Learning Objectives

- **Setup a blog application in Django**
- **Configure the dev server to run inside the Codio platform**
- **Run database migrations**
- **Create the `Tag` and `Post` models**
- **Register the `Tag` and `Post` models with Django Admin**
- **Create a post through Django Admin**

Connecting to GitHub

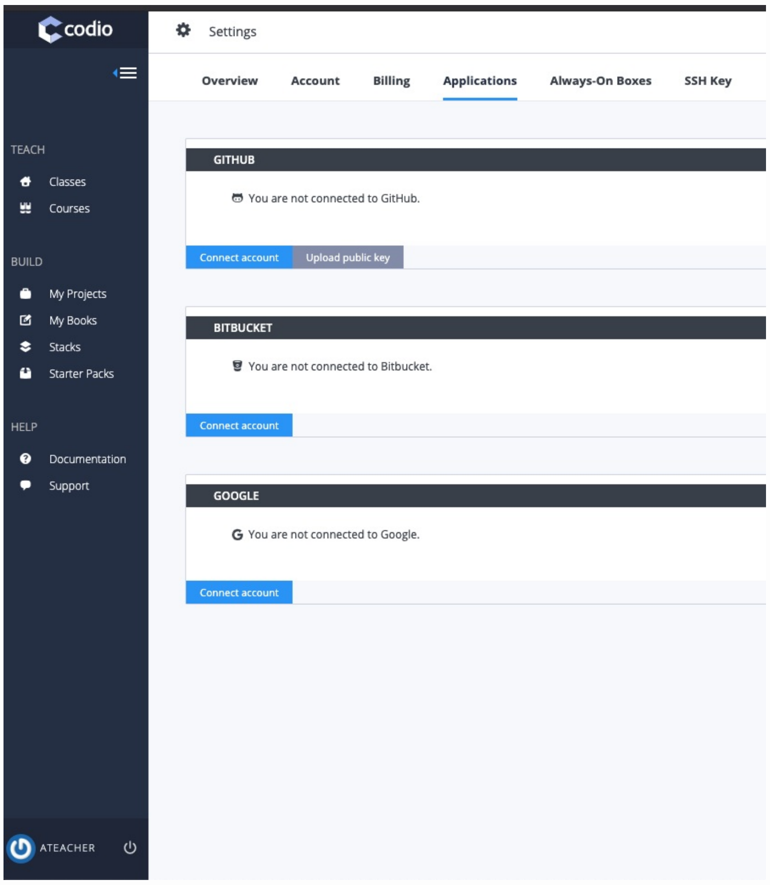
Connecting to GitHub

Over four courses, you are going to build a Django app. In order to get your code in every assignment, you are going to use GitHub. If you do not yet have an account, please create one now. We are going to clone a repo that will contain the code for your Django app.

Connecting GitHub and Codio

You need to connect GitHub to your Codio account. This only needs to be done one time.

- In your Codio account, click on your username
- Click on **Applications**



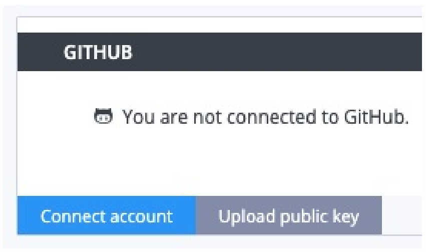
You are not connected to GitHub.

You are not connected to Bitbucket.

You are not connected to Google.

Codio Account

- Under GitHub, click on **Connect account**



connect account

- You will be using an SSH connection, so you need to click on **Upload public key**

Fork the Repo

- Go to the blango repo. This repo is the starting point for your blog.
- Click on the “Fork” button in the top-right corner.
- Click the green “Code” button.
- Copy the SSH information. It should look something like this:

```
git@github.com:<your_github_username>/blango.git
```

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Introduction

Introduction

Hello, and welcome to Advanced Django. This course is aimed at those who have some understanding of Django, and perhaps have even completed a project or two.

This course is divided into four modules. We'll be building blog project in Django throughout the course, gradually adding features to it. The first module expands on the Django fundamentals. We'll look at the admin section, generic model relationships and the Bootstrap HTML framework, then how using custom template tags and filters can cut down the amount of code in your templates.

Module 2 introduces *12-Factor Apps*: an informal standard for configuration applications. We'll look at the most important of these factors to build into a Django application (configuration and logging). Then we'll give you a better understanding of some of Django's security features, then discuss some different ways to deploy your application.

Module 3 is about performance. We'll introduce caching and how to use it with your Django application, then look at how to optimize your database and queries for more speed.

Module 4 is about the user. You'll learn about custom user models, and how they differ from just using a profile object. We'll then look at some third party modules to help with users and registration: Django Registrations (to make users validate their email address after signing up), and Django All Auth to allow authentication via third party services or social networks. We close the module by walking through how to set up authentication against Google OAuth.

Code

All code is PEP8 formatted using the Black code formatting tool for consistent formatting.

The Project: Blango

In this course (and through Course 3) you'll be building a Blog app in Django. It's called *Blango* (*Blog* + *Django*, of course!)

Hopefully the concepts behind the app are quite familiar. It’s been deliberately chosen to let you focus on the Django side of things rather than the domain/data model. Each author is a Django user and can create blog Posts which have Tags assigned to them. Comments can be added to Posts, and files can be attached to either Posts or Comments.

Not everything will be built at once, we will gradually add features to make the application more useful.

Project Setup

Project Setup

Because this is an advanced Django course, we are not going to go over installing Django and starting a project. The GitHub repo that you forked is a starting point for the `blango` project.

Change in to the `blango` directory and start an app called `blog`.

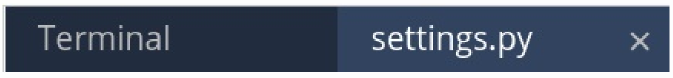
```
cd blango
python3 manage.py startapp blog
```

As usual, you'll need to add the `blog` app to your `INSTALLED_APPS` in your Django `settings.py` file.

important

Open `settings.py`

Click the link below to open the `settings.py` file. You will now need to click between the tabs to toggle between the terminal and `settings.py`.



Terminal and Settings.py Tabs

Open settings.py

The `INSTALLED_APPS` variable should now look like this:

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "blog",
]
```

Codio Specific Setup

Codio Specific Setup

info

Running Django in Codio

The changes on this page are done to allow Django to run on the Codio platform. By default, Django has strict security features that keep Codio from embedding Django.

The code samples below **only** apply for Codio. You **would not** make these changes when running Django outside of Codio.

Before setting up Django to run on Codio, you first need to import the `os` module.

```
import os
```

The two biggest obstacles in getting Django to run inside Codio are recognizing the unique host name for each Codio project and cookies. The changes below will tell Django the exact host name of the Codio project and alter how Django handles cookies.

```
ALLOWED_HOSTS = ['*']
X_FRAME_OPTIONS = 'ALLOW-FROM ' +
        os.environ.get('CODIO_HOSTNAME') + '-8000.codio.io'
CSRF_COOKIE_SAMESITE = None
CSRF_TRUSTED_ORIGINS = [os.environ.get('CODIO_HOSTNAME') +
        '-8000.codio.io']
CSRF_COOKIE_SECURE = True
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SAMESITE = 'None'
SESSION_COOKIE_SAMESITE = 'None'
```

Scroll down a bit and look for the definition of the variable `MIDDLEWARE`. Comment out the two lines that refer to `csrf`. CSRF stands for cross-site request forgery, which is a way for a malicious actor to force a user to perform an unintended action. Djano normally has CSRF protections in place, but we need to disable them. There are serious security implications to doing so, however your project on Codio is not publicly available on the internet.

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

]
```

Reminder: these changes only apply to working with Django on Codio. **Do not** make these changes to a project you plan on making available on the internet.

Project Migration

Project Migration

The final piece of set up is to run the database migrations with the `migrate` management command in the terminal.

```
python3 manage.py migrate
```

If successful, you should see the following output:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
```

A user needs to be created, to be able to log into the Django admin site. This is done in the terminal with the `createuser` management command.

```
python3 manage.py createsuperuser
```

If successful, you should see the following interaction. You can use your own username, email, and password. Be sure to remember them as this information will be used throughout this project.

```
Username (leave blank to use 'codio'):
Email address: codio@example.com
Password: password
Password (again): password
This password is too common.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

Note that you see the warning *This password is too common.*, because I've decided to literally use the password *password* here. This is fine if you're the only person using the application for testing, but definitely not recommended for production!

Models

Models

Now we can get started on our models. The data models we need to get started with are simple: there's `Post` which contains the blog post, and `Tag`, which contains the tag text. They are related to each other with a `ManyToManyField`. We won't spend too much time going through them as you should be familiar with Django models.

Be sure that you are importing `models` and `settings` from their respective modules.

```
from django.db import models
from django.conf import settings
```

Here's the `Tag` class.

```
class Tag(models.Model):
    value = models.TextField(max_length=100)

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

It's simple and just contains a single field for `value`.

The `Post` model is a little more complex but hopefully the fields aren't too unfamiliar.

```
class Post(models.Model):
    author = models.ForeignKey(settings.AUTH_USER_MODEL,
        on_delete=models.PROTECT)
    created_at = models.DateTimeField(auto_now_add=True)
    modified_at = models.DateTimeField(auto_now=True)
    published_at = models.DateTimeField(blank=True, null=True)
    title = models.TextField(max_length=100)
    slug = models.SlugField()
    summary = models.TextField(max_length=500)
    content = models.TextField()
    tags = models.ManyToManyField(Tag, related_name="posts")

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

Some of the things of note to point out are:

Author Foreign Key

The `author` is a `ForeignKey` to `settings.AUTH_USER_MODEL`: a Django setting which is a string. `ForeignKey` can be used either by passing the class itself, or passing a string which is parsed to load and refer to the class. By passing `settings.AUTH_USER_MODEL`, we'll be able to change the model class that's used for authentication by updating the Django settings, and all models that refer to this setting will update automatically to use the right model. By default, the value is `auth.User`, which refers to the `User` model in the Django `auth` application.

To use the class in our `ForeignKey`, we need to be sure to import the Django settings model by doing `from django.conf import settings` at the top of the file.

Date Fields

The field `created_at` sets `auto_now_add` to `True` which means that when a `Post` is saved, its creation date and time will automatically be set. Similarly, `modified_at` is instantiated with `auto_now` set to `True`, which means it will be set to the current date and time whenever a `Post` is saved.

Slugs and Slug Field

A *slug* is a short string designed to be used as an identifier, such as in a URL. Normally it is composed of lower case letters, numbers, and dashes. Slugs are used for search engine optimization (making pages rank higher in search engine results), as well as to make URLs more readable for humans. Taking one of the posts from the official Django blog as an example: *https://www.djangoproject.com/weblog/2021/may/26/django-irc-channels-*

migration-liberachat/. The slug from this URL is *django-irc-channels-migration-liberachat*. You can imagine that the Blog post probably has a numeric ID, but accessing it through a URL like *https://www.djangoproject.com/weblog/35* isn't very friendly.

The slug is usually generated from the title of the page, with common words removed. In the Django example the page title is *Django IRC Channels migration to Libera.Chat*. You can see the slug is generated by converting to lowercase, switching spaces with dashes and removing non alpha-numeric characters.

The Django `SlugField` inherits from `CharField` but adds some validation to ensure the value looks like a slug (lowercase, alphanumeric and dashes only) and by default limits it to 50 characters.

Django provides the `[slugify]` (https://docs.djangoproject.com/en/3.2/ref/utils/#django.utils.text.slugify) function to convert an arbitrary string to a "slugified" version. We can also configure the Django admin to convert one field into a slug. For example, converting `title` to `slug`. We'll see how to do this in the next section, *Django Admin Quickstart*.

Content

For simplicity we'll just be storing the actual blog content as HTML, so we don't have to worry about doing any kind of conversion when displaying it, we'll just render the content field verbatim. This is not the most secure approach, so it's only advisable if you trust your authors not to add malicious HTML. If you're building a site that will output user-supplied HTML, consider using something like Bleach to remove unsafe HTML.

Migrations

Before getting started with Django admin, we need to perform the usual steps when creating a new model. First, make the migrations with the `makemigrations` management command.

```
python3 manage.py makemigrations
```

If successful, you should see the following output:

```
Migrations for 'blog':
  blog/migrations/0001_initial.py
    - Create model Tag
    - Create model Post
```

Then apply the migrations to the database with the `migrate` management command.

```
python3 manage.py migrate
```

If successful, you should see the following output:

```
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes,
        sessions
Running migrations:
  Applying blog.0001_initial... OK
```

Django Admin Quickstart

Django Admin Quickstart

One of the features of Django the makes building websites with dynamic content so fast is its built in admin system. It provides a web interface that lets you create, edit and delete any Django model instances that you choose. The admin system is enabled by default when you start a Django project, but only models that you choose are exposed. By default the path to access the admin UI is `/admin/`, (e.g. http://127.0.0.1:8000/admin/ if running the Django debug server).

Disabling Django Admin

If you don't want Django Admin enabled, how is it turned off? There's two things to do:

- Remove `django.contrib.admin` from `INSTALLED_APPS` in the Django settings file.
- Remove the `admin/` URL rule from the project's `urls.py` file.

While on the subject of the admin URL, you can get a little extra security if you change it from the default `admin/` to something else. This would prevent attackers from guessing the URL. Since the admin site is password protected anyway, this is not something you'd normally have to do, but it could help if any of your admin users have weak passwords. We'll stick with the default `admin/` in this course.

Registering Models to the admin UI

Upon scaffolding a Djanog app, an `admin.py` file is automatically added. This is where Django will read the definition for the admin interface for your app. There are two ways to register a model and have it show in the Django admin UI. You can either register the model directly, which will just use the defaults and allow all fields to be editable. Most of the time, this is adequate, but sometimes you might want to create a model admin class by subclassing the Django `admin.ModelAdmin` class. This allows you to set options about how your model is displayed in the admin. Let's examine the `admin.py` file and then see how to implement both these methods.

When scaffolded the `admin.py` file contains just one code line (and a comment, which can be removed):

```
from django.contrib import admin
```

As you can probably guess, this imports the Django admin module, ready for use. To register a model into the admin section, we use the `admin.site.register` function.

info

What is `admin.site`?

`admin.site` is an object representing the admin site. We also saw it used in the `urls.py` file, routing the `admin/` path to `admin.site.urls`. We won't discuss its functionality in depth, but it's possible to customize the site to override things like the page titles, header text and template. You can find more details about this at the Django Admin Site Documentation

Let's start by looking at the simplest method of registering a model. We'll do this with the `Tag` model. First our model needs to be imported into the `admin.py` file:

```
from blog.models import Tag, Post
```

Then register it:

```
admin.site.register(Tag)
```

To configure how the admin site behaves with a certain model, a subclass of `admin.ModelAdmin` must be created. This subclasse's attributes determine how the model is displayed. First let's look at how we'll create one, for the `Post` model.

```
class PostAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("title",)}
```

Here we're setting just one attribute, `prepopulated_fields`. When used in this way, some JavaScript is inserted into the admin page so that the `slug` field updates when the `title` field changes. It will automatically "slugify" the title. But, there are many other ways to customise the ModelAdmin. Some of the more common customizations are:

- `exclude`: a list of fields to disallow editing of in the admin. For example, we might want to prevent users from manually setting the `slug`, and instead compute it when saving the Model. In which case, we would set `exclude` to `["slug"]"`.
- `fields`: this works the opposite way to `excludes`. If set, only fields in the `fields` list will be editable. Note that if a field requires a value, but is not editable (either by the use of `exclude` or `fields`, then saving the model instance will fail because the field will not be valid.
- `list_display`: a list of fields to include in the admin page list view. For example, we might want to see both a Post's title, and when it was published. We would do this by setting `list_display` to `["title", "published-at"]`.

There are many more options that can be used, and the full list can be viewed at the Django Model Admin Options Documenatation.

Now that the `PostAdmin` class is defined, let's return to the `register` function. The `PostAdmin` class is passed as the second argument:

```
admin.site.register(Post, PostAdmin)
```

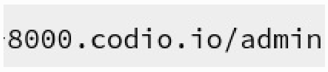
Launching the Blog

Launching the Blog

Now we're ready to check it all out. Start the Django test server by entering the following command in the terminal. The blog should load in the top panel.

```
python3 manage.py runserver 0.0.0.0:8000
```

Once, Django is up and running, you need to navigate to the admin page. In the project URL, add a `/admin` and press **Enter** on the keyboard.



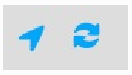
8000.codio.io/admin

Django Admin URL

info

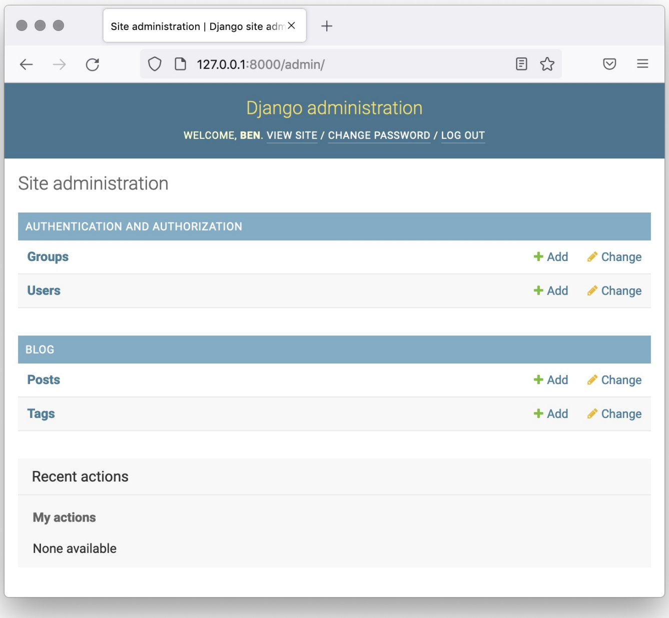
View in External Tab

If you want to see the Django application in its own tab, click the blue arrow in the top-left corner of the blog panel.



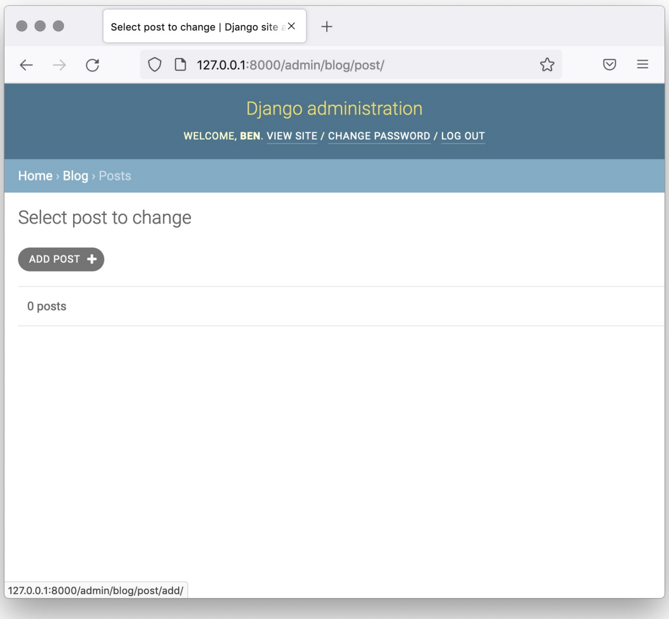
View Blog Project in an External Tag

You'll need to log in with the username and password you created earlier. Then, you should see a list of all the available models: Groups and Users (which are part of the Django auth system), then Posts and Tags, which are part of our Blog app.



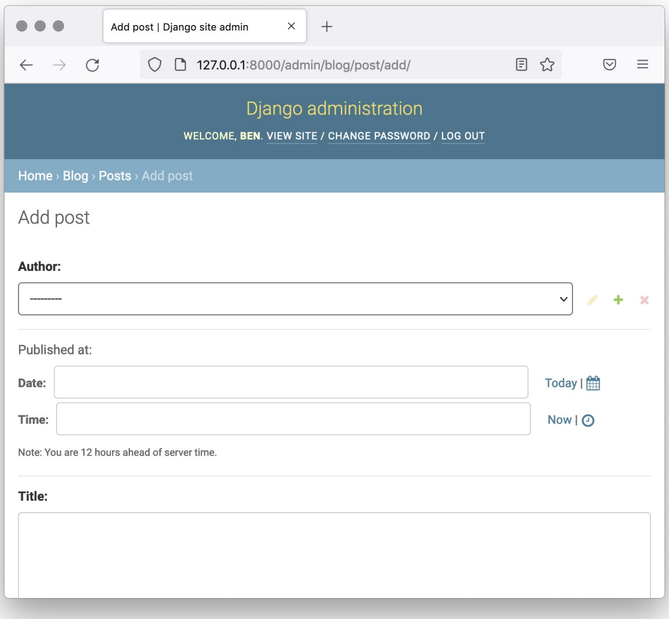
Models List

We'll now quickly walk through how to create a Post. First, click on `Posts` to go to the Posts list view. You should see an empty table, with an **Add Post** button at the top.


Select post to change | Django site
127.0.0.1:8000/admin/blog/post/
Django administration
WELCOME, BEN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Blog › Posts
Select post to change
ADD POST +
0 posts
127.0.0.1:8000/admin/blog/post/add/

Add Post

Click this button, and you'll see the Post edit form. Note that since `created_at` and `modified_at` are set automatically, they don't show up in the fields. Likewise, the model instance's `id`/`pk` (primary key) doesn't show as it shouldn't be changed.


Add post | Django site admin

127.0.0.1:8000/admin/blog/post/add/

Django administration

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

Home › Blog › Posts › Add post

Add post

Author: ---------

Published at:

Date: Today |

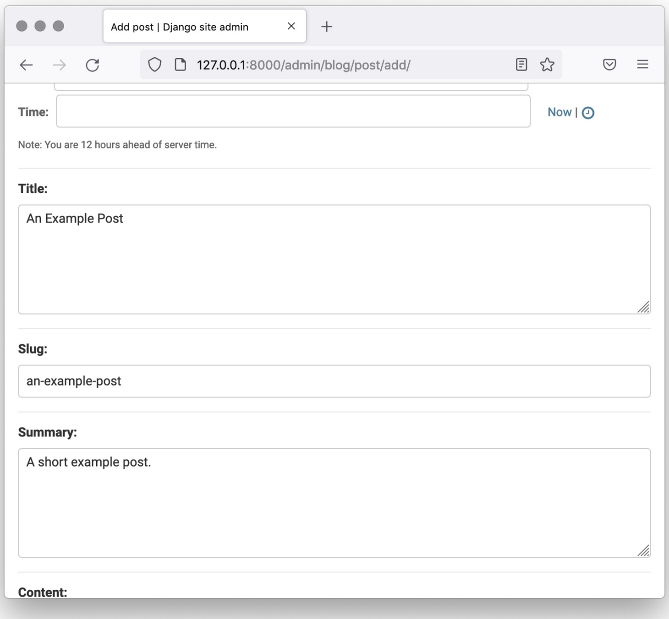
Time: Now |

Note: You are 12 hours ahead of server time.

Title:

Post Form

You can go through and enter some information to create a post. Note that as you enter a title, the slug automatically updates.


Add post | Django site admin

127.0.0.1:8000/admin/blog/post/add/

Time: Now |

Note: You are 12 hours ahead of server time.

Title: An Example Post

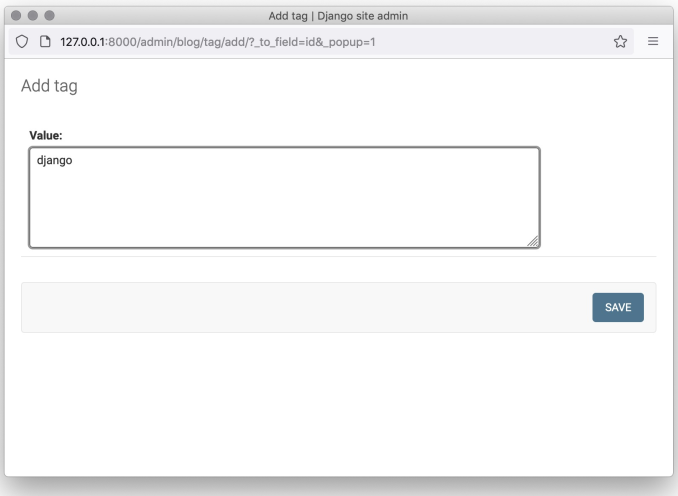
Slug: an-example-post

Summary: A short example post.

Content:

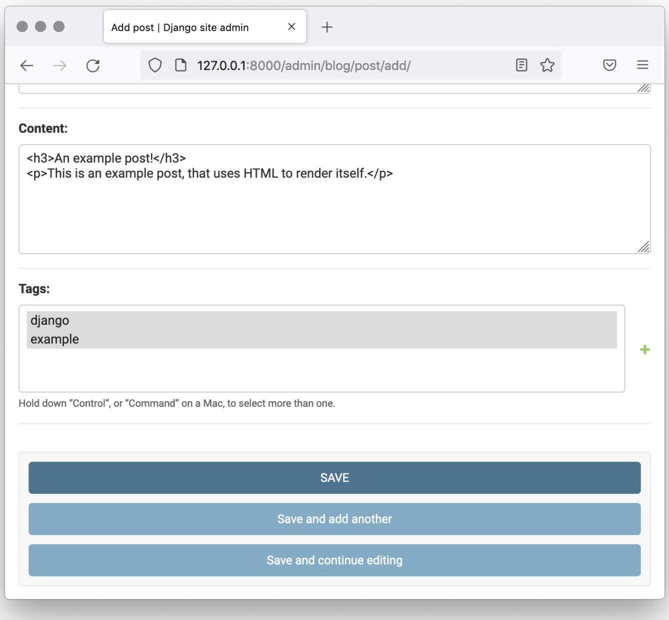
Slug Updates

You can add multiple tags by clicking the green plus icon to the right of the tags list. The tag editor opens in a new window.


Add tag | Django site admin
127.0.0.1:8000/admin/blog/tag/add/?_to_field=id&_popup=1
Add tag
Value:
django
SAVE

Tag Editor

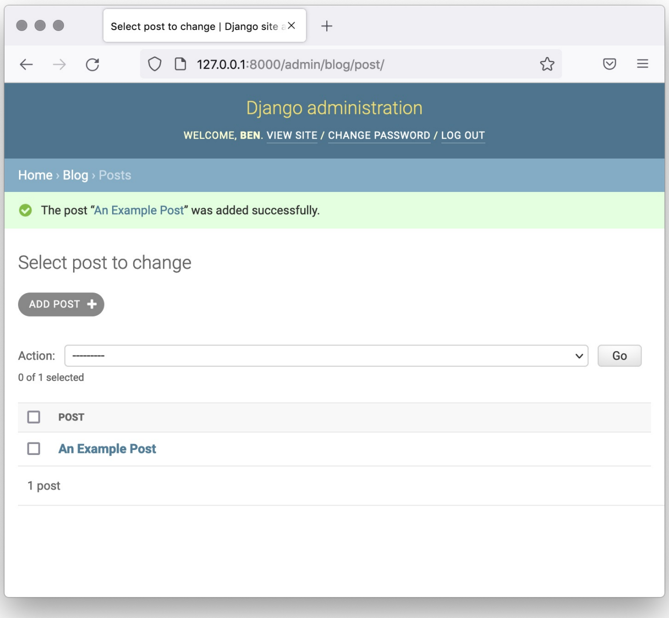
Click **Save** after entering a Value for the tag, and then the Tag editor will close and you'll return to the Post editor.


Add post | Django site admin
127.0.0.1:8000/admin/blog/post/add/
Content:
Tags:
django
example
Hold down "Control", or "Command" on a Mac, to select more than one.
SAVE
Save and add another
Save and continue editing

```
<h3>An example post!</h3>
<p>This is an example post, that uses HTML to render itself.</p>
```

Tags Entered

Make sure the tags are selected, then click **Save**. You'll go back to the Post list, and you'll see the Post you just created.



Select post to change | Django site
127.0.0.1:8000/admin/blog/post/
Django administration
WELCOME, BEN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Blog › Posts

The post "An Example Post" was added successfully.

Select post to change

ADD POST
Action: ---------
Go

0 of 1 selected

POST
An Example Post

1 post

Posts List

This was a quick intro to the Django Admin site.

challenge

Try this variation:

- Try changing the `PostAdmin` to show the `slug` and `published_at` in the Post list.

▼ Solution

Use `list_display` to change the information presented in the Post list. See the Django Model Admin Options Documenatation() for more information.

```
class PostAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("title",)}
    list_display = ('slug', 'published_at')
```

That was a brief introduction to the Django Admin system, a simple but powerful way to get content into your application. We'll often use it as it saves time having to build custom forms.

In the next section, we'll look at the Django generic relations system, and how it can be used.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish Introduction and Django Admin"
```

- Push to GitHub:

```
git push
```

Formative Assessment 1

Formative Assessment 2