Getting Started with Django CMS

by Brennan Davis

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Learnin

Course Overview

Course Overview

Hi everyone, my name is Brennan Davis, and welcome to my course, Getting Started with Django CMS. I am a front-end web developer at Endurance International. Django is a powerful Python-based web frame work, that allows developers to get a web application up and running quickly. And while it's amazing for building web applications, with large amounts of data, it lacks the basic features of a content management system. Django CMS is an add-on to Django that provides this missing content management functionality. In this course we're going to learn how to build, an easy to use, easy to maintain website, using Django and Django CMS. Some of the major topics that we're going to cover are why you would want to use Django CMS, over something more popular such as WordPress or Druple, how to start up a new Django project, with Django CMS already installed and configured, how to create custom page templates for your pages, how to create custom plugins for the site's content, and how to hook traditional Django apps into the Django CMS system. By the end of this course you'll know how to build a Django CMS powered site from the ground up. And we'll be able to easily hand those sites off to clients, to manage on their own. Before beginning the course you should be familiar with the basics of Django, Python, HTML, CSS, and running commands on the command line. I hope you'll join me on this journey to learn Django CMS, with the Getting Started with Django CMS course, at Pluralsight.

Installing and Configuring Django, and Django CMS

Course Introduction

Hello there, and welcome to Pluralsight. I'm Brennan Davis, professional web designer and developer. Throughout this course, we'll be learning how to supplement the powerful web framework Django with a fully customizable content management system called Django CMS. This course assumes you're already familiar with Python, Django, and basic web technologies such as HTML, CSS, and JavaScript. If you aren't familiar with Django, I suggest watching the Django Fundamentals course found right here on Pluralsight. Then, once you're comfortable with the basics, you can jump right back into this course. You can also check out Django's website at www. djangoproject. com for more information. By the end of this course, you'll know how to do the following using Django CMS. You'll be able to install and configure Django CMS in conjunction with Django, create page templates with content placeholders, create content plugins to go inside those content placeholders, and create app hooks to integrate traditional Django apps into the Django CMS system.

Why Django CMS?

Before we jump into the code, let's first discuss why we need Django CMS. If you're familiar with Django, you already know how powerful it is. What additional functionality could we need? Out of the box, Django is great for data-driven web applications. Django makes it very simple to create, receive, update, and delete data stored in the database. This makes Django a great fit for sites such as ecommerce sites with large catalogs of products or a daily news site that publishes multiple articles per day, but what if you're looking to have a smaller website such as a small local business might have with just a few pages of content about their services? Let's say these pages need to be updated every once in a while, so they can't just be hard-coded static pages. By itself, Django lacks the ability to create standalone pages from the admin system. They have to be hard-coded, which isn't convenient for a business owner. They'd have to call up their developer every time a change needed to be made to a page on their site. Now there are a lot of CMS options out there, and you could easily take your pick of any one of them to get the job done. One of the most popular content management systems by far is WordPress. It's easy for non-developers to get a site up and running in about 30 minutes or less. The problems come when you want to customize how your content is laid out on your page. WordPress only provides one large area to put your page content. This is great if you're writing a blog post, but on a static page, there may be specific areas of content you'd like to edit independently of any other. You can accomplish this with WordPress by writing short codes or custom metaboxes but these are often complex and confusing to create. Bearing all this information in mind then, we once again ask the question, "Why Django CMS? " In order to answer that question, I think now would be a perfect time for me to introduce the example project we'll be developing throughout this course. I'd like you to meet Rafael. He's the owner of his very own pizzeria, Rafael's Pizza. He's from one of those families who has a secret pizza dough recipe that's been handed down from generation to generation. Rafael has had a website for his restaurant for about a year. Some of the content he has on his site are his lunch and dinner menus and his daily signature specials. He currently has a website that provides one large content area for the front page. This has been a pain point for Rafael as he wishes to edit his menus every few weeks and his daily specials, well, he'd like to edit those on a daily basis. Both of these pieces of content currently reside in the same large content area. In order to preserve the layout of the page, HTML code has to be placed inside the content area. Rafael is not much of a technical person, so having code inside his content-editing area makes him rather nervous. He's already broken his home page on several occasions, requiring a developer to fix the issues for him. What he'd really like is a way to edit his menus and his daily specials independently from one another and not risk breaking his homepage. Here's where we finally get to know why you would use Django CMS over any other content management system. Django CMS allows you to custom-build individual content areas that are completely independent of one another. You can define the type and the quantity of content that can be placed in these areas as well, which significantly reduces user error. Django CMS integrates seamlessly with Django's existing functionality so you'll continue to get the fantastic data processing it already provides.

Install Django CMS

So now that we understand the why, let's get started creating a new site for our friend Rafael that'll be much simpler and easy to use than his existing one. I'll be using a Mac for the demonstration portions of this course. The first thing we need to do is set up our development environment. We'll be using a package called virtualenv to set up a virtual environment for our project, making it isolated from any other projects we may also be working on. If you don't already have virtualenv on your machine, you can install it using PIP. Type in pip install virtualenv. Let's create a directory for our project. I keep all my projects in a directory called Sites. I'll navigate into that now. Let's name our project rafs_pizza. Navigate inside the directory we've just created, then create your virtual environment. Type in virtualenv env_rafpizza - -no-site-packages. This will give as a clean, bare-bones environment to start with. Next, we're going to activate the environment. We do this by typing in source env_rafpizzza/bin/activate. As you can see, the name of the environment now shows up next to our user information, indicating that we're operating under that virtual environment. The creators of Django CMS provided an installation tool that greatly speeds up the process of getting all of Django CMS's dependencies installed. It also sets up our Django project for us. Double check to make sure your virtual environment is activated. Then type in pip install djangocms-installer. Let's quickly make another directory for our actual Django project to go in. This way the virtual environment isn't inside our Django project. Once you're navigated inside, we can run the installer. The command is djangocms -p, which stands for parent directory, dot to install in the current directory, rafaels_pizza. The installer will run through a series of options for setting up and configuring our project. I'm going to make sure to use the stable version of all the dependencies, and then I'm also going to use the default sqlite database configuration. Once you've gone through all the installer options, it will create your project for you. It will ask you to enter a username, email, and password for your admin user. You can make these whatever you'd like. Once it's finished, you can run the Django server, open up a browser, and navigate to 127. 0. 0. 1:8000. If all went as it should, then your Django CMS project will be up and running and you're ready to start developing.

Page Templates and Placeholders

Introduction

Hello there, and welcome back to Getting Started with Django CMS. In this next module, we'll be learning how to create page templates with placeholder areas for holding content. The main power of Django CMS comes from these customizable placeholder areas, so we'll be sure to take our time learning how to utilize them. Let's review where we are so far. We've just set up our virtual environment and installed Django and Django CMS. We've learned that custom content areas is the biggest thing setting Django CMS apart from other content management systems, and we've got an example project to work on in the form of a website for our restaurant called Rafael's Pizza. From our short amount of work in the previous module, we now have a basic website to work from for our Rafael. Let's begin by taking a tour of what Django CMS provides for us right out of the box.

Login and Page Creation

The Django CMS installer we used in the last module created a starter page for us that we use to log in to the admin system. You can also log in from any page by putting a? edit at the end of the URL. This will bring up the Django CMS toolbar and you can log in from there. This toolbar is available on all pages when logged in and provides useful links to the various parts of the system. You can also customize the toolbar with links to your own features. We'll get into that in a later module when we learn how to hook in our own Django applications. Let's go ahead and login using the user information that we created in the last module. After logging in, you'll see an Installation successful message. Below that, you should see a box that says, Add your first page. There are no pages currently created in the system, so we have to add one. Let's go ahead and do that now so we know what our pages look like in the Django CMS system. Click the blue Next button in the bottom right. Here, you can enter a title for your page. Let's just call this one Home. Enter a SLUG of home, making sure that home is not capitalized, and then for the Content we'll just say, Hello world! Go ahead and click the Create button when you're finished, and now we've got a page we can start working with.

Putting Content into Placeholders

The system has already placed us inside Edit mode, as you can see from the? edit in the URL. There are two views in edit mode, Structure and Content. Content will show you a preview of what the page will look like with your edits. Structure shows you all the places where you can edit the content on your page. These are the visual representation of the page's content placeholders. You can add, edit, and delete content plugins from here. There are several prebuilt ones you can choose from, including text and images. Let's go ahead and test some of these out to see how they work. Add a new plugin to this content placeholder by clicking on the plus button. This will bring up a list of all of the available plugins that this content placeholder can hold. Let's go ahead and just add another Text plugin. This particular text plugin provides some basic controls for formatting your text. Let's just type in something like, Hello world again! and just for the fun of it, we'll make the text bold. Once you're done editing you can hit Save, and then flip back over to the Content view, and here, you can see that our content has been added in. We'll be making our own content plugins in the next module. For now, let's create our page template with content placeholders for the home page of Rafael's website.

Creating a Page Template with Content Placeholders

It's high time we got into some code. Open up the Django Project Directory in your code editor of choice. I'm going to be using an open source editor called Brackets. Locate the Templates folder. The should be under rafaels_pizza, and then Templates. Here you'll see the templates that Django CMS gave to us right out of the box. The base. html file is the master file for our project. It's the one that all the other templates are going to be extending from. These other three templates are generic layout templates that Django CMS gave to us premade. The home page of Rafael's website has some unique requirements so we're going to be creating a template just for the home page. Go ahead and create a New file and call it home. html. Make sure that this file is inside this same Templates folder. So that we have some code to start with, let's grab the content from one of these other template files and copy it into our home. html. If you recall from the last module, the two main pieces of content that Rafael would like to have on his website are his daily specials and his menus. We're going to create content placeholders for those two pieces of content. We've already copied over two placeholders, so all we have to do is go in and rename them to what we want them to be. We'll rename sidebar to be daily_specials, and then we'll rename content to be menus. Believe it or not, that's it. You've just created your first content placeholders. We're not quite ready to apply this to a new page however. We first need to register our template in the settings. py file for our Django Project. Then, when we go to create a new page, our template should show up under the Templates dropdown. Open the Django Projects Settings file and search for cms_templates. Here we see the layout templates that the Django CMS installer created for us. On the left is the file name for the template, and on the right, is the name of the template that we want to appear in the template dropdown. Duplicate one of these entries, and on the left type in home. html, and on the right name it Home or Home Page. Save this file, and then let's head back over to the admin system in our browser.

Assign a Template to a Page

Once your page is refreshed, go to the toolbar, click on Page, Templates, and you should see your new Home Page template in the list. Click on it to assign this template to this page. Clicking on Structure in the right-hand of the toolbar will show us those two new placeholders that we created, our Daily_Specials, and our Menus. Let's add some of the content plugins to these placeholders. Let's add a Text plugin to the Daily_Specials, and let's just put in Extra large pepperoni pizza. Let me save that, and then let's go down to the Menus. We'll add a Text plugin here as well, and we'll just put in things like Pepperoni, Black Olive and Sausage, Ham and Pineapple, Barbeque Chicken, and any other flavor of pizza that you like. I'm going to go ahead and Publish these changes, and here's our content on our page. These two pieces of content were able to be edited individually from one another. Editing the Daily Specials will have no bearing on the Menus.

Add Styling

So far, our website's looking pretty boring, white background and just black text. So let's start sprucing up our website with some CSS. Head back over to your text editor, and inside the Static Directory inside rafaels_pizza, let's create a CSS Directory, and inside the CSS Directory, let's create a main. css file. I'm picturing the background of this website to be a dark red color. Mmmm, somewhere along these lines I think will work. Before we get too far, let's link up this CSS file with our base. html template, that way we can actually see the changes that we're making. We've got to make sure to link it the way that the Django system requires us to using the static template tag. Now that that's in place, let's go over to our website again and see what our new red background looks like. I think that's a good red color. We're probably going to want a logo of some sort here at the top, and then also some information in the footer along the bottom. Those two pieces of content are going to be consistent throughout the entire site, and we're going to want them to be editable by Rafael should he want to change them in the future. Say he may redesign his logo, or perhaps he'll want to change the year in the copyright information in the footer. For these pieces of content that need to be consistent throughout the entire site, Django CMS provides what's called a Static Placeholder. These placeholders are visible from every page of the website, and can be edited from every page. We'll be adding these placeholders to the base. html file. Let's go back over to our code editor and add those now.

Static Placeholders

Let's put the logo right above the nav. Making a static placeholder is really similar to making a regular placeholder, we just simply put static_placeholder instead of just placeholder. So now that we've got one for the logo, let's add one for the footer. We'll put that right underneath the content block. Save that file and then let's head back over to the browser. Refresh your browser and you should now see those placeholders showing up in the structure view of your page. You'll see that they have a little pin icon next to the name. This is what indicates that it's a static placeholder and is available on every page of the site. Let's go ahead and put in a logo for Rafael inside the logo static placeholder. We're going to use a Picture Plugin to do this. We'll go ahead and set a width of maybe 300, or 400, or so, we'll save that. Let's click on Content to see what it looks like. Looks pretty good, and then we can add some information in the footer. Probably Copyright 2016, and maybe All Rights Reserved, or something along those lines. All right, the site's looking a lot better already. As you can see, it doesn't take much to be able to get your content into the Django CMS system. Just a matter of a couple of lines of code for your placeholders and you're up and running.

Summary

As a quick review, here are the things we went over during this module, and that you should now know how to do in Django CMS. We had a brief introduction into how to navigate the Django CMS admin system. We created a new template by copying one of the prebuilt templates that Django CMS gives us out of the box. We updated the names of the placeholders so that they would better reflect the kind of content that we're going to be placing inside them. We registered the template in the Django Projects Settings file. We then applied the template to our home page, added some content to our placeholders with existing built-in Django CMS plugins. Created static placeholders for the logo and the footer. Added a logo to the logo static placeholder and some text to the footer static placeholder, and we also added some basic styles to spruce our site up a bit. In the next module, we'll look at how to create our own content plugins to better optimize the content that can be placed inside our placeholders. We'll also learn how to further customize our placeholders using the Django projects settings file.

Create Content Plugins

Introduction

We're getting into the real meat of Django CMS now. In this module, we'll be designing and building content plugins to go inside the placeholders we created in the last module. We're going to be going all the way from data structure through to front end templating. We'll also learn how to further configure our placeholders to optimize them with our new content plugins. Let's briefly review what we've learned in the first two modules. We've set up a Django CMS project locally by using a virtual environment. We've created placeholders for putting our customizable content, and we've provided some basic styling as a base for defining the site's look and feel.

Building a Plugin

Even though we've provided some basic styling in the last module, Rafael's site is still far from production ready. Black text on a red background is still pretty boring. By better defining the type of content that can go in each placeholder, and designing that content's look and feel, we'll be able to bring Rafael's site up to par. Our first content placeholder is the place we set aside for Rafael to put his daily specials. Let's create a content plugin that's optimized for displaying this kind of content. Open up a terminal window and start a new Django app. Type in. /manage. py startapp rafs_pizza_plugins. Now, open your code editor and go to the settings file. Find INSTALLED_APPS and add rafs_pizza_plugins to the bottom of the list. Make sure to put a comma after the current last app in the list, otherwise your Django application will throw errors. Open up the models. py file inside the rafs_pizza_plugins app. Here, we'll define the data structure for our plugin the same way we would any other model in Django. The difference here however, is won't be extending the models. model class like we normally would. Instead, we'll need to import the CMS plugin class. Somewhere in the top of the file type in from cms. models. pluginmodel import CMSPlugin. Then start a new class called daily_specials by typing in class Daily_Specials and in parenthesis put in CMSPlugin. Now we're ready to define our plugins fields. We'll want a name for our daily special, so let's do name = models. CharField max_length=200. An image would be good too, so let's do image = models. ImageField upload_to=daily_specials. This will create a folder called daily_specials in the project's media directory the first time an instance of daily specials is saved. A place to put a description of the daily special would be good too. So let's do description = models. TextField. We'll want there to be a link to the daily specials page, so we'll add a URL field. That should do it for the fields. Now let's configure the plugins metadata. We'll do class Meta:, and we'll want to define a verbose name of DailySpecial, and then verbose_name_plural of Daily Specials. Lastly, we'll add the Unicode function so our daily special instances are more user friendly when displayed in a list. Go ahead and save that file, and then we'll head back over to the terminal and type in. /manage. py makemigrations rafs_pizza_plugins. This will create a migration file for our app. Once it's done creating the file, type in. /manage. py migrate, and this will go ahead and create our tables in the database. In a normal Django app, we have a Views file for processing our web requests. In our content plugins app, the Views file is called CMS plugins. py. Rename the views. py to cms_plugins. py, and then go ahead and open that file. Remove the default code and put in from cms. plugin_base import CMSPluginBase, and then also from cms. plugin_pool import plugin_pool. Then, import the models we just created. Now, we'll create a class for our plugin. Call it class Daily_Specials_Plugin, and this will be extending from CMSPluginBase. There are a few variables we need to declare and provide values for. First, we need to tell what model this plugin will use, model = Daily_Specials, then, we need to put in a user-friendly name that will show when we go to select a plugin in our content placeholder. So we'll put name =, and then Daily Specials. For the last variable, we'll put in the name of the template for rendering the plugin, render_template = daily_special. html. We haven't created our html template for our plugin yet, but we'll be doing that in just a little bit. Next, we'll write out a render function that will build our context dictionary with the data we want to pass to the template. This function will be simply called Render. So we'll put in def and then render, and then the arguments that go in to it are self, context, instance, and placeholder. We're going to be updating our context object with the user inputted data. So we're going to do context. update, then parenthesis and curly braces, and then name will be the instance. name, image will be instance. image, description, instance. description, URL, instance. url, and then at the end of this function, we're going to be returning this context object. So you can see the keys for our dictionary match the fields we created in our model, and then returning the context object will allow us to tie the data of each instance of our plugin to the template, thus displaying it on the front end for each instance of the plugin. Then the last thing we need to do in this file is register the plugin. So we're going to do plugin_pool. register_plugin, and then the name of our class, which is Daily_Specials_Plugin. All right, go ahead and save this file, and now we're going to create that template file that we referenced. So let's create a templates folder inside rafs_pizza_plugins, and inside that we're going to create a file called daily_special. html. We're going to keep this pretty simple at the moment, but we could spruce it up a little later. For now, just put in the following HTML. Let's do an h1 tag, inside that we'll do some curly braces, and then name, so that's where our name will go, and then we'll do an image, and the source will be image. url, and then we'll set the alt attribute to the name, and then we'll put the description inside a p tag, and lastly, we'll have our url in an anchor tag, href= url, and then we'll just have the text be More or something like that. All right, and with that our plugin is complete. So let's head over to the browser and we can see what it looks like.

Create a Plugin Instance

Go to the Daily Specials placeholder on the Home page, and click the plus button to add a new plugin. Our new Daily Specials plugin should be in the drop-down menu, select it, and then give the daily special a name. For me, it's going to be BBQ Chicken. Put in an image of the pizza, and then write a brief description, You'll love our mouthwatering BBQ chicken pizza made with our homemade signature BBQ sauce, and we'll add a url, it doesn't really matter what it is at this point, we'll hit Save, and we'll flip over to our Content tab, and you see the plugin content rendered out on the page.

Defining Placeholder Configuration

You probably noticed the list of plugins available in the Daily Specials placeholder was quite long. We don't want to allow just any content being put in, and we want to eliminate as much potential for user error as possible. Django CMS provides the solution to that by allowing us to configure our content placeholders in the settings. py file. Let's head back over to our code editor and learn about the CMS placeholder conf setting. Once in your code editor, go to the Django Project Settings file, add a setting CMS_PLACEHOLDER_CONF =, and then we'll just have an empty object for now. This setting is going to allow us to configure each placeholder throughout the content. Start by putting the name of the placeholder you wish to configure. In this case, we're going to be doing the daily_specials placeholder. We're going to put in a name for our content placeholder, and we'll just call it Daily Specials, and then we're going to assign what plugins can go into this content placeholder. In this case, we only want our daily specials plugin to go in there, so we'll put that in an array, and then we can add a customized label to that plugin, so we'll put in Daily_Specials_Plugin, and then we'll put Add Daily Special. So when we go to add a daily special to this content placeholder, we'll see the words Add Daily Special instead of just the name of the plugin. Save the settings file, and then go back to the browser and refresh. Click the plus button again. The only available plugin should be our Daily Specials plugin. Now Rafael won't have to worry about adding the wrong kind of content because there's only one plugin to choose from. Not only can we restrict the type of plugins that can go in our placeholders, but we can also restrict the number of plugins. Let's put a restriction on the Daily Specials placeholder to only allow one daily special at a time. Head back over to the settings file, and let's add another key to the CMS_PLACEHOLDER_CONF dictionary. This one is called limits. The value for limits is another dictionary. Inside the limits dictionary, put in a key of global. Set the value of this key to 1. This makes it so that there can be only one of every kind of plugin this placeholder allows. Since we're only allowing one kind of plugin, our Daily Specials plugin, this will only allow one instance of our plugin. If you have multiple plugins and you would like to set a different limit for each of them, list the name of the plugin as the key, and then the number of instances of the plugin you wish to allow as the value. Save the Settings file again, and switch on over to your browser. Refresh, then try to add another instance of the Daily Specials Plugin. With our global limit of 1 set, we are no longer able to add multiple instances of the Daily Specials plugin. This will once again help guide Rafael when he goes to edit the content on his site.

Building a Second Plugin

Now that you know how to make plugins and configure the placeholders they go in, go ahead and create a plugin for the menu items. We'll head back over to the models file in the rafs_pizza_plugins app, and we'll add a class of Menu_Item. Let's make the fields for this plugin, name, and then an image, and we'll also do a price, a description, and a URL, and we'll put in our Unicode function so that we have our user-friendly name when we view these in a list, and we'll switch over to our cms_plugins file. Let's just copy the plugin from above. We'll rename it to be Menu_Item_Plugin. We'll change this information here to be Menu_Item, and the name to be Menu Item, and the render_template to be menu_item. html. And then the only thing that we need to do to our context. update is to add price because that's the only different field, and we need to not forget to register our plugin as well. All right, now that we've got that let's go ahead and create our menu_item. html, and we'll once again just do some basic HTML. We'll make it pretty similar to these Daily Specials. We'll put the price in a p tag, and then we'll go over to the settings file and we'll configure the content placeholder for the menu items. So we're going to do menu_items as the key, the name will be Menu Items, plugins that we allow will be the Menu_Item_Plugin, and we'll put in a plugin label, Menu_Item_Plugin, Add Menu Item. All right, go ahead and save that. Notice we didn't add a limit to the number of menu items because we want to be able to add as many as Rafael has available in his restaurant. All right, going back to our browser, we should now be able to only add the menu items plugins to the menu item placeholder. Go ahead and create a few instances. I'm going to create one for Pepperoni Pizza, Sausage and Black Olive, and BBQ Chicken, because even though it's the Daily Special, it should still appear on the menu.

Styling Plugins

Now that we've created a few menu instances, we can do some more styling. We're still going to keep it pretty basic since this course isn't about CSS, but I just wanted to demonstrate how to further integrate your plugins into your Django templates. Let's start with the daily specials plugin. Let's put a div around and give it an id of daily_special. Let's make the background of that div a lighter color of some sort to contrast with the dark red background. We want the daily special to pop, so let's add a subtle drop shadow. Nothing too in your face, just enough to make it seem like it's floating slightly. Now, let's line everything up so it's not all stacked on top of each other. Let's go back to our daily special plugin template and add some classes to our various elements. Now, we can use these classes to line everything up. (Working) That's looking a bit better. We obviously would want to do some more fine tuning when putting this into production, but now you know how you can make changes to both the plugin template and the page template to style your content any way that you want. Once again, since this is not a CSS course, we're not going to worry about styling the menu items right now. I'm just going to throw in some styles behind the scenes. There, Rafael's website is really beginning to take shape. With our custom plugins being set up to be the only content that could be added to our placeholders, we've created a system that will give Rafael much more confidence when he goes to update it.

Summary

Here's a quick summary of what we've learned in this module. We created some custom plugins for the specialized content of the site. We restricted our content placeholders to only allow our custom plugins. We gave our plugins structure in both the page template and the plugin templates. And we added just a little bit more styling. In the next module, we'll be learning how to hook traditional Django apps into Django CMS.

Apphooks

Introduction

We've got the basics of Django CMS pretty much covered. But what if you want to use an existing Django app within Django CMS? Django CMS include something called apphooks to let you incorporate traditionally built Django apps. In this module, I'll show you how to hook in an existing blog app, and then link that app's functionality into the Django CMS toolbar.

Build a Django App

First, we'll need an app that we can hook into the Django CMS system. Let's actually build a very basic blog app that we can work with. We'll start by going to our terminal and entering in. /manage. py startapp blog. This command should be run in the root directory of your project. Next, we'll head on over to our code editor. Open up the models for the new app that you created. Start a class called post. And we're going to add some of the basic fields for a blog post, title, date, category, and content. The title will make a CharField. The date will make a DateField. The category is going to be a ForeignKey field to the category model, and we'll write that model in a minute. And the content will be a TextField. Let's add the unicode functions so our post instances will have user-friendly titles when we review them in the admin. Next, let's create the category model. We'll go really simple and just add a title field, which will be a CharField. We'll also add a unicode function to make these user friendly in the admin as well. Now that we've got our models written out, we need to go to Settings and add our blog app to the list of installed apps. Once that's done, we can set up the database migrations. Go back to your terminal and type in manage. py makemigrations blog. Once that runs through, type in manage. py migrate. This will set up the blog tables for us in our database. We'll want our models to show up in the admin, so we'll need to modify the admin file in the blog app. We're not doing anything too fancy with these, so we'll just do the basic admin registration. Admin. site. register Post, and admin. site. register Category. Now onto views. Once again, we're keeping this basic. Let's start with the view for posts. Def post, and then request, and p_id. The p_id we're passing in will come from the URL, which we'll set up next. Create a context variable and set it to be an empty dictionary. Next, create a post variable, and we'll create a query set to grab the post that matches the p_id. Post equals post. objects. get pk=p_id. Then we'll add the object to the context dictionary. Context and post=post. The last thing for this view is to return the render function. Return render, and then put in the request and post. html for our template, which we'll create in a minute, and then the context. We'll do the same basic thing for our category view. Def category and request c_id. Make the context an empty object. Category will be Category. objects. get and the pk=c_id. And we'll add the category to the context object, and then we'll grab the posts that match this category, Post. objects. filter category=category, and then we'll add the posts to the context. And we'll return our render function with request category. html and context. We should probably have a view that lists all the blog posts. So let's create another view called blog. We'll do def blog just request, because we're not grabbing a specific post or category. Context will equal an empty object. The posts will be Posts. objects. all. Add those to the context object, and then return the render function, and we'll call this template blog. html. All right, let's set up those URLs. Create a urls. py file in the blog app. At the top, put in from django. conf. urls import url. We'll also need to import the views. So we'll do from. and then import views. Then we'll create the URL patterns list. As you can see, we're putting cid and pid inside our regular expressions so that we can grab those IDs, and pass those to the view. All right, and then last but not least, we've got to have some templates for our three views. So we'll create blog. html, extend from the base. html file, and put in a content block. We'll do a for loop that'll loop through our posts object. So we'll do for post in posts, put in an anchor tag, and the URL will be based off of that post. id, and we'll put this around an h2 tag. We'll put the posts title inside that, and then we'll do the date and the content for that post just inside p tags. All right, and then we'll next create the category page. This will be very similar to the blog page, but on this case, we'll want the category name be displayed, so we'll put that in h3. And then we'll do the same looping for our posts as we did on our blog page. And then when you create a detail page for our posts or a single view for our posts, so we'll do post. html. We'll just stick the post title inside an h2 tag, and then we'll output its date and content in p tags. All right, our really simple blog app is complete. Now is the point where we get to hook it into the Django CMS system.

Create the Apphook

We're going to create a file inside the blog app called cms_app. py. The imports that you'll need to put at the top are from cms. app_base import CMSApp and from cms. apphook_pool import apphook_pool. All right, let's next create a class with the following content. We'll call this class BlogApp and this one here from CMSApp. We'll call the apphook Blog. Urls will be blog. urls. And then the app_name is blog. And then we need to register our apphook by using apphook_pool. register, and in parenthesis, the name of our class, which is BlogApp. All right, this is all we need to make our app available to be hooked into Django CMS. So this is the kind of change that requires us to restart our development server. So we'll switch over to our terminal, hit Control + C. Then manage runserver again. All right, then head back over to your browser and create a new page in the Django CMS admin. Let's call this page Blog, and that hit Save and continue editing. Click Advanced Settings, and then go to the place where it says Application. Clicking on the drop-down should give the option of selecting our BlogApp. Hit Save and then publish the page. Now, all the URLs of our blog app will appear after /blog. To demonstrate this, let's quickly create some blog categories and posts. I'll just create a general category, and then I'll create a post called New Post. I'll assign it to the General Category. I'll add some content, just Hello World. I'll hit Save and add another. We'll just call this one Another New Post, set it to the General Category, and we'll add some basic content. Hello world again. Alright, two should be enough to illustrate the principle. Now let's navigate to the /blog. Here we're going to see the list of all our blog posts. And clicking on the title will take us to the detail view. And then going to blog/category the ID of the general category, which in my case looks like it's three, will show the list of posts in the General Category, which right now is both of them. This is just the first of several things you can do to integrate regular Django apps in Django CMS. The next one we'll look at is integrating our Django app into the Django CMS toolbar.

Extend the Django CMS Toolbar

For this, we'll need to create another file in our blog app. This one will be called cms_toolbars. py. To that file, we're going to add the following content. We're going to do from cms. toolbar_base import CMSToolbar. And we'll do from cms. toolbar_pool import toolbar_pool. And from cms. toolbar. items import break SubMenu. We're going to start this by putting in a decorator. So we'll do @toolbar_pool. register, and we're going to create a class for posts. So we'll do PostToolbar. We'll inherit from CMSToolbar, and then we're going to do a function def populate, and then we'll include self in this to create a variable called admin_menu. This will be equal to self. toolbar. get_or_create_menu. And we'll call this the blog menu, and the verbose name will just be blog. All right, and then we'll create a position variable, which will be admin_menu. get_alphabetical_insert_position and posts. This is a SubMenu. Alright, and then post_menu is our next variable. This will be equal to admin_menu. get_or_create menu post menu, Posts position=position. All right, and then the url is going to be admin/blog/post. That's where this menu item is going to take us. All right, and then post_menu. addm_modal_item Edit Posts url=url. So to recap so far, what we've done is we've created a blog menu in our toolbar. To that blog menu, we're adding a post sub menu, and then that post sub menu has an entry called edit posts that will take us to the post editing area of the Django admin. All right, and then we want to be able to add a new post from our toolbar as well. So we're going to do url=/admin/blog/post/add, add this to our post_menu. So post_menu. add_modal_item Add New Post url=url. All right, and then we need to add a beak to our sub menus at this point so that we can then add another sub menu afterwards. So we'll say post_menu. add_break. All right, we're going to want to do something very similar for the categories. So let's just copy what we've done for posts, and then we'll just go through and change all the instances to posts at this point to categories. We'll just speed through this real fast. So we got those all set up now. In order to see our changes we've just made, we have to restart our development server again. All right, once you've got it going again, head on over to your browser and hit refresh. You should now see this Blog menu option in your toolbar. Clicking on it is going to show us our post and category sub menus with the edit and add new functions under each of those. So now you have direct access to your blog from anywhere on your site, via the Django CMS toolbar. At this point, we've met all the functional requirements for Rafael's website. There are places for him to edit his daily specials and menus. And as a bonus, a place for him to begin blogging if he wants to.

Summary

As always, let's recap what we've done in this module. We created a blog as a traditional Django app. We created an apphook that allowed us to hook our blog app into a Django CMS page. We then added direct access to our blog's features directly into the Django CMS toolbar. In the next and final module, we'll summarize what we've accomplished throughout the entirety of the course, as well as identify additional resources for leaning more advanced Django CMS features.

Summary

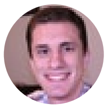
Course Summary

Congratulations! You've completed your first Django CMS project. As you've seen throughout this course, it's an easy framework to jump into especially if you already know how Django works. Let's review what we've learned from this course from start to finish. We discussed the benefits of using Django CMS over more widely-used systems, such as WordPress, with the biggest benefit being the creation of small, independently editable content areas. Learned how to get Django CMS up and running quickly by creating a virtual environment, installing the Django CMS installer tool, and using the installer tool to install all the dependencies and do the initial project setup. We became familiar with the Django CMS interface, both on the front-end, and in the Django admin area. Created our own templates with placeholders for our content. Created custom plugins to dictate what kind of content could be placed in our placeholder areas. We restricted the number of plugins that could be included in each placeholder. We then created a blog app using the traditional Django method of building apps, then hooked that app into a Django CMS page. Finally, we integrated the features of our blog app into the Django CMS toolbar.

What You Now Know

What you now know from going through this course are the core features of Django CMS. You have the ability to create fully functional Django CMS websites. Of course, there are more advanced features to learn as well but they only add to the core feature set. And while they will enhance the experience, are not necessary to provide an easy-to-use, fully-functioning website. As goes for any piece of software these days, Django CMS is a work in progress. It continues to evolve and get better with each new update. Throughout my time using Django CMS, I've seen small changes here and there to the core features. But for the most part, what you've learned during this course has largely stayed the same. You should be able to take what you've learned here and use it within future versions of Django CMS with little to no tweaking. If now that you've mastered the basics you would like to start digging into the more advanced features, you can visit Django CMS's documentation at docs. django-cms. org. Also too, the Django CMS installer tool we used has its own documentation which can be found at djangocms-installer. readthedocs. io. Thank you so much for taking the time to view this course and for your continued support of Pluralsight. Until the next one, farewell.

Course author



Brennan Davis

Brennan Davis is a front end web developer at Endurance International, where he leads a front end development team working on the Bluehost brand. He's been working professionally in the web...

Course info

Level	Intermediate
Rating	★★★★★ (24)
My rating	★★★★★
Duration	1h 2m
Released	15 Jun 2017

Share course