Code School: Try Django

by Sarah Holderness

Start Course

Bookmark Add to Channel Download Course

Getting Started

What Is Django?

Hitch up your wagon and mosey out west There's gold to find and knowledge to test We'll build an app for the treasure we seek Django's got the tools for our trek to the creek Prospect for riches with the model template view Route to your data and render HTML too Now pack up your gear it's time to make some dough You're fixing to strike gold as you try Django Hi, I'm Sarah Holderness and welcome to Try Django where we'll be learning about the Django Web framework and building an app along the way. in Level one section one, we'll be going over getting started with Django. Django is an open source MVC framework written in Python for building web applications. Django's website says it makes it easier to build better web apps more quickly and with less code, and it does this by handling common web development tasks right out of the box. Let's look at how Django works from a high level. Today, most web applications send a web request to the server for validation, processing and rendering, and then send HTML and assets back to the browser. Django is what's being run on the server and handles the validation, processing and rendering. If we look at this in a little more detail, we see that first, the user requests a URL, like instagram. com/codeschool. Then the request is sent to the Django app for processing. All of the user info is looked up and validated, and then data is passed off for rendering and HTML is sent back to the user. Django separates these duties into separate

components to make projects easier to maintain and collaborate on. Django has specific names for each one of these components. The part, which processes data is called the view. The part which stores and validates data is called the model, and the part that renders HTML is called the template. The model, template and view make up a Django app. If you're familiar with MVC or Model View Controller frameworks, then you can think of the view as the controller and the template as the view and MVC becomes MTV in Django. This might be a little confusing if you're used to MVC frameworks, but you'll get used to it. In this course, we'll be creating a site to keep track of our treasures found along our journey west. It'll have an image, the name, the material and the location found. If you're going to follow along with the challenges in this course, then you don't need to install Django. But if you like to play with Django on your own computer, you should install it first. If you have Python installed, it's as easy as typing Pip install Django in your terminal. You'll see some information about the progress of downloading and installing Django. Here's all of the different places we'll be writing code in this course. In the console, which uses a single caret; in the Django shell, which uses the three caret symbol; and in scripts inside your Django project. Before we create our Django project, it's important to know the difference between projects and apps in Django. For example, if we're going to create a project called Pinterest, inside we can have separate apps for the main. com, the blog and the jobs posting. In Django, each of the apps would live inside the outer Pinterest project but would have separate directories for each app. So before we start writing the actual code, we'll need to create a project and an app. We create our Django project with the command django-admin startproject, and the name of our project, Treasuregram. This will create a directory structure for our Django project. Under the folder Treasuregram, there will be several files created for us. Settings. py will hold our project settings, urls. py will hold our project's URLs, and manage. py is a utility for administrative tasks. Now we'll run our Django server to see our project running. To serve our site locally, we can type python manage. py runserver. Manage. py is in the top-level Treasuregram directory so you should navigate there to run this command. That output tells us our site is running at localhost port 8000. If we go there in our browser, we can see a congratulations message. It also tells us to start our first app by running python manage. py startapp. Now we'll create our first app by running python manage. py startapp and the name of our app, main app. This will create a directory for main app plus a bunch of files inside. We won't go into what each of these files does yet, but we'll focus on creating our first view in views. py. A view is simply a Python function that takes in a web request and returns a web response. For our first view, we want to keep it simple and just return a text response that says Hello Explorers! If we open up views. py, we can see there's a single import statement that imports render from Django shortcuts. Render is used to render templates which we'll do later, but for now we're just going to return a simple HttpResponse. To

do this, we'll need to import HttpResponse from django. http. This imports that specific class from a specific module. Now we can define our view function called index that takes in a request. This function will return an HttpResponse with an HTML string as the parameter. You might think you could go to your browser and load this view now, but there's actually one more step we need to do. We need to map this view to a URL. So let's say we want the URL /index to go to the index view we just wrote. Django has a URL dispatcher whose job is to map a URL to its corresponding view. So what this will look like is you will type the URL localhost port 8000/index. The URL dispatcher will map the URL to the corresponding view, and then the view will return the HttpResponse, Hello Explorers! The URL dispatcher lives inside the project's urls. py file. If we open that up, we see there's a URL patterns list and there's one URL in that list that goes to the admin. We won't need to touch that right now but we'll need to add a new URL for the index path. To do this, we'll create a URL object with two parameters. The first one is the path that we want to match, and the second one is the view function. This path might look kind of strange. It actually holds a regular expression that starts with r, to mean the string is raw and won't be escaped, and the caret means the start of the string. The second parameter just accesses the index function in the views file. And to access that view, we'll also need to import main app. views. If this pattern is matched, then the request will be sent to the view. This second parameter means go inside the main app, look inside that views. py file then call the index view function, which will send this to the index view function we wrote previously. Now if the user visits localhost/index, they'll be sent to main app's index view, which will return the HttpResponse. So if we preview that in the browser, we'll see the heading text Hello Explorers!

URL Dispatcher Best Practices

Welcome back to level one, section two where we'll be covering URL dispatcher best practices. Right now, localhost/index goes to our home page, but this is kind of an ugly URL. If you were going to go to Google. com, you wouldn't type Google. com/index so we'd like to get rid of that /index. First, we'll remove index/ from our regular expression and then send an empty path to load our view. Our pattern is now an empty string with just a caret. And if we load localhost, we see that our view loads without typing /index. In Django, it's a best practice to have a project URL dispatcher and an app URL dispatcher. Right now, our project's URL dispatcher is handling all of the requests. But we'd like to funnel requests for our main app to our main app' URL dispatcher, so the first thing we'll do is create that new urls. py file in the main app folder. So now in the project's urls. py file, instead of mapping to main app's view. index, we'll instead include all of main app's URLs. Notice to use that include function we need to import include from django. conf.

urls. So now we'll create our urls. py file in our main app. We'll still need to import URL, and we'll also want to import our views. If you want to import a specific module from your current app, you can leave off the package and just type. So now we want to create our app-specific URL patterns. We'll start with an empty string, but we'll also terminate it with a $ so that this doesn't match all strings. And then the second parameter will be our views index. Now if we load localhost with no path, we see our homepage. So things are working the same as they did before even though we refactored things behind the scenes. And this will allow us to stay more organized as our app grows.

Templates

The Template of Data

Hitch up your wagon and mosey out west There's gold to find and knowledge to test We'll build an app for the treasure we seek Django's got the tools for our trek to the creek Prospect for riches with the model, template, view Route to your data and render HTML too Now pack up your gear, it's time to make some dough You're fixing to strike gold as you Try Django Hi I'm Sarah Holderness and welcome back to Try Django, level two, section one, covering templates. If you remember, this is the app that we're trying to build, a list of our treasures. Now we're going to start building out the UI for this page and to do that, we'll need to use a template. Before we do that, let's look at our app flow diagram again. A request comes in, the URL dispatcher matches the correct view, and then the view should render a template. A template is just an HTML file that can have dynamic data. This is the part we'll be adding now. We need to create an HTML file since we already have the URL and view set up. To do this, we'll need a templates directory to store our templates. See, Django automatically looks for directories named templates for template files. But this is only if your app is registered with the Django project in settings, so we'll do that now. In the settings. py file, you'll see a list called installed_apps. We'll add our main_app to this list. You can see a few of the apps that come installed with Django already. Now we can get to creating our first template in a file called index. html. This contains just some basic HTML and the heading that displays TreasureGram. This template will render like this. Now we want to go into our index view and instead of returning an http response like we were doing before, we'll render our template. We can actually delete the import of HttpResponse and we'll return a render object that takes in a request and a string that contains the name of our template, index. html. If we load our page, we

can see the text TreasureGram and if we view the source of the page, we can see the HTML we just created. So, our template is working. We'd like to display a list of our treasures on our page, but to do that, we need dynamic data in our template. This is a two step process. First, we want to send data from our view to our template. And second, we want to access that data inside our template. Inside our view, we'll create some data, a variable name that holds the string gold nugget, a variable value equal to 1000, and we'll wrap these variables in a context, which is a dictionary that maps variable names to Python objects. Then we can pass this context as the third parameter of our render function. In the template, you can access a variable from the context by surrounding with double curly brackets, which is part of the Django template language. During the rendering of the template, all of the variables inside double curly brackets are replaced with their values. You can see in our rendered template that the values gold nugget and 1000 show up.

More Template Fun

Welcome back to level two, section two, where we'll be covering more on templates. We previously used a context dictionary to send two values to the template. This worked fine for that, but what if we have more complex data, like name, value, material, and location associated with a treasure or a bunch of treasures? How would we send that to the template in a context? We can use a class to store information about a treasure object and then use a list to store all of the treasure objects, and we can name that treasures. Then we have just one variable to put in our context, treasures. When we create our treasure class, we'll write class and then the name of the class, here it's treasure. Then we'll define a method called a nit which sets all of the attributes that we pass in. Then we can create a treasure object and set its attributes with just one line of code. t1 here is a gold nugget treasure and t2 is a fool's gold treasure, and they each are separate instances of the treasure class. We'll then move our treasure class to the bottom of views. py and then instead of creating individual treasure objects, we'll create a list called treasures to hold a few treasure objects. Then we can pass this treasures list in our context. Also notice, we put the context directly in line instead of creating a separate context variable first. How do we then display HTML for each object in the treasures list? If this was a regular Python script, we could write a for loop. But, we're inside a template, so we use the Django template language and its corresponding tags that start with a curly bracket percent and then we write our for loop and end with percent curly bracket. We can then close the for block with an endfor tag, which tells Django where the for block ends since we don't have indentation like we would in a regular Python script. You can see here we've added this for loop inside of our index. html template. Now when we view our page, we can see each treasure object's name and value displayed. Let's make another

change. You can see we have a zero for fool's gold and we want to replace zero with unknown, since that means we really don't know what the value of that thing is. To do this, we'll add a conditional which looks similar to Python except we have the curly bracket percent symbol wrapped around the conditional and we also add the endif tag to end the condition. Inside the first if statement, we'll check if the value's greater than zero. If so, we'll display the value. Otherwise, we'll display the text unknown. If we look at our page again, we can see fool's gold is unknown. That's because its value was not greater than zero.

Styling Templates

If we look at where we left off, we have the basic building blocks for our final page, but we'll need to add some style to get to our end goal. We'll start by adding a CSS file. Like with templates, Django automatically looks for static files in directories named static in your app. We'll create a directory called static and then create our style. css file in there. For now, it will just turn the heading green. Then in our template, we'll need to use the tag load staticfiles to make our static files available to the template. We'll also need to link to our style sheet. So inside the link tags href attribute, we'll add the static tag that tells Django where our style. css file is located. If we look at our rendered template, we can see our H1 is now green so our style. css is working. We can also add a CSS framework like Bootstrap to do some style work for us. We'll make sure the minified bootstrap file is in our static folder and then we'll link to it just like we did with style. css. We won't cover this in the slides, but we added a few Bootstrap customizations to our style. css file. We'll also add a static/images directory to hold some static images to style our site, like our logo. Now we'll start adding Bootstrap elements to our template like a nav bar that displays our static logo image, and a container and some panels to organize the page better. If we look at our page, it's looking way better, but we want to add a table to organize these attributes a little better. We'll add a table with some icons for the material location and value. You can see this page is getting a little complex but at its core, it's just HTML tags and our Django template tags loading dynamic data. Now our site is starting to look really nice, but it would look even better if we could add images for each treasure object. Since this is a new piece of data, we'll update our treasure class to add the image URL attribute. Then when we create our treasures, we'll also add the image URLs. Notice that these images aren't in our static folder. This is because these images aren't static and are hosted elsewhere. Now we'll just reference the treasures image URL inside an image tag. It's been a long journey, but we finally made it. We have a styled site with dynamic data and our object's images that matches our designer's mock up.

Models

The Missing Model

Hitch up your wagon and mosey out west There's gold to find And knowledge to test We'll build an app for the treasure we seek Django's got the tools for our trek to the creek Prospect for riches With a model template view Route to your data And render HTML too Now pack up your gear It's time to make some dough You're fixing to strike gold As you try Django Hi, I'm Sarah Holderness and welcome back to Try Django, level three, section one covering models. We've seen the URL to view to template process before but now we want to add in our model to organize and provide our view with data. Right now we're storing our treasures in a list in views. py but there's not a good interface for adding new ones directly from the website and we want something that's easy to maintain over time. A model is how you would do this in Django. So, we'll replace our treasure class with a treasure model. We'll create the treasure model in models. py following these three steps. First, we'll import models from Django. db, then we'll create a class called treasure that inherits from models. model so that Django knows we're creating a model. Then we'll create our same attributes we had before but using special model types that correspond to database types. If we compare our old treasure class to our new treasure model you can see a few differences but overall the structure's the same. Notice we didn't need to create a constructor in our treasure model because the model has one built in where you use the attribute names to set each one like so. We also use special Django field types that correspond to Python types and SQL types. For instance, models. charField corresponds to a Python string and an SQL varchar. And models. integerField corresponds to a Python int or an SQL integer and so on. There are many more Django model field types that you can find in the docs here. In our model we'll also need to define some rules for our attributes otherwise we would get errors like these. We'll set the max_length of our charFields and the max_digits and decimal_places for our decimal field. Our model now describes how we want to store our data in the database but do we actually have to write any SQL? No, actually we don't. That's covered by the Django ORM. The ORM will usually work automatically behind the scenes translating Python commands to SQL commands unless you edit the structure of your model. Then you'll need to do an extra step called a migration and we'll need to do that now since we just created our model. How do we then perform a migration? We actually have to run two commands in the console. The first one is python manage. py makemigrations which makes a migration file, then to apply the migration to the database we run python manage. py migrate. These separate steps let you review the

migration before you actually run migrate and since migrations are kind of like a version control system for your database that can come in handy. Let's actually run those commands on our project. We'll run python manage. py makemigrations and we can see this generates 0001_initial. py which is our migration file and then we can see a list of the things that that migration file does and ours just creates the model treasure. Before running migrate we can actually preview the SQL commands that will be run by doing an intermediate step. We can type this command and then see all the SQL that will get run as a preview. You don't need to do this but it can come in handy if you want to see if your Python code is doing what you expect. Now let's run migrate for real on our project. The output shows us all the migrations that are run and you can see there are a few we didn't create that are related to some other Django components. The last one main_app is the one we just created and is getting run now. Now, if you're wondering if your project is up to date with all of its migrations you can try running makemigrations and it'll say no changes detected if you're up to date and then if you run migrate it'll say no migrations to apply. Now that we've done our migration we can actually use our treasure model and create treasure objects and we'll do this in the Django interactive shell. We'll run python manage. py shell to open the Django shell. The Django shell is a great way to play around with creating models and writing queries before you write actual Python code. To start, we'll need to import our treasure model from main_app. models. Then we'll type our first query, treasure. objects. all and we can see we get an empty list since we haven't created any treasure objects yet. We did that treasure. objects. all which is a Django query set. A query set is like a select statement in SQL. All query sets start with the model name. objects, so here it would be treasure. objects and that's like a select statement in SQL. To retrieve all objects in SQL, we would write select star from treasure. As a query set we would write treasure. objects. all. We can also use filters like SQL where or limit by doing treasure. objects. filter and then the filter we want to create in parentheses like location equals Orlando, Florida. If you know there's only one element matching your query you can use get. Treasure. objects. get pk equals one is an example. PK is the primary key and we know it will be unique, so this will return only the treasure object with the matching primary key. Back in our shell we'll create our first treasure object with our model. Then if we use the query treasure. obejcts. all we see an empty list. That's because we forgot to save our treasure object. We can do that with t. save. Now if we repeat the query treasure. objects. all we'll see we have one treasure object in our list but this description treasure object isn't very specific. We can make this more specific in our treasure model. In our treasure model we'll add a method called __srt which will define a description for our treasure model and we'll just have it return the name of the treasure. Now the treasure's name will be shown whenever we output that object in the interactive shell. Let's create a couple more treasure objects. Gold Nugget and Fool's Gold and remember, after each one we

have to save. Then if we query treasure. objects. all again we see all the names of the objects we just created. Great, now these objects are in our database so we'll need to update our view to pull this data from our database and then pass it to the template for rendering. Since the way we interact with database objects is through our model, we'll need to import our treasure model into our view. Then inside of the index function we can call treasure. objects. all and store it in a variable called treasures. Since our template was already set up to handle a list of treasure objects we actually don't have to edit our template. We can see our rendered page and it already loads all of our treasure objects from our model.

The Admin

Welcome back to level three, section two where we'll be introducing the Django admin. One of the most powerful parts of Django is the automatic admin interface located at localhost/admin. It reads metadata from your models to provide a quick interface for viewing and editing them. This is where trusted users that are marked as staff can log in and manage content on your site. For instance, you can view the objects in your model, edit them, and create new ones. To use the admin site we need to create a super user. To do that we run the command python manage. py createsuperuser then enter your username, email and password. Now in our browser we can go to localhost/admin and log in. When we log in we can see our automatically generated groups and users but not our treasure model. We actually have to register our models with the admin for them to show up. In order to see our models in the admin, we'll need to register them in admin. py. We do this by importing our treasure model and then registering treasure with the command admin. site. register. Now when we log in we can see our treasure model. We can see all of the objects inside of our model and we can add more treasures using the form like an arrow head and a horse shoe. If we demo our app again, we have the new items we just added, the arrow head and horse shoe.

Course author

Sarah Holderness

Sarah just finished her PhD in Computer Science and in that process found two loves: inventing things and teaching. Working at Pluralsight allows her to do what she loves and work with some amazing...

Course info

Level	Beginner
Rating	★★★★★ (45)
My rating	★★★★★
Duration	0h 36m
Released	3 Aug 2016

Share course