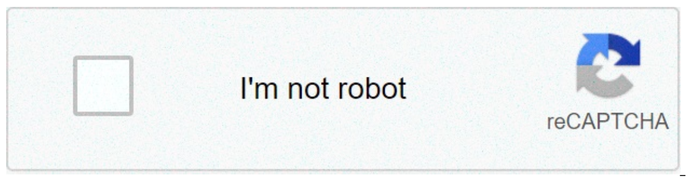

Continue

Django complete tutorial pdf

Django Overview Download Documentation News Community Code Issues About ♥ Donate Page 2 Django Overview Download Documentation News Community Code Issues About ♥ Donate Django is a fully featured Python web framework that can be used to build complex web applications. In this tutorial, you'll jump in and learn Django by example. You'll follow the steps to create a fully functioning web application and, along the way, learn some of the most important features of the framework and how they work together.In later posts in this series, you'll see how to build more complex websites using even more of Django's features than you'll cover in this tutorial.By the end of this tutorial, you will be able to: There are endless web development frameworks out there, so why should you learn Django over any of the others? First of all, it's written in Python, one of the most readable and beginner-friendly programming languages out there. Note: This tutorial assumes an intermediate knowledge of the Python language. If you're new to programming with Python, check out some of our beginner tutorials or the introductory course. The second reason you should learn Django is the scope of its features. If you need to build a website, you don't need to rely on any external libraries or packages if you choose Django. This means that you don't need to learn how to use anything else, and the syntax is seamless as you're using only one framework. There's also the added benefit that you don't need to worry that updating one library or framework will render others that you've installed useless. If you do find yourself needing to add extra features, there are a range of external libraries that you can use to enhance your site. One of the great things about the Django framework is its in-depth documentation. It has detailed documentation on every aspect of Django and also has great examples and even a tutorial to get you started. There's also a fantastic community of Django developers, so if you get stuck there's almost always a way forward by either checking the docs or asking the community. Django is a high-level web application framework with loads of features. It's great for anyone new to web development due to its fantastic documentation, and particularly if you're also familiar with Python. A Django website consists of a single project that is split into separate apps. The idea is that each app handles a self-contained function that the site needs to perform. As an example, imagine an application like Instagram. There are several different functions that need to be performed: User management: Login, logout, register, and so on The image feed: Uploading, editing, and displaying images Private messaging: Private messages between users and notifications These are each separate pieces of functionality, so if this were a Django site, then each piece of functionality should be a different Django app inside a single Django project. The Django project holds some configurations that apply to the project as a whole, such as project settings, URLs, shared templates and static files. Each application can have its own database and has its own functions to control how the data is displayed to the user in HTML templates. Each application also has its own URLs as well as its own HTML templates and static files, such as JavaScript and CSS. Django apps are structured so that there is a separation of logic. It supports the Model-View-Controller Pattern, which is the architecture on which most web frameworks are built. The basic principle is that in each application there are three separate files that handle the three main pieces of logic separately: Model defines the data structure. This is usually a database and is the base layer to an application. View displays some or all of the data to the user with HTML and CSS. Controller handles how the database and the view interact. If you want to learn more about the MVC pattern, then check out Model-View-Controller (MVC) Explained – With Legos. In Django, the architecture is slightly different. Although based upon the MVC pattern, Django handles the controller part itself. There's no need to define how the database and views interact. It's all done for you! The pattern Django utilizes is called the Model-View-Template (MVT) pattern. The view and template in the MVT pattern make up the view in the MVC pattern. All you need to do is add some URL configurations to map the views to, and Django handles the rest! A Django site starts off as a project and is built up with a number of applications that each handle separate functionality. Each app follows the Model-View-Template pattern. Now that you're familiar with the structure of a Django site, let's have a look at what you're going to build! Before you get started with any web development project, it's a good idea to come up with a plan of what you're going to build. In this tutorial, we are going to build an application with the following features: A fully functioning blog: If you're looking to demonstrate your coding ability, a blog is a great way to do that. In this application, you will be able to create, update, and delete blog posts. Posts will have categories that can be used to sort them. Finally, users will be able to leave comments on posts. A portfolio of your work: You can showcase previous web development projects here. You'll build a gallery style page with clickable links to projects that you've completed. Note: Before you get started, you can pull down the source code and follow along with the tutorial. If you prefer to follow along by writing the code yourself, don't worry. I've referenced the relevant parts of the source code throughout so you can refer back to it. We won't be using any external Python libraries in this tutorial. One of the great things about Django is that it has so many features that you don't need to rely on external libraries. However, we will add Bootstrap 4 styling in the templates. By building these two apps, you'll learn the basics of Django models, view functions, forms, templates, and the Django admin page. With knowledge of these features, you'll be able to go away and build loads more applications. You'll also have the tools to learn even more and build sophisticated Django sites. Now that you know the structure of a Django application, and what you are about to build, we're going to go through the process of creating an application in Django. You'll extend this later into your personal portfolio application. Whenever you are starting a new web development project, it's a good idea to first set up your development environment. Create a new directory for your project to live in, and cd into it: $ mkdir rp-portfolio $ cd rp-portfolio Once your inside the main directory, it's a good idea to create a virtual environment to manage dependencies. There are many different ways to set up virtual environments, but here you're going to use venv: This command will create a folder venv in your working directory. Inside this directory, you'll find several files including a copy of the Python standard library. Later, when you install new dependencies, they will also be stored in this directory. Next, you need to activate the virtual environment by running the following command: $ source venv/bin/activate Note: If you're not using bash shell, you might need to use a different command to activate your virtual environment. For example, on windows you need this command: C:\> venv\Scripts\activate.bat You'll know that your virtual environment has been activated, because your console prompt in the terminal will change. It should look something like this: Note: Your virtual environment directory doesn't have to be called venv. If you want to create one under a different name, for example my_venv, just replace with the second venv with my_venv. Then, when activating your virtual environment, replace venv with my_venv again. The prompt will also now be prefixed with (my_venv). Now that you've created a virtual environment, it's time to install Django. You can do this using pip: (venv) $ pip install Django Once you've set up the virtual environment and installed Django, you can now dive in to creating the application. As you saw in the previous section, a Django web application is made up of a project and its constituent apps. Making sure you're in the rp_portfolio directory, and you've activated your virtual environment, run the following command to create the project: $ django-admin startproject personal_portfolio This will create a new directory personal_portfolio. If you cd into this new directory you'll see another directory called personal_portfolio and a file called manage.py. Your directory structure should look something like this: rp-portfolio/ │ ├── personal_portfolio/ │ ├── personal_portfolio/ │ │ ├── __init__.py │ │ ├── settings.py │ │ ├── urls.py │ │ └── wsgi.py │ │ └── manage.py │ └── venv/ Most of the work you do will be in that first personal_portfolio directory. To save having to cd through several directories each time you come to work on your project, it can be helpful to reorder this slightly by moving all the files up a directory. While you're in the rp-portfolio directory, run the following commands: $ mv personal_portfolio/manage.py ./ $ mv personal_portfolio/personal_portfolio/* personal_portfolio $ rm -r personal_portfolio/personal_portfolio/ You should end up with something like this: rp-portfolio/ │ ├── personal_portfolio/ │ ├── __init__.py │ ├── settings.py │ ├── urls.py │ └── wsgi.py │ ├── venv/ │ └── manage.py Once your file structure is set up, you can now start the server and check that your set up was successful. In the console, run the following command: $ python manage.py runserver Then, in your browser go to localhost:8000, and you should see the following: Congratulations, you've created a Django site! The source code for this part of the tutorial can be found on GitHub. The next step is to create apps so that you can add views and functionality to your site. For this part of the tutorial, we'll create an app called hello_world, which you'll subsequently delete as its not necessary for our personal portfolio site. To create the app, run the following command: $ python manage.py startapp hello_world This will create another directory called hello_world with several files: __init__.py tells Python to treat the directory as a Python package. admin.py contains settings for the Django admin pages. apps.py contains settings for the application configuration. models.py contains a series of classes that Django's ORM converts to database tables. tests.py contains test classes. views.py contains functions and classes that handle what data is displayed in the HTML templates. Once you've created the app, you need to install it in your project. In rp-portfolio/settings.py, add the following line of code under INSTALLED_APPS: INSTALLED_APPS = ['django.contrib.admin', 'django.contrib.auth', 'django.contrib.contenttypes', 'django.contrib.sessions', 'django.contrib.messages', 'django.contrib.staticfiles', 'hello_world',] That line of code means that your project now knows that the app you just created exists. The next step is to create a view so that you can display something to a user. Views in Django are a collection of functions or classes inside the views.py file in your app directory. Each function or class handles the logic that gets processed each time a different URL is visited. Navigate to the views.py file in the hello_world directory. There's already a line of code in there that imports render(). Add the following code: from django.shortcuts import render def hello_world(request): return render(request, 'hello_world.html', {}) In this piece of code, you've defined a view function called hello_world(). When this function is called, it will render an HTML file called hello_world.html. That file doesn't exist yet, but we'll create it soon. The view function takes one argument, request. This object is an HttpRequestObject that is created whenever a page is loaded. It contains information about the request, such as the method, which can take several values including GET and POST. Now that you've created the view function, you need to create the HTML template to display to the user. render() looks for HTML templates inside a directory called templates inside your app directory. Create that directory and subsequently a file named hello_world.html inside it: $ mkdir hello_world/templates/ $ touch hello_world/templates/hello_world.html Add the following lines of HTML to your file: You've now created a function to handle your views and templates to display to the user. The final step is to hook up your URLs so that you can visit the page you've just created. Your project has a module called urls.py in which you need to include a URL configuration for the hello_world app. Inside personal_portfolio/urls.py, add the following: from django.contrib import admin from django.urls import path, include urlpatterns = [path('admin/', admin.site.urls), path('', include('hello_world.urls')),] This looks for a module called urls.py inside the hello_world application and registers any URLs defined there. Whenever you visit the root path of your URL (localhost:8000), the hello_world application's URLs will be registered. The hello_world.urls module doesn't exist yet, so you'll need to create it: $ touch hello_world/urls.py Inside this module, we need to import the path object as well as our app's views module. Then we want to create a list of URL patterns that correspond to the various view functions. At the moment, we have only created one view function, so we need only create one URL: from django.urls import path from hello_world import views urlpatterns = [path('', views.hello_world, name='hello_world'),] Now, when you restart the server and visit localhost:8000, you should be able to see the HTML template you created: Congratulations, again! You've created your first Django app and hooked it up to your project. Don't forget to check out the source code for this section and the previous one. The only problem now is that it doesn't look very nice. In the next section, we're going to add bootstrap styles to your project to make it prettier! If you don't add any styling, then the app you create isn't going to look too nice. Instead of going into CSS styling with this tutorial, we'll just cover how to add bootstrap styles to your project. This will allow us to improve the look of the site without too much effort. Before we get started with the Bootstrap styles, we'll create a base template that we can import to each subsequent view. This template is where we'll subsequently add the Bootstrap style imports. Create another directory called templates, this time inside personal_portfolio, and a file called base.html, inside the new directory: $ mkdir personal_portfolio/templates/ $ touch personal_portfolio/templates/base.html We create this additional templates directory to store HTML templates that will be used in every Django app in the project. As you saw previously, each Django project can consist of multiple apps that handle separated logic, and each app contains its own templates directory to store HTML templates related to the application. This application structure works well for the back end logic, but we want our entire site to look consistent on the front end. Instead of having to import Bootstrap styles into every app, we can create a template or set of templates that are shared by all the apps. As long as Django knows to look for templates in this new, shared directory it can save a lot of repeated styles. Inside this new file (personal_portfolio/templates/base.html), add the following lines of code: {% block page_content %}{% endblock %} Now, in hello_world/templates/hello_world.html, we can extend this base template: {% extends "base.html" %} {% block page_content %} Hello, World! {% endblock %} What happens here is that any HTML inside the page_content block gets added inside the same block in base.html. To install Bootstrap in your app, you'll use the Bootstrap CDN. This is a really simple way to install Bootstrap that just involves adding a few lines of code to base.html. Check out the source code to see how to add the CDN links to your project. All future templates that we create will extend base.html so that we can include Bootstrap styling on every page without having to import the styles again. Before we can see our new styled application, we need to tell our Django project that base.html exists. The default settings register template directories in each app, but not in the project directory itself. In personal_portfolio/settings.py, update TEMPLATES: TEMPLATES = [{ "BACKEND": "django.template.backends.django.DjangoTemplates", "DIRS": ["personal_portfolio/templates/"], "APP_DIRS": True, "OPTIONS": { "context_processors": ["django.template.context_processors.debug", "django.template.context_processors.request", "django.contrib.auth.context_processors.auth", "django.contrib.messages.context_processors.messages",] }, }] Now, when you visit localhost:8000, you should see that the page has been formatted with slightly different styling: Whenever you want create templates or import scripts that you intend to use in all your Django apps inside a project, you can add them to this project-level directory and extend them inside your app templates. Adding templates is the last stage to building your Hello, World! Django site. You learned how the Django templating engine works and how to create project-level templates that can be shared by all the apps inside your Django project. In this section, you learned how to create a simple Hello, World! Django site by creating a project with a single app. In the next section, you'll create another application to showcase web development projects, and you'll learn all about models in Django! The source code for this section can be found on GitHub. Any web developer looking to create a portfolio needs a way to show off projects they have worked on. That's what you'll be building now. You'll create another Django app called projects that will hold a series of sample projects that will be displayed to the user. Users can click on projects and see more information about your work. Before we build the projects app, let's first delete the hello_world application. All you need to do is delete the hello_world directory and remove the line "hello_world", from INSTALLED_APPS in settings.py: INSTALLED_APPS = ['django.contrib.admin', 'django.contrib.auth', 'django.contrib.contenttypes', 'django.contrib.sessions', 'django.contrib.messages', 'django.contrib.staticfiles', 'hello_world', # Delete this line] Finally, you need to remove the URL path created in personal_portfolio/urls.py: from django.contrib import admin from django.urls import path, include urlpatterns = [path('admin/', admin.site.urls), path('', include('hello_world.urls')), # Delete this line] Now that you've removed the hello_world app, we can create the projects app. Making sure you're in the rp-portfolio directory, run the following command in your console: $ python manage.py startapp projects This will create a directory named projects. The files created are the same as those created when we set up the hello_world application. In order to hook up our app, we need to add it into INSTALLED_APPS in settings.py: INSTALLED_APPS = ['django.contrib.admin', 'django.contrib.auth', 'django.contrib.contenttypes', 'django.contrib.sessions', 'django.contrib.messages', 'django.contrib.staticfiles', 'projects',] Check out the source code for this section on GitHub. We're not going to worry about URLs for this application just yet. Instead, we're going to focus on building a Project model. If you want to store data to display on a website, then you'll need a database. Typically, if you want to create a database with tables and columns within those tables, you'll need to use SQL to manage the database. But when you use Django, you don't need to learn a new language because it has a built-in Object Relational Mapper (ORM). An ORM is a program that allows you to create classes that correspond to database tables. Class attributes correspond to columns, and instances of the classes correspond to rows in the database. So, instead of learning a whole new language to create our database and its tables, we can just write some Python classes. When you're using an ORM, the classes you build that represent database tables are referred to as models. In Django, they live in the models.py module of each Django app. In your projects app, you'll only need one table to store the different projects you'll display to the user. That means you'll only need to create one model in models.py. The model you'll create will be called Project and will have the following fields: title will be a short string field to hold the name of your project. description will be a larger string field to hold a longer piece of text. technology will be a string field, but its contents will be limited to a select number of choices. image will be an image field that holds the file path where the image is stored. To create this model, we'll create a new class in models.py and add the following in our fields: from django.db import models class Project(models.Model): title = models.CharField(max_length=100) description = models.TextField() technology = models.CharField(max_length=20) image = models.FilePathField(path="/img") Django models come with many built-in model field types. We've only used three in this model. CharField is used for short strings and specifies a maximum length. TextField is similar to CharField but can be used for longer form text as it doesn't have a maximum length limit. Finally, FilePathField also holds a string but must point to a file path name. Now that we've created our Project class, we need Django to create the database. By default, the Django ORM creates databases in SQLite, but you can use other databases that use the SQL language, such as PostgreSQL or MySQL, with the Django ORM. To start the process of creating our database, we need to create a migration. A migration is a file containing a Migration class with rules that tell Django what changes need to be made to the database. To create the migration, type the following command in the console, making sure you're in the rp-portfolio directory: $ python manage.py makemigrations projects Migrations for 'projects': projects/migrations/0001_initial.py - Create model Project You should see that a file projects/migrations/0001_initial.py has been created in the projects app. Check out that file in the source code to make sure your migration is correct. Now that you've create a migration file, you need to apply the migrations set out in the migrations file and create your database using the migrate command: $ python manage.py migrate projects Operations to perform: Apply all migrations: projects Running migrations: Applying projects.0001_initial... OK Note: When running both the makemigrations and migrate commands, we added projects to our command. This tells Django to only look at models and migrations in the projects app. Django comes with several models already created. If you run makemigrations and migrate without the projects flag, then all migrations for all the default models in your Django projects will be created and applied. This is not a problem, but for the purposes of this section, they are not needed. You should also see that a file called db.sqlite3 has been created in the root of your project. Now your database is set up and ready to go. You can now create rows in your table that are the various projects you want to show on your portfolio site. To create instances of our Project class, we're going to have to use the Django shell. The Django shell is similar to the Python shell but allows you to access the database and create entries. To access the Django shell, we use another Django management command: Once you've accessed the shell, you'll notice that the command prompt will change from $ to >>>. You can then import your models: >>>>>> from projects.models import Project We're first going to create a new project with the following attributes: name: My First Project description: A web development project. technology: Django image: img/project1.png To do this, we create an instance of the Project class in the Django shell: >>>>>> p1 = Project(... title='My First Project', ... description='A web development project.', ... technology='Django', ... image='img/project1.png' ...) >>> p1.save() This creates a new entry in your projects table and saves it to the database. Now you have created a project that you can display on your portfolio site. The final step in this section is to create two more sample projects: >>>>>> p2 = Project(... title='My Second Project', ... description='Another web development project.', ... technology='Flask', ... image='img/project2.png' ...) >>> p2.save() >>> p3 = Project(... title='My Third Project', ... description='A final development project.', ... technology='Django', ... image='img/project3.png' ...) >>> p3.save() Well done for reaching the end of this section! You now know how to create models in Django and build migration files so that you can translate these model classes into database tables. You've also used the Django shell to create three instances of your model class. In the next section, we'll take these three projects you created and create a view function to display them to users on a web page. You can find the source code for this section of the tutorial on GitHub. Now you've created the projects to display on your portfolio site, you'll need to create view functions to send the data from the database to the HTML templates. In the projects app, you'll create two different views: An index view that shows a snippet of information about each project A detail view that shows more information on a particular topic Let's start with the index view, as the logic is slightly simpler. Inside views.py, you'll need to import the Project class from models.py and create a function project_index() that renders a template called project_index.html. In the body of this function, you'll make a Django ORM query to select all objects in the Project table: 1from django.shortcuts import render 2from projects.models import Project 3 4def project_index(request): 5 projects = Project.objects.all() 6 context = { 7 'projects': projects 8 } 9 return render(request, 'project_index.html', context) There's quite a lot going on in this code block, so let's break it down. In line 5, you perform a query. A query is simply a command that allows you to create, retrieve, update, or delete objects (or rows) in your database. In this case, you're retrieving all objects in the projects table. A database query returns a collection of all objects that match the query, known as a Queryset. In this case, you want all objects in the table, so it will return a collection of all projects. In line 6 of the code block above, we define a dictionary context. The dictionary only has one entry projects to which we assign our Queryset containing all projects. The context dictionary is used to send information to our template. Every view function you create needs to have a context dictionary. In line 9, context is added as an argument to render(). Any entries in the context dictionary are available in the template, as long as the context argument is passed to render(). You'll need to create a context dictionary and pass it to render in each view function you create. We also render a template named project_index.html, which doesn't exist yet. Don't worry about that for now. You'll create the templates for these views in the next section. Next, you'll need to create the project_detail() view function. This function will need an additional argument: the id of the project that's being viewed. Otherwise, the logic is similar: 13def project_detail(request, pk): 14 project = Project.objects.get(pk=pk) 15 context = { 16 'project': project 17 } 18 return render(request, 'project_detail.html', context) In line 14, we perform another query. This query retrieves the project with primary key, pk, equal to that in the function argument. We then assign that project in our context dictionary, which we pass to render(). Again, there's a template project_detail.html, which we have yet to create. Once your view functions are created, we need to hook them up to URLs. We'll start by creating a file projects/urls.py to hold the URL configuration for the app. This file should contain the following code: 1from django.urls import path 2from . import views 3 4urlpatterns = [5 path("", views.project_index, name="project_index"), 6 path("/", views.project_detail, name="project_detail"), 7] In line 5, we hook up the root URL of our app to the project_index view. It is slightly more complicated to hook up the project_detail view. To do this, we want the URL to be /1, or /2, and so on, depending on the pk of the project. The pk value in the URL is the same pk passed to the view function, so you need to dynamically generate these URLs depending on which project you want to view. To do this, we used the notation. This just tells Django that the value passed in the URL is an integer, and its variable name is pk. With those now set up, we need to hook these URLs up to the project URLs. In personal_portfolio/urls.py, add the following highlighted line of code: from django.contrib import admin from django.urls import path, include urlpatterns = [path("admin/", admin.site.urls), path("projects/", include("projects.urls")),] This line of code includes all the URLs in the projects app but means they are accessed when prefixed by projects/. There are now two full URLs that can be accessed with our project: localhost:8000/projects: The project index page localhost:8000/projects/3: The detail view for the project with pk=3 These URLs still won't work properly because we don't have any HTML templates. But our views and logic are up and running so all that's left to do is create those templates. If you want to check your code, take a look at the source code for this section. Phew! You're nearly there with this app. Our final step is to create two templates: The project_index template The project_detail template As we've added Bootstrap styles to our application, we can use some pre-styled components to make the views look nice. Let's start with the project_index template. For the project_index template, you'll create a grid of Bootstrap cards, with each card displaying details of the project. Of course, we don't know how many projects there are going to be. In theory, there could be hundreds to display. We don't want to have to create 100 different Bootstrap cards and hard-code in all the information to each project. Instead, we're going to use a feature of the Django template engine: for loops. Using this feature, you'll be able to loop through all the projects and create a card for each one. The for loop syntax in the Django template engine is as follows: {% for project in projects %} {# Do something... #} {% endfor %} Now that you know how for loops work, you can add the following code to a file named projects/templates/project_index.html: 1{% extends "base.html" %} 2{% load static %} 3{% block page_content %} 4Projects 5 6{% for project in projects %} 7 8 9 10 11 {{ project.title }} 12 {{ project.description }} 13 15 Read More 16 17 18 19 20 {% endfor %} 21 22{% endblock %} There's a lot of Bootstrap HTML here, which is not the focus of this tutorial. Feel free to copy and paste and take a look at the Bootstrap docs if you're interested in learning more. Instead of focusing on the Bootstrap, there are a few things to highlight in this code block. In line 1, we extend base.html as we did in the Hello, World! app tutorial. I've added some more styling to this file to include a navigation bar and so that all the content is contained in a Bootstrap container. The changes to base.html can be seen in the source code on GitHub. On line 2, we include a {% load static %} tag to include static files such as images. Remember back in the section on Django models, when you created the Project model. One of its attributes was a filepath. That filepath is where we're going to store the actual images for each project. Django automatically registers static files stored in a directory named static/ in each application. Our image file path names were of the structure: img/.png. When loading static files, Django looks in the static/ directory for files matching a given filepath within static/. So, we need to create a directory named static/ with another directory named img/ inside. Inside img/, you can copy over the images from the source code on GitHub. On line 6, we begin the for loop, looping over all projects passed in by the context dictionary. Inside this for loop, we can access each individual project. To access the project's attributes, you can use dot notation inside double curly brackets. For example, to access the project's title, you use {{ project.title }}. The same notation can be used to access any of the project's attributes. On line 9, we include our project image. Inside the src attribute, we add the code {% static project.image %}. This tells Django to look inside the static files to find a file matching project.image. The final point that we need to highlight is the link on line 13. This is the link to our project_detail page. Accessing URLs in Django is similar to accessing static files. The code for the URL has the following form: {% url '' %} In this case, we are accessing a URL path named project_detail, which takes integer arguments corresponding to the pk number of the project. With all that in place, if you start the Django server and visit localhost:8000/projects, then you should see something like this: With the project_index.html template in place, it's time to create the project_detail.html template. The code for this template is below: {% extends "base.html" %} {% load static %} {% block page_content %} {{ project.title }} About the project: {{ project.description }} Technology used: {{ project.technology }} {% endblock %} The code in this template has the same functionality as each project card in the project_index.html template. The only difference is the introduction of some Bootstrap columns. If you visit localhost:8000/projects/1, you should see the detail page for that first project you created: In this section, you learned how to use models, views, and templates to create a fully functioning app for your personal portfolio project. Check out the source code for this section on GitHub. In the next section, you'll build a fully functioning blog for your site, and you'll also learn about the Django admin page and forms. Congratulations, you've reached the end of the tutorial! We've covered a lot, so make sure to keep practicing and building. The more you build the easier it will become and the less you'll have to refer back to this article or the documentation. You'll be building sophisticated web applications in no time. In this tutorial you've seen: How to create Django projects and apps How to add web pages with views and templates How to get user input with forms How to hook your views and templates up with URL configurations How to add data to your site using relational databases with Django's Object Relational Mapper How to use the Django Admin to manage your models In addition, you've learned about the MVT structure of Django web applications and why Django is such a good choice for web development. If you want to learn more about Django, do check out the documentation and make sure to check out Part 2 of this series! django complete tutorial pdf. django complete tutorial for beginners. python django complete tutorial. django rest framework complete tutorial. django complete project tutorial