Django Fundamentals

by Reindert-Jan Ekker

Introduction to Django

Introduction

Hi. This is Riendert-Jan Ekker and welcome to this course about the fundamentals of web development with Django. In this module I'll tell you a bit about what Django is and why you'd want to learn it and what you need to know to be able to follow this course. I'll also tell you what you can expect from this course.

Why Django?

Django is a framework for writing web applications. It's slogan is, The Web framework for perfectionists with deadlines, and it focuses on being very productive without writing lots of code and on making the code you have to write as clean and elegant as possible. By the way, in case you were wondering, Django was named after Django Reinhardt, a famous German jazz guitar player so it's got nothing to do with the Tarantino movie. So, why would you want to learn Django? Well, first of all, Django is Python and Python is awesome. Python is a powerful, beautiful programming language that gives great productivity with little code. If you were to ask me why I program in Python, I would answer that writing code in Python makes me happy as a developer.

It's really a language that makes me feel at home and Python makes programming fun again. And apart from that, it also makes me feel very productive and it produces programs that run fast. Now, since Django is written in Python, you get to experience all that fun and use all its power when building web apps. It will make you more productive with less code and like Python, it comes with batteries included. There's a huge standard library with lots of functionalities to solve just about any problem. And what's more, there are lots and lots of Django-specific packages to make your life as a web developer easier.

Batteries Included

But let's just look at what Django itself has to offer. First of all, Django comes with a powerful object relational mapping API, which allow you to write pure Python clauses that represent your database tables and that means you won't have to write SQL queries to interact with your database. Then when you have written the model classes that represent your data, Django can generate an admin interface and that's a web-based user interface for editing content and it's generated completely automatically by Django and that's a very powerful feature that can really, really speed up your development cycle. Now Django also comes with a simple, but powerful template language for generating your web pages and a very simple and beautiful way to configure the URLs for your site. It will also let you handle HTML forms with a minimal of code, or often, without any code at all. And in case there's something you're missing in the template language or the standard form clauses, or any other part of Django, there's a huge amount of third-party packages that will probably offer what you need. And all of that's just the tip of the iceberg. Some other features that Django offers are user authentication, HTTP session handling, a powerful, very high-performance caching system, and some very nice internationalization support.

Django Principles

But apart from all this functionality there's another reason to use Django, because it has been developed with some specific principles in mind. Simply put, it tries to make you more productive with less code and during this course you will see that that's exactly what it does. But at the same time, it enables you to be very productive while writing beautiful, well-designed code. The Django development team is very passionate about well-designed code and here are just some of the principles they follow like loose coupling and the don't-repeat-yourself principle, and there are actually many, many more, and all of that results in a framework that doesn't just allow you to write beautiful code, but it really encourages you to do so without ever getting in the way. Now if you're interested in this sort of thing, I really encourage you to follow the link I included here and read more about the design principles behind Django. Now one very nice side effect of all of this is that it makes web development very enjoyable, which is probably one of the most important reasons to start using Django. Now finally, I just want to say that the Django documentation tends to be very readable, complete, and correct and in case you can't find what you need, there's a large and very friendly community of Django users that will probably have the answers to all your questions. Now to take a look at the documentation or find out more about the Django community, visit the Django homepage at djangoproject. com.

What you Should Already Know

So, now you know why you should learn Django, but before we start, here's a short overview of things that you should already know before watching this course. First of all, Django is a Python framework, so you should be familiar with Python. At the very least, watch the introductory Python course and write some simple programs. Although Django does use some advanced concepts, we'll just stick to the basics so a general understanding of the core language will probably be enough to get you started. If you also know the basics of web development and that means you should at least know how to write a simple HTML page. It may be helpful if you also know a bit about TSS styling and JavaScript, although you should be able to follow everything even if you don't. We'll also touch on the HTTP protocol a bit, which is the protocol that enables to ask a browser to ask a server for a web page, but if you don't know about HTTP, don't worry, it's not hard at all, and I'll explain everything you need to know. Finally, it's also important that you know the basics of databases. We will use a database to store some of our applications data so we'll need to create, view and modify database tables, for example, so you should know the basics for SQL and database design.

Course Overview

So let me give you a short overview of what you can expect from this course. First, of course, I'll show you how to install Django and set up your development environment. We'll see how to start a new project and some projects called apps and the general layout of these. You'll also learn how to use Django scripts to manage these environments. I'll also have a look at the Settings file for your project. We'll see how to define model clauses and generate a database from that and how to use the model API to get data from and save data to the database. We'll also see how to work with relations between these models which correspond with database relations between tables. And we'll see how Django can generate an admin interface for your database. And of course, we'll learn about views and templates as wel. Now speaking of views, you'll learn how to map a URL to a view, how to handle HTTP requests, and generate responses, and how to use the more advanced features of forms and generic views. We'll also see how to write Django templates, which includes learning about the template syntax and the tags and filters you can use and how to actually pass any data into the template. Well so far, that's what I'd call the core of Django, but there's also some other topics we'll come across and those include how to unit test your code, how to put your web site into production, and how to use third-party packages in your Django projects. And that's basically it for this intro module. Let's go ahead with the next module and see how to install Django.

Installing Django

Intro

Hi, this is Reindert-Jan Ekker and welcome to this module in which I'll show you how to obtain and install Django. The basic installation procedure is the same whether you are using a Mac, Linux, or Windows machine. Needless to say, you need to start with installing Python. Of course, in case you already installed Python, you may want to skip this step, but if you are running Mac OS or Linux, which have a preinstalled version of Python, you probably want to upgrade, because chances are that your OS doesn't come with the latest version of Python. Now after making sure you have the right version of Python installed, the next step is install the pip package manager. Pip will make installing Django very easy and it will also make installing third-party packages very easy. Up to here the steps for installation depended on your platform and I will show you how to do this on Windows, Mac OS and Linux, but from here, after installing pip, everything is pretty much the same regardless of your OS. So the next step is to install a tool called virtualenv. Now this tool is not strictly necessary for installing Django. You might just skip this step and go straight ahead to the installation of Django. But as we'll see later in this module, that's not really a good idea in a real development environment and that's why I'll have to complicate things a bit and show you have to install virtualenv and create a virtual environment for your first Django projects. You'll find out what that means when we get there. And finally, after all these steps we will be ready to install Django and start working on our first web application.

Choosing your Versions

Now when installing Python in Django, you need to choose which versions you want to use. Do you want to use Python 2 or 3? And which Django version do you need? Well, the first option would be to simply take the newest version of both. At the time of recording of this course Django 1. 6 was just released so if you want to be at the cutting edge of technology you should use Django 1. 6 with Python 3. This combination is what I will use primarily for this course, but you should be aware that Django 1. 6 may still have some bugs. For a production environment it may be wiser to get Django 1. 5, which is the previous version, and which has been around for a while. For Python I'd still choose version 3 because that's been around for a while now too, so it's stable and fast and it's where the future of Python is, but there still are some packages that don't support Python 3 yet, although most important packages do. However, if you need some specific library that hasn't imported to Python 3 yet, you will need to install Python 2. Now it doesn't really matter what you choose for this course since the difference when writing a basic web site are quite small. I'll write my code for Python 3 and Django 1. 6 and point out any major differences with Django 1. 5 and Python 2, but make sure that whether you choose Python 2 or 3, I recommend you install the latest version of either. At the moment that would be Python 2. 7 or 3. 3. These will work with either Django 1. 5 or Django 1. 6, but if you want to know exactly which Django versions are compatible with which Python version, check out this link. I will now go on to show you how to install Python and pip in Windows and then I'll do the same for Mac OS and finally I'll you the procedure for Linux. Of course, you can skip right the part that applies to you, or if you already have the Python version you like with pip installed, you can skip to the part about virtualenv.

Installing Pip and Python on Windows

So let's start with the first two steps of our installation on Windows. To install Python I'd go straight to the source and download an MSI installer from python. org. By the way, it's also possible to download prepackaged Python distributions that include all kinds of packages. One specific example is called Anaconda and it comes with pip installed so in case you installed Anaconda or another distribution that includes pip, you can skip this part and go straight to installing virtualenv. After you have installed Python, we'll simply follow the instructions from pip-installer. org. That site will give you two scripts you can run, the first one will install something called setuptools, which we need to install pip. So, let me show you how this works in a little demo video.

Demo: Windows Installation

So let me show you how to install Python and pip on a clean Windows machine. Here you can see the MSI installer I just downloaded from python. org and let's simply run that. I'm not going to change the path, but if you want to, you can, of course. And I also like to add Python to my path. Very well. Let's wait until that finishes. Okay. So far, so good. So now we have a Python install and we can go to pip-installer. org as you can see here and follow the instructions and it says here, download ez_setup_py. That's exactly what I'm going to do. I'm going to save that to my downloads folder. Okay, go back to the site and download get-pip. py as well. Again, save that to my downloads folder. And now if I take a look at my downloads folder, there they are and I can simply double click them and remember to first double click ez_setup because that's a prerequisite for running the get-pip script. And after that finishes we can run get-pip script like this. And when I start to see a little ______ like this I can run my Python interpreter. Okay, that works. (typing) And let's see if we can also run pip now. That's in my Python 3. 3/scripts folder (typing). Very well, and pip is there as well. So from here we can go to the next step and install virtualenv.

Installing Pip and Python on Mac OS X

Let's see how we can install Python and pip on a Mac. Of course Mac OS comes with the Python interceptor installed, but that's usually an outdated version so an upgrade is in order, although you can download and install and stand-alone Python version like on Windows, I would actually recommend to use Homebrew, a package manager from Mac OS, which will make it real easy to install new Python versions as well as many different kinds of open-source software. Basically, it's similar to the package-management software you might know from Linux. After installing Homebrew, installing Python is as simple as typing a single command into the terminal, brew install python. If you want Python 3 and not Python 2, you should add a 3 at the end. Homebrew will take it from there and install pip as well so you won't need to do anything else. Let me show you what this looks like in practice.

Demo: OS X Installation

So to install Python and pip on Mac OS, we go to brew. sh, which is the home page of the Homebrew, a package manager for OS 10. Now if we scroll down, here's a command line, which we can copy, and then I open a terminal. Now in case you don't know how to open a terminal in Mac OS, let me show you. It's in the applications folder under utilities and then terminal. By the way, don't open console because that's a completely different application. Simply open terminal. And then you get a window like this and you can just paste the command line from Homebrew in there and it will ask you whether you want to install. It shows you what it's going to install and then when you press enter, that's what it does. Of course, first you have to enter your password (typing). Now also, in case you don't have Xcode installed yet, it asks you to do so and you can simply click install and it will do so automatically. Very well. So this installation was done and now I'm going to press a key here and the Homebrew installation is going to continue. And now we see that the installation is successful and that means we can go on to the next step and the thing I'm going to do right now is install Python and that's as simple as telling brew to install python and in this case I'm going to ask it to install Python 3. Now in case you want to install Python 2, simply leave out the 3. So I'm going to install Python 3, press enter, and again, wait until brew finishes, and finally, now brew finishes, we can see that it didn't just install Python, but it also installed setup tools and pip and that means that now we can use pip to install other stuff. For example, pip install virtualenv. And right now I'm not going to press enter because I'm going to explain more about virtualenv in a moment.

Installing Pip and Python on Linux

Of course, the majority of Django development is done on Linux so let's see how we install pip there. Now, matters are complicated a bit by the fact that not all Linux distributions use the same package manager so I cannot possibly tell you how to install pip on every single one of them, but the most popular package manager is apt, which is used by Debian, Ubuntu, and Linux Mint, so I'm going to focus on that package manager. Now, like Mac OS, Linux comes with Python preinstalled and there's a good chance that it's not the latest version so you probably want to upgrade your Python version. The apt-get command for this would be sudo apt-get install followed by the name of the package. That name is simply python for Python 2, and if you want to install Python 3, you add a 3 at the end of the package name. Now after that has been installed, you can install pip the same way, so sudo apt-get install python-pip. So in this case, the name of the package is python-pip. And again, let me show you a short demo of how this works.

Demo: Linux Installation

So here's my Linux desktop and the first thing we'll have to do is start a terminal emulator. Usually you'll find that in the menu under accessories and in this case here's my terminal emulator. Let's start that. There we are. And then I'm going to type a command, sudo apt-get install python, and in this case I want to install Python 3 so I'm going to append a 3 to the command and press enter and it's going to ask for my password and then we'll just let it run. So now the steps for installing pip are pretty much the same. So I'm going to do sudo apt-get install, and then package name is python-pip and again, enter. And that's just about it. Let's see if we can start pip. Very well, that runs so now we're ready for the next step, which is installing virtualenv.

Virtualenv

Now, before we start with the next step and install virtualenv, let me explain why we actually need to install virtualenv. Here's an example that you might encounter in a development environment. Suppose I'm a developer and I'm doing some maintenance on a Django project that uses Python3 and Django 1. 5. I did a straightforward system-wide Django install and everything works, but at a certain point we start a new project and we choose to use Python 3 and the new Django 1. 6. Now we need two Django installations and have to do some configuration to get everything right. It's not that nasty and it can be done and eventually I will get everything working and so far, so good. At some point, however, I decide I need to use a library called Requests and I installed the newest version of that library, version 2. Again, I simply do a system-wide install of Requests and that's where things go wrong. You see, I didn't realize that my other project has a dependency on Requests as well, but on a different version, so now I have installed Requests version 2, my other project breaks because Requests 1 has been replaced by a different version. And that's just a very simple example. You can imagine how messy things get when you're working on 10 different projects, across Python versions and with many dependencies on different versions of libraries, etc., etc. But, fortunately, there is a nice solution and that's called virtualenv and not only will it prevent these kinds of dependency conflicts, but it will also make it easier to run different versions of Django. Virtualenv allows us to create isolated Python environments. It's good practice to create such an environment every single time you start a new project. You can then install all the dependencies for that project inside the virtualenv environment so they won't cause conflicts with other projects and that basically means that you should never install any Python packages on a system-wide scale ever again. You will save yourself lots of trouble by making a habit of always working inside a virtualenv environment. If you do so, projects with conflicting dependencies can coexist peacefully. Installing virtualenv is easy. You simply ask pip to do it for you. So virtualenv is the last system-wide Python package we're going to install. Now the command is pip install virtualenv and in case you're using Windows, pip won't be on your path by default. You can call it like this. It's in the scripts directory of your Python installation. Of course, you can also add the directory to your path if you like. Now on Linux you will need to use sudo. By the way, if you didn't use Homebrew for the installation on Mac OS, you may need to use sudo on your Mac as well. After installing virtualenv, creating a new virtual environment is as simple as calling virutalenv with the name of the new environment and like with pip, Windows users will find the executables in the scripts directory of the Python installation. Now virtualenv takes several options, but there's one that I like specially and that's -p, which tells virtualenv which Python interpreter to include in the new virtualenv. So for example, if you give it path to your Python 3 install, inside your new environment that will be the default interpreter and that's very helpful if you have multiple versions of Python installed and you want to set up your projects to work with a specific version. We'll see how this works in a moment. Now, calling virtualenv with our without the -p switch will cause virtualenv to create a new directory and set up several things in there. Let me show you a demo so you can see this in practice.

Demo: Virtualenv

So here we are again on the command line and I'm going to start by showing you how to install virtualenv using pip. (typing) And as you can see, it's really easy. And again, let me remind you that in case you use Linux you probably have to use sudo pip install virtualenv. And in case you used something else on the Mac than Homebrew to install Python and pip, again, you probably need to use sudo. Now as you can see, pip does give some warnings and some messages, but at the end here, it says successfully installed, so now virtualenv is there. And now before going on, what I want to show you is that right now my default Python interpreter, if I run Python, is Python 2. 7. 5. Please keep that in mind for a moment. I'm going to exit here and let me check where my Python interpreter version 3 is. (typing) It's in user local bin python3. So now let's start a new virtualenv. Let's say I'm going to start a new project and I want to use Python 3. I'm going to say -p and I'm going to copy this path here (typing). Copy/paste that into the command line, so now it says virtualenv -p and then the path to Python 3. And let's call my new project demo. Very well. Virtualenv as well gives some messages. For example, it's telling me it's making a new Python executable and it's installing pip. Very well. Let's go and see what's inside the demo directory. I'm going to go in there and I'm going to list it, and as you can see, there's a bin and an include and a lib directory here. And I'm particularly interested in the bin directory. And if I list the bin directory, here you can see that it has its own pip and its own Python executables and here Python as well. So that means all of that is contained inside of the virtual environment. And now if I want to start working inside that environment, what I'm going to do is I'm going to say dot, which means I want to read in a script file. I'm going to say. bin/activate. Then I'm going to press enter. And now as you can see to signal to tell me that I'm inside the demo virtual environment, my prompt here is prefixed with the name of my virtual environment inside parentheses. So in this case it says demo. And the funny thing is that now my default Python interpreter is actually Python 3. 3 and that's because, let me scroll this a bit up here. And that's because earlier here when I made the virtualenv, I told it I want to use Python 3. So here inside this project, whenever I activate this particular virtual environment, I'm going to use Python 3 instead of Python 2. Now you can also see if I do an ls of the lib directory here, that it contains a Python 3 directory and if I look in there we can see that it contains links to the entire standard library of Python and also to a site packages directory. And that's because if I'm going to run pip right now when I'm inside this activated virtual environment, pip is going to install these packages inside this environment and they will end up here inside the side packages inside my virtual environment and that means I'm going to be able to install packages that are completely isolated from packages inside other virtual environments. So let me show you that this really works. I'm going to say pip install Django (typing). And again, let's check that it's in here now. Let's check the sites packages, the directory under lib. And as you can see here, now it lists Django here. And now to check that it really works, I'm going to start Python and I'm going to say import Django, and you can see that we actually can import Django and we can also ask Django for its version and it's telling me it's version 1. 6. Now just to show you that this is an isolated version of Django, let me start new terminal. I'm going to minimize this one. And now I have a terminal that's not inside an activated virtual environment. And now if I start Python, not only is it Python 2. 7, but if I say import Django, it doesn't work. It says, no module named Django, and that's because the Django version I just installed isn't available outside of the virtual environment. So what we have seen is to start working in the virtual environment, first you have to move into the new directory with the CD command and then you have activate the virtualenv with the activate scripts. Now on Linux and Mac OS this is done with the dot command followed by bin/activate, but on Windows this is done by typing scripts/activate. So everything that's inside the bin folder on Linux and Mac OS ends up inside the scripts folder on Windows and that's really an important difference that Windows users should be aware of.

Installing Django

And as we have seen in the demo, the final step of installing Django is very simple as well. You only have to tell pip to install Django and that will install the newest Django version. If you need a specific version, like 1. 5, use the double equal signs like in the example here, to install that specific Django version. And of course, you should run this command inside an activated virtual environment so that the Django install will be local to that virtual environment.

Summary

So what have we seen in this module? Well, first of all I told you how to choose which Python and Django versions you need. Then I showed you how to install the pip package manager on various operating systems, and then from there, how to install virtualenv and how to use virtualenv to create and activate a new virtual environment for your projects. And finally, once we had this activated virtual environment we were ready to actually go ahead and install Django, and that means that we're now ready to go on to the next module in which we'll actually start our first Django project.

Starting a Django Project

Introduction

Hi, I'm Reindert-Jan Ekker, and in this module we'll start a new Django project and make our first simple web application. So in this module I'm going to take you through all the steps needed to create our first real web application with Django. First we'll have to create a new project, and then we'll add a new page to that project, a Hello, World page, and to do that we need several steps. First we'll have to make a new app, which is a specific Django component I'll tell you more about, and then to that app we'll add a view and a template. And afterwards, I'm also going to show you how to add some styling and some scripting and some extra files to that project. Of course we'll also have to see how to run the application. And finally, we need to take a look at the design pattern around which Django revolves, and that's called MTV, or Model Template View. This is a Django-specific flavor of the very popular Module-View-Controller pattern, or MVC, which you may already know about.

Demo: Starting a New Project

So of course we'll have to start with creating a new project. You do this with the django-admin script that comes with your Django installation. You can simply call it like this, django-admin. py startproject, and then the name of the project. Now for Windows users, make sure you have your path set to your Django installation or otherwise you won't be able to call the django-admin script like this. Of course, if you don't have your path set, you can also call the django-admin script with the full path. So let's go ahead and make our first project. I like to group all my projects in a dev folder, so that's where we are right now, and to create a project I simply say django-admin. py startproject, and I'm going to name my project boardgames. Now this command makes a new directory called boardgames and we can cd into there. Now before showing you what's in here, I'm going to show you how to run our project, and to do that I say python manage. py runserver. And as you can see it says Starting development server, with a URL. So I'm going to copy this URL and get a browser window, very well, and I'm going to paste the URL in here, and now Django tells us, It worked, Congratulations on your for Django-powered page. So that's basically all you need to do to make a new Django project and run it, just these two simple commands. Now let's take a moment and look at what files this created. So these are the contents of my newly created boardgames directory, and as you can see it contains a script called manage. py, which is the script we just called here for running the server, and it also contains a subdirectory, which is also called boardgames. So inside the boardgames directory there's a second directory called boardgames, and this directory holds some files for configuring the projects in general. First of all there's this file called _init_. py, and that's a file that comes with every Python package, I'm not going to go into that any further, and then there's the _pychache_ folder, and then there's a settings. py, and this holds general project settings, we're going to see a lot of this file in this course. Then there's urls. py, which we're going to use to configure URLs for our project, and there's a file called wsgi. py, which is used for deploying your projects to a production server.

Running the Development Server

So to run a Django project, you simply move into the project directly with the cd command, and then you run the manage. py script with the runserver command. And this will start the built-in Django development server. It's a fast and lightweight web server that will reload the changes in your code automatically, which is nice because it really speeds up development. You should never use this server in a production environment though, because it was really not intended for that and it may cause problems.

Django Apps

Now before adding any new code, I have to introduce you to a new, but very important Django concept, and that's the idea of a Django app. Now the word app, when used by a Django developer, means something very different from say an app for your mobile phone, because an app is not a standalone piece of software. So let me explain what an app actually is. A Django app is just a Python package that is specifically intended for use in a Django project. You can break up the functionality of your project into multiple apps that each act as a little web application of their own, with their own models, views, templates, etc., etc. So that means that a typical Django project consists of multiple apps. Now in a moment I'm going to add a new app to our project that adds a home page, and after that I'm going to add an app that will encapsulate the functionality of playing a tic-tac-toe game. Now if I wanted to implement a forum where users can exchange their information, I would add a forum app, and for the web shop where I'm going to sell board game related merchandise, I would add a shop app, etc., etc. Now this helps keep our code modular and organized. But of course you're completely free to organize your apps any way you like, so you can even decide to put everything in one huge app, or not to use apps at all, but the generally accepted best practice is to follow the Unix philosophy to do just one thing and do it well, so keep your apps simple and small. Now something else that's nice about writing apps is that sometimes you can make them reusable. So I might just reuse the shop app or the forum app from another site in our current project, without any problems. I have to say that designing an app to be reusable is a little too advanced for this course, so I'm not going to be doing that right now.

Demo: Adding a New App

So to start our app I'm going to follow the instructions that the default page gives us and run python manage. py startapp. So you can quit the server with Ctrl + C and get back to the command-line, and then I'm going to say python manage. py startapp, and I'm going to call my app main. And again, let's see what that does. Here's our directory again. This is the general boardgames directory and there's a new directory here called main. And that again holds an _init_. py, it holds something called admin. py, which we're going to use for the admin interface, we'll see about that in the next module, there's models. py, which is going to hold our models, and again we'll see more about that in the next module, tests. py for unit testing, and views. py, which is going to hold our views. And to add a new page to our project, I'm going to have to edit this file called views. py.

Demo: Adding a Page

So let's go ahead and edit views. py. I have to open it in an editor here. This is what it looks like by default, and now I'm going to add some code that will serve a page that says Hello, World. (Typing) Now this method called home is what we call a view in Django, and it returns something called an HttpResponse, which holds a string, Hello, World! Now I'm not going to explain this in detail right now, let's just see what happens if we run the server with this code. Well nothing happens, and there's two reasons for that. First of all, let me go back to the editor. Here in the file settings. py under boardgames I'm going to have to add my new app to this list called INSTALLED_APPS. So here I'm going to add my main app. Very well. And then the second thing I have to do is to open this file called urls. py and configure my new page to be served on a certain URL. So let's start with saying url, and I'm just going to say it's going to be served on the empty url, which is like the home url, and then I'll say main. views. home. And now we see Hello, World!

Adding an App

Let's go over what we just saw. First of all, we can add a new app with the python manage. py script and then using the startapp command. After the startapp command, you use the name of the app you want to start. Now don't forget you also have to add the name of your app to the INSTALLED_APPS variable in settings. py, otherwise Django won't pick up your app as one of the modules in your project.

URLs and Views

So what did I just do? What happens when I view a simple web page with Django? Well suppose that like in the demo, we visit the locally running server at 127. 0. 0. 1 at port 8000, and we use a browser of course to visit that URL, and first of all the browser would send something called an HTTP GET request to that server. Now if you don't know what that means exactly, let me put it like this. An HTTP request is a message from the browser to the server where the browser requests that the server does something, and in the simple case of viewing a web page, it will send a GET request, which means that it asks the server to send it the content for some URL, and the specifics of that data are determined by whatever comes after the host name and the port number. In this case, there's nothing behind the host name and port number, and that's why the browser will ask for the document host at that slash. The string you see here, GET / HTTP/1. 1 is the most important part of the HTTP request, although the browser will usually send lots of extra data. So this request will end up at the Django development server, which will show our Hello, World view. So our development server receives some requests and then what it does is it searches in a file called urls. py, we've already seen that for a URL mapping that matches that request. Here you see an example that matches that request and it maps this URL to a view, in this case the view is called main. views. home. So what Django does next is it searches for a file called views. py, which holds a view function by the name of home. So in this case it will search in the main application for views. py, and in there for a home function. And as you can see, the home function takes a request object and returns a response, and its response contains a string Hello, World, and that's exactly what's sent back to the browser and what the browser shows. Now you may notice here that the actual URL in the URL mapping is different from the one I showed you in the demo. Let me try to explain to you how that works.

Demo: URL Mappings

So let's take a better look at the URL mapping. The first argument of the URL function is a regular expression, and that's the kind of expression that you can use to match strings. Regular expressions themselves are strings as well, and we prefix them with an r because we want them to be role strings. That means we don't want the special characters we use in regular expressions to have any special meaning. So whenever you use a regular expression, prefix your string with an r. Now the expression I used here is simply an empty string, and I map that to the home view we already made, which displays Hello, World. And actually, the empty string here matches anything. Let me show you. As you know already, the empty string matches the empty string of course, but it can also say, let's say test, and that matches as well because we see Hello, World now. Or I can say test/hello/world, and again, that matches. But of course I don't really want that, I want the home view to be shown only on my home page, and to do that I want my expression to match the empty URL only, and the solution is like this, use a caret and then use a dollar sign. The caret here means the beginning of the string, and the dollar sign means the end of the string. So this expression will match exactly the empty string and nothing else. So if I save now, and I don't have to restart the server because the production server will pick up all my changes in my Python code automatically, now this URL will give you 404, Page not found, because my URL mapping doesn't match this URL anymore, it will only match the empty URL like this. So of course I can use regular expressions to match many, many more things. If I only want to match exactly the string hello, I can do this, save again, and now reloading this will give a 404, and saying /hello will say Hello, World! Now if I remove the dollar sign, this will match any string starting with hello, so I can match hello or I can match helloworld. Very well. Or I can match anything ending with hello, like this. So now helloworld won't match anymore, but hello by itself will, or let's say hellohello will match as well, and that's just a very short introduction to the power of regular expressions, because there's a lot of things to know about regular expressions, and you can match just about anything.

URL Mappings

So whenever someone browses to a page on your server, his or her browser sends an HTTP request for that URL, and then Django looks in the urls module to find a urlpatterns variable. This variable holds a list of url mappings. It will go through that list to find a pattern that matches the URL that was asked for. Now as I said in a demo, we use regular expressions for these urlpatterns, and if you want to know more about regular expressions, follow this link. They're really quite powerful and a very nice tool for any programmer, not just a Django developer.

Django Views

So then we find a URL mapping and Django calls a view. And in case you're familiar with other web frameworks, you might have heard the term controller, and a Django view is what other frameworks call a controller. Basically it's callable, so a function, that takes a request and returns a response. So it handles the HTTP request and decides what content is going to be shown to the user. So let's say we want to show the user an HTML page, and if everything works successfully, the status code of the response is going to be 200, OK. Now the simplest way to do this is to use the HttpResponse class that Django gives us and return that. So as you see here, we have a home function that takes a request and returns a new HttpResponse instance. And I just have a simple string here, but of course this string could contain HTML. Now you can imagine that if you do everything this way that your code is going to be very ugly because you're going to write a lot of Python code to generate all your HTML, and we really don't want our views to contain presentation logic. So what we normally do is we move the generation of HTML to a template, which is another kind of Django component that's specifically designed for generating your HTML.

Demo: Templates

So I want to use a template to render my page. A template is simply a file that will help me create HTML. First I'm going to make a new folder in my app called templates, which is where all the templates go, at least for this app, and then inside the templates I'm going to make another folder that's named after my app, so that's going to be called main, so all the templates for my app go in here. And let's make a new file and let's call that home. html. Very well. So this is going to be an empty file, and I can put any text I want in here. So I can simply say Hello, World, like we did before, but let's make it a little bit more fancy and let's make it an HTML page. (Typing) Very well. So this is a very nice HTML page. And let's go back to views. Now I want my view to call this template and to have the HTML rendered and returned to the browser. And for that Django comes with a very nice utility method called render in django. shortcuts, so let's use that instead of this line. I'm going to remove all of this and say render, and render takes, as you can see here, a request and then a couple of arguments. First I'm going to say request. And then the second argument it takes is the location of my template, and that's taken relative to the location of the templates directory. So in this case the location of this template will be main/home. html. And this should also make clear why I put my templates for this app inside a main directory, because if I'm going to start another app that also has a home. html template, I don't want the names to clash, so another home template may be, let's say in forums/home. html. So that's why you always put your templates inside the templates that are inside a directory that's called after your app. So very well. This will call the template engine, the template will look at this file, and render HTML and return an HttpResponse, and then we return that as the result of our view. So let's see if this works. Let's go to the browser and reload. Again, we don't have to restart the server, and now we have a real HTML page. As you can see, this is our h1 tag, this tag here, and we have a title as well, so that's very nice. Now let's go one step further because this template engine supports a lot of other stuff. For example, it can render variables. So let's say message here instead of just Hello, World, and these double curly braces are Django template syntax and they will render a variable called message. And to give the template that variable, I can add a dictionary here and give it all the variables with values I want. So in this case I'm going to say message, and let's say that the message is going to be hi, there. Very well. And again I don't have to restart the server, I can simply reload and now it says hi, there! So this dictionary here that I give as an argument to the render method will end up as what we call the context for this template and the double curly braces here will have the template engine look up an element of the dictionary with the name message and output it into the HTML. And in a similar way we can also have a template use a for loop or use if, then, else, etc., etc., to really generate very nice and powerful HTML pages, but more on that later.

Django Templates

So this is what our view function looks like now. Instead of returning an HttpResponse instance directly, we call the render method, and the render method will use the template engine to parse our template file and generate the HTML, then it will put that HTML into a response object and that will be returned as the result of our view. So basically a template can render HTML or any kind of text-based format, so you can also use it to render JSON or CSV files, or whatever you like. And again, to render a template from a view, use the render function and you can find that in django. shortcuts. Another thing I showed you is that templates go in the templates/ dir of your app. Also I showed you where the templates should go because they should go in another dir under templates that's named after your app. So generally if you look at the complete structure, we first have your projects directory, and then in there is your app directory, that will contain a templates/ dir that will contain another dir named after your app, and that will contain your actual template. So you will have project/app/templates/app/template. html. Or, in our concrete example from the demo, you may have a project called boardgames in which there's a main app, which has a templates directory, which has another main directory, which holds our templates, and in this example that was home. html.

Demo: Static Files

So let's apply some styling, some CSS, to our home page so I can show you how easy it is to actually do this kind of thing in a Django template, and this is what our home page is going to look like. Well first of all of course we need CSS, and for that I'm going to use the Twitter Bootstrap CSS with the team I downloaded from a site called bootswatch. So files like CSS and JavaScript files and images are not the kind of stuff you want to include in your templates, because each time you render a template, the template engine has to read the template and generate your page, and you really don't want to do that for your CSS and your JavaScript. So what you do instead is you make a static folder. Here as you can see it's on the same level as our app, and put all that stuff in there. So here in the bootstrap folder is all the CSS and the JavaScript I downloaded from CSS Bootstrap, and here are some extra fonts, and here's our jquery JavaScript, and there's also an image here I'm going to use later, and all that stuff ends up in a folder called static. But of course then we have to tell Django how to use that. Let me show you how we do this. There's an app that comes by default with Django and it's called staticfiles, so if we look in settings. py, here in the INSTALLED_APS variable you find django. contrib. staticfiles, and it's enabled by default so you don't have to do anything here to get that functionality. Then if you scroll all the way down, there's a setting here that says STATIC_URL, and it's set to /static. I'm going to explain to you what that does in a moment. Now this is the one you have to add. It's a variable called STATICFILES_DIRS, and it's a list of directories that are going to be searched for static files. So in this case I take the BASE_DIR of our project and and append static to that. And that's going to point to our static folder with all the files in there. So then to use these files, in my template I start with saying load staticfiles, and this syntax here with the curly braces and the percent signs is called a template tag, so this is the template tag called load and it loads other template tags from the staticfiles app. Then if I scroll down a bit, here for example where I'm going to include my stylesheets, here we see in the href attribute, instead of using a URL here I use a static template tag, again with the curly braces and the percent signs, and this points to a file bootstrap/css/bootstrap. min. css. And if I scroll down further where I include my JavaScript, for example here I include my jquery, again, instead of using a URL I use this static template tag. So what will happen is that the static template tag here will replace this syntax with a URL that points to this file, so our jquery file here in the static directory. And going back for a moment to our settings file, this setting here, STATICFILES_DIRS, tells our static tag where to find the files, which is here, and the STATIC_URL here is the first part of the URL that gets generated. So if we go back to our page and let's look at the page source for a moment, we find for example that the URL that's used for our stylesheet starts with /static, and then the path to our file. And that's how we can include static files like CSS and JavaScript and images in our pages.

Static Files

So any real world web application is going to use non-dynamic content like CSS, JavaScript, and images, and that content may be hosted separately on a different server, so there's a good chance when you put your site in production you're not going to host this content on the same server where your Python code runs. And to make that easy, you can use the staticfiles app. Now as I showed you during development, you can put your static files in a static directory and they will be served by the development server, but in a deployment situation you will probably choose another solution. Now, I also showed you that the staticfiles app is enabled by default and that there's a static URL setting in settings. py, and it also showed you that you have to add this line to your settings. py if you want the staticfiles app to find your files in the static folder. Then, when you did that setup, you can refer to static content from your templates by first loading the staticfiles templates at the start of your template, and then everywhere you want to refer to that content, use the static tag, and that will look like this.

Model-Template-View

Now finally I want to tell you a bit about the design pattern that underlies all of Django, and it's called model template view. So as I said before, MTV is a variation on the well-known and widely used Model-View-Controller architecture. Model, view, and template, those are the names of three types of components, and let's start at the bottom two and explain those. First of all, the model represents your data. We'll write a Django model for each table in your database, so the model layer will contain a bunch of classes that represent the data in your database. We haven't done anything with this yet, but I'll show you more about this in the next module. Now one level higher there's the view layer. In Django a view is a component that handles an HTTP request. Such a view may call the model layer to store or retrieve data in or from the database. Also, to generate an HTML page and display the data we get from the database, it may call a component in the template layer. So let's look at that layer. Well generally, templates are used to generate HTML. This is purely a presentation layer, so templates should normally only contain presentation logic. We'll find out more about what this means later in the course. For now, it's okay if you just remember the names and general function of models, views, and templates. Now in case you're familiar with other web development frameworks, the model template view pattern may surprise you because most other frameworks use a similar pattern called Model-View-Controller, and in these frameworks what Django calls a template is called a view, and what Django calls a view is called a controller, but apart from that, the general patterns are the same, and there's no need to be confused. Just remember that in Django, we use the word view where other frameworks would say controller, and we say template where others say view.

Summary

And that concludes this module. We saw how to create new projects and run it. I also showed you how to create a new app and how to add templates and views, which are part of the model template view architecture, how to map URLs to a view, and how to write your own views, and finally how to render a template from a view. And the last thing I showed you is how to actually use static files like CSS and JavaScript from your templates.

Models

Introduction

Hi, my name is Reindert-Jan Ekker and in this module we'll add some real functionality to our project. We'll make Model classes and generated database from there and see how Django makes it very easy to work with persistent data. After looking at ______templates and views, we're ready to start adding some models. We'll see how to write Model classes and define database field types, how to save your objects to the database and delete them, how to do queries on your database, and how to handle relations between objects. When we've written our models we can let Django generate the database and even have it auto-generate a user interface for editing the contents of our database.

Demo: Adding Models

A model is the single definitive source of data about your data. It contains the essential fields and behaviors of the data you're storing. So we're going to write some simple classes that will represent a tic-tac-toe game and from there we'll generate a database and the user interface to enter and edit game data. Now of course I'm going to start with adding a new app to the project. (Typing) Now let's add our ______ models. I�m going to open my Editor again and here's our new app. And there's a file called models here. So I'm going to open that and it says, Create your models here. Now let's start with adding a Move class. Now all Django models need to inherit from the base Model class, (Typing) which makes this an official Django model and that means that it will represent a table in the database that holds moves. Now how does Django know which columns that table will hold? Let's tell it. To begin with a move should have the coordinates on the board where the move was made. And these should be stored in the databases integers. Let's tell Django to make integer fields in the move table. (Typing) Notice how we made two class attributes, x and y, which are instances of the IntegerField class. Django will inspect these and create a Model class that has x and y attributes with methods to save move instances to the database and do database queries to retrieve rows from the database and more. Django will also add a primary key field so the current Move class has three fields in total. Now let's say that the player can make a comment with every move (Typing). Of course a comment is character data, not integer data, and as you can see we can give the Field class options like the maximum length of the field. Again, this will be used by Django to generate SQL. Now let me add a Game class. I'm going to copy that in here (Typing). Now there are some new things here. First of all, the game has to have players. And the players of our games will be the users of our site. As you can see, I made a first player and a second player field, which will hold foreign keys to a user model. Now the ForeignKey Field class takes the model that the foreign key references as its first argument. But I didn't make a User class yet. And that's because Django comes with a default User class. Let's import that. (Typing) Now since the foreign key represents a one to many relation, Django will add attributes to the User class as well to represent the other end of the relation. So, for example, on the User class there will be a field called games_first_player, which holds a collection of all the games in which that user is the first player. Now apart from the first and the second player I also added a field so we know which player can make the next move. Then there's a DateTimeField for the start_time with the auto_now_add option that makes sure that this field is set when the instance is created and another DateTimeField which is called last_active, which will only be updated every time that the model instance changes. And for that we have the auto_now option. So now that we have the Game class, we can also add a relation from moves to games (Typing). Of course, again, this is a many to one relation, which means that many moves can have a relation to the same game, which is exactly as it should be. I'm leaving out the related name here so the set of moves for a game will be called move sets. So let's check out how Django creates a database from this. We need the syncdb command for this (Typing) and here you can see the tables Django is creating for us. Now there are a lot of tables, but our game and move tables are not showing up. Let me explain to you why that is. But first I have to create a super user because that's what Django is asking me for. (Typing) Oh and I forgot, I can't use a blank password, so let's do something else. Very well. Now the reason why Djando didn't create the tables for my models is because I forgot to add our new app to the installed apps setting. Let's fix that. Of course I have to go to settings. py and go to the installed_apps and add my new app here (Typing) and now I can run syncdb again. So now it tells us that it created our tables. As you can see, the names of our tables are prefixed with the name of the app, which in this case is tictactoe, and then the name of the model. Now by default Django will create a SQLite database in the top level directory called db. sqlite3. Of course, if you want to use MySQL or another database engine all you have to do is configure this in settings. py. Now what do our tables look like? I can ask Django for the sql statement it has executed with the sql command. And I have to give the sql command the name of the app I want to see the sql for. If I run it, I see what my tables look like. And here I see all my fields, including a new field called id, which is the primary key field, as well as some constraints. Every single field here has a not null constraint. Of course, if you don't want a not null constraint you can set that on the field instance as an option. You can also see here how the foreign keys are defined. So these are the related names, they get a suffix_id and they reference, in this case, the auth_user table and here in this case the tictactoe_game table. So that's the sql that Django creates for us and all I did is write some very simple code and add my app to the installed_apps list and from there Django generates my database.

Django Model Classes

So let's review what we just saw. Basically speaking, each Model class maps to a table in your database and is a subclass of the Django Model class. The class attributes of the model represent the columns of the database table. Each attribute is an instance of a Field class, like IntegerField or DateTimeField. You can also give the fields options, like default values or constraints. There are many different field types and you can even define custom ones. The link I included here will take you to the reference documentation for Django Field classes, so you can find out everything there is to know about them. Now these field types are used by Django in several ways. First of all, Django uses the Field class to determine the type of the corresponding database column, but there's more. Later in this course we'll see that Django will also render HTML to generate nice input fields in web forms, depending on the type of data our fields accepts. But more importantly, Django will generate an API for working with our database models, which makes writing database queries and storing data very easy. We'll see more about this in a moment.

Manage.py Database Commands

After you write a Model class you can use the manage. py script to generate database tables for you. Before you do that, it may be wise to check whether Django will generate the sql statements you want it to. The manage. py script has a sql command for this, which lists to create table statements for an app in your project. That way you can check whether the sql it generates is as you intended. When you're satisfied you run the syncdb commands. This will add any database tables that haven't been created yet, but you should be aware that syncdb doesn't do database migration. That means that adding or removing a field or changing constraints, like not null, or adding an index will not cause syncdb to do anything at all. Now fortunately the Django developers are working hard to add database migrations to Django 1. 7. For now, if you need to change a model you can simply drop the table and run syncdb again or for a more professional migration solution, you should check out a python package called south, which is the de facto standard for Django database migrations.

Demo: The Admin Interface

Let me show you another feature of Django that will boost your productivity. Django can generate very nice and powerful user interface for your Model classes, enabling you to enter and edit your data without having to write any code for it. Now the admin app comes with Django by default and as you can see, here it is in the installed_apps setting in settings. py. So this is where the admin app actually gets ______. Now there's also some admin related code in urls. py, let's open that. As you can see, the admin app is imported and the auto_discover method is called, which tells the admin sites to discover all models to generate a user interface for. Now you can also see here that this URL pattern that matches the word admin includes the URL patterns from the admin app. In other words, the admin app is hosted under admin followed by a /. Now also note that there is no dollar sign here, so this actually doesn't just match the word admin, but it matches anything starting with admin. So this line includes the URL's module from the admin site into our URL configuration and there can be several pages listed in this URLs module under admin/ and whatever comes after that. Now to enable a user interface for our models we need to register them with the admin interface. Now Django has generated an admin. py file in our app, so let's open that and register our models. Of course, I should first import them. (Typing) And now I can register them with the admin interface so that they will show up in the user interface. (Typing) Okay, let's go and start the development server. And let's open a browser. As we saw in the URL configuration, the admin site is hosted under admin, so let's check that out. Unfortunately I have to log in with the super user account I made when the database was created. Now the interface you see here is completely auto generated by Django. As you can see, we can add users, groups, games, and moves. Let's start by adding a couple of games. There's an Add button here and let's click that. Django knows how to generate the right HTML forms for your model fields. From the foreign fields for players it makes a dropdown menu where we can select a user. We're looking even at a user on the fly here. So let's create two more users, I'm going to call them alice and let's create another one called bob. And for now I'm just going to say that I'm the first to make a move here. And let's just create some more moves her. Very well. Now we're back at the game list and I don't really like how this list looks. It just says game object about every game. But I need it to tell me a bit more. The solution is to add a str method to our class so that we can display these objects. So let's go to our models. py and add a method here, I'm going to call it strength. Very well. (Typing) So now our Game class has a method to display itself, we can immediately reload here in the browser and there we have a very nice display of our games. But I'd also like some status information. I want to know if a game is running or if it has ended and if it has ended what the result was. Let's add a new field to our model for that. (Typing) Now this is a field that holds a single character. As you can see, it's a character field with max_length=1 and the default value is A, which means a game starts in the active status. I also like to make the possible values explicit and to do so I can make a choices list. Let me show you. So this list holds all possible values for the fields with a descriptive string. Django will generate a nice little dropdown list from this, but we have to set it as an option for the field and that's what I did here. So I gave it the choices option pointing to game_status_choices, which is this list. Now I cannot show this to you right away because we have just added a field to the model, which means we have to update the database. Unfortunately Django doesn't give us a command to migrate the database model. What it does give us is a command to start a shell for our database. So let's do that. Of course, we have to stop the server first. Then I'm going to say python manage. py dbshell. This will start the shell for our database. Whether that's ______, MySQL, SQLite, it doesn't matter. Now I can, of course, type an ______ table command here, but I'm going to do it quick and dirty and simply drop the entire games table. (Typing) Sorry, it doesn't have an s at the end, of course, and let's exit here. Now I can run syncdb again to recreate the table including our new status field. So let's check out the admin interface again. And now if I Add a game, we can see that the users are still there because I only dropped the games table, not the users table. So let me add a game. And here's the status dropdown and these strings here are taken from the choices field I just made. So, let's add some other games. Very well. (Typing) And of course, if I go back here, we can add moves as well. Let's look at the generated form for that. There's an X and a Y and a Comment field and again a dropdown that has been generated for the foreign key to the Games. So that should give you a quick impression of the admin interface. Of course, we'll let users input their moves in a real tic-tac-toe board soon, but for now we can at least put some data in the database.

The Django Admin Interface

So Django will generate a very nice admin user interface for any model you'd register in admin. py. This admin user interface is extremely configurable, by the way, and I could really talk a couple of hours about all the different things you can do with it, but that's really beyond the scope of this course. So if you want to know more about customizing the admin interface, visit this link and it will tell you all about all the power the admin interface has to offer. Now don't forget to implement the underscore underscore str function for your Model classes because that will make sure that both in the admin interface or when debugging or maybe even in templates your mobile instances have a nice default representation and will remake life a lot easier for you.

Demo: The Model API

Django gives us a very nice API for manipulating our data from our python ______. Let's check it out. I'm going to show you this in an interactive python session, but instead of running python normally I'm going to use the manage script again with the shell commands. This starts a normal python shell, but it also reads the Django settings file, so we won't have to do that ourselves. But what we do have to do is import our models. (Typing) Now every model has a manager instance, it's set as an attribute on the class and it's called objects, like this. (Typing) We can use these manager objects to perform queries on the table. For example, to get all games we say this (Typing), or I can get them by primary key, like this (Typing). And this is the game with primary key 1. To check that I can assign it to a variable and ask for its id. Very well. So it's id 1. Now as you know, Django automatically gives every model a primary key field called id, even though we don't set it explicitly. We can also construct more intricate queries. For example, let's ask for all active games. (Typing) To do that I use the Filter method. (Typing) Now I can simply use the name of the field I want to filter on as keyword argument to the filter function, like this (Typing), and Django will generate the appropriate sql statement. Now the same is true, by the way, for get, so here we filtered on a primary key field of 1. Pk here is a special argument and it will always match the primary key. So I could also have used the field name id here because that's the actual field name of the primary key. So if for both get and filter you supply keyword arguments with the values you want to filter on. I can also ask for all the games with a status different than A with the Exclude method. Very well. Now let's ask for all games in which I am the first player. (Typing) Again, we're simply using a keyword argument. Note the double underscores here, they are a special Django feature. The first part of the argument is first_player, which is the name of a field on the model object, but that's a foreign key pointing to a record in the users table and I want to look up all games where the username fields on that record in the user table equals reindert. Now the double underscore here enables me to ask for the value of a field on that record, in this case, username. So we are getting all games where the username field of the user record that this ForeignKey field points to equals reindert. So let's make a move. This starts with simply instantiating a move object. (Typing) And as you can see, we have to use the fields that we are setting as keyword arguments. Now to save this in the database I say m. save. Now I can ask our game instance for a list of moves made in that game. We have a one to many relation because of our foreign key and from the side of the move instance, the related game instance is simply a field called game. But from the point of view of the game, there's a set of related objects. Django calls this move_set by default. So this points to a RelatedManager object on the move table that only represents the moves related to this game. So because it's also a manager, I can ask for all the moves in this set. Or, for example, I can count them (Typing) or ask for all moves where there's a comment. (Typing) So that will exclude all empty comments. Now let's change the status of the game to something else than A for active, let's change it to D for draw. Sorry, of course, I have to use quotes. And again, to save this game, now we can say g. save. And the difference between this save and the previous save is that the previous save was a new object, so this will be an sql insert statement, and this object already existed, so this will be an sql update statement.

Save and Delete

So Django reads the class attributes that represent your database fields and generates the Model class for us, which enables us to create instances of the model. In my demo those would be user move or game instances. To create such an instance, you call the constructor with keyword arguments for the fields you want to set. So this example here sets the x, y, and game fields of a new move. The game instance is a foreign key relation, but we don't have to worry about that in our code, we simply set the fields to a reference to a python object, which is, in this case, a game. Now storing the new data in the database is straightforward. You simply call the Save method on the object you want to save. In case it's an object that was not saved in the database yet Django will set the primary key field and insert the object in the database. In case the object was already present in the database and you only changed some of its fields, Django will detect this and do an sql update instead of insert. You can also call delete on an object to remove it from the database.

The Model API

Each Django Model class gets a class attribute called objects, which is a manager. It enables you to do queries on a table level, so that's why it's an attribute of the Model class, not of Model instances. So basically you can think of a Model instance as representing a row in your database and a Model manager as representing the entire table. Now some of the more important methods that the manager offers you are get, which returns a single instance, which you can select by specifying a field and a value you are looking for. By the way, this will raise an exception if the number of matching objects is not exactly 1. So if there are either no matches at all or more than 1 for the Get method, you will get an error. Now all returns all rows in the table and filter allows you to select a set of matching objects. You tell it what you want it to match in the same way as with get. And exclude does the opposite of filter. It returns all the objects that don't match. Now if course, this is just the tip of the iceberg. If you want to know more about the models API follow this link.

Database Relations

The ForeignKey fields I showed you in the demo represent one to many relations from game objects to moves and from users to games. These relations are defined in our model by a ForeignKey field that is on the side of the relation that represents the table that actually holds the foreign key reference. In other words, the field is defined on the one end of the relation. The other side gets an attribute that represents a set that can hold any number of objects. It will be named something_set where the something is the name of the related model. So in our demo, a game has a move set, but a move simply has a single game field. Now this move set object is a manager object, just like Model classes have, but in this case the manager only represents the related objects. It works just like any other manager though, so you can call get, all, and filter on it. To set a foreign key relation you assign the object you want to reference on the one side. So in this case, we assign a game object to a move. So you don't have to bother with actual foreign keys because Django does that for you. We are working with real python objects here. You can also set the relation from the other side. To do so you call the Add method on the set to add the new move to the game. In the database, this has the exact same effect. Now Django also offers OneToOne and ManyToMany fields. If you want to know more about them, follow this link.

Summary

So, that brings us to the end of this module. What have we seen? Well first of all we've seen how to write Model classes and when you're writing Model classes you will have to describe the types of fields your model has. From there you can let Django generate a database and once you did that, you can save and delete data. Of course, you can also do database queries. Something else you cannot do without if you're working with databases are relations, which are represented by foreign keys. And we've also seen how Django makes working with these very easy. And finally, last but not least, we've seen how Django generates a very nice admin user interface so you can edit your database content in a very user friendly way.

Adding a User Home Page

Introduction

Hi, I'm Reindert-Jan Ekker and in this module we're going to add a user homepage to our Django projects. So, we're going to add some user homepages and I'd like to add some content from the database to those pages so you can see how we can retrieve that data and display it using a template. We'll also need to use authentication so that the user can log in and that we can check on the home page whether he is actually logged in as a user. Furthermore, we'll need some more advanced template language. We'll need to do some logic, some ______ statements, we need to generate URLs. For example, a link to the log-in and the log-out page, and we'll also see how to do template inheritance, which is a very powerful feature that will enable us to apply styling to the whole site at once. I'll also go a bit deeper into URL mappings.

Demo: Adding Login and Logout Views

Now I'm going to start by adding a login and a logout view and since Django comes with a login and a logout view, we don't have to do a lot to configure them. We only have to add them here in urls. py. Let me show you. And there's a couple of things I have to explain about this. First of all, as you can see here, we can use the += operator to add new patterns to the URL patterns variable, so that gives you a very nice way to group different groups of views. Then we see why there's a string that's the first argument of the patterns function. You might have wondered before why there's an empty string here. Well, in this case we're actually using that argument. And as you can see, there's a package here. It's called django. contrib. auth. views and basically this string is the common prefix for the all the views in this patterns invocation. So here we say that the login URL is mapped to the login view. This isn't just a login view, it's actually Django. contrib. auth. views. login, and the same here. This is Django. contrib. auth. views. logout. So basically using += and the prefix, you can make really nice groups of URL patterns with common prefixes. Furthermore, you can also see that I'm now giving arguments to my views, so the login view takes an argument called tamplate_name with a template. So apparently we'll have to provide the login view with our own template. I'm going to show you that in a moment. And the logout view gets an argument called next_page and that's actually the page where we're going to redirect after someone has been logged out. Finally you can see that we can name our URL mappings so this mapping is called boardgames_login and this mapping is called boardgames_logout. And this next page here also refers to a URL name and I'm going to have to add that here so I'm going to say here name is boardames_home. So now when the user logs out he will be redirected to this page. Now let's add the template for the login. html. I'm going to add a top-level template folder called templates, of course. So that's at the top level of our project and I'm going to add a file called login. html and let's open it. So here's the html I just pasted in here and it's a very simple login form. Don�t worry about the HTML right now. This will become clear later in the course. The important part here is that the view passes a form variable here and we render that form in the template. We also add our own HTML form tag and a submit pattern. Very well. Let's try this. Of course, my links don't work yet so I have to go to a login manually (typing). And now I get a "template does not exist" exception and that is actually because this new template folder I made isn't configured yet so I'll have to add that in the settings, going down here and I'm going to add template_dirs and let's copy this here (Typing) and let's not call it static, but let's call it templates. Very well, and why do we need to do this? Well, the template logger by default only looks inside apps to find templates folders and of course, this new template folder isn't inside an app so we'll have to configure the template_dirs variable to look in our base_dir templates as well. So let's reload the browser. Very well. Now this doesn't look like much, but don�t worry. Soon we'll see how to apply styling across the whole site.

More about URL Mappings

So we've just seen some new features of URL mappings. First of all we saw that you can add patterns to an existing patterns variable with the += operator and that gets more powerful if you also use the prefix string that's the first argument of the patterns function. And that will be the prefix for all the views in that patterns invocation so if you're including multiple views, which are in the same package, you can move the package name to the prefix string. Furthermore we've seen that views can receive key word arguments like in this example, template name and next page. And finally we've seen that URLs can get names with the name key word argument like here, boardgames_login and boardgames_logout. Now let's go on with the demo and among other things we'll see how we can use these names for URLs.

Demo: A Template for the Home Page

Now let's log in (typing). And now we see that the login will redirect to URL accounts/profile, but of course, I don't have a view that's mapped to that URL and I want it to simply redirect to the home page instead so let's fix this. Again, I have to go back to settings and I'm going to add some more lines here (typing). Now, what we're doing here is we're setting the URLs for the login and the logout pages, which will be used, for example, when a user gets redirected to the login page when the site requires him to log in. Now this setting here, the last one, is where the user gets redirected after logging in, which currently, of course, is the home page. Now to make sure we can see that the user is logged in, let's show a message. I'm going to home. html and I'm going to change this message here (typing). So what you see here is a template tag, again with the familiar curly braces and percent signs and this is the if tag and it simply checks whether the current user is authenticated, so basically whether we are logged in. If we are logged in, I'm going to say, "hi, " and then I'm going to print the user name of the current user. Now if the user isn't logged in, then we're just going to show the normal message that we show to everyone who isn't logged in. Now, of course, the login link here doesn't make sense anymore when the user is logged in, so let's change that as well (typing). And again here we check whether the user's authenticated and if he is, I'm saying log out and if he isn�t, I'm giving a login link. And as you can see, now in the links here, in the href attribute, I'm using another tag called URL and that takes the name of one my URL mappings. So what this does is is it will generate the correct URL to my logout view and this will generate a correct URL to my login view. So if somewhere along the way I desire to change my URL mappings and move the login to some other URL, this template will keep working because the correct URL is generated here automatically. So that's a very nice way to handle links inside your sight. Now let's also fix the login link in the default welcome message. Here, Click here to log in. I'm going to remove this and add a URL tag and let's see if this works now. So let's see if this link works now. So this is my login form (typing). I'm going to log in here, and it redirects me to my home page. It says, Hi, Reindert!, so this is a different message when I'm logged in. And you can also see that this link here now says log out. If I click it, again, I'm on the home page.

Authorization with Django

So we have seen the complete process of setting up simple login and logout views. The default login views that comes with Django is in Django. contrib. auth. views and remember you have to provide the template for the login form yourself. There's a default logout view in the same package and what that needs is a next page. There's no such thing as a logout page, but we need to know where the user has to go after he logs out, so that's the next page. Then we saw that we need some settings, the login URL and the logout URL and the login redirect URL in settings. py to make everything work. Then after all of that is set up, you can use the if template tag inside your templates to check whether a user is authenticated and depending on that you can show different kinds of content to a logged in or a non-logged in user. Now in case you want to know more about authentication with Django, follow this link.

More about Django Templates

We've also seem more template syntax. Let's go over the things we've seen. The first is called a variable and it's denoted by double curly braces. It will render the value of the item in the braces to the output. Of course, that item has to be present in the template context. Now if you don�t remember what that means, we'll see more of that later in this module. Now there's another kind of template syntax and that's called a tag. The syntax for a tag is a pair of curly braces with percent signs. Now Django comes with a nice set of default tags and in this module we'll see the for, if, and url tags, but there are a lot more and the link here takes you to the reference documentation of all the built-in tags. If you're not satisfied with what the built-in tags have to offer, you can add your own custom tags, either by importing them from a third-party app, or by writing them yourself. Now this is the if tag, which just works like a normal if statement. If has optional ______ if and else parts, but remember that you always need to add the end if tag at the end, so in that respect it's different from a normal if statement. We've also seen the url tag and the url tag will generate a url for some named url mapping, so in this example it will generate the url for our login view.

Demo: Adding the Home View

So as you know, I want to add a user home page or a profile page and to do that I started with adding a new app called user. You might also have called it profile or account or whatever you like, but I like to call it user. And for now I simply want it to render a template so I copied our home view, this is basically the same view we already had. The only difference is it renders a template called user/home. html. And of course, we can find this template in the templates folder of our user app under another folder called user and there it is, home. html. Then I added a urls. py, so this app has its own URL configuration. It looks exactly the same as we're used to. Here's the prefix. We tell Django to look in user. views, which is this file and I'm telling it that the home URL is going to map to the home view. I'm going to give it the name user_home (typing). That's a mistake. User_home. So I like to prefix these names with the name of my app. Very well. So then there's two other steps I have to take to make sure that Django picks up my new app and my new view. First of all, of course, in settings. py I have to add my app to installed apps. So here it is, user. Very well. And then in urls. py I say, include. So this is a URL mapping where I say everything that starts with user (note, there's no dollar sign here-- so this only matches the start of the string), everything that starts with user/, will be mapped unto the URLs that are included here. So I'm including the URLs from user urls. py and mapping them onto user/. So, going back to this urls. py in my app, this view here called home will actually be found under user/home because this is included in the general URL configuration of my site. So let's test that. My home page is still here. Very well. Let's log in. (typing) I got redirected to the home page again and now if I go to /user/home, (typing) I see the message Hi, Reindert! Now of course, I want to be redirected after log in to this new home page so I'm going to change the login redirect url to user/home. Very well, let's test that. I'm going to log out. And I end up on the welcome page and then when I say log in (typing), I end up on my home page.

URL Mappings for Apps

So every app can have its own URL configuration and we can include our app under some prefix in the project URL configuration. Note that you should use a pattern that doesn't end with a dollar sign and a call to include instead of a reference to a view class.

Demo: Template Inheritance

So, let's have another look at our two home templates. Basically what I did is copy the entire template. The only change I made is that I removed the if and else stacks around the contents, so now the main home is always showing the welcome message and the user home always shows Hi and the user name. Basically all this other HTML and styling is copied and I don't really like that. I like to keep that in one place and use it for the entire site and we can do that with template inheritance. Let me show you how. In the general templates folder I'm going to make a new file and I'm going to call that base. html and I'm going to copy almost everything from here into the base HTML file. Very well. So load static files, etc., etc. And there's here the actual content and instead of putting that into the base html, I'm going to say block here (typing), so this is a block I'm going to call content (typing). I'm also going to say end block (typing). And as you can see, I did something similar here with block JavaScript and a little further up here with block styling. So let me explain to you what this does. If you go back to the main home page I can now remove all of this shared content here, so let's just remove this, and instead of using this I'm going to say, extend base. html. And let's remove this as well. So now this template inherits from base. html and that means it's going to use all of its contents, but it can differ if it wants where I use blocks. Now I don't want to change the styling block. Neither do I want to change the JavaScript block, but I do want to change the content block, so here in my home page I'm going to say that I want to override the content block (typing). So now the template engine will take almost all HTML from the parent from base. html apart from the content block, which will come from home. html. Now I'm going to do the exact same thing on the user home page (typing). Very well. So now we have two very simple templates for both of the home pages and we have one base. html which contains all the JavaScript and CSS styling, etc. Now let's test this again. Oh, apparently I have a syntax error. Oh, yes, of course, the block tag is call extends, not extend, so let's change that. (Typing) and here as well (typing) And now everything works again. If I log out I get sent to the welcome page and if I log in (typing) I go to the user home.

Template Inheritance

So you can use the extends tag to inherit from a base template. Now make sure that whenever you use extends it's the first tag in your template. Now then if you use template inheritance you can define blocks that can be overridden by child templates so you define a block, let's say block content, in the base template and you use the same block tag to override the content of that block in the child. Now if you want to know more about the template language,

Demo: Login Required

go here. Now there's one important bug here and that's when I log out I can still go to /user/home and although the user name is still empty I can still visit this page. So how do I prevent non-logged in users from accessing this page? Well, of course, Django has a very nice mechanism for that and it's called a decorator (typing). So I import the login required decorator and I decorate my home view with the decorator. So now I'm saying that only people who are logged in can have access to this view. And trying again, now I'm logged in. If I go to /user/home I'm presented with a login page. Then if log in, now I have access.

Demo: Showing Game Data on the Home Page

So, let's finish our page and show some of the games for the current user. As you can see, I changed my home view a little bit. First of all, I'm asking my game. objects, my manager, for the games for the current user. Now this is something I added myself and I'll show you in a moment. Then when I have the games for the current user in the my games variable, I filter them on the active status so that I get a list of active games. I also can get a list of finished games, so games that don't have an active status, by excluding the active status from my games. Then the list of active games get filtered a little more here by checking who's the next one to move and if that's me, so the user that sent a current request, then I'm putting them in waiting games, and otherwise I'm putting them in other games. Now these three lists, other games, waiting games, and finished games, I'm putting in a dictionary and this dictionary I'm calling context and I'm putting that as an argument here in the render method. So this dictionary containing three lists of games with the names other games, waiting games, and finished games, are going to be sent as to template context to the template. So, let's take a look at the new template for my home page, and there's several new things here. For all of the three lists, so the waiting games list, the other games list, and the finished games list, I'm using an include statement. Let me put returns in here so you can see the entire statement. It says, include tictactoe/game_list_snippet with header="games waiting for your move" and games_list=waiting games. Now this games list snippet is another template so with the include tag we can include a separate template. Let's look at this other template called game list snippet. Now this template doesn't return a complete HTML page. Instead it returns a snippet of HTML that only contains a list of games. It starts with a header tag and then with dif, which contains a for loop, or actually a for tag that will go over games_list variable, assign each game in this list to the game variable and then for every game in there it will make a link. By the way, that link doesn't go anywhere right now, and then display the game, and depending on the status of the game, there's a whole if, elif, elif, elif elif chain here. It prints the status of the game. So it might be You won! It might be a draw. It might be You lost! Or it might even be Waiting for opponents move, or Your turn, depending on whether it's my move or someone else's turn. Now after all of that I also have a span where I show the move count, so how many moves have been made in this game. All of that is inside an anchor tag. So what this snippet will do is with the for loop it will generate a list of anchor tags for each game and show the game itself, show the status of the game, and show the move count. (typing) Also notice that the for tag takes an empty tag as well, so in case there's no games in this list, it will say, No games available. So back to our home template. So, I'm including this snippet for each list, so I'm including it for waiting games, I'm including it for other games, and I'm including it for finished games as well. And as you can see, I'm passing the other template to parameters. I'm saying the header. For the waiting games the header should be, games waiting for your move and here it should be, other active games, and here it should be finished games. And I'm also passing it different lists so this game list snippet template will be called three times and will generate three different lists. And let's look at what that does. And now this is the new home page. I see three games. Games waiting for my move. Other active games, waiting for opponents move, it says, and two finished games, one where I lost and one that ended in a draw. Now I can't click these links right now because they don't point anywhere, but we'll add that in the next module. So for our last overview, first here in the view I retrieve all the data from the database and select the games I want to show in the various lists. Then I put these lists in a dictionary that will become the template context, and I sent that context to the template as an argument to the render method. Then there's the template. This will receive my three lists and I put them here as arguments to the include tag and for each list I include a game list snippet, which has a for loop and loops of all the games in the games list. Very well. One other thing I didn't show you yet is that I also have a block title here that says overview for my user name and if I look in my base template, here in the title tag, that's where it gets replaced, so the title tag here says block title and so this block will be replaced with this overview for the current user. Now of course, I can also log out and log in as someone else, so let's log in as Alice and here we see Alice's games. Here's the game I lost and Alice won. And here's our draw and she is also waiting for a move from Bob. And again if I log out and log in as Bob, (typing), we see Bob's games.

Demo: A Custom Manager Class

Now the last thing I need to show you is what I did to models. py in the tictactoe app because I added a custom manager class. You can see it here. It's called GamesManager and it inherits from models. Manager. And I did this because as you know, the managers represent the whole table and getting all games for the current user is something you do on a table level so that's not something that should go with the game class because the game class represents database rows and I want this to work with database table, so that's why I'm inheriting a manager and I'm adding the games for user method. And then here there's an advanced query that gives me all the games where the current user is either the first player or the second player. And then of course I have to make sure that this manager is actually set as the manager of the games class and I can do that by assigning it to objects, so I'm just saying objects is a games manager and now my game class has a manager of this new type and that's why in my views. py I can ask for game. objects. games_for_user because I implemented that myself.

The Template Context

So the data the template renders to HTML is provided by an instance of a class called RequestContext. For basic usage you can think of the RequestContext as a being a dictionary that maps names to objects and we've already seen how to send our data to a template with a dictionary and a call to the render method as shown here. Now because Django only allows limited execution of logic in the templates, this means all the data you want to display should be provided by you in the context. In other words, the code you need to find the model data and to do business logic and manipulations on the data should be in the view and model and not in the template. The context should simply hold the data that needs to be presented to the user and there should be no need in the template to do a lot of work on the data. For a complete reference of the template language, follow this link. And the second link tells you a little bit more technical stuff from the perspective of a Python programmer.

Templates: For and Include Tags

Now finally we've seen the for tag and it looks like this. You need to start with a tag that has the for and end key words and that works just like a standard Python loop. Now the HTML after the star tag is rendered for every item in the list. Of course, that can contain other tags as well. Don't forget to end the loop with the endfor tag. We've also seen the empty tag, which is optional and which you can use to print a message if the list is empty. Then we've also seen the include tag and you can use that to render one template from another template. So in this example we're rendering the tictactoe/game_list_snippet-html template and we're also giving it arguments with the with keyword and then some key value pairs. Now the with keyword and everything after it are optional so you don't need those.

Summary

So in this module we've added a user home page to our projects and on that home page we displayed some contents from the database. Now in the process of adding this page to our project, we've seen how to do authentication with Django, and we've seen some advanced features of the template language to do logic, to link to other URLs, and how to do template inheritance, and also we've seen how to do some advanced URL mappings.

Forms

Introduction

Hi, my name is Reindert-Jan Ekker and in this module we'll add some HTML forms to our application to enable user interaction. So in this module I'm going to add invitations where a user can invite another user to a new game and to do that we need to do several things. To start I'm going to add an invitation model and then we'll have to generate an HTML form from there. Then we'll see how Django validates user input from an HTML form. We'll see how to style the form. We'll see how to use the form in a view, and how to write the templates to display the form. Along the way we'll also see how to pass arguments to view methods.

Demo: Adding a HTML Form

So, I've started by adding an invitation model to our models. py in the tictactoe app and like you would expect, an invitation is from user, the one who initiates the invitation, and the to user, which is the user who gets invited, and we can add a message to say hi to the user, let's play a game, and I added a timestamp field as well, that will automatically be set to the time that the invitation is created. Very well. So of course now we have to do is syncdb to update the database schema and add our new table. But when I try that I get an error. It says, "one or more models did not validate. " The "accessor for field from user clashes with related field user invitation set. " Now, let's see why that is. We have two foreign keys here on the invitation class and as you know, the normal name of the related attribute on the user class would be invitation_set and we can't have two attributes with the same name, so that means I have to make sure that these two fields have related names on the user class that are different and I can do that by saying related_name, and in this case it will be invitations_received and in this case it will be invitations_sent. Very well. Let's try again. Oh, and there's a syntax error here. Let's fix that. And now our table tictactoe_invitation has been created. Now based on this model class, Django can generate HTML forms for us and it will look at the field definitions for each field of this model class and generate appropriate HTML input. Now we need to use a class called ModelForm this and we'll put that in a file called forms. py. So here in the tictactoe app, I'm going to make a new file called forms. py. Okay. And let me copy some code in here. So now we have an invitation for a class that inherits from ModelForm and to let it know which model we want to generate a form for, I have to add a class meta and that gets the model attribute that points to the class that we want to generate the model for and in this case it's invitation. And of course, don�t forget to import that class as well. So now we have a form. Of course, I need a view and a template to show this form to the user. So let's start with the view. So here's a view called new_invitation. Let's go over how this works. First of all, of course, only logged-in users can invite other users to a game. So here's a login require decorator. Now looking at the view code, there are three main flaws through this function. First, when the user goes to the URL to create a new invitation, for example, by clicking a link to the new invitation page. In that case the browser sends a get request. So this part, where we check if it�s a post request, is not true and we end up in the else part here of the if. We generate a new form from our new InvitationForm class and we send that to the template here in the context. Now let's see what this template looks like before we look at the rest of the view code. Of course, I have to make a new template here and let's call it new_invitation. html. And here we see all the common elements we're used to. It extends from base. html. It has a title block and a content block and it does some styling here with a diff, and then here there's the HTML form tag. And the action is empty, which means we will send the user input to the same URL form it's hosted at, but the method here is post and that means that when the form is submitted the browser will send a post request instead of a get request. Now here's a new tag we haven't' seen before called csrf_token and it's needed to prevent a class of a tag called cross site request forgery, or csrf. To prevent this form of attack, Django will include a hidden field with a special token and to generate that you need this tag. You need to include this tag with every form you write so don't forget that. Then here we render the form itself. As you can see, it simply renders the form variable and that knows how to generate its own HTML. And then finally we add a submit button. So when the user inputs some data the browser sends that data to the server in a post request. How do we handle that in the view? Now that's the general idea of this if statement. We can detect whether the user has filled in the form and submitted it by testing whether the method is post. If it is, we put the data from the request into a new invitation form instance. Then the form class can tell me whether the user data is valid with this call here and that will return false in case a required field is left blank or when an integer field contains tags that is not a number, etc., etc. But in case the form data is valid, we call save, which in turn will call save on the invitation model class. Then we redirect the user back to his or her home page. Now if the form is not valid we will simply end up here again so we will render the template again, but now the form will contain validation errors and show them to the user. Now of course, before we can try this form we need to add our view to our URL config. And this is a very simple URL invite and we name it tictactoe_invite and we map it to our new view method. So let's also include it in the top-level URLs and now we can try this form. Now I didn't make any links yet so I have to type in the entire URL (typing). Okay. And right now the form doesn't look very nice, but we'll fix that soon. So all of these inputs here have been generated by the form class, dropdowns for the form key fields, and text input for the message, but the submit button was added by me. Now if I simply submit the form without entering any data it will show some error messages because currently all fields are required and I didn't fill in anything. But when I do fill in everything (typing) I get redirected to my home page and that means presumably an invitation was saved to the database so, let's go back to the view. When I typed in the invite URL, the browser sent a get request and this was the code that was executed and we simply saw an empty form. Then when I pressed submit, the post request was sent, but form. is_valid returned false so we ended up here again and again I saw the form but this time with errors. When I actually filled out the form, again, the post request was sent and now is_valid returned true so the form was saved and I was redirected to my home page.

Using Django Forms

So, the ModelForm class will generate HTML for your model and the generated HTML will contain HTML inputs that are appropriate for each field. Now this is how you use ModelForm. You inherit your own class from it and you add to it a meta class with a model attribute where you specify the specific model you want to generate HTML forms for. Now if you want to know more about ModelForm, follow this link. So how do you use a form in a view? Well, there's several scenarios. First of all, when the user first visits the forms page and he sees the empty form, that will typically be a get request from the browser and in that case in the view you want to simply initialize an empty form instance and then of course, render the template. Now after the user submits the form you have HTTP POST data, or well actually that depends on what you do in the HTML form tag because you can specify the method there so you can also have the form submitted data as get, but let's just assume it submits as post. In that case you want to initialize your form instance from the post data and then you can call form. is_valid to check whether all the fields have been filled in correctly. Now if that's not the case, you will get validation errors and you want to render the template again including all the errors so that the user can see what he did wrong, but on the other hand, if everything is okay and all the fields have been filled in correctly, what you want to do is say form. save to save the data to the database, and then most of the time you'll want to redirect the user to some other page. Now if you want to know more about how to use forms in views, follow this link. And then finally, of course, you want to display the form in a template. You simply render the contents of the form object and that object will know how to render the right HTML. Now you shouldn't forget to also add the csrf_token tag. If you don't, Django will give you an error. Then of course, you have to surround your form with an HTML form tag and you to provide your own submit button to make sure the user will be able to submit the form.

Demo: Adding Styling to the Form with Crispy-Forms

Now there's several things about our new form we need to fix and let's start by making it a bit more attractive to look at. There's a very nice third-party package called Crispy Forms, which will make our form work nicely with Bootstraps version 3. Now of course, first I have to install the package with pip. As you can see here, I'm saying pip install Django-crispy-forms. And now that it says successfully installed, let's add the styling to our forms and that's actually really simple. First of all, I'll have to register Crispy Forms in settings. py in my installed apps. And then I need to tell it that we are working with Bootstraps version 3. Very well. Now all we have to do in our templates is load the Crispy Forms tag library and apply a filter to the form, like this (typing), and this will pass the form through a filter function from the Crispy Forms package which will transform the HTML in CSS to apply the Bootstraps styling. So let's check what the new form looks like. This is the old one. If I reload, this is the new one. Now, doesn't that look nicer? Even the error messages are nicely styled now. I did the same thing to the login form, so load in Crispy Form tags and applying the filter, and if we log out and press login, this is what the login form looks like now. And again, if I just submit, I get very nicely styled error messages. So now that the forms look nice, we can start fixing the functionality.

Demo: Field Options

Now, isn't it kind of strange that we can select the From user for an invitation. That should of course always be the current user. And I also think that the message fields should not be required because sometimes you'll just want to leave it blank. Now we can tell the ModelForm class that we don't want to see the from_user field by adding an exclude attribute, but now of course, this field won't be filled, so in the view the is_valid method here will always return false. We have to fix this by making a new invitation instance ourselves and we'll preset the from_user field to the currently logged-in user. And now we have to pass this instance to the form, like this. (typing)

Now because from_user has been added to the exclude options, the from class won't try to get that field from the submitted user data and it will use the value provided with this instance. Now by the way, let's go back to the forms. There a lot of other things you can do with a for meta class. If you want to know more about it, check out the forms documentation I linked in the slides. Now I also don't want the message fields to be required anymore. Remember that the ModelForm class validates its data based on the model class. Let's check out the invitation model. Now Django disallows null and blank values for just about every field type by default, but if I want to allow a blank message, we can enable that easily by adding blank=True to the message field. Now I should be allowed to submit the form without leaving a message. We can also add some nicer field names and messages to make the form a little more user friendly. The field names can be provided as the first argument or for key fields with the verbose name argument. And we can also specify a help text if we think a little explanation is necessary. So let's refresh our form and doesn't that look at lot friendlier? And you can also see the star is missing here so now the message is optional. Let's uncheck that. If I submit, I only get an error on the user field now. Now finally, let's fix some of the links in our base templates. This should be a URL tag that points to the tictactoe invite view. So now the user will actually be able to find the new invitation page.

Field Options

So we've seen several new things we can do with model fields. First of all we can add a verbose_name and that will show a more friendly display in forms for our fields. And you can add it as the first argument of a field definition or on foreign key fields you should use verbose_name as a key word argument. Secondly, we've seen you can also add a help_text will show up in the form as some extra text to help the user. I also mentioned that Django by default disallows null or blank fields and if you want to allow null fields, you will have to give the option null=True. If you want to allow blank fields, you can give the option, blank=True.

Demo: Showing Invitations in a List

Now let's show a list of received invitations on the user's home page. We'll start with the view. Now I'm using the many side of the relation here so I simply ask the user object to return all the received invitations. I could also have used the invitation manager class with the filter on the current user, but this seems more clear to me and we don't introduce a dependency on the invitation class. Now of course, we shouldn't forget to pass the list of invitations to the template. Now let's update the template as well. (typing) So in case there are any invitations, we check that with the if statement here. This code will add a list of invitations to the home page. Now let's see what that looks like. So currently I have no invitations. That means I have to add one. I'm simply going to invite myself. Let's say, Hi, let's play a game. And now I have an open invitation and I can click here, but nothing will happen because my links don't work yet. So, let's see how to add links that will enable us to accept invitations.

Demo: Accepting Invitations

So, going back to view. py I'm going to add a new view function. So, I've added a view here called accept_invitation and it takes an extra argument called pk, which is, of course, the primary key of the invitation we want to accept. Now I could write my own logic to try to retrieve the object with that primary key and show the user an error if the object doesn't exist, but since that is such a common pattern, Django provides a nice shortcut called get_object_or_404, which will either retrieve the object or show a 404 not found error if it doesn't exist. Then I check whether the currently logged in user is actually the user who was invited and if not, we tell them that they don't have permissions to view the page. And finally, we have similar logic as before. In case of the initial view of the page we have a get request and simply render the template and if the form was submitted already and the user accepted the invitation, we create a new game based on our invitation, and then we move the invitation from the database and then redirect for the page for the new game. Now if the invitation was not accepted, we simply remove it and redirect to the user home page. Now of course, I shouldn't forget to add my get_object_or_404 import here and I also shouldn't forget to import PermissionDenied error. So what does the template look like in this case? I have to add a new template called accept_invitation. html. And again, here we see some of the stuff we're familiar with. We extend from base. html. We have a title block and a content block. Here display some of the information from the invitation so this is the user that actually is inviting me. Here we check if there's a message, and if so, we show it. And here's our actual form. And as you can see, I'm not using a form class here. I'm simply using two buttons. One with name accept and the other called deny and in that way I can check in the view which one was clicked. So let's go back to the view. So here we check whether the accept button or the deny button was clicked. Now this game-related logic where we actually create a game is not implemented yet, so for now I'm just going to remove it and show a simple message instead. Now let's do that by simply returning an HTTP response, invitation accepted. And again, I have to import an HTTP response. Very well.

Demo: Named Groups

Now as always, the final step is to add the view to the URLs configuration. Now this here is a special piece of regular expression syntax called a named group and it mentions one or more digits. That's this part here. So the /d means digits and the + means 1 or more. And the braces around it, the parentheses here, make it a group. By saying question mark p here, we can give the group a name, and in this case the name is "pk. " So in the URL it's invitation/ and then some digits and then another /. We gather these digits by using this group and give them the name "pk" and that will be passed to the view as the pk argument. So, this argument here will come from the URL and that's how we will find out which invitation to show to the user. Now let's make links to invitations so the user can actually reach this view. And that's what we'll do here. So this is the name of the view, tictactoe_accept_invitation, and now I can pass it some key value arguments, and I'm going to pass it the primary key, which of course, is the idea of the invitation. So let's test this. If I click this now, I go to this URL, tictactoe/invitation/3. And this is actually the database ID of this invitation instance. And here we see the message I entered and if I accept it we will see, Invitation Accepted! And also the invitation will be removed from the database. So now if I go back to my overview, the invitation will be gone.

Named Groups in URLs

You can capture parts of the URL in a regular expression and give the captured data a name. You make a group by putting a expression in parentheses and to give the group a name you start with a question mark and a "p" and then put the name of the group inside angled brackets. A simple example would be this where we match one or more digits and call the group pk. Now the data you capture with named groups is passed to the view as a key word argument, so a view that handles the lost URL example would look like this. Remember need a request parameter and then we have the pk, which was captured by the named group. You could also pass arguments to views using the URL tag. In that case, the named value pair simply follows the name of the view.

Summary

So in this module we've added invitations to the project. Along the way we've seen how to generate HTML forms for models, how to do form validation, how to style forms, how to use forms from a view, and how to use forms in a template. We've also seen how to pass an argument to a view.

Making Moves

Introduction

Hi, this is Reindert-Jan Ekker and welcome to this module in which we'll add move making and finish the tictactoe module. So, we'll finish our tictactoe app and to do so we'll have to add move making to our apps functionality. We'll do this with a move form and to that form we'll add custom validation of the user input. We'll have to add a lot of logic to the application as well to do things like draw the board or decide who's move it is or won the game. While adding logic we'll learn about a Django principle called fat models. We'll also learn about some advanced template features along the way.

Demo: Creating a New Game

So, at the moment we can accept an invitation and the invitation gets deleted from the database, but we're not starting a new game yet, so let's see how to do that. First of course, I'll have to import the game model and then I'm going to stay (typing). So here I'm doing a function call called new game and the manager of the game model. And let's see how that works. I'm going to switch to models. py and here's our games manager and as you can see, there's no new game method here so let's implement that (typing). So basically this is a factory method for creating a new game and here we call the game constructor from an invitation so we take the first player and the second player from the invitation and we also assign the next to move fields that points to the player whose turn it is. Now of course I could also have overridden the init method for the game class and have it accept an invitation and generate a new game from there, but overriding in it on a model class is a very subtle thing to do right and this is actually the preferred way to do this so making a factory method on the manager class. I'm also going to add a new method on the game model and I'm going to call it get absolute URL (typing). Now this function in here, get absolute URL, is the default way to tell Django what the canonical URL is for a model instance. So basically if Django needs to know what the URL is to view a game, it will call this function and we're calling the reverse function and that basically does the same as the URL tag. It constructs the URL for the mapping with this name, tictactoe game detail. Now of course we shouldn't forget to import a referred function (typing). Now this get absolute URL method is used automatically by Django in several ways. For example, if I go back to the view, now here we can say (typing) redirect game. And so now this redirect function will see that this is a model instance and call get absolute URL on that model instance and then redirect to that URL. And let me show you something else. In the admin interface, if I go to game now, and I click on a game, here now we have link, view on site, and again that will call the get absolute URL on the game. Now this link, of course, only shows up if your model actually has a get absolute URL implementation. Otherwise it wants you up. Now if we click it, we get a no reverse match and that's because I didn't actually implement the view yet. So here I'm saying tictactoe game detail, but there's no URL mapping for that yet.

Fat Models, Skinny Views

So there's a Django based practice called fat models, skinny views. Basically it means that you put your logic in your model classes and keep your views short and simple. Now there are several reasons for this. First of all, it follows the DRY principle. It reduces the chance you have to repeat yourself when you use the same model in multiple views because you won't have to repeat the same logic in different views. You simply put it in the model and each time call the same function. It also makes your code more testable because breaking up logic into small methods on the model makes your code easier to unit test. Finally it makes your code more readable. By giving your methods friendly names you can abstract ugly logic into something that is easily readable and understandable.

URLs: Reverse and get_absolute_url

So one example of the fat model principle is the get absolute URL method you can define on your models so that a model instance can return the URL where it can be viewed. Now there are several places where Django will automatically pick up this URL for a model instance like in the admin user interface. Now typically when implementing the get absolute URL method you use a function called reversed, which will result view name into a URL. This works pretty much the same as the URL template tag and just like that tag you can pass it keyword arguments. So if you want to know which URL corresponds with some view, use the reverse function. It's the counterpart of the URL tag for use in your views in models.

Demo: Displaying the Game Board

So, when an invitation is accepted we're adding a new game and then we redirect to that game, and of course, to show the game we need a view so I added a very simple view called game detail, which takes a primary key and calls get object or 404. And of course, that will try to get an object with a given primary key and if there is none, it will throw a 404 error and then we render a game detail template and pass it the game in the context. Now looking at the URL mapping, we see a familiar pattern. The URL is game/ followed by primary key, which we capture with a group. We call that group pk, and that will be passed as an argument to the view. This is a view method name called game detail and the name of the mapping is tictactoe game detail. So that should be familiar. Now let's look at the template. First of all, there's a new template feature here. I'm adding my own styling with the styling block. As you can see, here's my new CSS rules and I'm using something called block. super here. And what that does is it includes the contents of the block from the parent so if we look at the base template, here in the styling block there's some CSS as well. We include bootstrap. css and of course I want to include that in the child template as well. So here, before I add my own css, I add the css from the parent and then add to that and to do that we use the block. super variable. Then I want to display the game, of course. And here's my content block and I want to generate a table because that's the right way to represent the tictactoe board. Now you might want think, well, tictactoe is a 3 x 3 game board so we can simply say for x in range in 3 and for y in range 3. But actually, Django won't let us call the range method from a template because it restricts the kind of logic you can write in a template. So instead of doing that kind of logic, I'll have to call a method on the game model and that does the logic for us. So again, there we have the fat model principle. So let's go to the models. py and I'm going to start with showing you that I added some constants here and an X for a first player move and an O for the second player move and a board size. And then if we go to the models, here's our as board method and it generates a nested list. So it's a list with a list of squares for every line and then we can go over the moves that were made and depending on whether the move that was made was by the first player, we add an X or a Y at the coordinates of that move. So that's kind of simple, but of course, our moves didn't have a by first player field yet, so I have to add that and that's a Boolean field. It will be true when the move was made by the first player and false otherwise. And I also added a time stamp field so we can know what the order was of when the moves were made. Then I added a meta class to a move model and the meta class can be used to add some options to your model and in this case I added the get_latest_by option. That will tell Django what fields to look at to determine what the latest and earliest moves are. So in this case I say get_latest_by is time stamp and Django will know that to see what the latest move was it can look at the timestamp field. And that way, on the game model, I can add a last_move method that will return the latest of its move set. So this latest is a function that's defined by Django on every model manager and it will use the get_latest_by field. So now with these two lines of code here we have defined a last_move method on the game class that will look at the timestamp to see what the last move was. And then there's some more logic here on the move class, I define something called player, and depending on the value of the by_first_player Boolean, it will either return the first player object or the second player object of the game, so it will return actual user instance. Now all of these are examples of the fat model, skinny view pattern so we're adding logic to our models to make writing our views and templates a lot easier. Okay, so let's go back to the template. Here I have the game. as_board. We now know it's a nested list so it loops over the lines in the board and for every line it adds a table row and then for every square in that line it adds a table cell. And then it gives that table cell css class and adds the string that's actually in the square so that can be the empty string or an X or an O. Now of course, I changed the move class I added to fields, so I dropped the database and did a new syncdb, and here's my home page and let's look at the nice game. Here's a game where nine moves were made. If I click it, this is what the board looks like. So we can look at the source of this page and here we have the table that's generated by my template. So for every line of table row and then for every square the table cell that holds either an X or an O or nothing at all, but we don't have empty squares so we don't see that right now. So now we know how to display the board, we'll have to see how to make moves.

Templates: Lookup

Now let's have a short look at the template language again. A lot of the times you will use dot notation when you use Django templates, like when you say user. name or game. move_set, and although it looks like a normal Python attribute lookup, it isn't. It does a lot more than that. When you use a dot for an objects attribute, the template system tries four things until it finds a value. First it will assume your variable is a dictionary and tried to use the part after the dot as a key. Then it tries to look up the attribute as a normal attribute for an object and if that fails it tries to call a method. Now if that fails too, it tries to do an index lookup. So, what does that mean? Well, basically it means it doesn't matter if the thing you're looking up is not a real attribute on your object, but a method or an item you a list or a dict. You can simply use the dot notation and Django will figure it out. So if we have a list, a list of five will retrieve the item with index 5. And if we have a dict, we can use the dot notation to do a key lookup. And we can even call methods this way. For example, we can ask for the count method on a query set like the move set on a game. You should note though that you can only call methods without arguments this way. So you should be aware that the template language doesn't allow you to call a method and provide it with an argument so it means that you cannot pass the current user to a method in the game that tells you whether it is his move. And also you can't call the range function to generate numbers. Now this isn't just an inconvenience, it's by design. You should really do that kind of that stuff elsewhere. Templates should be relatively simple and deal with generating output. They're not for doing intricate logic, that's why we have fat models, or if you need to loop over a range of numbers, generate that range in the view and put it the template context.

Templates: block.super

We also saw a new template feature and that's block. super. It's a variable defined within template blocks, which will contain the contain the contents of the parent block, so that way you can override a block from the parent while still including the parent content. And we also saw that we can add a meta class to your model to add some extra options.

Models: The Meta Class

For example, you can add the meta class to set the default ordering for your table. Or with the code in the example you can tell Django which field should be used for the latest and earliest methods, or you can do things like set the table name, which can be nice if you have to use and existing database. And there's a lot more. As always, here's a link you can follow if you want to know more.

Demo: Making a Move

So let's start implementing move making functionality. First of all, in case a user visits the detail page for a game, but it's his turn to play in that game, we'll have to allow him to make a move. Let's start with testing that it's actually the user's move. (typing) So here I say, if game. is_users_move, and I pass the current user, and then I redirect to a new view. Now of course we need to implement this is_users_move method on the model so let's look at the model. I already put it here and we do two things here. First we check that the game is actually active because we don't want to make moves on a finished game and then we check that self. next_to_move so the player that is to make the next move is actually the user that was passed in. So I think it is better to put this sort of logic on the model class than in the view, because it keeps the view simple and this is actually the reasoning about the model so I think this kind of logic should be in the model class. So now of course we need to add the do_move view that we're redirecting to. (typing) So right now this view is pretty much the same as the detail view. The only difference is that when it's not my turn we raise a PermissionDenied error, and of course, we need to map a URL to this view too (typing). Now this is very similar to the game_detail view. The main difference is that it ends with do_move and of course, it maps to the do_move view and not to the detail view. So finally, let's add a template (typing). So what I do here is I simply inherit from the game_detail page. Right now the only difference is that we change the mouse pointer and the color of the square on the board when the mouse hovers over the board, so I still use the styling from the game_detail page with block. super here, but I add some styling of our own too. Now of course, for user input I want to use a form, and looking at the game_detail page, I want to put the form inside this div here. So I need to add a block here where I can put my code from the do_move template. (typing) So now let's add the form here (typing) in the do_move template (typing). Now currently we don't actually have a form in the context so only this submit button here will show. Now let's test this. And I see here an error occurred. And let's look at the error. Here it says indentation error, and apparently, this, by the way, this is the log from the run server command so by scrolling up in the log I can find what the error is and here there's an indentation error (typing) so let's reload. Very well. Now if I click on a game where it's my turn, I get redirected to do_move and here's a submit button. Very well. So right now the only difference between the do_move page and the normal game page like here, is the css and it should highlight squares when my mouse hovers over them, but it doesn�t do that and that's because I still miss some styling (typing) so let me add that here in a square. I'm going to add the empty class if the square is empty. And now if I click on a game where it's my move, we see that empty squares are highlighted and that my mouse turns into a hand. Very well. So the next step is to add a form class for making a move so we'll actually use this form tag here. (typing)

Demo: Custom Form Validation

So, let's go to forms. py and add another model form class for entering a move (typing). Now I'm going to leave out some fields. For example, the game and the by_first_player Boolean, as well as the comment. Now I'm leaving out the game field and by_first_player field because they should be set in the view and not by the user and I'm not going to implement game comments for now so I'm leaving that out as well. Now as you know, the view will call a method called is_valid on the model form, which will trigger validation of the user input and to prevent cheating we need to check whether the input actually is a valid move by writing some custom validation code. Now there are several ways to add custom validation. If you want to do validation checks on specific fields, you can specify validators on those fields, so let's add some bounds on the X and Y fields for moves. And we'll do that where those fields are defined and that's in models. py. So here's the X field and what I can do is add an option called validators and in this list I should add a list of the validators I want for this field. So first of course, I have to import some validators (typing) and now I can say that I want the X field to at least have a value of zero and at most a value of the board size, minus one. And let's copy that to the Y field because the same is true for the Y field.

So this code tells Django that these fields must have at least a value of zero and at a max a value of two, which is board size minus one. So to test this, first of course we must finish the view. So here we inner slice the template context with the current game and in the case of a get request, here we add an empty move form, but in case of a post, we fill the form with the data from the user and then validate. Now I'm not saving anything yet, we'll do that later, but you should notice how I pre-fill the move instance again with a new move instance that holds the current game because that's not provided by the form. Now of course I shouldn't forget to import MoveForm as well, and of course I should import the move class as well, so let's test this. Here's a game where it's my move. And let's input an illegal value and now we see a very field-specific error here. So this error is specific to the X field and this error is specific to the Y field and the errors are coupled with the fields themselves because we put the validators on the model fields. Now we also need to check whether the coordinates points to an empty square and that's a check that's not specific to a single field. We can add this validation to the ModelForm class by overriding the cleaned method. So let's see how to do that. I'm going to forms. py and I'm going to override the cleaned method. Now Django validators raise a validation error with a message when something's wrong and the cleaned data attribute holds the data after converting it from the user input string to a Python type like an int and at this point it's also already done the first validation steps so I get the X and the Y values from the cleaned data and then I check that this square is actually empty with the is_empty method. Again, this is a method on the game model that checks whether any moves were made at this square. Finally, the cleaned method has to return the cleaned_data dictionary. So let's test this again. Of course, first I should import some stuff (typing). Of course, we need to implement the is_empty method as well, so let's do that here (typing). So what I do here is I take the set of moves that were made in this game and then I filter them for any moves that were made at X coordinate X and Y coordinate Y and then it calls exists and that will return true if any of these moves exist. Now to test this I have to log out and I'm going to log in as Bob (typing) and let's look at this game. And now I can try to make a move at this position where a move was already made so the coordinates will be 1 and 1. And it tells me this is an illegal move. So that shows us how to do some validation. Let's review that and then finish up and finish the game.

Custom Form Validation

Because the ModelForm class is based on your form and so it knows the kind of data your model expects, it can ultimately do a lot of validation, but sometimes you just need to add some validation of your own. No in case you want to check the input on a single field, you can use a validator, which is basically a method that calls a validation error when something's wrong. Now Django comes with a default set of validators, which you can assign to a field by adding a validator option to the field. You can even write your own custom validator, which is very simple. If you want to know how, follow the link shown here. But sometimes you need to do validation that's now restricted to a specific field and for that case you can override the cleaned function on the form class. Now when doing that there are some more things you need to take into account, like how to handle the cleaned data attribute. If you want to read more about it, here's a link for you.

Demo: Finishing Up

So let's finish our tictactoe game app. Now I'm going to start with making the form hidden and I do that like this. (typing) So basically form. x will generate HTML for the X field on the form and form. y will generate an input for the Y field and we can call the as_hidden method on those fields to generate HTML for our hidden input so we won't see these input fields and I'll also make the submit button hidden (typing). So now I'm going to add some JavaScript that detects a click and fills in this form and then unhides the submit button, so the user won't be able to see these two fields. He'll only be able to click submit. (typing) So, here we have a block JavaScript and again we include the JavaScript from the parent template and then here I have a script that will handle a click on an empty square. And it takes the X and Y coordinates from the HTML elements, actually from a data attributes on the HTML so that means I need to set these attributes in the HTML. I'll do that in a moment. Now after detecting the click it will put the X and Y coordinates in the input fields that are generated by the form and then it will remove the hidden class form and submit button, so then the form will be filled in and the user will be able to click commit. (typing) Now of course, now I have to generate the HTML attributes and I'm going to do that in the parent view, in the game_detail view, and then here on the table row I'm going to say data-Y, which is data attribute, and now I want to generate the actual Y coordinate here (typing). Inside a for loop, a counter variable is defined and I'm using that to set this attribute to the Y coordinate of the current line so for the first line this will be zero and the second line this will be one and after that it will be two. And I can do basically the same thing for a cell. So for a table cell I'm going to say data-x and again now this counter counts this in our for loop. So again this will be the X coordinate of the square. So let's test that this JavaScript now works and now if I click here, for example, a cross is put in and the submit button appears and we can also inspect the elements and here are my hidden inputs, X and Y, and as you can see, the values for the current coordinates have been filled in by my JavaScript. So again, suppose I click here, and I inspect again. Now we see new coordinates. So my JavaScript basically works very well. Now I'm going to add an idea attribute to a cell if the last move was made at that position. So basically what we do is we check whether the current Y coordinate is equal to the Y coordinate of the last move and the same for the X coordinate, and the X coordinate of the last move. And if that's true, we give this square the ID of last move and as you can see here, that cell will get a different background color, so that way we can highlight the position of the last move. Now finally, let's change the X's and O's here into some nicely styled icons (typing). Very well. And let's test again. And now here we have a game where there's nice icons and the last move has been highlighted and we also interactively highlight empty fields when I hover over them. And I can click and submit. Very well. Now as you can see, this move isn't actually done because the view isn't handling that, so let's also finish the view so we actually process the move. (typing) So here instead of making the instance myself, I'm going to use the factor method on the current game (typing) and I'm also going to save the move, of course, if the form was valid (typing), and then I'm going to have to update the game state (typing). So let's go to the models again 9typing). And let's start here with toggle_next_player. It simply switches the player so it's currently next move is the first player, we switch to the second player and otherwise we switch to the first player. Then there's a get status function, which will check the board state after the new move. It will check whether we have three X's or O's in a row and it will also check the diagonals and then return the correct new status so that will be an F if the first player or an S if the second player wins, a D for a draw, and an A for active instead, and of course, this is a bug. So if I go up all the way, you can see that here in the game_status_choices we have defined these values, so A is active, F is first player wins, etc., etc. You might remember this from the start of the course. So then in update_after_move, first I call toggle_next_player so I switch to the next player and then I determine my status after the move. And then finally here's the create_move method and it will return a new move with the game instance pointing to self and it also determines the correct value for the by_first_player Boolean. So going back to the view, again, first we make a new move based on the current game and we fill in the rest of the form into the MoveForm. Then we check whether the form is _valid, and if it is, we save the move and we update the game after that. Now of course, we shouldn't forget to save the game is well. By the way, there's a bug here. I need to save this where I save the information from the form, so this will cause the information from the form to be saved to the database and will return the move instance after that we save that in the move variable. So let's test this again. Here's my game and if I click here I can click submit and now we see that the last move was highlighted and the move was actually made.

Summary

So, we finished the tictactoe app and right now we can add a game and play a game and we'll see what the status of the game is, moves will be handled and validated, etc., etc. And to get to that point, in this module we started with making a move. We added the functionality to make a move with a form and some custom validation. We also saw about fat models and I put a lot of the game logic in the game class and one of the things I put in there was the get absolute URL method that tells Django what the URL is for one of the model instances. We also came across a lot of new template stuff. Among other things we saw how to use block. super and we saw some new usages of the form in a template and I told you how field lookup works in the template language.

Odds and Ends

Introduction

Hi, I'M Reindert-Jan Ekker and welcome to this module of the Django course in which I will conclude this introduction to Django with mentioning some odds and ends and with that we will come to the end of this course. By now, you know all the basics of Django. You know how to work with models, templates, and views, how to map URLs to views, and how to work with HTML forms. Now's there's three last three last things I'd like to discuss with you in this last module. First of all, I want to tell you about generic views. They are an important, although somewhat advanced advanced Django feature that'll help you keep your view code as short as possible. Secondly, I want to show you a little bit about debugging your Django code, and then I'd like to show you some pointers about everything else there is to know about developing Django applications and where to go from here.

Class-based Views

All the views we've seen so far have been functions that accept a request object and return a response object. We call these function-based views. Class-based views are very similar, but the difference is that, as the name suggests, they're classes and not functions, which means that because they are classes, functionality can be inherited and extended. Now I'm not actually going to show you how to write your own class-based views, but if you want to know how to do that, it's quite simple and you can read about them here. Now instead of showing you how to write a class-based view, let's see a nice application of them in a demo.

Demo: Class-based Views

Now we've seen several examples of pages where we load data from the database and show that to the user, or show a form to edit the data from the database, etc., etc., and in a way we're repeating ourselves here. For example, things like getting something from the data base and putting it in a template context, those are things we do for many views again and again. And the same is true for handling forms so checking if we have a POST request, initializing a form, validating it, saving the data, etc., etc. All of these show up just about every time we need to handle form data. Now Django wouldn't be Django if we didn't have something nice to avoid repeating ourselves. We can add a class-based view that inherits from another view that already knows about getting data from the database and putting it in a context, etc., etc. You see, the power of classes over functions, as I said, is that you can reuse the functionality of a super class in a child class. So, in case of class-based views we can reuse the functionality of a view class. Let me show you a simple example by adding a view that lists all games in a database. Django provides a nice generic view class called ListView and we can use it like this (typing). Now this class inherits from the generic ListView and we set the model for the ListView to our game model class and the ListView is a Django generic view, which will get all game objects from a database, put them in a template context, and render a template. So let's go ahead and map our new view to a URL. Now one difference is that we cannot refer to a class-based view as a string and instead we need to import the view class (typing) and now we can refer to it by name. Now the URL mapping expects a callable function here so we call the as_view function on the view class, which will return a view function for this class. So there are two things that are different when you're mapping a class-based view instead of a function-based view. Now if you visit this URL in the browser (typing) now we get a TemplateDoesNotExist error for a template called game_list. html. The default template name for a generic list view is model_list, so that's why it looks for a template of the name game_lists. We can simply add a template (typing) and this is a very simple template that generates a list of game and note that there's a variable here called object_list, which will hold the list of objects retrieved by the generic list view. So refreshing the page we get a list of all the games in the database. So going back to the views we can get the ListView by extending the generic ListView class like this, and as you can see, the view code is very, very short. So that's very nice. You get a lot of functionality for almost no code and there's other generic view classes as well, like a detail view that will retrieve a single object from the database, or a template view that simply renders a template and nothing more. And there are even some more powerful view classes that work with forms. Let me show you how to use one of these classes to enable user sign-up for our site (typing).

Demo: Adding User Signup

Now I'm moving in to the views module of the user app and let me show you another view that inherits from a generic view. It's called signup view and it will allow new users to sign up for the site and create a new user account. Now as you can see, this inherits from a generic CreateView. That's a class that uses a form to create a new model object and then validate it and save it to the database. Here I'm telling it that we want to use the user creation form class that's provided by Django and that will generate a form for creating a user that allows us to enter a user name. Now of course you can also provide your own form class here. Then I set the name of the template to user signup and finally I have to configure the success URL, which is the URL where the user will get redirected after the form has successfully been submitted and the new data has been saved to the database. Of course, I also imported everything I need here at the start of the file. So basically this view uses the functionality from the CreateView generic view class to show a form and then create a new user. Of course we also have to map this in urls. py. Here it is. The URL is signup. And again I use the class-based view by name with the as_view function, and don't forget to import it because that's necessary for class-based views. And then finally I have a quite simple template here. Basically there's a dif and a create and account header and then here's the form with the submit button and that's about it. So then if I go to the home page and I click signup, this is what I'm presented with. This form gets generated by the user creation form class, which is provided by Django and then the CreateView handles validation so if I click submit I get nice validation messages and if we fill out the form correctly (typing), the view now has created a new user and we can log in (typing).

Generic Views

So we've seen that we can create views with common functionality with almost no code when we make use of Django generic views. Some examples would be TemplateView, which will let you show a template, DetailView, which will retrieve a single object from the database and show it to the user, and ListView will do the same for a list of objects. And then there's also the generic form-editing views that will generating a form and logic for a model class. For example, CreateView will let you create a new instance of your model and UpdateView will edit a model instance and DeleteView will remove an instance. And of course if you want to know more about how to make use of generic views, go here.

Debugging Django

The second topic I want to talk about is how to debug your Django code. If you're a Python developer with some experience, you're probably aware of the common technique of importing pdb, which is the Python debugger, and then calling set trace to check what's happening in your program and that's something you may very well do in your view code. If you insert this in a view and visit the corresponding URL, you can step through your code and see what happens, but there's a much more powerful way to debug your Django project and it's available as a third-party package, called the Django-debug-toolbar. It will show you lots of extra information and let's see how that works.

Demo: The Django Debug Toolbar

Like with other third-party packages we saw in this course, I'm going to start with installing the Django-debug-toolbar with pip (typing). Very well. That's all you have to do. And in the settings for your project you add the debug toolbar to the installed apps. And that's basically everything you need to do. And now when you go to your site, there's something new injected here in all your pages. That's the debug toolbar's site bar. Now this shows a lot of information that's helpful when you're debugging your project. For example, here on the SQL page, you can see all the SQL queries that were made while generating this page and it also includes for every query the time that it took to retrieve the data, so this can be very helpful when you're debugging your model or when you have SQL performance problems. Another nice thing is the templates page here, which will show you the template context for your template. So here we can see all the variables that are in the current template context, as well as in the parent templates context. So basically you can inspect all the data that's available to your template. You can also look at your HTML headers and the settings for your project and the general time that it took to generate your page. So basically every time I start a new Django project, I install the Django-debug-toolbar because it's a very powerful tool that will let me inspect everything that happens when I generate my pages.

Resources

So where to go from here. Well, first of all, the authoritative source of all Django information is of course, the site of the Django project at djangoproject. com Another very nice resource is this site, which will give you a very nice Django cheat sheet. Then there's a very nice site at ccbv. co. uk, which has a very nice overview of class-based views and how they work and as I said before, those are really quite an advanced topic and they're not always very easy to wrap your head around. Here at djangopackages. com there's an overview of all Django packages you can download. It's very nice to see all the different kinds of functionality available to you just by installing a third-party package. And finally, there's djangosnippet. org, which has a very nice collection of Django code snippets.

Summary

And with that we come to the end of this course. In this last module we've seen a little bit about generic views, how to install the Django debugging toolbar to help you debug your project, and finally I showed you a short list of nice Django resources. So in this course you've seen all the basics of Django and now you're ready to start making your first Django project. Thank you for watching. I really hope you enjoyed it.

Course author

Reindert-Jan Ekker

After studying Artificial Intelligence, Reindert-Jan has worked in many different roles in the software development life cycle, from tester and developer to Scrum Master. Then he decided to go...

Course info

Level	Intermediate
Rating	★★★★☆ (572)
My rating	★★★★★
Duration	3h 9m
Released	31 Jan 2014

Share course