factory

With the module definitions in place, we are ready to create the `Profile` service that will communicate with our API.

Create `static/javascripts/profiles/services/profile.service.js` with the following contents:

```
/**
* Profile
* @namespace thinkster.profiles.services
*/
(function () {
  'use strict';

  angular
    .module('thinkster.profiles.services')
    .factory('Profile', Profile);

  Profile.$inject = ['$http'];

  /**
  * @namespace Profile
  */
  function Profile($http) {
    /**
    * @name Profile
    * @desc The factory to be returned
    * @memberOf thinkster.profiles.services.Profile
    */
    var Profile = {
      destroy: destroy,
      get: get,
      update: update
    };

    return Profile;

    ////////////////////

    /**
    * @name destroy
    * @desc Destroys the given profile
    * @param {Object} profile The profile to be destroyed
    * @returns {Promise}
    * @memberOf thinkster.profiles.services.Profile
    */
    function destroy(profile) {
      return $http.delete('/api/v1/accounts/' + profile.id + '/');
    }


    /**
    * @name get
    * @desc Gets the profile for user with username `username`
    * @param {string} username The username of the user to fetch
    * @returns {Promise}
    * @memberOf thinkster.profiles.services.Profile
    */
    function get(username) {
      return $http.get('/api/v1/accounts/' + username + '/');
    }


    /**
    * @name update
    * @desc Update the given profile
    * @param {Object} profile The profile to be updated
    * @returns {Promise}
    * @memberOf thinkster.profiles.services.Profile
    */
    function update(profile) {
      return $http.put('/api/v1/accounts/' + profile.username + '/', profile);
    }
  }
})();
```

Create a new factory called `Profiles` in `static/javascripts/profiles/services/profiles.service.js`
We aren't doing anything special here. Each of these API calls is a basic CRUD operation, so we get away with not having much code.

Add this file to `javascripts.html`:

```
<script type="text/javascript" src="{% static 'javascripts/profiles/services/profile.service.js' %}"></script>
```

Include `profiles.service.js` in `javascripts.html`

Making an interface for user profiles

Create `static/templates/profiles/profile.html` with the following content:

```
<div class="profile" ng-show="vm.profile">
  <div class="jumbotron profile__header">
    <h1 class="profile__username">+{{ vm.profile.username }}</h1>
    <p class="profile__tagline">{{ vm.profile.tagline }}</p>
  </div>

  <posts posts="vm.posts"></posts>
</div>
```

Make a template for displaying profiles in `static/templates/profiles/profile.html`
This will render a header with the username and tagline of the profile owner, followed by a list of their posts. The posts are rendered using the directive we created earlier for the index page.

Controlling the profile interface with ProfileController

The next step is to create the controller that will use the service we just created, along with the `Post` service, to retrieve the data we want to display.

Create `static/javascripts/profiles/controllers/profile.controller.js` with the following content:

```
/**
* ProfileController
* @namespace thinkster.profiles.controllers
*/
(function () {
  'use strict';

  angular
    .module('thinkster.profiles.controllers')
    .controller('ProfileController', ProfileController);

  ProfileController.$inject = ['$location', '$routeParams', 'Posts', 'Profile', 'Snackbar'];

  /**
  * @namespace ProfileController
  */
  function ProfileController($location, $routeParams, Posts, Profile, Snackbar) {
    var vm = this;

    vm.profile = undefined;
    vm.posts = [];

    activate();

    /**
    * @name activate
    * @desc Actions to be performed when this controller is instantiated
    * @memberOf thinkster.profiles.controllers.ProfileController
    */
    function activate() {
      var username = $routeParams.username.substr(1);

      Profile.get(username).then(profileSuccessFn, profileErrorFn);
      Posts.get(username).then(postsSuccessFn, postsErrorFn);

      /**
      * @name profileSuccessProfile
      * @desc Update `profile` on viewmodel
      */
      function profileSuccessFn(data, status, headers, config) {
        vm.profile = data.data;
      }


      /**
      * @name profileErrorFn
      * @desc Redirect to index and show error Snackbar
      */
      function profileErrorFn(data, status, headers, config) {
        $location.url('/');
        Snackbar.error('That user does not exist.');
      }


      /**
        * @name postsSucessFn
        * @desc Update `posts` on viewmodel
        */
      function postsSuccessFn(data, status, headers, config) {
        vm.posts = data.data;
      }


      /**
        * @name postsErrorFn
        * @desc Show error snackbar
        */
      function postsErrorFn(data, status, headers, config) {
        Snackbar.error(data.data.error);
      }
    }
  }
})();
```

Create a new controller called `ProfileController` in `static/javascripts/profiles/controllers/profile.controller.js`
Include this file in `javascripts.html` :

```
<script type="text/javascript" src="{% static 'javascripts/profiles/controllers/profile.controller.js' %}"></script>
```

Include `profile.controller.js` in `javascripts.html`

Making a route for viewing user profiles ∞

Open `static/javascripts/thinkster.routes.js` and add the following route:

```
.when('/+:username', {
  controller: 'ProfileController',
  controllerAs: 'vm',
  templateUrl: '/static/templates/profiles/profile.html'
})
```

Make a route for viewing user profiles

Checkpoint

To view your profile, direct your browser to `http://localhost:8000/+<username>`. If the page renders, everything is good!

Visit your profile page at `http://localhost:8000/+<username>`

Updating user profiles

The last feature we will implement in this tutorial is the ability for a user to update their profile. The updates we offer will be minimal, including updating the user's first name, last name, email, and tagline, but you will get the gist of it and can add more options at will.

ProfileSettingsController

To get started, open `static/javascripts/profiles/controllers/profile-settings.controller.js` and add the following contents:

```
/**
* ProfileSettingsController
* @namespace thinkster.profiles.controllers
*/
(function () {
  'use strict';

  angular
    .module('thinkster.profiles.controllers')
    .controller('ProfileSettingsController', ProfileSettingsController);

  ProfileSettingsController.$inject = [
    '$location', '$routeParams', 'Authentication', 'Profile', 'Snackbar'
  ];

  /**
  * @namespace ProfileSettingsController
  */
  function ProfileSettingsController($location, $routeParams, Authentication, Profile, Snackbar) {
    var vm = this;

    vm.destroy = destroy;
    vm.update = update;

    activate();


    /**
    * @name activate
    * @desc Actions to be performed when this controller is instantiated.
    * @memberOf thinkster.profiles.controllers.ProfileSettingsController
    */
    function activate() {
      var authenticatedAccount = Authentication.getAuthenticatedAccount();
      var username = $routeParams.username.substr(1);

      // Redirect if not logged in
      if (!authenticatedAccount) {
        $location.url('/');
        Snackbar.error('You are not authorized to view this page.');
      } else {
        // Redirect if logged in, but not the owner of this profile.
        if (authenticatedAccount.username !== username) {
          $location.url('/');
          Snackbar.error('You are not authorized to view this page.');
        }
      }

      Profile.get(username).then(profileSuccessFn, profileErrorFn);

      /**
      * @name profileSuccessFn
      * @desc Update `profile` for view
      */
      function profileSuccessFn(data, status, headers, config) {
        vm.profile = data.data;
      }


      /**
      * @name profileErrorFn
      * @desc Redirect to index
      */
```

```
      */
      function profileErrorFn(data, status, headers, config) {
        $location.url('/');
        Snackbar.error('That user does not exist.');
      }
    }


    /**
    * @name destroy
    * @desc Destroy this user's profile
    * @memberOf thinkster.profiles.controllers.ProfileSettingsController
    */
    function destroy() {
      Profile.destroy(vm.profile.username).then(profileSuccessFn, profileErrorFn);

      /**
      * @name profileSuccessFn
      * @desc Redirect to index and display success snackbar
      */
      function profileSuccessFn(data, status, headers, config) {
        Authentication.unauthenticate();
        window.location = '/';

        Snackbar.show('Your account has been deleted.');
      }


      /**
      * @name profileErrorFn
      * @desc Display error snackbar
      */
      function profileErrorFn(data, status, headers, config) {
        Snackbar.error(data.error);
      }
    }


    /**
    * @name update
    * @desc Update this user's profile
    * @memberOf thinkster.profiles.controllers.ProfileSettingsController
    */
    function update() {
      Profile.update(vm.profile).then(profileSuccessFn, profileErrorFn);

      /**
      * @name profileSuccessFn
      * @desc Show success snackbar
      */
      function profileSuccessFn(data, status, headers, config) {
        Snackbar.show('Your profile has been updated.');
      }


      /**
      * @name profileErrorFn
      * @desc Show error snackbar
      */
      function profileErrorFn(data, status, headers, config) {
        Snackbar.error(data.error);
      }
    }
  }
})();
```

Create the `ProfileSettingsController` controller
Be sure to include this file in `javascripts.html` :

```
<script type="text/javascript" src="{% static 'javascripts/profiles/controllers/profile-settings.controller.js' %}"></script>
```

Include `profile-settings.controller.js` in `javascripts.html`
Here we have created two methods that will be available to the view: `update` and `destroy` . As their names suggest, `update` will allow the user to update their profile and `destroy` will destroy the user's account.

Most of this controller should look familiar, but let's go over the methods we've created for clarity.

```
/**
 * @name activate
 * @desc Actions to be performed when this controller is instantiated.
 * @memberOf thinkster.profiles.controllers.ProfileSettingsController
 */
function activate() {
  var authenticatedAccount = Authentication.getAuthenticatedAccount();
```

```
  var username = $routeParams.username.substr(1);

  // Redirect if not logged in
  if (!authenticatedAccount) {
    $location.url('/');
    Snackbar.error('You are not authorized to view this page.');
  } else {
    // Redirect if logged in, but not the owner of this profile.
    if (authenticatedAccount.username !== username) {
      $location.url('/');
      Snackbar.error('You are not authorized to view this page.');
    }
  }

  Profile.get(username).then(profileSuccessFn, profileErrorFn);

  /**
   * @name profileSuccessFn
   * @desc Update `profile` for view
   */
  function profileSuccessFn(data, status, headers, config) {
    vm.profile = data.data;
  }

  /**
   * @name profileErrorFn
   * @desc Redirec to index
   */
  function profileErrorFn(data, status, headers, config) {
    $location.url('/');
    Snackbar.error('That user does not exist.');
  }
}
```

In `activate`, we follow a familiar pattern. Because this page allows for dangerous operations to be performed, we must make sure the current user is authorized to see this page. We do this by first checking if the user is authenticated and then checking if the authenticated user owns the profile. If either case is false, then we redirect to the index page with a snackbar error stating that the user is not authorized to view this page.

If the authorization process succeeds, we simply grab the user's profile from the server and allow the user to do as they wish.

```
/**
 * @name destroy
 * @desc Destroy this user's profile
 * @memberOf thinkster.profiles.controllers.ProfileSettingsController
 */
function destroy() {
  Profile.destroy(vm.profile).then(profileSuccessFn, profileErrorFn);

  /**
   * @name profileSuccessFn
   * @desc Redirect to index and display success snackbar
   */
  function profileSuccessFn(data, status, headers, config) {
    Authentication.unauthenticate();
    window.location = '/';

    Snackbar.show('Your account has been deleted.');
  }


  /**
   * @name profileErrorFn
   * @desc Display error snackbar
   */
  function profileErrorFn(data, status, headers, config) {
    Snackbar.error(data.error);
  }
}
```

When a user wishes to destroy their profile, we must unauthenticate them and redirect to the index page, performing a page refresh in the process. This will make the navigation bar re-render with the logged out view.

If for some reason destroying the user's profile returns an error status code, we simply display an error snackbar with the error message returned by the server. We do not perform any other actions because we see no reason why this call should fail unless the user is not authorized to delete this profile, but we have already accounted for this scenario in the `activate` method.

```
/**
 * @name update
 * @desc Update this user's profile
 * @memberOf thinkster.profiles.controllers.ProfileSettingsController
 */
function update() {
  Profile.update(vm.profile).then(profileSuccessFn, profileErrorFn);

  /**
   * @name profileSuccessFn
   * @desc Show success snackbar
```

```
    */
  function profileSuccessFn(data, status, headers, config) {
    Snackbar.show('Your profile has been updated.');
  }


  /**
   * @name profileErrorFn
   * @desc Show error snackbar
   */
  function profileErrorFn(data, status, headers, config) {
    Snackbar.error(data.error);
  }
}
```

`update()` is very simple. Whether the call succeeds or fails, we show a snackbar with the appropriate message.

A template for the settings page

As usual, now that we have the controller we need to make a corresponding template.

Create `static/templates/profiles/settings.html` with the following content:

```
<div class="col-md-4 col-md-offset-4">
  <div class="well" ng-show="vm.profile">
    <form role="form" class="settings" ng-submit="vm.update()">
      <div class="form-group">
        <label for="settings__email">Email</label>
        <input type="text" class="form-control" id="settings__email" ng-model="vm.profile.email" placeholder="ex. john@example.com" />
      </div>

      <div class="form-group">
        <label for="settings__password">New Password</label>
        <input type="password" class="form-control" id="settings__password" ng-model="vm.profile.password" placeholder="ex. notgoogleplus" />
      </div>

      <div class="form-group">
        <label for="settings__confirm-password">Confirm Password</label>
        <input type="password" class="form-control" id="settings__confirm-password" ng-model="vm.profile.confirm_password" placeholder="ex. notgo
ogleplus" />
      </div>

      <div class="form-group">
        <label for="settings__username">Username</label>
        <input type="text" class="form-control" id="settings__username" ng-model="vm.profile.username" placeholder="ex. notgoogleplus" />
      </div>

      <div class="form-group">
        <label for="settings__tagline">Tagline</label>
        <textarea class="form-control" id="settings__tagline" ng-model="vm.profile.tagline" placeholder="ex. This is Not Google Plus." />
      </div>

      <div class="form-group">
        <button type="submit" class="btn btn-primary">Submit</button>
        <button type="button" class="btn btn-danger pull-right" ng-click="vm.destroy()">Delete Account</button>
      </div>
    </form>
  </div>
</div>
```

Create a template for `ProfileSettingsController`
This template is similar to the forms we created for registering and logging in. There is nothing here worth discussing.

Profile settings route

Open up `static/javascripts/thinkster.routes.js` and add the following route:

```
// ...
.when('/+:username/settings', {
  controller: 'ProfileSettingsController',
  controllerAs: 'vm',
  templateUrl: '/static/templates/profiles/settings.html'
})
// ...
```

Checkpoint

And that's our last feature! You should now be able to load up the settings page at `http://localhost:8000/+:username/settings` and update your settings as you wish.

Try updating your tagline. If it works, you will now see your tagline displayed on your profile page.

Update your tagline and view the new tagline on your profile page

Congratulations, you did it!

During this tutorial you accomplished a lot.

For starters, you build an entire authentication system by yourself! You extended Django's built-in `User` model and added various attributes and did so in a way that makes adding other information an easy feat when it becomes necessary. You went on to built both the front and back ends for registration, logging in, logging out, and updating the user's profile.

In addition to building the authentication system you also create a way for users to add their posts to our application and view other users' posts.

This is the stuff that we do as engineers of the web. There will be times when you will need skills learned outside this tutorial and there are certainly best practices that we did not touch on, but what you've done here is the gist of web development!

Be proud of what you've accomplished here and tell you friends by tweeting about it (https://twitter.com/intent/tweet?text=I%20just%20built%20a%20Google%20Plus%20clone%20with%20%23django%20and%20%23angularjs!&url=http%3A%2F%2Fthinkster.io%2F&via=gothinkster). We hope that you enjoyed this tutorial and will come back when you want to learn more. As always, our inbox is open to your comments, suggestions, and feedback.

Happy hacking!

Contributors

Before you go, I want to give a shoutout to all of the people who were kind enough to send us emails and pull requests.

Here is a full list of contributors who helped with the current release:

Albert Pai, Christophe Blefari, Diego Martinez, Eric Simons, Ernest Ezis, Iulian Gulea, James Brewer, Lorenzo Cinque, Martin Hill, Martin Oosthuizen, Matt Green, Ronald Paloschi, Seth Clossman, Vladimir Vitvitskiy, Zach Reinhardt