**Prerequisites:**

Let's talk about what you need to know to take this course. For starters, you need to know Python. You need to know the basics of Python, as well as object-oriented programming concepts such as classes, inheritance, polymorphism, and so on. You should also know the basics of relational databases, so you should be familiar with concepts such as tables, columns, primary and foreign keys, relationships, and so on. If you need to refresh the fundamentals, I have a couple of beginner-level tutorials on my YouTube channel, as well as comprehensive courses on my website codewithmarsh.com. I'll put the links down below in case you're interested.

**What is Django?**  
So what is Django and why is it so popular? Well, Django is a free and open source framework for building web applications with Python. It's not the only web framework for Python, but it's the most popular one. Because it helps us build a website in less time with fewer lines of code. That's why a lot of companies like YouTube, Instagram, Spotify, and Dropbox use Django in their tech stack. Now, Django is what we call a batteries included framework, which means It comes with a lot of features out of the box, so we don't have to code them from scratch. For example, it gives us an admin interface for managing our data, which is a huge time saver. It also has an object relational mapper that abstracts the database so we can query or persist data without writing a lot of SQL code.

It also comes with an authentication package for identifying users. It also has a package for caching data and much, much more. So since Django offers all these amazing features, we can focus on our application and its requirements. We don't have to reinvent the wheel and code all these features from scratch. Now, once in a while, you might come across someone like our popular superstore developer, John Smith, who thinks Django is a piece of crap because it's old and bloated with so many features. He used it, hated it, and then rewrote his entire project with a new shiny framework that is faster. Well, saying a framework is better than Django because it's faster is kind of like saying a Ferrari is better than a truck

because it's faster. What if you want to move something? You're not going to use your Ferrari for that, right? So a wise software engineer doesn't pick up a framework merely based on its performance. There are so many other things you need to take into account like the maturity of the framework, how stable it is from one version to another, its learning curve, the size of the community and so on. Django has been around for a long time and it has a huge community. So there are tons of Django packages built by the community that you can reuse in your applications. Also, If you get stuck, there's always someone out there to help you. If you want to hire people, there are tons of people to choose from.

So these are the benefits of having a huge community around a framework. Now, regarding Django features, yes, Django comes with a lot of features, but you don't have to use or even learn all of them, because all these features are optional. So you can use the features that make sense for your own application. So, in my opinion, debates about the best framework in the world are useless. We all like different things and Just because someone doesn't like Django, it doesn't mean it's bad. A lot of companies are using Django and are looking for Django developers. Here in the US, the average salary of a Django developer is just over $117,000 a year. So if you like Python and want to get into web development, learning Django is a good investment for your future.

**How the web works?**

So you learned that Django is a framework for building web applications with Python. Now let's talk about some of the fundamental concepts you need to understand to build web applications. Let's say we're going to build an online store and publish it at moshbuy.com. Now this website is going to have two parts or two applications. A frontend and a backend. The frontend is the part that is loaded inside a web browser on a client machine. It's the part that the user sees and interacts with. The backend is the part that runs on a web server and is responsible for data processing, validating business rules, and so on. Now let's imagine that Alice wants to visit our website. So she points her browser to moshpy.com.

This address is also called a URL, which is short for Uniform Resource Locator. It's basically a way to locate a resource on our internet. A resource can be a webpage, an image, a video, a PDF, and so on. So Alice types moshpy.com in her browser and presses enter. At this moment, her browser sends a request to the web server that hosts our website and says, hey, Alice wants to see the homepage. So the web server should take this request, process it, and return a response back to the client. This data exchange is defined by a protocol called HTTP, which is short for Hypertext Transfer Protocol. It defines how clients and servers can communicate. So this is the big picture. As Alice navigates our website, for each page, her browser sends an HTTP request to the server and receives an HTTP response.

Now, as part of building the backend for this website, we need to decide how we're going to respond to clients. One option is to generate the requested page on the server and return it to the client. We use HTML for that. HTML is short for hypertext markup language. It's a simple language for representing web pages and their content. Every web page we have seen on our internet is built using HTML. So one option is to generate the page on the server and return an HTML document to the client. The other option is to return only the data needed on the requested page and have the client generate the page. So instead of putting a complete page or a complete HTML document in an HTTP response, we only return the data like the list of products.

Now, what is the difference? Well, if we push this responsibility to the client, we can free up the server so it can serve more clients. Our application will be more scalable. That's why over the past few years, this approach has become more trendy and is now considered the industry best practice. These days, we have tools like React, Angular, and Vue for generating web pages on the client. These are all client-side tools that are used by front-end developers. In contrast, we have server-side tools for building backends. Django falls in this category, so you should not compare Django with, let's say, React. You may compare Django with other server-side frameworks like ASP.NET Core, which is used by C-Sharp developers, Express, used by JavaScript developers, and so on.

So if we push the responsibility of generating web pages to the client, the server essentially becomes a gateway to the data. On the server, we can provide endpoints that the client can talk to to get or save various pieces of data. For example, we can provide one endpoint to get the list of products and another endpoint to get the list of orders someone has placed. Now, all these endpoints together represent the interface that clients use to talk to the server. In technical terms, we say the server provides an API or an application programming interface to clients. This API is essentially like the buttons on a remote control. All these buttons together represent the interface or the API we use to interact with a TV, okay?

So in this course, our focus will be on using Django to build an API for our online store. Client applications can use this API to get or save the data. How these clients are built is irrelevant here. We can use React, Angular, or even plain JavaScript. That falls under front-end development, which has nothing to do with Django. Once we build this API, if you know front-end development, you can always build a client app that talks to this API. Alright, now that you understand the big picture, we are ready to set up our development environment. And that's what we will do next.

**Setting up the Development Environment:**

Alright, let's make sure you have the right setup before we start coding. The first thing I want you to do is upgrade your Python to the latest version. So head over to python.org slash download and download the latest version of Python for your operating system. Once you do that, then open up your terminal window and run python dash dash version to make sure you have upgraded your Python correctly. Now here we see Python 2.7 because on Mac we have two different versions of Python running side by side. We have the old Python 2 which is now deprecated, and we also have Python 3. So if you're using a Mac, you have to run Python 3 dash dash version. But if you're on Windows or Linux, you have to run Python dash dash version.

Okay? So here I'm using Python 3.9.5. Great. Next, we're going to use pip or pip 3 if you're on Mac to install pipenv. I talked about pipenv in my Python course. It's basically a dependency management tool for installing our application dependencies in virtual environments. This way, our application dependencies will not clash with other applications dependencies. Okay, so let's go ahead and install this. Now in this course, just like my other courses, I'm going to use Visual Studio Code or VS Code as my editor. It's just my personal preference. If you have a preference for a different tool like PyCharm, that's totally fine. Having said that, I highly encourage you to use VS Code, because throughout the course, I'm going to show you a lot of shortcuts and time-saving tips that will not apply to your editor.

In case you don't have VS Code, you can get it from code.visualstudio.com. Now once you run it, I want you to go to the extensions panel over here, and search for Python. Make sure to install this extension, because with this we get features such as IntelliSense, debugging, code navigation, and so on.

**Creating Your First Django Project:**

Alright, now that we have installed all the necessary tools, let's create our first Django project. So here in the terminal window, I'm going to go to my desktop. You can go anywhere on your machine, it doesn't really matter. Now, we're going to create a directory called storefront. That is the name of our project. Next, we go inside this directory and run pipenv install django. So we're going to install django using pipenv inside a virtual environment. Now, let's go ahead. All right, now take a look. So over here, you can see that pipenv has created a virtual environment for this project. Down below, you can see the path to this virtual environment. If you're on Windows, you're going to see something like c drive, backslash, whatever.

Now, if you're on a mac and using z Shell, which is this fancy, colorful terminal window, we can hold down the command key and click on this path to go into it. Otherwise, we can just copy this and use the CD command to go inside this directory. Now take a look. So here's our virtual environment. In this environment, we have this bin folder that contains a bunch of binaries. For example, we have Python, pip, Django admin, and so on. So back to the terminal, pipenv created a virtual environment and installed Django inside that environment. But in addition to this, pipenv also created two files in this directory, pip file and pip file dot log. So I'm going to open this directory using Visual Studio Code by running code period.

If this doesn't work on your machine, just drag and drop this folder onto VS Code. Alright, so here's our pip file. This is like package.json for JavaScript projects. So in this file we can see that our project needs this particular version of Python and under packages you can see the packages our application is dependent upon. So here we have Django equals asterisk. That means any version or the latest version of Django. If we installed a particular version of Django, we will see that version here. Now back to the terminal window, we need to activate this virtual environment. So we'll use the Python interpreter inside this virtual environment, not the one that is installed globally on this machine. To do that, we're going to run pipenv shell.

Okay. Next, we're going to use django admin to start a new project django admin is a utility that comes with Django. So if you run it, we can see all these commands that we can use to work with django projects. As we go through the course you will become familiar with these commands. In this lesson, we're going to use django admin start project, and we're going to call that project store front now back to vs Code. Django admin created this directory for our project and Inside this directory it created another directory that is the core of our application in this directory We have this class which I'm going to talk about in a second But before I do so let's get rid of this redundancy because we have three Storefront directories the top one is the one that we created earlier in the terminal and then we have these two other directories one for the project one for the core of our application So I'm gonna delete

this project directory, now we are back in the previous step. So we don't have a Django project at this stage. Now, back in the terminal, let's bring up the last command, but type a period at the end. This tells Django to use the current directory as the project directory. So Django is not going to create an additional directory for our project. Okay, now take a look. So, we have this directory, which is the core of our application. In this directory we have this init file, which defines this directory as a package. We have this settings module where we define our application settings. We have the URLs module where we define the URLs of our application. And these two other modules that are used for deployment.

For now, don't worry about them. Now, next to this directory we have this file manage.py. This is a wrapper around Django admin. So going forward, instead of Django admin, we're going to use manage.py. The reason for this is that manage the PI takes the settings of this project into account. Let me show you what I mean So now that we have a project we want to run a web server. So another command that Django admin provides is Run server. However, if you're on Django admin run server We get an error of saying request that setting debug but settings are not configured because at this point if Django admin doesn't know about the settings of our project. So now that we have a project, instead of Django admin, we're going to run Python manage.py.

Now look, we see the same commands that Django admin provides. So here we can say, Python, manage.py, run server. Now optionally, we can supply a port number. If we don't supply this, by default, this will use port 8000. But you might have another application running on port 8,000. In that case, you might want to explicitly set the port number. So let's go ahead with that. Now we don't get an error because manage the PI knows about the settings of our project. Now here we get this warning saying you have 18 unapplied migrations. Don't worry about this yet. We'll talk about this soon. So if you look over here, you can see that Django has started a development server at this address. So we can copy this and paste it into browser.

Or if you're on a Mac and using Z shell, you can hold down the command key and click this. Alright, beautiful. Our first Django project is running successfully. So let's move on to the next lesson.

**Using the Integrated Terminal in VSCode:**

So far, I've been using a separate terminal window, but VS Code also has an integrated terminal window that is easier to use. But to use that, we have to properly configure it to use the Python interpreter inside our virtual environment. Let me show you how to do this. So back to our terminal window. First, we're going to stop the web server by pressing Control and C. OK? Now, back in VS Code, on the top, under the View menu, look, we have this command palette. The shortcut on Mac is Shift, Command, and P. Now here we're going to search for Python interpreter. So currently, VS Code is using this Python interpreter that is installed globally on this machine. We don't want to use this.

We want to use the one inside our virtual environment. But how can we find the path to our virtual environment? Very easy. So back to the terminal. Let's run pipenv dash dash venv. alright, here's the path, so let's copy this, good, now back to VS Code, we're gonna select, enter, interpreter path, then we're gonna paste the path, and append, slash, bin, slash, python, at the end, if you're on Windows, instead of a forward slash, you have to use a backslash, okay, good, now, in this project, we have an extra directory, VS Code, we have a settings file, and here we have the path to a python interpreter. Good. With this, we can go to the view menu. Now look, we have this integrated terminal.

The shortcut on mac is control and back tick now vs code automatically activates the virtual environment for this project. So here we can run Python, manage.py, run server. Good. So going forward, I'm gonna use the integrated terminal window here, it's easier. We can always hide it by pressing control and back tick or bring it back. We can maximize it, move it around and so on. Now one quick tip, sometimes when running this command, you might get a syntax error saying invalid syntax. This happens every now and then when VS Code fails to activate the virtual environment for this project. To solve this problem, all you have to do is open a new terminal window and look, VS Code executed this command to activate the virtual environment for this project.

Now we can run Python, manage.py, run server. Great.

Creating Your First App:

Let's talk about apps. So every Django project is essentially a collection of various apps, each providing certain functionality. Just like the apps on your mobile phone, each app provides a certain functionality, right? Django projects are exactly the same. So in this project, let's open up the storefront directory and then look at our settings module. Now we can collapse this panel by pressing command and B on Mac or ctrl and B on Windows, Okay. Now in this module, We have a bunch of different settings. In this lesson, we're going to look at the install apps. So every Django project by default includes these apps. The first app is the admin app, which gives us an admin interface for managing our data. Then we have the auth app, which is used for authenticating users.

Then we have content types app, which we'll talk about later in the course. Next, we have the sessions app, which is kind of legacy. We don't use sessions anymore. A session is a temporary memory on the server. managing users data. These days, when building api's with Django, we don't use the sessions app. So we can come here and delete this app. Then we have the messages app, which is used for displaying one-time notifications to the user. And finally, we have the static files app for serving static files, like images, CSS files, and so on. So each app provides a certain piece of functionality, okay? Now we can also create our own apps here. So let's save this file and Now let's open up the terminal window by pressing Ctrl and backtick.

So here's our terminal window where we have our development server running. We're going to open a new terminal window. Okay, now let's expand this. To clear the window, we simply press Ctrl and L. Now we're going to run Python manage.py start app. And we're going to call this app playground. Okay, so let's close this window and open up Explorer panel. Alright, look, here's our new app, which is represented using a folder with a special structure. So every Django app has the exact same structure. Here we have the migrations folder for generating database tables. We'll talk about that later in the course, we have a complete section about this topic. We have the admin module, where we define how the admin interface for this app is going to look like.

Next, we have the apps module where we configure this app. So the name is misleading. I don't know why Django developers decided to call this module apps. It would be better if it was called config. That's just my two cents. Next we have the models module where we define the model classes for this app. We use model classes to pull out data from the database and present to the user. Next we have the tests module where we write our unit tests. And finally we have the views module which we'll talk about in the next lesson. Again, the name is misleading. This is not the view that you think if you come from a front-end development background. So What we have here is essentially a request handler.

It's not a view. It doesn't have a template or HTML. Again, we'll talk about that in the next lesson. So we created a new app. Now we need to register this app in the settings module. So every time you create a new app, you need to register it here in the list of installed apps. We simply add the name of the app, which is called Playground. Save the changes. Done. In the next lesson, we're going to talk about views.

**Writing Views:**

Alright, let's talk about views. So earlier you learned that HTTP is a request response protocol. So every data exchange involves a request and a response. This is where we use views in Django. So here in the playground folder, let's open up the views module. This is where we define our views or view functions. A view function is a function that takes a request and returns a response. So more accurately, it's a request handler. In some frameworks, it's called an action. In Django, it's called a view. But I don't agree with this name because from an architectural point of view, a view is often associated with something that the user sees. That part in Django is called a template. And we'll talk about that later in this section.

Now, that aside, let's create our first view function. So, we're going to create a function. We can call it anything we want. I'm going to call it say hello. Now, this function should take a request object return a response. So first, on the top, from Django that HTTP package we're going to import the HTTP response class. Now in this function, we can do anything we want. In a real-world scenario we can pull data from a database, we can transform data, we can send emails and so on. For now, let's just return a simple response. So, we're going to return an instance of the HTTP response class and And in this object, in this response, you want to add a simple string. So let's say hello world.

Okay, so this is our first view or first view function. Now we need to map this view to a URL. So when we get a request at that URL, this function will be called. And that's what we're going to do next.

**Mapping URLs to Views:**

here's our Django project. Now, let's say whenever we send a request to playground slash hello, our view function should be called and return hello world to the user. Let's see how we can do that. So back to our project, here in the playground folder, we're going to add a new file called urls.pi you could call it anything, the name doesn't matter, but by convention, we call it urls. Now, in this module, we're going to map our urls to our view functions. So, on the top, we're going to import the path function from Django that URLs. So from Django that URLs We're going to import the path function. that don't worry about memorizing any of these as you practice, all of this will become second nature.

Also, from the current folder, we should import the views module so we can reference our view function. Now, we should define a special variable called URL Patterns all in lowercase. Make sure to split it properly because this is what Django looks for. We should set this to an array of URL pattern objects. We use the path function to create a URL pattern object. So if we call this function, look at the signature of this function. This function has a bunch of parameters. The first one is route, which is a string. The notation you see here is called type annotation, which is kind of a new feature in Python. With type annotation, we can see the type of parameters and the return value of a function.

So the first parameter is route, which is a string. The second parameter is view, which is a function that returns HTTP response object. Now look over here. This notation represents the return type of this function. So the path function returns a URL pattern object. So I'm going to call this function and give it two arguments. A route or a URL, which is playground slash hello, and a View function. So from the views module, we're gonna get say hello and Note that I'm not calling this function. So here we don't have a pair of parentheses. We're just passing a reference to this function. Okay? So, what we have here is called a URL conf module that basically means URL configuration. So every app can have its own URL configuration.

But now we need to import this URL configuration into the main URL configuration for this project. Where is it? You saw it earlier. So, back to the Explorer panel and Here in the storefront folder, we have another URLs module. Up here we have a bunch of comments about how we should use this module. Now look over here. If you want to include another URL conf, there are two things we need to do. First we need to import the include function from django.urls. So over here we have from django.urls. We're going to import the include function. Next we need to add a URL to URL patterns. So in this module, we also have a URL patterns object, which is an array of URL patterns object.

So here we're going to call path, give it a route, and then we use the include function to reference the URL configuration module in this app. So we're going to call path and say any URLs that start with playground followed by a forward slash should be routed to Our playground app. So this is where we use the include function. Now the include function requires a parameter. There's a string here. We're going to reference playground dot URLs. So if we send a request to playground slash hello, Django knows that all requests that start with playground should be handled by this app. So it's going to chop off the first part of this URL and Pass the rest to the URL configuration module. the playground app.

So back to this URLs module, we no longer need to add playground here because we added it once in the main URL configuration module. So we simply add hello, followed by a forward slash. I forgot to add that. We always end our routes with a forward slash. So let's save the changes. Now, whenever we change our code, Django web server automatically restores itself. So let's open up the terminal window and Make sure we don't have any errors. Good. So back to the browser. Let's send a request to this endpoint. Great. We see hello world on the screen. So this is how we can map URLs to view functions. Next, we're going to talk about templates.

**Using Templates:**

So I told you that views in Django are not really views. They're more like request handlers or actions. What we often call a view in other frameworks is called a template in Django. So let's see how we can use a template to return HTML content to the client. So here in the Playground app, we're going to add a new folder called templates. And in this folder, we're going to add a new file called hello.html. And of course, we could call it anything. And here we can write some HTML markup. For example, we can type h1 and press tab to add an h1 heading and say hello world. Now don't worry if you don't know HTML, we're not going to use it in this course.

I just want to show you how Django templates work. So let's save the changes. Now back to our view function. Instead of returning a plain HTTP response, we're going to use the render function to render a template and return HTML markup to the client. So let's remove this line and call the render function now look at the return type of this function it returns an http response object so here we're going to return the response now let's remove the brackets and type them again so we can see the function signature the first parameter here is a request object and the type of this is http request so we're going to pass this request object here now the second parameter. That's the name of our template, which is a string.

So, here I'm gonna say hello dot HTML the other parameters are optional, so don't worry about them at this point. Now, save the changes. Let's make sure our project works up to this point. So, back in the browser, refresh. Beautiful. Now we see HTML content. We can verify it by right-clicking on this page and looking at the page source. So, look. We are returning HTML content. Great. Now, let's make this a bit more interesting. So, back to our template. Instead of hello world, we can dynamically render some value. So, back to our view function. Let's type a comma to see the next parameter. That is a context object and the type of this is a mapping of string to any. That means we can pass any mapping object that maps

a string value to any other type of object. So here we can pass a dictionary. So let's pass a dictionary. The type of the key should be a string, so we can call that name. And we're going to set that to a value like, let's say, mosh. Now, back to our template. Instead of hello world, we can render the name that we passed here. So we type two pairs of braces, and in between, we type the name of the key. Save. Refresh, beautiful. Now, in this template, we can also write some logic. So we can type an if statement. So we type braces, and in between them, we add two percentage signs. In between them, we can write an if statement. So we can say, if name is set, we can render it here.

Otherwise, we can render hello world. And of course, we need an end if clause as well. So, and if. Now this syntax is kind of ugly, but the good thing about Django is that it's modular. So you can easily replace Django's default template engine with your preferred template engine. But once again, we don't really use templates in Django projects that often these days. There are special cases for them, but for the most part, we use Django to build APIs that return data, not HTML content. So I just included this lesson so you know what templates are and how they work. We're not going to spend any more time on templates for now unless we encounter a situation where we really need them.

**Debugging Django Applications in VSCode:**

Let's talk about debugging Django applications in vs Code. This is particularly useful if our application doesn't produce the right result and we want to run it line by line and see where exactly Something has gone wrong. So, click on this icon to open the run and debug panel. Now, the first time you see this message saying we should create a launch that JSON file Basically, we need to create a launch profile. So vs code knows how to run or debug this application. So, let's click on this now In this list, select Django. All right, here's our launch.json file. What you see here is kind of like a dictionary in Python. We have a bunch of key value pairs. These key value pairs define a launch profile.

So VS Code knows that to run this application, it should use the current Python interpreter to run manage.py, which exists in our workspace folder. And then as an argument, it should pass run server to it. Here we can add an additional argument to specify the port so it doesn't clash with port 8000, which is currently in use, okay? Save the changes. We're done with this file. So let's close it. Now if you're curious, let me show you where this file exists. It's added to our project up here inside the VS Code folder. So here's our launch profile. Now that we have a launch profile, if we go to the run and debug panel, We see something else. So here we can start our application for debugging.

But before doing so, I want to write some dummy code here in our view function. So let's set x to 1 and y to 2. Now I'm gonna click on this line to insert a breakpoint. When we add a breakpoint, VS Code will execute every line of code until it hits our breakpoint. From that point onward, we can execute our code line by line. So let's add a breakpoint and And then start the application for debugging. Alright, let me collapse this window. Good. So, in this new terminal window, we have a development server listening on port 9000. So, let's hold down the command key on Mac or control on Windows and click on this link. Alright, here's what we get. The homepage of our project is gone because we have registered a custom route.

That is playground slash hello. So we see the homepage only the first time we run our Django project. So let's go to playground slash hello. Alright, our breakpoint is activated and we are on this line. Now on the left side, in the variables section, you can see the local variables. So currently we have the request object, that is the request object that we receive in this function. So we can expand it and inspect its various attributes. We don't want to do that for now, so let's close this. We want to execute our code line by line. So here we have a bunch of functions. The first one is step over and the shortcut for this is F10. With this we can step over the current line.

So if we press f10 this line gets executed and now we are on the second line. Now look over here in the list of local variables we have x and its value is one this is very useful when debugging applications. So if something is not calculated properly, we can inspect it here. Now, most of the time you can see your local variables here, but if not, you can always add them in the watch window. So before recording this video, I was practicing. That's why you see x here. Let me select this and delete it. So in your watch section, you're not going to have any variables. To add one, just simply click on this and then type the name of our variable. Okay.

So this is how we can step over various lines. So can press f10 again and again. So this function is executed. And now back in the browser, we see the final result. Now let's do something more interesting. So back to our code. Let's close the terminal window so we have more space. I'm going to define a function here called calculate. And here we're going to set x to 1, y to 2, and return x. Just some dummy code. Now over here, we're going to call the calculate function. Now, save the changes, back in the browser, let's refresh, so our view function gets called. Now we are right here. Now this time, instead of stepping over this line, we want to step into it.

Because if we step over this line, we're not going to see what happened in the calculate function. We see the final result, so we see x is set to 1, and we can verify that over here as well. But sometimes we need to step into a function to see What is happening there? Maybe there is a bug inside the calculate function. So let's restart the debugger. Okay. I'm going to close this. Back in the browser. Let's refresh this page. Okay. We're back to this line. Now this time, we're going to step into this function. So look over here. This is the icon for stepping into a function. And as you can see, the shortcut is F11. So if you press F11, We go inside the calculate function.

Now, We can execute each line using f10 or step over and see where something has gone wrong. Now, Let's imagine this calculate function is a large function with a lot of code. At some point, You want to step out of it without having to execute every line of code in this function? There's a shortcut for this. So, look over here. That is step out. The shortcut is shift and F11. So, if we press shift and And F11, we get back to the previous function. So this is how we can debug Django applications in VS Code. Now, once we're done, it's always a good practice to remove these breakpoints, otherwise they get in the way. So as you debug your applications, you place various breakpoints in different parts of your code, and you will hit them all the time.

So always remove your breakpoints once you're done with them. So we're done with this debugging session. Now, we can disconnect by pressing Shift F5 or clicking on this icon All right. Now one last tip before we finish this lesson on the top under the run menu Look, we have a command called run without debugging the shortcut on Mac is ctrl and f5 So if we use this shortcut, we can start an application without having to run Python manage the PI run server. Let me show you what I mean. So first let's bring up our terminal window and Here's the second terminal window that we use for debugging. I'm going to delete this. Here's the first terminal window where we started our application by running Python manage.py run server.

So let's press control and C to stop the server. Good. I'm also going to delete this terminal window. Let's imagine we just open this project in VS code day one. Now to run this application without debugging, we can press control and F5. now our application has started on port 9000. So we can go to this address and then hit playground slash hello. So this is how we can run our application without debugging. Just remember that if you use this command, your breakpoints are not going to get hit. So if you want to debug your applications, you should start it in the debug mode.

Using Django Debug Toolbar:

is another way to debug django applications using a powerful tool called django debug Toolbar. So if you go to google and search for django debug Toolbar, you're going to find this page. Over here, you're going to find the installation instructions. So I highly encourage you to follow along with me and repeat the same steps so install this toolbar together. So, first we have to use pip or pipf to install django debug Toolbar. So back to vs Code. I'm going to open up a new terminal window, and as you can see, VS Code used the source command to activate our virtual environment. So now, let's run pipenv install django-debug-toolbar. Good. Now, back to the documentation. The next step is to add debug toolbar in the list of installed apps in our settings module.

So, Back to vs Code. We can press command and P on Mac or control MP on Windows to bring up the search box. Now we search for our settings module. Good. Now in the list of installed apps, at the end. We're gonna add debug underline toolbar. Good. Yeah the next step. The next step is to add a new URL pattern in our main URL conf module. So I'm going to copy this line and back to vs Code. Let's bring up the URLs module. We're gonna pick the one in the storefront folder. Okay. Now we add a comma here and paste that line. Now here we have a compilation error because debug toolbar is a module that we have to import on the top.

So if you look at the documentation, you can see that over here, Import debug toolbar. So copy this and paste it right here. Good. Now back to the documentation. The next step is to add a middleware. We use middleware to hook into Django's request response processing. So in our settings module, we have a setting for middleware. We're going to add this line in that file. So copy this back to the settings module. Here we have the install apps. Right after that, you can see the middleware setting. So we're going to add that on the top. Now the final step is to add our IP address in the internal IP setting. So for local development, we should use 127 001. So I'm going to copy this entire setting, because by default, this doesn't exist in a new Django project.

So copy this, and I'm going to paste it anywhere in this settings file, it doesn't really matter. Okay, save. So we have completed all the steps. Now back in the browser, if you go to this URL, playground slash hello, we don't see the toolbar, it should appear here, because the toolbar only appears when we return proper HTML document. If you right click here and look at the page source, you can see that we are not returning a proper HTML document. So we don't have the HTML head and body elements. So let's open up hello that HTML now on the top, we type HTML, press tab. Now inside these elements, we're gonna add a body. And then inside the body, we're gonna add the code that we had earlier.

So let's move that right here. Good. Now we have a proper HTML document. Of course the head element is missing, but it doesn't really matter Now back to the browser refresh and here's Django debug toolbar here We have different panels like history or seeing the URLs. We have hit so far We have versions to see the version of Django Python and debug toolbar. We're running we have processing time We have our settings we have request headers and so on now my personal favorite panel is the sequel or SQL panel. Here we can see the queries that are sent to the database. So later in the course, where we talk about querying the database using Django's object relational mapper, we're going to come back to this panel and see what queries Django will generate and send to the database.

So that's it for this lesson. We're done with this section. In the next section, we're going to talk about building a data model. So I'll see you in the next section.

**Building a Data Model:**

**Introduction:**  
  
Welcome back to another section of the ultimate Django course. In this section, we're going to talk about models which are used to store and retrieve data. We'll start off by a quick introduction to data modeling to make sure you know the essential concepts. Then we're going to design the data model for an e-commerce application. Next, we're going to talk about organizing our models in apps to improve reusability. This is one of the areas that is often misunderstood, so I'm going to demystify it once for all. And then we'll spend the rest of this section encoding model classes. We'll use these classes to store and retrieve data throughout the course. So, let's jump in and get started.

Introduction to Data Modeling:

first step in every Django project is figuring out the pieces of data we want to store. So, what entities or concepts do we have in an e-commerce application? Well, for starters, we need the concept of a product with attributes like title, description, price, and inventory. Now, in a real scenario, a product entity might have other attributes. It really depends on the requirements of our application. For now, we just want to keep things simple, okay? Now, quite often, our products are divided in different categories, like Shoes, beauty products, fruits, and so on. So we need another entity called collection or category with an attribute called title. Now we need to add a relationship or an association between these entities so we can start from one end and navigate to the other end.

For example, we can get a collection and find all the products in that collection, okay? Now, let's assume that in our application, a product can belong to one and only one collection and a collection can have multiple products. So an asterisk represents many. We could also use an n here, okay? Now we have a one-to-many relationship between collection and product. So relationships can be one-to-one, one-to-many, and many-to-many. Back to our diagram. Now over here we can add a label called products, and that means when we code these entities or these classes in the collection class, we're going to have an attribute called products. For now, I just want to hide this for simplicity, okay? Now, sometimes we can have multiple relationships or multiple associations between two entities.

So let's imagine that a collection can optionally have a featured product. That is the product whose picture we want to show to the user. So we can add a second relationship between these entities. Now look at this number here. 0 to 1 means a collection can have 0 to 1 product and we're going to call that attribute featured product. So once again, that means in the collection class, we're going to have an attribute called featured product. Okay? Now, you might ask, what about the ID attribute? Shouldn't every entity have an ID attribute? Yes. But Django is going to automatically create that for us. That's why I haven't shown it here. Now a little exercise for you. Grab a piece of paper and identify the other essential entities we need in an e-commerce application.

Just the bare minimum. Don't go ahead and create a complex diagram like this. I don't want this. There are only five entities that I want you to identify. So spend a couple minutes and Identify these entities along with their attributes and relationships then come back see my solution.

**Building an E-commerce Data Model:**

Alright, so this is what we ended up with in the previous lesson. We have a one-to-many relationship between collections and products. Now, don't take this as a hard and fast rule. Maybe in your application, a product can belong to many collections. Then you're going to have a many-to-many relationship between products and collections, okay? So, back to our model. I'm going to remove the collection from here so we have some free space on the screen. Now, what else do we need here? Well, in every e-commerce application, We have the concept of a shopping cart, right? So we're gonna have an entity or a model or a class called cart with an attribute called Created at. Let's assume that we need to know when each cart is created So every now and then we can do a cleanup and remove carts that are 30 days old.

I just made this up. But in a real scenario, you should always design your models based on the requirements of your project. Don't make up your own rules. okay? Now, we need a relationship or an association between products and carts because a cart can have multiple products and a product might be in different carts. So here we have a many to many relationship between products and carts. Now, sometimes the relationship between two entities can have attributes. For example, if a product is in a shopping cart, we need to know how many instances of that product we have in the shopping cart. So this relationship itself should have an attribute called quantity. So we're going to add a new entity here called cart item.

Now look at the notation I've used to represent this concept. I've connected cart item to the relationship using a dashed line. That means cart item represents the relationship between these two entities, product and cart. This is what we call an association class. So cart item is an association class, okay? Now there's another way to show this. Instead of using an association class, we can add a relationship between cart cart and cart item because a cart might contain multiple items and an item belongs to one and only one cart. Similarly, we can add a relationship between product and cart item because each cart item represents a single product and a product might be referenced by many cart items. So here we have two one-to-many relationships between these entities.

Now compare this with the association class notation. So we can either add many-to-many relationship with an association class or to one too many relationships it's a matter of personal preference, okay? So we're done with the shopping cart, but you might ask, what about the user or the customer that owns this cart? Where is that in the diagram? Well, I haven't connected cart with an entity like customer because we don't want to force people to register and login before they can add products to their shopping cart, okay? So there is no relationship between carts and customers. So, cars are essentially anonymous. Anyone can have a cart whether they're anonymous users or registered customers, okay? So, we're done with the cart now let's talk about customers.

So, our customer entity can have attributes like name, email, username, password, and so on. Now, for simplicity, I'm only showing name and email here. Now, a customer can have many orders, and each order belongs to one and only one customer. So, we need a one-to-many relationship between customers and orders. Now, once again, the order entity might have multiple attributes. For now, we just want to know when this order was placed. Now, an order can contain multiple products, and a product can be in multiple orders. So we have a many-to-many relationship between orders and products. And similar to the shopping cart example, this relationship itself needs attributes. So here we need another entity called orderItem with an attribute called quantity. That's our association class.

Alternatively, you could add a relationship between order and order item, because each order may contain multiple items, and each order item belongs to one and only one order. Similarly, an order item references one and only one product, while a given product might be referenced by multiple order items. So two one-to-many relationships. So these are the entities that I wanted you to identify. But to make this project more interesting, I decided to throw in one more entity, tag. Here we have a minute-to-minute relationship between products and tags, because a product can have many tags, and a tag might be referenced by different products. We'll come back to this entity in the next lesson.

Organizing Models in Apps:

So I told you that a Django project contains one or more apps, and just like the apps on your phone, each app in a Django project provides a specific piece of functionality. So that means each app is going to have its own data model. So in this lesson, we're going to look at a few different ways to organize our entities in different apps. One way is to have a single app called the store and drop all of our entities here. Now we can bundle and distribute this app via pip. And this way, anyone can install this app in their project get all these models and the functionality around them. So next time we're working on an e-commerce project, we don't have to rewrite the same functionality over and over.

We simply install this app in our project and write extra code for customization. Great. But there's a problem here. As this application grows and gets more complex, it gets bloated with too many things. Like too many models, too many viewers and other stuff. This is what we call a monolith. Like a large, heavy piece of stone. So at some point I application might become hard to understand, hard to maintain, and hard to reuse. It becomes like one of those remote controls with too many buttons. When designing these apps, we want to follow the Unix philosophy. So each app should do one thing and do it well. So here's another solution. We can break down this project into four small apps. The products app, which represents a product catalog.

So here we're gonna have three entities. Product, collection, and tag. Then we're gonna have the customers app for Managing customers, we're gonna have the shopping cart app for adding the shopping cart functionality and the orders app for adding the ordering functionality. Not bad, right? Well, this is actually a poor way of breaking down this project for a number of reasons. Look at the coupling or dependency between these apps. The orders app is dependent on the shopping cart app, Which is dependent on the products app. So next time we're working on an e-commerce project, we'll have to install all these apps one by one and First, we have to install the products app, followed by the shopping cart app, the customers app, and the orders app.

Ideally, each app should be self-contained so we can easily drop it into a new project. The other problem is that if we publish a new version of the products app, that might cause a breaking change in the shopping cart and Subsequently orders apps. So anyone using these apps have to upgrade all these apps together. Again, more and more work. Also, it doesn't really make sense to have the shopping cart functionality and without having a product catalog, right? Why would we ever want to add the shopping cart functionality to a project without a product catalog? So shopping cart functionality and product catalog always go together. By the same token, why would we ever want to add the shopping cart functionality to a project without the ability to place an order?

Just to allow people to add something to their shopping cart? Then what? Again, it doesn't really make sense. So these concepts are highly related and should be bundled together. If we separate them, we increase coupling between apps and we end up with the problems I just talked about. So it doesn't mean we have to go back to the monolithic design. No, there is a middle ground here. Look at the tag entity. The ability to tag products is optional. We don't necessarily need it in every e-commerce application. In fact, we might need it in other types of applications like a blog or video platform and so on. So tagging is not specific to products or an e-commerce application. It's a separate property. piece of functionality.

So I'm going to move the tag entity to a separate app called tags. And here we're going to have another entity called tagged item, which represents an item that is tagged. It can be a product, an article, a video, and so on. It's an abstract concept. Now with this separation, each app is self-contained and provides a specific piece of functionality. So we can use either or both of them in a new project depending on what we're going to build. Plus, we have zero coupling between these apps, so we can independently change and deploy them without affecting other apps. This is the proper way to break down this project into multiple apps. So here's the takeaway. If our app boundary is so large, we'll end up with a monolith that gets bloated and becomes hard to reuse.

On the other hand, if our app boundaries are so small and fine-grained, we'll end up with a lot of coupling between them. A good design is one with minimal coupling and high cohesion, meaning high focus. So each app is focused on a specific piece of functionality and includes everything needed to fulfill that piece of functionality, okay? So back to VS Code, we're gonna create two new apps. So we run Python, manage.py, start app. The first app is store, and the second one is tags, okay? Now, as I told you before, every time we create a new app, should add it in the list of installed apps. So let's close this window and go to our settings module. Here's the list of install apps i'm going to add those apps right here.

So store and tags. All right, we're done with this step. Starting from the next lesson, we're going to create the model classes for these apps.

Creating Models:

Alright, in this lesson, we're going to create our first model class. So let's go to the store app and open the models module. Now on the top, we have an import statement from Django.db, which is a package. We're importing the models module. In this module, we have a bunch of useful classes that you're going to learn about as we go through this section. So over here, we're going to create our first model class. So we define a new class called product and have it inherit the model class in Django. That class is defined in the models module. So models dot model Now here we need to define the fields of this class. What fields do we need here? Well, every product needs a title.

So it's a title too Then we go to the models module now here. We need to choose a fill type. So let's take a break here Now here on Google, let's search for Django filled types Okay, here's the model filled reference and We're gonna get back to this page over and over. So on the right side, Over here. You can see all the built-in fill types. For example, we have boolean field for storing boolean values. We have char field for storing a sequence of characters or more accurately short or medium sized strings. We have date field, date time field, decimal field, and so on. So as we go through this section, you're going to become familiar with this field types. So back to our code.

So We want the title to be an instance of the char field class. Now, all these field types have a bunch of options. So, back to our documentation, on the top, look over here, field options. These options are common to all field types. For example, using null, we can determine if a field is nullable or not. We also have db index. With this, we can create a database index on this field. We can also set the default value and so on. So all these options are available to all fill types. Now some fill types, depending on their type, have extra options. So let's scroll down and find the char field. Okay, now read over here. Char field has two extra arguments. You have max length, which is required, and db collation, which is optional.

So using max length, we can specify the maximum number of characters we can store in this field. So here we're gonna set the max length to let's say 255 characters. So this is our first field. Now in the next section where we talk about migrations, you will see that we can generate our database tables based on the definition of this model. So then we're going to have a table called product and in that table we're going to have a column called title and the type of that column is going to be var char of 255. For now let's not worry about this. So that's our first fill. What else do we need here? We need a description. that description can be a long text.

So instead of a char field, we're going to use a text field. Okay? The text field doesn't have any required arguments. So here we don't have to set the max length. Okay? What else? Next we need to add the price field. For this i'm going to use decimal field. Always use decimal field for monetary values. We also have float field, but floats have rounding issues. So for storing monetary values, we should always use Decimal field now decimal field if you look at the documentation has two required arguments So as an example, let's say the maximum price with support in this system is going to be nine nine nine nine point nine nine I'm just making this up. So how many digits do we have here six digits?

How many digits do we have after the decimal point two digits? We need to specify these values as arguments here. So max digits equals Six, and decimal places equals two. These two arguments are always required for decimal fields, okay? Now, let's remove the comment. Another field is inventory. We can set this to an integer field. We also have positive integer, we have small integer, big integer, and so on. And the last field i'm going to define here is last update. Let's say every time we update this product, you want to store the current date time here. So we're going to set this to models.dateTime field. And optionally we can set auto underline now to true. That means every time we update a product object, Django automatically stores the current date time in this field.

We also have auto now underline add. With this, only the first time we create a product object, Django stores the current date time here. So we don't want this. We're done with the definition of the product class now a little exercise for you. I Want you to create the customer class with these fields first name last name email, which should be unique phone and Birthdate which should be nullable. So I spent a couple minutes on this and then come back see my solution All right, here's my implementation of the customer class so first and last name fields these are both char fields with a maximum length of 255 characters and Next, we have email, which is an instance of the email field. And here I've said unique to true, So we don't end up with duplicate emails.

Next, we have phone, which is also a char field with a max length of 255 characters. And finally, we have birth date, which is an instance of date field. And here I've said null to true, So this field is nullable. Now for birth date, I've used a date field, not a date time field, because we don't care about the time someone was born. Okay, so Now you might be curious why we don't have an ID field in any of these classes. Because as I said earlier, Django creates it for us automatically. So every entity or every model class is going to have an ID field that is going to be a primary key. But what if we don't want to have an ID field?

What if in this product class? We need a field called SKU and This is going to be our primary key. So we can set this to let's say a char field with a max length of let's say 10 characters and Now here we can set primary key to true. With this Django is not going to create an ID field and make it the primary key. So SKU is going to be our primary key. Okay. So since we don't need this field for now, I'm going to remove it. Good. So this is how we create model classes. Next, we're going to talk about choice fields.

Choice Fields:

Sometimes we need to limit the list of values that can be stored in a field. For example, here in our customer class, let's define a new field called membership and set it to models.char field with a max length of one. Now let's imagine in this field, we can have one of these three values. We can have B that is short for bronze or S for silver or G for gold. How can we implement this? Well, let's go back to our documentation. Earlier I told you that all field types share these options. One of these options is choices. And we use this to specify the possible values for a field. So what we need to pass here is a sequence consisting of iterables of exactly two items.

Now I know this is a mouthful. Let's look at a real example. So over here, we have an array of tuples where each tuple includes two values. The first value is the actual value we're storing in the database. And the second value is a human readable name. So back to our customer class, I'm going to define a new attribute called membership underline choices. I'm using uppercase letters here to indicate that this is a fixed list of values. We shouldn't mess with it. Okay, so we're going to set this to an array. In this array, we're going to add three tuples. Here's the first tuple. For the value, I'm going to use B that is for bronze and for human readable name, I'm going to use bronze.

Now let's duplicate this a couple times. and change this to s and silver and finally g and gold now that we have this array, we can go to our membership field and set choices to membership choices. Okay, like this. now we can also set the default value. So we can set default to, let's say B, that is for bronze. Now this works, but there is a tiny problem in our implementation. If tomorrow we decide to change B to something else, then there are two places we have to modify. Here's one place and here's the other place. So the best practice is to define this value separately. So here we can define another attribute called membership underline bronze. We set it to B. So this is where we store the actual value and now we reference that in multiple places.

So here we use membership underline bronze. and also, by setting the default value, we use membership, underline bronze, okay? Now, for consistency, we should also define separate attributes for s and g i'm going to pause the video and do this on my own. All right, here's the end result. We have three attributes, membership bronze, silver, and gold, and we have used them here. Now, you might be wondering, Why we need the human readable name? Well, later when we talk about the admin interface, You will see that this human readable name will be used in a drop-down list to populate this field. We'll talk about that later in the course. Now, a little exercise for you. I want you to create the order class with two fields, placed at and payment status.

Place that should be a daytime field and it should be auto populated at the time. We create an order object. Payment status should be a char field with these values and So P for pending, C for complete, and F for fail. Spill a couple minutes on this, then come back and see my solution. Alright, here's my implementation of the order class. So first we have our constants, which are payment status pending, complete, failed, and choices. And then we have our fields, which are placed at, which is a daytime field. And here I've said auto now add to true. So the first time we create an order, Django automatically populates this field. Next, we have payment status, which is a char field with a max length of one.

I've set choices to our area of choices and the default value to payment status pending.

Defining One-to-One Relationships:

So you have learned how to create model classes. Now let's see how we can implement a one-to-one relationship between two models. So here in the models module, let's define a new class called address, and assume that every customer should have one and only one address, and each address should belong to one and only one customer. So here we have a one-to-one relationship between customers and addresses, right? Now let's add a couple of fields here, like street, which is going to be a char field with a max length of 255 characters and city with the same definition. Now, as you probably know, a database relationship has two ends, a parent and a child. The parent should exist before we can store the child, right?

I've got a question for you. In this relationship, which class or which entity is the parent? It's the customer because we say the customer has an address. So the customer should exist. before we can create an address. So here in the child class or the child entity, we need to specify the parent. So we add a new field called customer and set it to models dot one to one field. Now this field has a couple of required arguments. The first one is the type of the parent model. So here we type customer. Now the second argument is on delete. And with this we specify the delete behavior. What should happen when we delete a customer. So if we said undelete to models.cascade when we delete a customer the associated address will also be deleted.

This is the cascade behavior. Alternatively if this field accepts null values we can use set null. So when we delete a customer or the parent record the child record, in this case the address is not going to get deleted. It's going to stay in the database and the customer field or the customer column is going to get set to null. In this case, this field doesn't accept null value, so it doesn't make sense to use set null. Alternatively, we can use set default. So this field will be set to its default value. And one other option we have is protect. And with this, we can prevent the deletion. So if there is a child associated with this parent, we cannot delete that parent.

First, we have to delete the child now Now, which value we use really depends on the requirements. There is no one size fits all. In this case, I believe we should use cascade. Because if we delete a customer, the associated address should also be deleted. Now there is one more argument we need to set here. And that is primary key equals true. If we don't set this, Django will create another field here called ID. So every address is going to have an ID. And that means we're going to end up with a one-to-many relationship between customers and addresses, because we can have many addresses with the same customer. But if we make this field the primary key, we can only have one address for each customer, because primary keys don't allow duplicate values, right?

So this is how we can implement a one-to-one relationship between two models. Now, you might be wondering if we should define the reverse relationship in the customer class. In other words, do we have to go to the customer class and add a new field called address for the reverse relationship. No, we don't have to do that because Django automatically creates this for us. So, we're done with this lesson. Next, I'm going to show you how to implement a one-to-many relationship.

**Defining a One-to-many Relationship:**

Now let's assume that a customer can have multiple addresses. So we want to change this one-to-one relationship to one-to-many relationship. How can we do this? Really easy. First we need to change the type of the field to foreign key. So we're telling Django that customer is a foreign key in this table. Okay? Next we need to remove primary key equals true. Because we want to have multiple addresses for the same customer. So we want to allow duplicate values in this column, right? So, remove this option. This is how we can define a one-to-many relationship between two models. Really easy. Now it's your turn. I want you to create a one-to-many relationship between these entities. So a collection can have multiple products, a customer can have multiple orders, an order can have multiple items, and a cart can have multiple items as well.

So spend a few minutes on this, then come back and see my solution. all right let's look at my solution. So I've defined a new class called collection with a single field called title. Pretty basic, but note that I have defined this class before the product class, so we can reference it over here. So in the product class, we have a new field called collection, which is a foreign key, and this is where we are referencing the collection class defined earlier. Now sometimes we cannot arrange our code in such a way that the parent class is further up. In those cases we we can pass a string. Just a little tip for you. Now, for the delete behavior, I've used protect. So if you accidentally delete a collection, we don't end up deleting all the products in that collection.

Okay? What else do we have here? Here in the order class, we have a new field called customer, which is also a foreign key. And for the delete behavior, I'm using protect. So if you accidentally delete a customer, we don't end up deleting orders. In fact, we should never delete orders from our database, because all these orders represent our sales. So they should never be deleted. Okay. Now, once again, in this case, I had to rearrange my code. So the customer class is defined before the order class. So we can reference it here. Now, we have a new class called order item with four fields, we have order, which is a foreign key to order. And once again, we are using protect here.

So if you accidentally delete an order, don't end up deleting the order items. So if an order has at least one item, we will not be able to delete it, okay? Then we have product, which is also a foreign key. And here we're using protect. So if you accidentally delete a product, we don't end up deleting the associated order items. Next we have quantity, which have set to positive small integer field. This way we can prevent negative values from getting stored in this field. And you also have unit price which which is a decimal field. Now you might be wondering why I have defined unit price here. Don't we have it in the product class? Yes, we do. But the price of products can change over time.

So we should always store the price of a product at the time it was ordered. Okay. What else do we have here? We have a new class called cart. We could also call it shopping cart. Here we have a single field created at which is a daytime field and I've said auto now add to true. So this field gets auto populated when we create a new cart. And finally, we have a new class called cart item with three fields, cart, which is a foreign key to cart. And here I've used cascade. So if we delete the cart, we don't need it anymore, we should delete all the items automatically. Then we have product, which is a foreign key to product. And here I've also used cascade.

So if you can delete a product, meaning if that product has never been ordered before, then that product should be removed from all the existing shopping carts as well. That's why I've used cascade here. And finally, we have quantity, which is again, a positive small integer field.

Defining Many-to-Many Relationships:

Let's talk about many-to-many relationships. So I'm going to introduce a new class called promotion and define a many-to-many relationship between promotions and products. So a product can have different promotions and a promotion can apply to different products. So let's define a new class called promotion, which is a new model. Now here we're going to have fields like description, which is going to be a char field with a max length of 255 characters. we're also going to have discount, which is going to be a float field. We can also have start date, end date, and so on. Now we can define the relationship in either of these classes. And as i said earlier, Django will automatically create the reverse relationship. Now in this particular case, it makes more sense to define the relationship in the product class because we might have a product that we're showing to the user.

And at the same time, we want to show all the promotions that apply to that product. So I'm going to go to the product class. And over here, I'm going to define a new field called promotions. So we use a plural name here, because we might have multiple promotions. Now we're going to set this to models dot many to many field. And just like before, we need to specify the name of the target model, which is promotion. Now with this implementation, Django is going to create the reverse relationship in the promotion class. So in this class, we're going to have a field called Product underline set that returns all the products that a particular promotion is applied to if you don't like this name You can always change it.

So in our product class where we define our many-to-many field over here, we can supply a keyword argument Related name if we set this to products Django will use this name as the name of the field in the promotion class. So instead of product set we're gonna have a products. Just remember, if you change the default convention in Django, it's best to change it consistently everywhere, otherwise in some classes you're going to have a field like products, and in some other classes you're going to have a field like product on the line set, and this makes your code really ugly. So in this implementation I'm not going to use related name, I'm going to stick to the default convention. Okay, so this is how we define a many to many relationship between two models.

Resolving Circular Relationships:

I told you that sometimes we can have multiple relationships between two models. So here we have two relationships between collection and product. Now, out of the two, You have only implemented the one-to-many relationship. Now, as part of implementing the other relationship, we're going to face a situation called a circular dependency, which is something we should avoid. A Circular dependency happens when two classes depend on each other at the same time. So the product class is going to be dependent on the collection class and And at the same time, the collection class is gonna be dependent on the product class. So let me show you how to handle this situation. So back in the code, here in the product class, we have this field called collection, which is a foreign key to the collection class.

So here we have a dependency from the product class towards the collection class. Now, to implement the other relationship, we have to go to the collection class and add a new field here called featured underline product. which is going to be foreign key to the product class. Now here we're going to set undelete to well, if we delete a product and that product happens to be the featured product for a collection, here we want to set this field to null. So I'm going to set this to models.set null. And that means we should make this field nullable. So here we pass another keyword argument null equals true. Now look over here, we have an error saying product is not defined. We are seeing this error, because the product class is defined after the collection class.

Now this is where we have a circular dependency. So in the product class, we have a dependency to the collection class. And that's why we define the collection class before the product class. However, because of the circular dependency, now the collection class also needs to be dependent on the product class, which is not possible. To solve this problem, We need to wrap this in quotes and pass this string here. So this solves our problem. However, this introduces a new kind of problem. If tomorrow we decide to rename this product class to something else, this string is not going to get updated. Let me show you. So we put the cursor here, press F2 to rename product to something else. Let's say product one.

Now look, this string is not updated. So use this technique only if you have to. Now let's rename this back to Product. Good. Save the changes. Now let's open up the terminal window. We have an error saying reverse query name for featured product clashes with field name collection. What is going on here? Let me explain. So in the collection class we have a field called featured product, right? Now as you know Django is going to automatically create the reverse relationship for us. So that means In the product class, we're going to have a field called collection, which represents the other end of the relationship. And this is exactly where the problem happens. Because in the product class, we already have a field called collection.

So Django cannot create the reverse relationship because of the name clash. So here we have two choices. One solution is to set related name to something other than collection, or if we don't care about the reverse relationship, we can simply type a plus sign here. This tells Django not to create that reverse relationship. Now, I'll save the changes, look at the terminal, the error is gone, and our server has started at this address. Perfect.

Generic Relationships:

Let's talk about generic relationships. So I told you that we want to design the tags app such that we can reuse it in any projects. Anywhere we want to have the ability to tag items. That's why we have two models in this app. Tag, which represents an actual tag, and tag item, which represents a tag applied to a particular item, which can be a product, a video, an article, literally anything. So let's see how we can implement these models. So back to VS Code. Let's go to the tags app and open the models module here we're going to define two classes the first one is tag and we're going to give it a field called label which is going to be a char field with a max length of 255 nothing new so far now the second class is the one that is interesting tagged item now using this class we can find out what tag is applied to what object

Now, to determine the tag, we simply add a field here, tag, which is going to be a foreign key to the tag class. Now, here we need to set undelete to what? Well, if we delete a tag, we want to remove it from all the associated objects, right? So here we're going to cascade. Again, nothing new so far. Now, the interesting part is identifying the object that this tag is applied to. here's the poor way of implementing this. If you add a field called product, then we need to set this to a foreign key to product. Now with this implementation, we have to import the product class on the top and that means the tags app is going to be dependent on the store app.

So look, if we say from store that models import product, now the tags app is dependent on the store app. We don't want to do this. What if tomorrow we want to tag articles or videos? These are completely different apps. So the tags app should know nothing about these models. So delete this line. What we need here is a generic way to identify an object. To do that, we need two pieces of information. The first one is the type of an object, which can be, let's say, product, video, article, whatever. The second attribute we need is the idea of that object. Using these two pieces of information, we can identify objects in our application, or in database terms, we can identify any record in any tables.

Because using the type, we can find the table, and using the ID, we can find the record, right? So instead of using a concrete model like product, we should use an abstract model called content type, which comes with Django. So earlier, we talked about the list of installed apps in the settings module. Let's go there real quick. So over here, we have an app called content types. Using content types, we can create generic relationships within our models. So content type is a model that represents the type of an object in our application. Now, we need to import this on the top. So, from Django, that contrib, which is short for contribution, then we're going to go to content types app, then we're going to go to the models module and import,

the content type class. So this is a model just like any model that we have in our application. But this model is specifically made for allowing generic relationships. Now back to the business. So we're going to have a foreign key to content type and that means instead of product we should call this field content type. Now for undelete once again we're going to use cascade because if that object type is removed we want to remove all the associated tags. So models dot cascade. Now, the second attribute we need here is the idea of that target object. So we can call this field object underline ID. And this has to be a positive integer field. Because we're assuming that every table is going to have a primary key.

And all primary keys are positive integers. Now what if in a particular table, our primary key is not an integer, let's say it's a good, Well, this solution is not going to work. So this is the limitation of this solution. So now, with these two pieces of information, we can identify any object in our application. Now, one last thing. When querying data, We might want to get the actual object that this tag is applied to, like the actual product to do that We're gonna add another field here called content underline object, And of course we could call it anything. that we should set this to a particular type of field called generic foreign Now, we need to import this on the top. So, I'm going to duplicate this line and change models to fields and content type to generic foreign key.

Now, using this field, content object, we can read the actual object that a particular tag is applied to. So, let's quickly recap. To define a generic relationship, there are three fields we need to define. Content type, object ID, and content object. Now a little exercise for you. I want you to create an app called likes for tracking the objects that a user likes. So in this app, we're going to have a model called liked item. And using an instance of this model, we can tell what user likes what object. So here we need a field called user, which is going to be a foreign key to the user class defined in this module, Django dot contrib, that auth that models. So auth is another app that is automatically installed in every Django project.

And using this app, we can authenticate and authorize users. So go ahead and spend a couple minutes on this, then come back and see my solution. Alright, here's my solution. We have a new app called likes. And in this app, we have a new model called like item. Now here we have four fields. We have user, which is a foreign key to the user class defined. in django.contrib.auth.models. Now here for undelete I've used cascade, so if a user is deleted, we want all the objects that the user has liked to be deleted as well. Then we have three fields exactly like before, no changes. So we have content type for identifying the type of an object that the user likes, we have object ID for referencing that particular object, and content object for reading an actual object.

So that was pretty simple. Let's move on to the next lesson.

Setting Up the Database:

Introduction:

Welcome back to another section of the ultimate Django course. In this section, we'll use Django migrations to generate our database schema based off of the models we've built in the previous section. So we'll talk about making migrations, running them, reversing them in case we make mistakes. We'll also talk about populating our database. So by the end of the section, we'll have a production grade database that we can query in the next section. So let's jump in and get started.

Supported Database Engines:

Django supports many database management systems. The most basic one that is set up by default is SQLite, which is a very basic, lightweight database engine. You should use it only for development or for low-traffic websites, nothing serious. For any serious projects, you should use Postgres, MySQL, MariaDB, or Oracle. These database engines are officially supported by Django, but they're also third-party libraries for other database engines like Microsoft SQL Server. Now, out of all these options, Postgres and MySQL are the two most common database backends used in the Django community. In this section, we'll start off by having a quick look at SQLite, and then shortly after, we'll switch to MySQL for the rest of the course. Now, don't worry if you have never worked with MySQL, we're not going to use any fancy MySQL features.

All you have to do is install MySQL. Now, if you don't want to use MySQL and prefer a different database engine like Postgres, that's totally fine with me. But I'm not going to cover connecting the project to Postgres. You need to research it on your own. So next, we're going to talk about creating migrations.

Creating Migrations:

let's talk about migrations. In Django, we use migrations to create or update our database tables based off of the models we have in our project. So in django projects, we're not going to manually create or modify our database tables. We're going to let django take care of that. Let me show you how this works. So open up a new terminal window and run python manage.pi make migrations. Make sure to spell it properly all right let's see what's going on here. Django looked at all the installed apps in our project, and for each app, it created a new migration file. So here we have the migrations for the store app. We have a new file at this location, store slash migration slash, here's the name of the file.

So every migration file has a sequence number, like one, two, three, four, and a descriptive name. If you're on a Mac, you can hold down the command, or if you're on Windows, you can hold down the control key, and then click to open this file. So let's have a quick look here. As you can see, this migration file is just a Python module. So here we have a class called migration with a bunch of operations. So here's the first operation for creating a model called cart with two fields ID, which is a big auto field, which translates to a big auto incremented column in a database. And as you can see, this field is used as a primary key. So earlier I told you that we don't have to explicitly define a primary key for each model.

Because Django is going to do that for us. Okay. Now in the same model, we have another field called created at which is a daytime field. Now similarly, we have other operations for creating other models like collection, customer and so on. Now at some point, we're going to run these migrations against a database. At that point, Django is going to translate this Python code into SQL code and run it on top of our database. We'll see that soon. So here's what I want you to take away. As you build new models or modify existing ones, you run the make migrations command to generate a new migration. Let's go through this exercise a couple more times. So let me show you a cool trick.

Let's say we want to quickly jump to the product class. We can hold down the command key on mac or control key on windows and press T. This is the go to symbol command. You can find it on the top under the go menu. So here we have go to symbol in workspace and the shortcut is command and t on Mac. So if you search for product, We can find all symbols that contain the word product. A symbol can be a class, a function, a variable, and so on. So, let's jump to the product class and Rename the price field to unit price. So here we press f2 to rename this to unit underline price. I'll save the changes. Back in the terminal window.

We run make migrations one more time. That Django is asking. Did you rename product of price to product of unit price? Yes, we did. So we type Y. Good. Now we have a new migration in the store app with this sequence number. And this description, rename price product, unit price. If you don't like the description, you can always change it. So let's go to the Explorer panel. Here's the store app, the migrations folder. Now, we can simply rename this to rename price to unit price. Just remember, if you rename a file here, make sure to update all references to that file in other migration files. Let me show you what I mean. So, if you open this migration file over here, You can see this migration is dependent on This other migration in the store app.

So, if tomorrow we decide to rename this file from initial to, let's say, first migration, Then we'll have to find all references to this file and update them as well. Okay, so So now we have a new migration and in this migration we have a different kind of operation instead of a create model operation We have a rename field operation. So when we run this migration Django will go to the product table and Rename the price column to unit price. Okay Now back to the terminal window. We haven't made any more changes So if you run make migrations one more time Django says no changes detected now sometimes you see this message and Even though you have changed your models, why does this happen?

This happens if you forget to add the app in the list of installed apps. So, one more time, we're going to use our shortcut, which is command and t, to jump to the installed apps variable. Okay? So over here, we have all the apps we have created so far. Store, tags, and likes. If likes was missing here, Django wouldn't detect any changes in this app. Okay? Now, let's go back to the product class. and make a different kind of change you want to introduce a new field called slug now a lot of people don't know what a slug is let me show you so here on stack overflow if you search for a question in the address bar you can see the idea of the question and the actual question itself what you see here is called a slug but this luck can only contain letters numbers underscores on hyphens so any other character is going to be replaced or removed for example if

Here we have a space between each word, but in the URL, the spaces are replaced by hyphens. Now the whole point of adding a slug here is to make it easier for search engines to find our content. So this is a search engine optimization technique. Now similarly, we want to give each product a slug so search engines can easily find our products. So we're going to set this to models.slug field. Save the changes. Back to the terminal. Let's run make migrations one more time. Now this time we see a different kind of message, saying you're trying to add a non-nullable field slug to product without a default. Now here we have two options. One option is to provide a one of default value right now.

The other option is to quit and add a default in our model file. Let me show you both options. So if we type two we can quit. Now back to our model. We can set a default value. Let's say hyphen. The other option is to make this nullable. If we make this nullable, Django is not going to complain that this field needs a default value. Now, let's remove this and create the migration one more time. So, back to the terminal, make migrations. Alright, this time I'm going to select the first option. So we're going to provide a one of default value. Now Django is asking, what is that default value? So we're going to supply a string that contains a hyphen, exactly like before.

Alright? the Django created a new migration file with this sequence number. Let's open this file real quick. Now in this file, we have a new operation for adding a field for an existing model or table. So when we run this migration, Django is going to go to the product table and it's going to add a new field called slug and it's going to populate it with a hyphen. So the difference between this option and the previous option is that this default value is not in our model it It's only in the migration file. It will be used only once. Okay? So this is all about migrations. Now, before we finish this lesson, let's give this migration file a more descriptive name. So because this is our last migration file, we don't have any other migrations that are dependent on this file, so we can safely rename this file to let's say add slug to product.

Okay. So now we have a bunch of migrations in our project. Next, I'm gonna show you how to run these migrations.

Running Migrations:

So now we have a bunch of migrations where each migration describes a set of changes. So each migration is like a commit in a version control system like Git, right? Now we're going to run these migrations to generate our database schema. So here in the terminal, we run Python manage.py migrate. All right, so Django went through all the installed apps and executed all the pending migrations. So as you can see, we have a migration that comes from the content types app. We have another migration that comes from the auth app. We have a bunch of migrations from the admin app and so on. Now, where is our database? Well, if you look at the Explorer panel over here, you can see a file called db.sqlite3.

So this is our SQLite database, which is a very lightweight database you should only use for development or very small websites. Now, here in VS Code, I have an extension called SQLite, which we can use. for opening a SQLite database. So if you don't have this extension, go ahead and install it. There are also other tools like there is a tool called DB browser for SQLite, which you can get from SQLite browser.org. It's a free but crappy application, but it gets the job done. We're going to use it only in this lesson, because from the next lesson, we're going to switch to MySQL. Okay. So install this extension, and then bring up the command palette. So on the top, you can go to

view menu and open the command palette. The shortcut is shift command and p on mac or shift ctrl p on Windows. So we open this up, search for SQLite, open database. You can see our database in the list. So let's select it. Good. Now let's open the explorer panel and collapse our project. We have a new node here called sqlite Explorer. Now here you can see our database. as well as all the tables we have in this database. So we have a bunch of tables that belong to the auth app. These tables are used for authenticating and authorizing users. We'll talk about them later in the course. Then we have three tables that are used by Django to do its job. Again, we'll look at this soon.

We have one table that belongs to our likes app. We have a bunch of tables that belong to the store app. And as you can see, all these tables are named using singular names. So we have address, cart cart item collection and so on. If you don't like singular names, you can always change them. I'll show you how to do that later in this section. We also have a couple of tables that belong to the tags app, okay? Now, let's open the migrations table. So we right click on it and select show table. This table keeps track of the migrations that have been applied to this database. So the first migration comes from the content types app. Here's the name of the migration and

Here you can see the date and time this migration was applied. So currently we have 22 migrations applied to this database. And that means if we go back to the terminal and run the migrate command one more time, nothing is going to happen because all the migrations we have in this project have been executed. So using this table, Django can tell which migrations have been applied and which migrations haven't. So this is how we can run our migrations. Now we have another command called SQL migrate. And using this command, we can see the actual SQL code that is sent to our database at runtime. For example, if you want to see the SQL code for the third migration in the store app, here we type the name of our app and the sequence number of our migration file.

So this is the actual SQL statement that Django will send to our database. And this will be dependent on the database backend we use. So currently we're using SQLite. Later when we switch to MySQL or a different backend, the SQL code that will be generated will be different, okay? Now, one last thing before I finish this lesson. Is it SQL or SQL? Well, this language was originally called SQL, which is short for Structured English Query Language. But SQL was a trademark of an aircraft company, so IBM didn't want to get into legal issues. That's why it changed the language to SQL. But some people prefer to call it SQL, other people call it SQL. So Don't hate me for calling it SQL, that's just my preference.

Now what about MySQL? Isn't that MySQL? Well, according to MySQL team, they prefer their tool to be called MySQL, but they don't mind it if you call it MySQL. So I personally call this tool MySQL, but I prefer to refer to the language as SQL. And now a little exercise for you. Add the zip fill to the address class, create a migration, run it, and then inspect the migrations table so you see the whole workflow in action.

Customizing Database Schema:

Sometimes you need more control over the database schema. For example, you may want to overwrite the name of a table or you may want to add an index to a couple of columns and so on. Let me show you how to do this. So let's press Command and T on Mac or Control and T on Windows and jump to our customer class. Okay, now in this class, we're going to define metadata. So inside this class, after our fields, we're going to add an inner class called meta. The name of this class is important. So this is where we define metadata about this model. Now, here we have a bunch of options. If you simply search for Django model metadata on this page, you can find all the available options here.

So we have DB underline table for specifying the table name. We can also set the ordering that will be used by default when querying objects. We can also create indexes and so on. So let's look at a couple of examples. back to our code. Here I'm gonna set DB on the line table to store on the line customers so here I'm using a plural name. Now even though this is possible, it's not something that I personally recommend. Because if you want to use plural names, then you would have to set the table name for every model in your project. I find it unnecessary. The whole reason we use Django is because it comes with a lot of conventions that we can follow to quickly build applications.

If you want to constantly override these conventions and customize things, maybe Django is not the right tool for you. That's just my personal opinion. So here we can set the table name. We can also create indexes. So we set indexes to an array of index objects. To create an index object, we can use models.index. Now here we set a keyword argument called fields. We set it to an array of strings. So we can add an index on last name and first name columns. Now if you're not familiar with indexes, basically we use them to speed up our queries. I cover them in detail in my SQL course in case you want to learn more. So we have made a couple of changes. Now it's time for what?

It's time for creating a migration. So save the changes. Back to the terminal, we run Python manage.py make migrations. Great. So Django created a new migration file. But it couldn't come up with a good name for this file because we have mixed two different types of operations here. On the one hand, we're creating an index. On the other hand, we are renaming a table. So it's best if you don't mix up different types of operations. Write a bit of code to solve a specific problem, create a migration, and then move on to the next problem. So I'm going to leave this file as is. Now let's run this migration. So python manage.py migrate. Good. Now, let's open up our database one more time.

So here in Explorer panel, here's our database, I'm going to refresh good. Now look at the name of our table. Now we have a plural name here. But look, this is not consistent with other tables. That's why I personally think we should stick to the default convention in Django. So in the next lesson, I'm going to show you how to undo the last migration.

Reverting Migrations:

Let's imagine that the migration that we applied in the previous lesson was a mistake. So how can we undo it? Well, there are a couple of different ways to do this. If you want to selectively undo what we did in the previous lesson, we have to create a new migration. For example, if you want to keep the index, but remove the new table name, then we have to remove this line, save the changes, create a new migration and apply it. That's the only way. However, if you want to undo everything that we did in the previous lesson, we can completely undo revert the last migration. Let me show you how to do this. So let's look at the explorer panel. Here in the store app, look at the migrations folder.

This is the last migration that we want to undo. Migration number 4. So we want to downgrade the database to the previous migration, which is migration number 3. So we open up a terminal window and type python manage.py migrate. We want to migrate the store app to migration number 3. Okay, beautiful. The last migration is unapplied. Perfect. So now our database is back to the previous state. Let's verify it. So here in SQLite Explorer, let's refresh. Look, the customers table is gone and now we have the customer table. And also, if you look at the migrations table, look, the last migration, which was migration number four in the store app is gone. So we properly downgraded our database, but the changes are still in the code.

So if we go to the Explorer panel and look at the migrations folder in the store app, look, the new migration is still there. So if we open up a terminal window and run Python manage.py migrate, then the last migration, which is currently unapplied, will be applied. So to properly revert the last migration, we should delete this migration file. as well as all the changes we made in our code. So, let's go to the customer class. Earlier, we added this meta class over here. If we keep this code, but remove the migration file, the next time we create migrations, Django is going to create a new migration for applying this change to our database. We don't want this to happen. So, the proper way to undo the last migration is delete this code as well as the associated migration file.

Now, this is where we use version control systems like git. Undoing code using a version control system like git is a lot easier than manually removing code from different places. So in this project I've been using git and after each lesson I made a commit. So let's open up the terminal window and run git log dash dash one line. This shows the list of commits I've made so far. So here's my last commit, customizing database schema. and this is where i created metadata for the customer model. Now, here we have this head pointer pointing to this commit. So to undo this commit, we have to make the head point to the previous commit. Now, if this is all new to you, don't worry, just follow along with me.

And if you're interested, I have a comprehensive course about git on my website. So, let's press q here. Now, to reset the head, we type git reset dash dash hard head tilde one. So we want to move the head pointer one step back. So it points to the previous commit. Okay, let's go with that. Great. Now all the changes we made in the previous lesson are gone. Let's verify it. So here in the migrations folder, we no longer have the force migration. And also, if we go to the customer class, The metadata is gone. Great. Now here's a little exercise for you. In the customer model, rename the first name field, to give a name. Create a migration, apply it, and then revert it.

Installing MySQL:

Alright, from this lesson, we're going to switch to MySQL. If you don't like MySQL, you can use someone else's SQL. That's totally fine with me. But let's see how we can install MySQL on your machine. If you already have MySQL, feel free to move on to the next lesson. So open up your browser and head over to MySQL.com. Then go to Downloads. On this page, down the bottom, you'll see a link called MySQL Community Downloads. Now over here, we're going to download MySQL Community Server. Now this page you can select your operating system. So I'm on a Mac and Then we're going to download the latest version Now on this page, you don't have to sign up or login. There is a link down the bottom So let's just start the download great.

So open this DMG file or setup wizard on Windows So here we have a set of wizard pretty simple, but there is one thing that I want to highlight here Let's click on continue and again I agree and install. Now, here it's asking for your computer's password. This is the password that you use to login. So let's type that in. Now, on this page, we can select a password for login into MySQL. This is different from the password you use to log into your computer. So we're gonna select use strong password encryption. Click on next. And then here we type the password for the root user of MySQL. So I'm going to type a complex password here. And then finish. Now once again it's asking for my computer's password.

So let's type that in. Alright, the installation was completed successfully. Great. If you have any problems, if you get any errors, the best way is just to google the error. Because every computer is different, unfortunately I cannot help you resolve the errors on your machine. So now that we have MySQL, Next we're going to talk about GUI tools for connecting to MySQL.

Connecting to MySQL:

All right, now we need a tool to connect to MySQL and manage our database. There are many different tools out there. You can use MySQL Workbench, which is absolutely free and you can get it from mysql.com. Personally, I've used it a lot in the past. It gets the job done, but quite frankly, it's not the best tool out there. We also have TablePlus, which is a great tool. It's very lightweight and it supports many different database engines. So it's not limited to MySQL. And of course, that comes with a price. At the time of recording this video, The license for TablePlus is $59. And by the way, I'm not an affiliate for TablePlus or any other products. These are the tools that I've personally used and introduced in my courses.

And of course, there are many other tools out there. You can research and find your favorite tool. We also have DataGrip, which is my favorite tool. I believe this is the best database tool in the market because it's built by JetBrains, which is the same company that has built IntelliJ, WebStorm, ReSharper, and many other amazing tools. And of course, This is the most expensive product. You can get a license for $89 for the first year, but for the second and third years, the price will reduce to $70 and $50. So every year you have to pay to use DataGrip. Having said that, you can download a trial version and use it for free. So if you want to follow along with me, you can just download it.

You don't need to provide a credit card. Just download it and use it for 30 days. So the first time you open it, you're going to see this window. You're going to evaluate it for free. Now you can optionally supply your email address and evaluate. Great. So this is our tool. We're going to create a new project. We can call that store front. Now over here, you're going to click on the plus sign to create a new data source, which is going to be MySQL. So look, there's so many data sources available here. You can use Amazon Redshift, Apache, Cassandra, and so many other data sources. So let's go with MySQL. Now on this dialog box, you can give your connection a name.

I'm going to accept the default. Next, we're going to specify the host, which is localhost. The port by default is 3306. The user should be root, and the password should be the password that you set when installing MySQL Server. So I'm going to type my password here. Now pay attention to this error. We have missing driver files. So let's download them. Good. Now, let's test the connection. We get this error saying server returns invalid time zone. So let's set the time zone over here. Let's set the time zone to UTC press tab and Test the connection one more time Okay, we successfully connected to mysql server. Perfect. Let's click on OK Now let's right click on localhost and go to new query console Now in this console window we can type sequel commands

So for this course, we're going to create a database called storefront. We have to create the database first before we can connect with it using Django. So even if you're not using DataGrip, let's say if you're using MySQL Workbench, you need to create a database called storefront or whatever you want to call it. Now to execute this, we click on this icon. Good. So we have a new database called storefront and we're going to use this in the next lesson.

Using MySQL in Django:

Alright, the next step is to connect our Django project to MySQL. And to do that, we have to install a package called MySQL client. So here in the terminal, let's run pipenv install MySQL client. Make sure to spell it properly. Alright, the installation was done. Now before recording this video, I was practicing these steps and there I got a weird error when installing MySQL client. And quite honestly, it took me over an hour to troubleshoot the issue. So if you encounter any errors, don't give up. Keep going. This is a natural part of software development. You have to be patient and determined. Just google the exact error message and I promise you, you're going to find something on stackoverflow.com. You can also post your questions on our forum at forum.codewithmosh.com.

So in my case, it turned out that my installation of MySQL was corrupt. So before going further, let's make sure you have installed MySQL properly. Open a new terminal window, and run mysql so we're going to connect to mysql using the root user and a password that we're going to supply now so let's type our password all right if you see this that means you have installed mysql properly if you get an error saying mysql command not found that means either you didn't install mysql properly or mysql is not in your path so then depending on the operating system you're running you need to research how to add MySQL to your path. Alright, so let's stop this by pressing Ctrl and D. Good.

Now, we installed MySQL client package. Great. The next step is to change our database settings. So we're going to go to our settings module. Now let me show you a new shortcut. Under the go menu, look at this command. Go to symbol in editor. The shortcut is shift command and O on Mac. So if you press this, We can see all the symbols in this file. These symbols can be variables, classes, functions, and so on. So all these symbols you see here, these are various settings in this module. So if you type databases, we can jump to the database settings straight away. So look at our default database engine. That is SQLite. And here's the name of our SQLite file. So the project directory followed by db.sqlite3.

Now we're going to change this to my SQL. And this is the reason why we installed mysql client package. If you don't install this, you cannot change the engine to MySQL. Okay? Now, we're going to change the name to the name of our database. This is the database that we created in the previous lesson. Next, we're going to set the host to localhost. Next, we set the user to root and password to the password we used when setting up MySQL. Now, this is purely for development. So we're not going to use this in production. In production, We're not going to include our password as plain text in this file. So later in the course where we talk about deployment, I'll talk about a different strategy for database settings.

For now, Don't worry about it. So, let's save the changes. Now, Let's go back to our terminal window where we're running our server. You should not get any errors at this point. So your server should be still up and running. If you get any errors at this Then you have to stop here and troubleshoot. Otherwise, you should run the migrations. So all the tables are created in our MySQL database. So let's open a new terminal window and run Python manage.py migrate. Okay, all the migrations are applied to our database. So if you go back to data grip and do a refresh, look, our storefront database has these tables. Perfect. So hopefully you didn't encounter any issues. If you did, just be patient and determined.

Don't give up. I deal with these issues all the time, even with more than 20 years of software development. So our database is ready. Let's move on to the next lesson.

Running Custom SQL:

Sometimes we need full control over generating or updating our database schema. This is really easy with Django. All we have to do is create an empty migration, and there we can write any arbitrary SQL code. Let me show you. So here in the terminal, we type Python, manage.py, make migrations. We're going to go in the store app and create an empty migration. Okay, so now we have a new migration, migration number four with this name. of course, we can change the name to something more meaningful. Now, let's open this file. In this file, we have an empty migration, so the operations array is empty. Now, here we can use migrations dot run sql so this is a class that is defined in the migrations module that is imported on the top.

We can create an instance of this class. And here in the constructor, we supply two sql statements. The first one is for upgrading our database. The second one is for downgrading. So here we can use triple quotes in Python to break down our SQL statements into multiple lines. So for this demo, I'm going to insert a new record into the collection table. So store underline collection. We want to set the title column. And for the value, I'm going to use collection one. Okay. Now using the same technique, we could also create a stored procedure, a function, a view, and so on. Now, as I said, this first argument is for upgrading our database. So as a best practice, we should supply a second argument, even though it's optional, because if you fail to do this, then you won't be able to revert this migration, okay?

So in the second statement, we want to undo what we did earlier. So we want to delete the collection with this name. so we say delete from store underline collection where title equals collection one. Now, let's save the changes. Back in the terminal, we run python manage.pi migrate. Okay, the last migration is applied. So let's go to data grip and look at our collection table. So double click on this and here's our new collection, beautiful. Now let's unapply the last migration to make sure our implementation is right. So we want to downgrade the store app to migration number three. So here in the terminal, we say Python matters of pipe, migrate store to migration number three. Perfect. Now back to our database. Let's refresh the collection table.

Our new collection is gone. Great.

Generating Dummy Data:

The last thing we're going to do in this section is populating our database with some dummy data. And for that, we're going to use a great website called makaru calm. It's one of my favorite tools. So let's say we want to populate our customers table. Over here. We can specify our columns and their type. So we have ID, which is a number. We have first name, which is a first name column. So this tool will generate a bunch of random first names for us. okay? Similarly, we have last name, email. Now, instead of gender, We're gonna use phone. So I'm gonna change the type of this column to phone. Now we can optionally specify the format for the phone. I'm gonna leave it as is.

Next we need to add the birthdate column and for that we're gonna use a date column. And I'm gonna change the format to This format. Four digits for the year, followed by two digits for a month and the day. Now we should also add another field for the membership. So if you recall, in this column, we can have B for bronze, S for silver or G for gold. So I'm going to change the type of this to regular expression. And over here, we can type any valid regular expressions. If you're not familiar with regular expressions, you can click on this and see some examples. So what I'm going to type here is a pair of parentheses. And inside the Or G. So one of these characters will be used to generate the values for this column.

Okay. Now over here, we can specify the number of rows to generate, I think 1000 is good for customers. Next, we can specify the format, which can be CSV, JSON, tab, delimited, SQL, and so on. I'm going to select CSV for now. And preview our data. So this is what we're going to get, we're going to get 1000 customers with these values. now we can download the csv right away or we can change the format to sql and over here we specify the name of our table which is going to be store underline customer and then we can download our data. So now we have a sql file we can open this with data grip or mysql workbench to populate our table. So now i'm going to drag and drop this file onto data grip

so we have a bunch of insert into statements. Now on the top, make sure the current schema is storefront. If it's not, just select it over here. Then select all these rows and execute them all right so now we're done. Let's open a new query console for the storefront database and find the number of records in the customer table so let me type select count of everything from store on the line customer. So as you can see over here, we have 1000 records in this table. Beautiful. So we have reached the end of this section. Now I'm going to stop recording and generate data for a bunch of tables we're going to work with in the next section. So at the beginning of the next section, I will give you a SQL file to populate your database with dummy data.

Okay, so that's it for now. I hope you learned a lot and I will see you in the next section.

Django ORM:

Introduction:

Welcome back to another section of the Ultimate Django course. In this section, you will learn how to query and manipulate data using Django ORM or Object Relational Mapper. We'll start off this section by a quick overview of Django ORM and the problem it solves. Then we'll spend the rest of the section on various ways to query and manipulate data. We'll cover filtering, sorting, grouping data, and much, much more. So by the end of this section, you'll be able to confidently pull out data or store it in a relational database. So let's jump in and get started.

Django ORM:

Let's start off this section by a quick overview of object relational mappers and the problem they try to solve. So you know that in relational databases, data is stored as rows in tables. So when pulling up data from a relational database, we need to map these rows into objects. In the past, we used to do this by hand and this was pretty repetitive and time consuming. So we would have to write a SQL query, send it to the database, read the result, and map it to a bunch of objects. So for each record, you'll have to create a new object and set its attributes. This is where an object relational mapper comes into the picture. An object relational mapper, as the name implies, maps objects to relational records, and that frees us from writing a lot of repetitive code.

So using an object relational mapper, we don't have to write SQL code to query or manipulate data. We can code in an object-oriented programming language like Python. The URL will then translate our Python code into SQL code at runtime. So, does it mean we never need to write SQL? No. When dealing with complex problems, ORMs can't produce efficient SQL queries. So that's when we need to jump in and write optimized SQL queries by hand. But overall, ORMs help us reduce complexity in our code, make it more understandable, and get more done in less time. A good example of that is Django migrations. Using these migrations, we generated our database tables almost instantly. We didn't have to write any sequel code or use a tool like data grip to create our tables add relationships indexes and so on Django migrations took care of all of that for us.

So migrations are part of Django RM. Also all the model classes we have created so far inherit from the model class in Django This model class is also part of Django RM So as you will see in this section all our model classes inherit the functionality to query or manipulate data from the base model model class. Now, once in a while, you see someone like our popular John Smith, who hates ORMs and prefers to write every bit of code by hand because he believes ORMs are slow. What John is missing here is that he doesn't understand that the more code he writes, the more bugs he or someone else has to fix in the future. There is an old saying that says, the best code is no code.

Writing and maintaining all that extra code costs time and money. Someone is paying for it, your employer or your client. If you were that person, you wouldn't want someone else wasting your money writing code that was unnecessary, right? Throughout my career, I've seen so many projects over-engineered by people like John that never made it into production. In my opinion, a good software engineer is one who delivers working software in time. Building the best, most optimized solution that will never make it to production is not something to be proud of. So use the right tool for the job. As I said, Orms don't perform well when dealing with complex queries, but that doesn't mean we should never use our apps We can use them in a lot of cases, perhaps for the most part.

Remember the old saying, premature optimization is the root of all evils. So don't optimize your code unless you have proof that is slow. So that's it for the Django. Orm. Let's move on to the next lesson.

Resetting the Database:

Alright, before we get started, let's make sure that we're on the same page. I want to make sure that we're both using the exact same code and database, so what you see on your machine will be exactly the same as what I'm going to show you in the videos. So please follow everything I'm going to show you in this lesson. So at the beginning of the course, in the first section, I gave you a zip file that contains all the resources for this course. So here we have a folder called code that contains the code for each section. Now we're going to go to the fifth section, and open the starter code. So here we have the storefront project. Let's drag and drop it into vs Code.

Okay, we get a message saying no python interpreter is selected. Don't worry about this. So let's close this. Now, let's go to the settings module. And here in the section for our database settings, change the password of MySQL. Otherwise, you're going to get a connection error. So use the password for the root user. Next, We open a terminal window, and here we're going to run pipenv install. So this is going to create a new virtual environment and install all of our dependencies there. Okay, great. Now let's open up DataGrip or any tool you use for managing your database. Here we have the storefront database. I'm going to delete this database and recreate it from scratch. So delete. Good. Now you're going to right click on localhost.

and open a new query console. And here we're going to type create database store front. Make sure to spell it properly because this is the name that we use in our project, okay? So now we have a new database. Let's verify it. Great. So this database is currently empty. The next step is to run all of our migrations to create our tables. So back to our project. Here in the terminal, we run Python, manage.pi Migrate. Now we get an error saying invalid syntax because in this window our virtual environment is not activated. So back to our project. Let's activate our virtual environment. Good. Now we're going to run python manage.py migrate. Okay. All of our migrations are applied. So back to data grip.

Let's refresh. All of our tables are created. Beautiful. But these tables are currently empty. So the next step is to populate them with data. But before doing so, I want to close all the sessions I have open in data grip. So down the bottom, over here, we have the services tab. I'm going to right click on local host and close all sessions. Now I'm going to right click on storefront and open a new query console. Now back in the resources folder, we have this data folder. And here we have a seed file. that will populate five of our tables. Collection, product, customer, order, and order item. So let's drag and drop this into this window. Good. Now on the top, we have to select the schema.

So I'm going to select storefront. Good. Now make sure that all these statements are selected. So currently, all of them are selected. If not, just press command and a on mac or ctrl a on Windows. And finally, we're going to execute them all in one go. Good. So now five of our tables are populated with data. Let's look at the customer table. All right. So here we have 1000 customers. Beautiful. And now the final step is to start our server. So Python, manage.py, run server. Great. So now that we're on the same page, let's move on to the next lesson.

Managers and QuerySets:

Alright, the first thing you need to understand about Django RM is the concept of managers and query sets. So let's press command and T on Mac or control and T on Windows and jump to our say hello function. So we're going to go back to our playground and write all the code here. Now for this section, we don't really need the calculate function. So I'm going to delete it. And I'm going to delete this line as well. On the top from store dot models, we're going to import the product class. Now, every model in django has an attribute called objects. And this returns a manager object. A manager is an interface to the database. It's like a remote control with a bunch of buttons that we can use to talk to our database.

So here we have a bunch of methods for querying or updating data. For example, we have all for pulling out all the objects in the products table. We also have get for getting a single object. We have filter. filtering data and so on. Now, most of these methods, like the all method, return a query set. So when we call this method, we don't get a list of products. We get a query set object. A query set is an object that encapsulates a query. So at some point, Django is going to evaluate this query set and this is when Django will generate the right SQL statement to send to our database. But when will this happen? Well, under a few scenarios. One scenario is when we iterate over a query set.

So if you write code like for product in query set, at this moment, Django will evaluate this query set and get the result from the database. So for this demo, let's just print the product on the terminal. I'll save the changes back in the browser here on the Hello page. Let's refresh. Okay, now let's open Django debug toolbar. So Look at the SQL tab. Over here, you can see the actual SQL statements that Django sent to our database. So here we have three queries. The first two are just used by Django to do its job. The third query is our select statement to get all the products from the products table. So let's expand this. This is the exact query that Django has sent to our database.

So select ID, title, slug, and other fields from product table. Now over here, we can click on the select button to see the actual result that is returned from the database. So this is very useful when debugging applications. Now back to our code. So one scenario where a query set is evaluated is when we iterate over it. Another scenario is when we convert it to a list. So if we call the list function and give it this query set, this query set will be evaluated. Another scenario is when we access individual element, like the first element. Or when you slice this, so if we use the slice operator and pick the first five elements, then the query set will be evaluated. So because of this, we say query sets are lazy, which means they're evaluated at a later point.

Now you might be wondering why query sets are lazy. Why doesn't Django simply call the database the moment we call the all method? Well, the reason for this is that we can use query set methods to build a complex queries. So over here, we can call query set that filter to filter the result. And this will return a new query set. So once again, the query is not going to get evaluated at this point. So we get a new query set. And then right away, we can call the filter method to apply a second filter. Similarly, we can call the order by method to sort the result. So by changing all these methods, will build a complex query, and at some point, when we iterate over that query or convert it to a list, that query will be evaluated.

Now imagine if query sets were not lazy. The moment we call the all method, the query would be evaluated, and this would result in thousands or even millions of objects returned from the database. But what if you were only interested in a subset of these products? Then calling the filter method would filter the objects in memory. So we would get thousands or millions of objects will store them in memory and throw most of them out. So to recap, every model has an attribute called objects, which returns a manager, which is an interface to the database. Now, the manager has a bunch of methods for querying and updating data. Most of these methods, like the all method return a query set. In contrast, we have some other methods that return result immediately.

For example, we have the count method, which returns the number of records in the product table. Now, why don't we get a query set here? Well, because this method returns a number, so it doesn't really make sense to do something extra with that number. We're not going to apply a filter to it, we're not going to sort the result or build a complex query around it. So this was the basics of managers and query sets. As we go through the section, you will learn more about these query sets. Next, we're going to talk about retrieving objects in detail.

Retrieving Objects:

Alright, in this lesson, we're going to talk about a few different methods for retrieving objects. The first method is the all method that we briefly talked about in the previous lesson. This method returns a query set, and when this query set gets evaluated, we get all the objects in a given table, like the product table. Now, sometimes we want to get a single object, like the product with the id 1. To do that, we use the get method, and here we pass a lookup parameter like id equals 1. Then we could also use pk, That's a special argument. So when we use pk, Django will automatically translate this to the name of the primary key field. So our primary key field might be id, it might be code, whatever.

We don't have to remember. This is the benefit of using the pk argument. Now unlike the all method, the get method returns an actual object, not a query set. The reason for this is because once we get an object, it doesn't really make sense to do anything extra with it. We're not going to sort the result. We're not going to apply additional filters and so on. So here we get a product and that means we should rename this variable to product. So always pay great attention to the name of your variables and functions. I cannot emphasize this enough because this is one of the main issues I see in a lot of people's code. They have a function like get airplane. So you see that function and you think you're going to get an airplane.

But when debugging the application, you realize that this function actually returns an orange And that orange ends up getting stored in an object called book. Don't do this. So always pay great attention to the name of your variables. Okay. So let's save this and see what we get back to the browser refresh. Let's look at the SQL tab. So here's our select statement. So we're selecting all these columns from the product table. And here we have a where clause to select the product with ID one. Now Django also adds a limit clause, a limit of 21. I'm not entirely sure why they use 21 here. To me, It looks like a magic number, but this is something you can find in Django source code.

So for now, don't worry about it. This limit doesn't really make a difference for us. Now, one thing you need to know about the get method is that if it cannot find this object, it will throw an exception. So if we set PK to 0 look, we get an exception of type does not exist. So So to handle this, we have to wrap our code inside a try-catch block. So, back to the code, first we're going to import the type of the exception on the top. So, from django.core.exceptions, we're going to import object does not exist. Now note that I've put all my Django import statements next to each other, so my code is cleaner and more organized. Compare this with this. So, we're importing something from Django, then something from

another app, then we're going back to Django. Again, this is a little bit ugly. So I prefer to put all Django import statements next to each other. Now we need to wrap this line inside a try catch block. Don't tell me that you don't know what the try catch block is because I covered it in detail in my Python course. So here we type try. Then we type accept, followed by the type of exception, which is object does not exist. Also, Note that you don't have to type the full name of the class. So here I'm typing OBJDNE. And with that we can easily look up this class. So we can use abbreviations. Okay. Enter. Now in a real scenario, here we want to show an error to the user.

But for now don't worry about it. So I just want to pass. Okay. Now back to the browser. Refresh. This time we didn't get an exception. Great. Now this try catch block looks a little bit ugly. We don't want to repeat this pattern over and over in our code. So let me show you a better way. First I'm going to delete the try catch block. Good. Now instead of the get method, we can use the filter method. Now this method returns a query set. So right away we can call the first method of the query set. Now if the query set is empty, the first method returns none. So in this case, product is going to be none. We're not going to get an exception.

Check this out. So save and refresh, great. Now sometimes we want to check the existence of an object. So over here, right after the filter method, instead of the first method we can call another method called exists. So first we apply a filter and then we check to see if there is an object in our query set with this criteria. So in this case we're not going to get a product object, we get a Boolean value. So we should rename this variable to something like exists. Okay? So these are various ways to retrieve objects. Now in the next lesson, we're going to talk about filtering in detail.

Filtering Objects:

Alright, let's talk about filtering data. So let's say we want to find all the products that are $20. So here we pass a keyword argument and say unit price equals 20. Pretty straightforward. Now, what if you want to find all the products that are more expensive than $20? We cannot use our logical operators here. Look, we have an error. Because this expression, this piece of code is a Boolean expression and returns a Boolean value which can be true or false. But when calling the filter method, We need to pass a keyword argument. So we need to pass a keyword followed by a value, right? So to solve this problem, after the field name, we type two underscores followed by a lookup type. So in Django, we have GT, which is short for greater than.

We also have greater than or equal to. We have less than, less than or equal to. So for this demo, I'm going to change this to GT equals 20. Now we're passing a keyword argument to this method and But the name of our keyword is kind of special, right? Save the changes. Now to find all these lookup types, we can search for query set API. And on this page, look over here, where is it? It's right here, filled lookups. So we have various lookup types for numbers, strings and dates. In this lesson, we're going to cover a lot of them. And I'm going to leave the rest to you. So back to our code. Another useful lookup type is range. With this we can find the products whose price is in a given range So over here, we cannot pass a single value.

We need to pass two values a minimum and a maximum So we need to pass a tuple with two values. Let's say 20 and 30 now out of curiosity Let's render these products in our template and see what we get. So here's the context The object that we're passing to our template in this object in this dictionary. I want to add another key value pair called Result, you could also call it product. That's better and set it to query set. So we're passing the query set here Now it would be better to convert this to a list So the name of the key reflects the type of the value once again We don't want to get an airplane and store it in a book, right?

So save the changes now. Let's go to hello dot HTML and after our if block we're gonna render an unordered list which represents a list of bullet points so tab and Inside the unordered list, we're going to add a for loop. So we add a pair of braces with ampersands. And here we say for product in products, we don't need a colon here, just the expression. Now, in our for loop, we're going to type a list item for rendering a bullet point. And over here, we add two braces to render product.title. Finally, we need to close our for loop. once again braces with ampersand, and here we type and 4. Now back to the browser, here are the products whose price is between $20 to $30.

Great. Now we can also filter across relationships. So let's remove this argument. Let's say we want to find all the products in collection number 1. So here we type collection, followed by two underscores, so we navigate the relationship, and over here we can type the name of any of the attributes or fields of the collection class. So we can say id equals 1. We can also use our lookup types. So we can add two underscores followed by a lookup type, like greater than or range. And then we can set this to a tuple of, let's say, three values. So with this, we get all the products in any of these collections. So these were the lookup types for dealing with numbers. Now let's look at an example involving a string.

So I'm going to remove this argument. So let's say we want to find the products that contain coffee in their title. So over here we can type two underscores and use the contains lookup type. We set this to coffee and with this we don't get anything because this lookup type is case sensitive. So to perform a case insensitive search we have to use I contains. Now take a look. So here are all the products that have coffee in their title. Now, we also have starts with ends with as well as their case insensitive variation. Okay. Now, for dates, we have a bunch of lookup types. So let's say we want to find all the products that were updated in 2021. So last update, double underscore, here, we can use year and study to 2021.

Now take a look. So this is what we get. So we can extract individual components and dates like year, month, Second and so on. We can also compare this with a date value. Not something we use that often. And the last lookup type I'm gonna cover here is checking for null. So, to get all the products without a description, we can say description double underscore is null equals true. We don't get anything because all of our products have a description, okay? Now, it's your turn. I've attached a PDF below this video. In that PDF, you're gonna find a bunch of exercises and their solutions. So make sure to do these exercises before moving on to the next lesson where we talk about complex filtering.

Complex Lookups Using Q Objects:

Alright, let's see how we can apply multiple filters. So let's say we want to find all the products with inventory less than 10 and unit price less than 20. There are a couple of different ways to implement this query. One way is to pass multiple keyword arguments here. So we can say inventory less than 10 and here we pass a second keyword argument and say unit price less than 20. So Let's go back to the browser, refresh. Here are the products we get. Let's look at our SQL query. So here's our select statement. As you can see, in the where clause, we have two conditions combined using the and operator. Okay. Now, there is another way to write the same query. Instead of passing multiple keyword arguments, we can chain the call to filter method.

So I'm going to remove the second keyword argument. So here we have the filter method. The filter method returns a query set. Now we can get that query set and apply a second filter on it. So here we can say unit price less than 20. Now here we get a new query set, which we store in this object. And the moment we call the list function, this query set will get evaluated. So if we go back to the browser and refresh, we get the same result as before. So let's take a look at our query. So exactly like before, where we have two conditions combined using the and operator. Now, how can we combine these conditions using the or operator? Well, to do that, we have to use Q objects.

Back to our code. First, we need to import the Q class from Django. So from Django.db.models import Q. So Q is short for query. And using this class, we can represent a query expression. or a piece of code that produces a value. Okay. So using this queue class, we can encapsulate a keyword argument. So let me remove all this code and start from scratch. So we want to combine these conditions using the or operator. So here we say filter. Now instead of passing a keyword argument here, we're going to pass a queue object. So we create a queue object. And right here we pass a keyword argument. So we say inventory less than 10. So each q object encapsulates a keyword argument or a query expression.

Now we can combine this q object with another q object using the bitwise operators. So if we use the bitwise or operator, this will translate to a logical or in a sql query. So here we pass a second q object and add the second condition. Unit underline price less than 20. now let's save the changes so here's what we have implemented. Back to the browser. Refresh. Let's look at our query. There you go. So, two conditions combined using the or operator. Beautiful. Now back to our code. Here we can also use the and operator, but this syntax is a little bit too verbose. If you want to combine multiple conditions using the and operator, we don't really need q objects. We can simply pass multiple keyword arguments to the filter method, just like before.

Okay, we can also negate a queue object. So this will translate to the not operator in sequel So when we execute this, we'll get all the products whose inventory is less than 10. And Their unit price is not less than 20. Okay, so this is how we can build complex lookups using queue objects. Next, we're going to talk about referencing fields using F objects.

Referencing Fields using F Objects:

Sometimes when filtering data, we need to reference a particular field. For example, let's say we want to find all the products where their inventory equals their unit price. Now, I know this doesn't really make sense in terms of a business rule, but let's just imagine that we want to compare two fields. So here in the filter method, if we say inventory equals unit price, we get an error because unit price is not a valid value for this keyword argument. Where is unit price defined? we don't know, right? Now, if we convert this to a string, that's not going to work either because the inventory field is a number and we cannot compare a number with a string. So to solve this problem, we need to use an f object.

So first on the top, from the same models module, we're going to import the f class. Now using this class, we can reference a particular field. That's why it's called F. So over here, we're going to create an f object and in the constructor, we're going to pass the name of a field, okay? So, let's run this query and see what we get. Refresh. Obviously, we don't have any products that satisfy this criteria, but let's look at our sql query. So, over here, look at the where clause. We are finding products whose inventory equals their unit price. Now, using f objects, we can also reference a field in a related table. For example, here we can say, Collection, double underscore ID, and with this we'll compare the inventory of a product with the idea of its collection, okay?

So this is how we can reference fields using f objects, pretty simple. Next we're going to talk about sorting data.

Sorting:

Alright, let's talk about sorting data. So here we have this order by method that I briefly mentioned before. Using this method, we can sort the result by one or more fields. For example, here we can get all the products and sort them by their title in ascending order. Take a look. So save and refresh. There you go. So all the products are now sorted in alphabetical order. Beautiful. Now let's look at the SQL tab. So here's our query. Now with this implementation, We have an order by clause for sorting the products by their title in ascending order. If you want to change the sort direction, We simply add a negative sign here. that we can also sort by multiple fields. For example, We can sort by unit price in ascending order, and then by title in descending order.

So with this implementation we we're going to sort our products from the cheapest to most expensive ones, and if we have multiple products with the exact same price, within that group, our products are going to be sorted by their title in descending order, okay? Now, the order by method returns a query cell object, and one of the methods of query set is reverse. So if we call the reverse method here, this will reverse the direction of the sort. So with this implementation, we're going to sort the products by unit price in descending order and then by title in ascending order now we can also call the order by method after filtering data because the filter method returns a query set object and order by is one of the methods of query set objects so here on query set api reference over here you can see all the methods of query set objects so we have filter exclude order by and so on so as an example we can

filter all our products to get products in collection one and then sort them by their unit price. Pretty straightforward. Now, sometimes we want to sort the result and pick only the first object. So let's remove the filter method here. We're simply sorting our products by their unit price and then we're going to pick the first product. So with this implementation, we're not going to get a query set because we're accessing the an individual element. So up to this point, when we call the order by method, we get a query set object. But the moment we access an individual element in this query set, the query set gets evaluated, and then we get an actual object. So we should rename this variable to product.

And of course, we need to make a change here, because we're not going to convert this to a list. So instead of products, we're going to pass product or context. And of course, we need to change our template to accommodate this, but let's not worry about that in this video. What I want to show you instead is a convenience method called earliest. So another way to rewrite the same query is like this. Instead of order by, we can call The earliest method to sort the result by unit price and get the first object. So with the second implementation, we're not going to access an individual element and because the earliest method returns an object. In contrast, order by returns a query set. Now similarly, we have another method called latest, which sorts the products by unit price in descending order, and then it will return the first object.

Okay? So this is all about sorting data. Next, we're going to talk about limiting results.

Limiting Results:

So our product table has a thousand products. Now quite often, we don't want to show all these products to the user in one list. We want to show pages of products. So let's say our page size is five and we want to show the products on the first page. To do that, we use Python's array slicing syntax. So if you type colon five here, this is going to return the first five objects in this array. So more accurately, we're going to get objects at these indexes, zero, one, two, three, and four. So excluding five. Okay. Now, let's go back to the browser and refresh. So here are the first five objects or first five products. Now let's look at the SQL tab. So now in our query, we have the limit clause for getting the first five products.

Now to get the products on the second page, we can start from five and finish before 10. So this is going to return the product objects at these indexes, five, six, seven, eight, and nine. Now, refresh. With this implementation, our query is going to have an additional clause. So we have limit 5 followed by offset 5 for skipping 5 records.

Selecting Field to Query:

So you have seen that when we query objects, by default, all of their fields are read from the database. But what if you're only interested in a subset of these fields? For example, look at our product table. Our product table has a bunch of fields or columns. Some of these columns, like the description column, contain a lot of text. What if we don't care about the values in this column? What if we only want to return the ID and the title of each product to the user? This is where we use the values method. So we call the values method specify the fields we want to query. Let's say ID and title. Now, refresh. We get the same result as before, but look at our query.

With this implementation, we're only reading the ID and the title of each product, okay? Now, we can also read related fields. For example, we can go to the collection and read the title field. So using the double underscore notation we we can access the related fields. Now let me show you something interesting. So back to the browser, refresh. Let's look at our query one more time. With this implementation, now we have an inner join between the product and the collection tables because we are reading a related field. So this is how we can use the values method. Now, one thing you need to know about this method is that with this method, instead of getting a bunch of product instances, we get a bunch of dictionary objects.

Let me show you what I mean. So, let's go to our template. Instead of rendering product.title, let's just render the product object itself. So, save, and refresh. See what's happening? So each object in the result is a dictionary. It's not a product instance. So in this dictionary, we have three key value pairs. We have id, title, and collection, double underscore, title. Now, we have another method called values.title. And with this method, we get a bunch of tuples instead of dictionaries. So, take a look. Refresh. Now, each object is a tuple of three values. So, here we have the ID, followed by the title of the product, and the title of the collection. Now, here's your exercise. I want you to write a query to select products that have been ordered and Sort them by their title.

now. Let me give you a hint before you get started. So, in our database we We have this product table that contains all the products in our system, but not all products have been ordered. So to find the products that have been ordered, you need to go to the order item table. Here we have a column called product ID. You need to select all these values, all these product IDs, and use them as an argument when filtering products. So spend a couple minutes on this, then come back, see my solution. All right, the solution. As we said, We should start with the order item table. So first we go on the top and import the order item class. Now we go to order item, the objects.

Now here we need to select all the product IDs from this table. So this is where we use the values or values list method. It doesn't really matter. So I'm going to go with values because it's shorter. So we go to values. Next we go to the product field and select the ID field. that we can also use a single underscore, because Django will automatically create this field for us at runtime. So even though we didn't declare a field called product ID, on the order item class, look, we only have product, right? But Django will automatically create a field called product underline ID at runtime. And essentially, this is our foreign key field. Okay, so back to our query, we select product IDs and

store the result in query set. Let's test our implementation up to this point. So, in the browser, this is what we get, a bunch of dictionaries, and as you can see, we have a lot of duplicates. Now, before i show you how to resolve duplicates, let's look at our query. So, we have a simple query for selecting product id from the order item table. Beautiful. Now, to get rid of duplicates, here we can use the distinct method. So values returned a query set and distinct is another method of query set objects. With this, let's go back to the browser and refresh. We no longer have duplicates. Beautiful. Now the last step. We want to go to the product table and select all products with these IDs.

Very simple. So we're going to say product that objects that filter id double underscore in. So here i'm using the in lookup type to find all products whose id is in a given list. And we're going to set this to this expression over here where we select product ids from order item table. So cut. Move it here. And we're going to set the query set right here. So this is the end result. Let's see what we get in the browser. So refresh. We get a bunch of product objects. So product with ID 1 4 5 and so on now let's go to our template and Render the title of each product. So instead of rendering the product. We're gonna run their product the title and There you go.

So here's the list of products. Now the last thing we need to do is sort the list So after we filter our products, we're gonna call the order by method to sort the products by their title and Now take a look. Here's the final result. These are the products that have been ordered.

Deferring Fields:

In the last lesson, you learn how we can optimize our queries by specifying the fields. We want to query. Now, We have another technique called deferring fields. So we have a method called only, and With this method, we can specify the fields. We want to read from the database. Let's say id and title. Now, you might be wondering how this method is different from the values method. Well, with the only method, will get instances of the product class, whereas with the values method, will get dictionary objects, okay? Now, let's save the changes. Back in the browser, refresh. So, we get the same result as before. Let's look at our SQL tab. So, here's our query. Pretty simple. We're selecting the ID and title from the product table.

Great. So, this is how the only method works. Now, you've got to be careful with this method. If you don't know what you're doing, you'll end up with a lot of queries sent to the database under the hood. Let me show you what I mean. So, let's go to our template. And after the title of each product, render its price. So we add a dollar sign here, followed by two braces. And here we render product dot unit underline price. Let's save back in the browser. Refresh. Look what's happening. Our application is freezing. So it's going to take several seconds until we see the result. All right, here's the result. Now let's look at our SQL tab. How many queries do we have here?

1,002. Let's see what's going on here. So here's our main query where we read the ID and title from the product table. Now, after the main query, for each product, you have a separate query to read its price. So look at this query. Here we are reading the ID and unit price from the product table where ID equals 1. We have a similar query for every other product in this list. So because we have a thousand products in this list, we have a thousand extra queries for reading the price of all these products. This is what I was talking about. So you have to be careful when using the only method. If you don't know what you're doing, you'll end up with a lot of extra unnecessary queries.

Now we don't have this problem with the values method. Because with the values method, we get dictionary objects and these dictionary objects don't have this behavior. So if you access a field that doesn't exist in a dictionary, that dictionary is not going to issue a query to the database, Okay. Now we have another method, which is the opposite of the only method, and that is called defer. With this method, we can defer the loading of certain fields to later. So let's say our product table has a bunch of columns. You're interested in all of them except the description field. So we can defer this to later. But once again, you have to be careful. So if somewhere else you have, let's say, a for loop, and in that loop, you're iterating over the products and rendering the description of each product, once again, you're going to end up with all those extra queries.

Selecting Related Objects:

Sometimes we need to preload a bunch of objects together. For example, here we're loading all the products. Now let's go to our template and render the collection of each product next to its title. So here we add a hyphen followed by two braces to render product.collection.title. Now save, back in the browser, refresh. Look, we have the same problem as before. Our application is hanging because Django is sending a thousand extra queries Queries to the database to read the collection of all these products. Because when we ask for products, Django is only going to query the product table. It's not going to query the related tables unless we specifically instructed to do so. So here we want to preload the products with our collection.

To do that, before we call the all method, we call select related. And Here we specify the field. We want to preload. So that field is collection and Then we call the all method now back to the browser Refresh our products loaded instantly now. Let's look at our sequel tap So here's our query See what's going on here. We're selecting all these fields from the product table followed by these three fields from the collection table and Here we have a join between product and collection So, when we use the select related method, Django creates a join between our tables. Now, we can also span relationships. So, let's say collection has another field that we want to preload as part of this query. So, we add two other scores here and then type the name of that other field.

Okay? So, this is select related. now. We have another method called prefetch related. So, We use select related when the other end of the relationship has one instance and Like in this case, a product has one collection, right? But we use prefetch related when the other end of the relationship has many objects. What is an example of that? The promotions of a product. So let's go to the product class real quick. Look, each product has one collection, but it can have many promotions. So to preload the promotions, we use the prefetch method. So back to our playground. I'm going to replace this with prefetch related. And the field we want to preload is promotions. Now, before we go back to the browser, I'm going to go back to the template and remove the collection of these products.

Otherwise, we're going to end up with all those extra queries. So save. Now refresh. All right, let's look at our sql tab. So now we have two queries. Here's the first query to read these columns from the product table. Now, after that, we have another query to read the promotions of these products so you're reading three columns from the promotion table, and we have a join between promotion and product. So, essentially, we have two result sets. The first set includes the products. The second set includes promotions. Django reads these sets and then populates our objects in memory. Now, we can also combine these two methods. So let's say we want to load all the products with their promotions and collection. So after the call to prefix related, we can call select related to load the collection of each product.

So both these methods return a query set. And that's why we can chain all these methods to build a complex query. Okay, now the order of these methods doesn't really matter. So we can put select related first or last. It doesn't really make a difference. So now let's save the changes and add the collection back in our template. So product.collection.title. Now let's test the result. Great. So we see the collection of all these products. Now in our SQL tab, we have two queries. The first query is to read the products and their collection. So here we have a join between product and collection. And the second query is to read the promotions of all these products. All right, here's your exercise. I want you to write a query to get the last five orders with their customer and items, including the product reference in each order item.

So pause the video, spend a couple minutes on this, and come back see my solution. all right, here's the solution. We want to get a list of orders. So we should start with the order class. Then we go to objects. Now we want to preload these orders with their customer. So this is where we call select related to preload the customer field. Now we don't want to show all the orders, we want to show the last five orders. So first, we need to sort them by place that in descending order. So the latest orders come first. And then we use array slicing syntax to pick the top five orders. Okay. Now let's test our implementation up to this point. So we store the result in query set.

And then to keep our code clean, we should change the name of this key from products to orders. Next, we go to our template and rename products to orders and product to order. Here we can render the idea of each product followed by, let's say, the first name of the customer. So two pairs of braces to render order that customer that first name save now here we can see the last five orders and the customer. Beautiful. Let's look at our sequel tab to make sure we don't have any extra queries. So we have a single query to read the orders and their customer. So we are selecting all these columns from the order table, followed by all these columns from the customer table.

And here we have a join between order and customer tables. Great. So let's move on to the next step. Now we should preload the items of these orders. So here we call prefetch related because each order can have many items. Now, what is the name of the field we're going to query? Well, let's have a quick look at our order class. So each order has Three fields place that payment status and customer. We don't have a field called items, but look at the order item class Here we have order which is a foreign key to order So Django is going to create the reverse relationship for us. Now. What is the name of that relationship? That is order item underline set This is the convention that Django uses to create the reverse relationship If you don't like this name, you can set it using the related name argument.

So over here We can set related name to items. So with this implementation, the order class is going to have a field called items. Now, I don't want to set this because we're going to end up with Inconsistency in our code. And I don't want to change the name for every relationship. So for the purpose of this exercise, let's go with Django's convention. So back to our playground. We want to prefetch order item underline set. So The name of the target class in lowercase followed by an underscore and set. Okay Let's see what we get after this point. So Refresh our page loaded instantly. Let's look at the sequel tab We have one extra query to load These order items. So we're loading order items for Orders in this list.

So everything is fine so far Now the last step is to load the product referenced in each order item So, over here, where we are prefetching the order item, we can span the relationship by adding two underscores, and then we type the name of the target field, which is product. Save. Now, refresh. Let's look at our query one more time. Now we have one extra query to read the products referenced in these order items. So, we are reading all the columns from the product table where the idea of the product is in this list. And this list is the list of order items we have queried so far. So this is how I want you to write code. Write a little bit of code, test your code, make sure everything works before doing a big step.

With this technique, we can prevent a lot of crazy issues down the road.

Aggregating Objects:

Sometimes you want to compute summaries like max or average price of our products. This is where we use the aggregate method so called product that objects that aggregate and Here we pass an aggregate object. So first on the top we import an aggregate class so from Django that DB that models that Aggregate we import one of these classes. So we have count we have max min average and sum. So let's say we want to count our products. Here we create a count object and specify the field we want to use for counting. If we use ID, we can count the total number of products because every product has an ID. But if we use a different column, let's say description, and assume that description can be null, this will count the number of products that have a description, okay?

So the proper way to count the total number of records To use the ID or the primary key field. Now the aggregate method doesn't return a query set because once we calculate a summary value, It doesn't really make sense to do anything extra with it. We're not gonna add filters or sort data and so on. So here we get a dictionary. Let's call that result and add that to our context object over here. So I'm gonna add a new key called result and set it to result, Okay. Now we go to our template and I cleaned up the code from the previous lesson, so we don't have a for loop for rendering the list of orders. So right after our hello world message, we can render the result object.

Now, let's see what we get. Refresh. So we get a dictionary with one key value pair. Look at the name of the key. It's ID double underscore count, because we use the ID column for counting objects. And as you can see, we have a thousand products in this table. Now we can easily change the name of this key. So back to our code, over here. We can pass a keyword argument. We can say count equals this. Now, Take a look. So the key is your name pretty simple Now we can also calculate multiple summaries. So in addition to the total number of products, We can also calculate the minimum price of our products. So we can pass another keyword argument, say min price, and set it to min of

unit underline price. So here's what we have. you have two keyword arguments, count and min price. Let's see what we get in the browser. So refresh. There you go. Min price is a decimal object with this value. Now, one last thing before we finish this lesson. Since aggregate is one of the methods of query sets, We can apply it wherever we have a query set. So we can filter our products and pull out the products in our given collection. Let's say collection number one. And then calculate these summaries over that data set all right so that's all about aggregating objects. Now I've attached a PDF below this video where you can find a bunch of exercises and their solutions. Make sure to do these exercises before moving on to the next lesson.

Annotating Objects:

Sometimes we want to add additional attributes to our objects while querying them. This is where we use the annotate method. So we go to customer.objects and call the annotate method. Now let's say while querying customers, we want to give each customer a new field called is new and we want to set it to true. So let's get the result and store it in a query set. Now in our context object, I'm going to add a new key value pair called result and Set it to list of query set. Now, in our template, we are not doing anything in this lesson. So let's just focus on the code here now Here in the browser, refresh, we get this error saying query set that annotate received none expression.

What this error is telling us is that we cannot pass a Boolean value over here. We need to pass an expression object. So, in Django, we have this expression class, which is the base class for all all types of expressions. Derivatives of this class are value, for representing simple values, like a number, a boolean, a string. We have f, which you have seen before. Using the f class, we can reference a field in the same or another table, right? So an f object is essentially an expression object. We also have func for calling database functions. So all these database engines have a bunch of functions for manipulating data. To call those functions, We use a funk object. I'm gonna show you that in a second.

We also have aggregate, which is the base class for all aggregate classes. So those aggregate classes you learn about, like count, some max admin all these classes derived from the aggregate class. Okay, So let's go through a few examples. So back to our code, we cannot pass a boolean value here. We need to pass an expression object. The simplest expression object is a value object. So on the top, from Django, dot db dot models we're going to import the value class. And now we're going to wrap this value inside a value object. So we pass value of true. Okay. Now save. Take a look. No more errors. Let's look at our query. So I'm going to execute this query. So in the list of customers, now we have a new column called is new, which is populated by one.

which is the numeric equivalent of a boolean true. So when Django reads this data, it's going to give each customer a new field called is new and it's going to set it to true. Let's look at another example. This time we want to give our customers a new field called new ID. And for this demo, we want to set this to the same value as the ID field. So we need to reference another field in this model. To do that, first we need to import the F class from django.db.models. And then over here, we can reference the id field. Now, let's run our query one more time. So, take a look. Now we have a new column called new ID, which is populated with the same value as the primary key.

Now, we can also perform computations here. So, for example, we can add one to the id to generate the new ID. Now, take a look. So now the new ID is the original ID plus one.

Calling Database Functions:

Alright, now let's see how we can call a database function. So from the same module, we're going to import the font class. Now we're going to give our customers a new field called full name. And this is where we're going to call the concat function of a database engine. So pretty much all database engines have this function for concatenating strings. So here we're going to create a func object and give it a bunch of arguments. So we want to specify two fields here. first name and last name. How can we reference fields? Using f objects. So f of first name and f of last name. Next we need to give this a keyword argument that specifies the target function. So we set function to concat.

Now if we run our query first name and last name are going to be concatenated but we also need to add a space in between. So we cannot pass a white space here. We need to pass an expression object. So we need to wrap this inside a value object. Okay. Now, let's run our query and see what we get. So refresh. And here's our query. Take a look. So in our query, we are selecting all these columns from the product table. But we also have a new computed column using the concat function of MySQL. So we're concatenating first name with a space and last name. And here in this table, we have the full name of each customer. Great. Now, there is also a shorthand to achieve the same result.

So back to our code. First, we're going to import the concat class from Django. So from Django.db.models.functions, we're going to import the concat class. And then we're going to use that class instead of a func class. So let me duplicate this query so you can compare them side by side. In the second implementation, we are not going to create a func object. We're going to create a concat object. Now here we need to specify our fields. So first name and last name. Now note that over here, I'm not wrapping these fields inside an F object. So that is the benefit of using the concat class. But to add a white space, We need to wrap it inside a value object because otherwise Django thinks this is a column in our table.

So let's wrap this inside a value object. Good. Now, unlike the first method, we don't have to specify the target function because that is reflected in the class itself. So the code is shorter and gives us the exact same result. Now, if you're curious about Django database functions, just Google Django database functions on this page. You can see various functions in different categories. For example, we have a bunch of functions for working with dates We have mathematical functions. We have functions for manipulating text and so on These functions are common across all database engines, but different database engines have their own unique functions so to call those functions you have to go back and use a func object.

Grouping Data:

Now let's say we want to see the number of orders each customer has placed. So we're going to annotate our customers with a new field called orders count. And here we're going to use the count class. Now look, VS Code offers to automatically import this class for us. So we just press enter and it's imported on the top. Now let me show you a shortcut. We can also import the count class from the models module. And with this we can avoid an extra import statement. Now, what is the difference between these two classes? Nothing. The models module simply imports the count class from the aggregates module, and then re exports it. Okay, so let's delete this line. Good. So we create a count object.

Now, what field should we use to count the number of orders? Well, let's go to our customer class. So our customer has these fields, but we don't have a field called orders. So let's look at the order class. Here we have customer, which is a foreign key, right? And you know that. With this implementation, Django is going to create the reverse relationship for us. So here in the customer class, we're gonna have a field called order on the line set so the singular name of the target class Followed by on the line set and this is all in lowercase. Now, for some reason that is known to Django developers, we cannot use the same to count the number of orders for each customer and

Let me show you what I mean. So back to our playground. If we type order on the line set here, we get an exception. Look, cannot resolve keyword order set into field. Our choices are address, birthdate, email, first name, ID, last name, membership, order and phone. So the name of the field that we should use for counting is order. Why? Honestly, I don't know. So back to our code, let's change this to order. and rerun our query, great. Now let's look at our query. So, now we are selecting all these fields from the customer table, and we're also counting the number of orders. Now look, we have a left join between customers and orders, because not every customer has an order if you are not familiar with left joins, you really need to look at my sql course, I covered this in detail there.

And finally we have, a group by clause for grouping the number of orders for each customer. So if you run this query, we can see the number of orders for each customer.

Working with Expression Wrappers:

So let's quickly recap what you have learned so far. We talked about the expression class, which is the base class for all types of expressions. Derivatives of this class are value for representing simple values like a boolean, a number, a string. We have f for referencing fields, func for calling database functions, aggregate, which is the base class for all aggregate classes like count, sum, and so on. Now in this lesson, we're going to talk about another derivative of the expression class called expression wrapper. We use this class when building complex expressions. So here's an example. Let's say we're going to annotate our products and give them a new field called discounted price. So here we're going to get the unit price and multiply it by let's say 0.8.

Now, if we run this query, we're going to get an exception. So take a look. So we get this error saying expression contains mixed types, because we have mixed decimal field, which is our unit price field, with a float field, which is our 0.8 number. Now the error is saying that we must set the output field. So to solve this problem, first we need to import the expression wrapper class from the same module, from the models module. Now we should wrap this expression inside an expression wrapper object, and that's where we specify the type of the output field. So to keep our code clean, I would prefer not to wrap this expression right here, I would prefer to create an expression wrapper object over here, then we move this right here.

And as the second argument to the constructor of this class, we specify the output field. So we need to decide what is the type of the output field. For monetary values, we should always use a decimal field. because float fields have a rounding issue and are not very accurate. So, back to our code, we set the output field to a decimal field object. So make sure to create an object, we're not simply passing the name of the class, we're creating an object, okay? Now, we can store the result in a separate object like discounted price, and then we use that over here. Now, let's run our query. Okay, no more error. So let's see what is sent to the database. There you go.

We have a new computed column, which takes the unit price and multiplies it by 0.8. So this is how we can use expression wrapper objects. Now once again, There is a PDF below this video where you can find a bunch of exercises and their solutions. So spend a few minutes on those and then come back, see the next lesson.

Querying Generic Relationships:

let's talk about querying generic relationships. So earlier in the course, we created the tags app with two models. We have tag and tagged item. Now if you remember, we decided to use the content type framework to decouple this app from the store app. So this app knows absolutely nothing about the store app. It doesn't know we have a model called product. So with this decoupling, we can reuse this app in any kind of project. Tomorrow we can use it to tag articles, blog posts, videos and so on. But now let's see how we can find the tags for a given product. So let's go to our database. And look at this table Django content type. In this table, we can see all the models we have in our application.

So in the admin app, we have a model called log entry, we didn't create this, this is built into the admin app that comes with every Django project. Similarly, in the auth app, we have three models group permission and user. In the likes app, we have a model called liked item. And in the store app, we have all these models that we have defined so far. Okay. So this is the content type table. Now let's look at the tagged item table in the tags app. So here we have a few columns, object ID, content type ID, and tag ID. So to find the tags for a given product, first we have to find the content type ID of the product model. in this table, currently in my database, we have the product model and the content type ID for this model is 11.

So once we find this, then we can go to this table and write a query to filter all records where content type ID equals 11 and object ID equals the ID of the product whose tags you want to find out. That is the big picture. Now let's implement this in code. So back to our playground and First we have to import a few classes. From django.contrib.contenttypes.models, we're going to import the content type model. This is the model that represents the content type table that you just saw. Then from store.models, we're going to import the product class. And finally from the tags.models module, we're going to import the tagged item class. Now first we need to find the content type ID for the product model.

So here we say content type dot objects. So because content type is a model, it has the objects attribute just like all the models we have created so far. Now here we have a method called get for model. This is a special method that is only available in this manager. Remember I told you that the objects attribute returns a manager object which is a gateway to the database. So content type manager has a special method called get for model. So we call this method and give it our product class. Now we get a content type instance. Now more accurately, this object represents this row in our content type table. So this is how we can find the content type ID for the product model.

Now you might be wondering why we're not just using the number 11. Because we cannot rely on these numbers. In your database, in your development database, the content type ID for product might be 11. But in your production database, it might be something else. That's why first we have to find the content type ID for the product model. Now that we have this, we can use it to filter tagged items. So we say tagged item that objects that filter, you're going to give it two filters. First is content type, which we set to our content type object. And then we set object ID to the ID of the product those tags you want to query let's say product number one. This should be calculated dynamically.

For example, depending on the product that the user is looking at, we're going to get the product id from the url and pass it here. But for now, let's not worry about that part. So this will return a bunch of tagged item objects. Now, back to our database, look at our tagged item table. In this table, we have a field called tag ID, which is a foreign key to the tag table. So the actual tag is not stored here, is stored in this other table. So we need to preload this field. Otherwise, we'll end up with a lot of extra queries to the database. So back to our code. Here we're going to call select related to preload the tag field. Now let me show you a better way to format this code.

It's getting a little bit ugly. So we can put this on a new line. Now we get an error because Python interpreter doesn't know that this line is the continuation of the first line. to solve this problem, we add a backslash here, and now the error is gone. So I would like to indent this line, and then one more time, we put filter on a new line and add a backslash here, and then format our code like this. I think this is nicer, especially when we're building a complex query by chaining a bunch of method calls here. So this returns a query set. Let's get it. And now we can add a new key value pair, in our context object, so we can call that result or we can call it tags and set it to list of our query set, list of query set.

Now, let's run a query and see what happens. So refresh, good, no errors. So we have two queries. The first query is for finding the content type id for our product model. And the second query is for reading the tags for a given product. Now currently we don't have any data in these tables, but if you're curious, you can just populate these tables and render the tags on the page. So this is how we can get the tags for a given product. Now in the next lesson, I'm going to show you how to encapsulate this logic inside a custom manager so our code is cleaner and easier to work with.

Custom Managers:

So our tags app is decoupled from the store app, but writing code like this is not ideal. Every time we want to find the tags for a given object, first we have to find the content type, then we have to go to the tag item model, we have to preload the tag field and apply a filter. There must be a better way. So wouldn't that be nicer if we had a method like tagged item dot objects dot get tags for product one. Obviously, calling this method is much nicer and cleaner than writing all this code. So let me show you how to implement this method. To do this, we have to build a custom manager for the tag item model. So we want to replace the default manager that is returned from the objects attribute, you want to replace that with a custom manager.

So let's go to this class. Before this class, we have to create a custom manager. So class tagged item manager, this class should inherit from the manager class in the models module. So this class is the base class for all managers. Now here we're going to implement that custom method. So get tags for, now the first parameter should be self, so this becomes a method of this class. We also need two more parameters, object type and object id. Now let's go to our playground and Grab all this code and move it to our new method Okay, let's fix the indentation good now we're going to replace product with object type and One with object ID as simple as that Now finally, we're going to return this query set so we can return it right away So we're done with this

now we need to use this manager in the tagged item model. So here's the tagged item class. I'm going to add a new attribute here called objects. We're going to set it to an instance of the tag item manager class. As simple as that. Now we can go back to our playground and use this new method to get the tags for a given object, like a product.

Understanding QuerySet Cache:

Let's talk about the caching mechanism built into query sets. So let's call product that objects that all and Store the result in a variable called query set. Now, you know that when we convert this query set to a list, Django is going to evaluate this query set and that's when it's going to go to the database to get the result. Now, this is an expensive operation because reading data from the disk is always slower than reading it from the memory. So when Django evaluates this query and gets the data from the database and It's going to store it somewhere in memory called the query set cache. So the second time we convert this query set to a list, Django is not going to evaluate this query again.

It's not going to go to the database. It's going to read the result from the query set cache. Let me show you. So back to the browser. Refresh. Now let's look at our queries. Look. We have a single query for reading the products. So the second time we convert this query set to a list, Django is going to read the result from the query set cache. Now the same thing happens if you access an individual element from this query set. Again, Django is going to read this object from the query set cache. Now one thing you need to know about caching is that caching happens only if you evaluate the entire query set first. For example, over here, first we're evaluating the entire query set and then accessing an individual element.

In contrast, if you accessed an individual element first, Then converted our query set to a list. you would end up with two queries to the database. Take a look. So save, refresh, and look. here. We have two queries. With the first query. We are reading the first product in this table. And With the second query you're reading all products. So keep this in mind. Even though caching is a great optimization technique, if you don't structure your code properly, Caching is going to come back and bite you. So this is all about caching. Next we're going to talk about creating objects.

Creating Objects:

so far we've been only querying data. Now let's see how we can insert a record in the database. So let's create a collection object and set its title to let's say video games. Now every collection has a featured product which is optional, so we don't have to set it, but let me show you how to set this field and create a relationship. There are basically two ways to set this field. We can set this to a product object, like the product with id 1 here we could also use PK, and with this we don't have to remember the name of the primary key field, so it can be ID, code, or whatever. Now the second way to set this field is like this.

We can set collection that featured product id to one so we can either use a product object or use the value of the primary key field. Now either way, this product should exist before we can create this collection. In other words, we cannot create this collection and this product at the same time. this is how relational databases work. The parent record should exist before we can create the child record, right? So, assuming that we have a product with this ID, now i'm going to remove the last line and set the feature product to a product object, okay? Now, there's another way to initialize this collection. Instead of setting these fields individually here, we can set them using keyword arguments when creating a collection object.

So, here we can set title to video games. Now, there are a couple of problems with this approach. One problem is that if you noticed here, we didn't get IntelliSense. So as I was typing title, look, we are not seeing the name of this field. We're just seeing the name of other classes imported in this module. So that's the first problem. In contrast, when we use the dot operator, look, here we can see the name of the fields in this class. Okay. The second problem with using keyword arguments Is that these keyword arguments don't get updated if you've renamed these fields. Let me show you what I mean so let's go to the collection class and Rename the title field to name now back to our playground.

See what happened The title field is updated here, but not over here. So the keyword arguments don't get updated That's why I person don't like this approach even though sometimes it can make your code shorter. So Let's set The fields individually here. Now, to insert this collection our database. All we have to do is call the same method. So every collection has this method for saving data. Now, in this case, because we haven't set the idea of this collection, Django will treat this as an insert operation. So it's going to create this collection, Okay. Now, there is another way to write this code. There's a shorthand. So instead of writing these four lines, we can call collection that objects that create. And here we have to set the fields using keyword arguments.

So we set name to whatever and featured product. Look again, we don't have IntelliSense. So we can set feature product id to one. So the create method is available in the manager object. And what happens inside this method is exactly what we have on the top. So this method is going to create a collection object It's going to initialize this object based on this keyword arguments. And then it's going to call the save method to insert this collection in the database. And of course, at the end, it's going to return the actual collection object. So either way, here we can read the idea of this new collection. That's going to be available here as well. So the second approach appears shorter. But again, here we have the problem with our keyword arguments.

So if you rename this field, our code is not going to get updated. So I always prefer to use the traditional approach. And before we finish this lesson, let's go back to our collection class and rename this field back to title. Good. So let's make sure everything works up to this point. Save, back to the browser, refresh. Now let's look at our query. So we have an insert into statement for inserting a new collection with these values.

Updating Objects:

Let's talk about updating objects. So here we have this collection. Now to update it. All we have to do is set its ID or primary key. So in my database, this new collection that we created, its ID is 11, So I'm gonna set PK to 11. And by the way, even though I'm using a keyword argument here, This is not gonna be an issue in the future, Because if tomorrow we decide to rename the ID field to something else, our code is not gonna break. PK will always work, okay? So here we have an existing collection now, let's set its title to games and featured product to none save Refresh good. Let's look at our query So here we have an update statement for setting the title and featured product ID Perfect.

Now, let me show you something tricky Let's say we only want to update the featured product for this collection. So we don't want to set the title Save Refresh. And Look at our query one more time. Now, something crazy is happening here. Can you tell? Look at this line. Django is setting the title of this collection to an empty string. And this causes data loss. So as part of updating this field, we end up losing the title for this collection. But why does this happen? Here's the reason. This collection object that we have in memory, by default, is title is set to an empty string. So even if we don't explicitly update this field, Django is going to include it in our SQL statement.

And this is where Django is different from some of the other ORMs you might have worked with. Some other ORMs out there have this feature called change tracking. So the SQL statement that is generated here will be based on the fields that are updated. So to properly update an object in Django applications, first we have to read it from the database. So we have all the values in memory, then we can update it. over here, instead of creating a collection object, first we have to get it from the database. So get pk equals 11, good. Now we have a collection with all the values, so we can update certain fields. Now you might worry that reading this object first may cause a performance penalty.

But speaking from experience, I'm telling you, this is not an issue in most cases. So don't try to prematurely optimize your code. Always profile first, see if you really have a performance problem in this part of code, before optimizing it. Okay. So once again, reading this object first before updating it is not an issue in most cases. But let's say in your application, you really have a performance problem in this part of your application. So let me show you how we can avoid this extra read. In the previous lesson, we talked about a convenience method called create. So collection that objects that create. Now, on this manager, we have a similar method called update for updating objects. And with this method we can avoid reading this collection first before updating it.

So we can update it directly in the database. So we call this method and pass one or more keyword arguments. So here we can set featured product to not. And that means here we have that problem with keyword arguments that i told you about. So with this new method we can avoid reading an object first but at the cost of making our code a little bit fragile. Okay? Now this update method we'll update this field for all objects in our query set. So if we use it like this, it's going to update the featured product for all collections in the collection table. But here we want to target a particular collection. So before calling the update method, first we have to filter to get the collection with the ID 11 and then we can update its featured product.

So let me comment out these few lines and run our code one more time. Refresh. Good. So take a look. With this new implementation, our update statement is only setting the featured product, okay? All right, that's it for this lesson. Next, we're going to talk about deleting objects.

Deleting Objects:

All right, let's talk about deleting objects. Again, we have two options here. We can delete a single object or multiple objects in a query set. So if you have a collection, we can delete that collection by calling collection.delete. Pretty straightforward. Now to delete multiple objects, first we need to get a query set. For example, we can say collection.objects.filter. Let's say we want to delete all collections whose ID is greater than five. So here we say ID greater than five. This returns a query set. Now on that query set, we can call the delete method to delete all these objects in one go. So these are two ways to delete objects in Django. Now it's your turn. Download the PDF below this video and spend a few minutes on the exercises.

When you're done, come back and see the next lesson.

Transactions:

Sometimes we want to make multiple changes to our database in an atomic way, meaning all changes should be saved together, or if one of the changes fails, then all changes should be rolled back. A typical example is saving an order with its items. So let's create an order object. Here we should set the customer or the customer ID field, and then save the order before we can save its items. This is how relational databases work. We should always create the parent record first before we can create child records, okay? Now, after this, we can create an order item. Here we should set a few fields. First, we should set the order to the order that we just created. Then we should set the product or product ID.

We should also set the quantity and unit price. With all this, now we can save this order item. Now, imagine While saving this order item, something crazy happens. We get an exception what's gonna happen our database is gonna be an inconsistent state. So we'll have an order without an item. We don't want this to happen. So this is where we use a transaction. So we're gonna wrap both these operations inside a transaction, and Either both of this will be committed together, or if one of these operations fails, then both changes are gonna get rolled back. So on the top, from Django the DB package We're going to import the transaction module. Now in this module, we have a function called atomic, which we can use as a decorator or a context manager.

So we can apply it as a decorator to this view function. And this will essentially wrap this entire view function inside the transaction. So all the code that is here will be run inside a transaction. Okay. Now sometimes you want to have more control over what parts of your view function need to be inside a transaction. In those cases, You can use this as a context manager. So I'm going to remove this decorator from here. Let's say we want to wrap these lines inside a transaction. But let's say we have some other code here that is not going to be part of a transaction. So we're going to leave this out and then wrap the rest of the code inside a with block.

So we say with transaction.atomic. This returns a context manager. Now we need to indent this code. Now let's test this. So I'm gonna run the code. Refresh. Good. So now in our database, let's open a new query console and say, select everything from store underline order and order by ID in descending order. So here's our new order order 1001 in this case, both the order and its item were saved successfully. But to demonstrate an exception, let's go back to our code and set product ID to negative one. So we're going to get an exception here because we don't have a product with this ID. Now, back to the browser, refresh. So here's the integrity error that we just got. Now let's look at our database.

So since we have wrapped both these operations inside a transaction, now we should not have a new order because we couldn't save its item. Take a look. So I'm going to rerun this query. See, there is no new order here. So this is how transactions work.

Executing Raw SQL Queries:

Alright, we covered a lot about Django RM. The last thing we're going to cover in this section is executing raw SQL queries. So sometimes implementing certain queries using Django RM can get overly complex. You might end up with crazy annotations or filters. In those cases, you can always write your own query by hand and execute it directly using Django RM. Let me show you. So we're going to call product.objects.raw. So every manager has this raw method for executing raw SQL queries. So we call this and give it a query like select everything from store underline product. So Django is going to execute this query, then it's going to map the column names to fields of product objects and return a bunch of products.

Well, more accurately, it's going to return a raw query set. So we're going to get the query set, but this query set is different from the other query sets you have seen so far. so here we don't have the filter method. We don't have annotate because these methods don't really make sense here. Okay. So for this demo, I'm going to add a new key in this context object called result and set it to list of query set. Now back to the browser, refresh. So here is our query. Great. Now a few things i need to highlight here. In this case, there is really no point writing this query because django can perfectly generate this for us. So use this approach only when dealing with complex queries.

If you end up with complex annotations and filters and realize writing the same query using raw sql is easier and cleaner, then go for this approach. Or if the query that django generates doesn't perform well, then again, this is another case for using this approach, okay? Now, here we can also specify column names. So Instead of selecting all columns, we can select ID and title. And Django is going to load only these two fields. So the other fields are going to be deferred. Okay. Now, sometimes we want to execute queries that don't map to our model objects. In those cases, we can access the database directly and bypass the model layer. So instead of using the raw method, we're going to use a different approach.

First on the top, from Django dot DB, we're going to import the connection module. In this module, we have a function cursor for creating a cursor object. So we get a cursor. Now this cursor has an execute method and here we can pass any sql statements, no limitations. So we can have a select, we can have insert, update and delete. Now after executing our query, we should always close the cursor to release the allocated resources. So the proper way to use a cursor is to wrap it inside a try finally block. So inside a finally block, we close the cursor. So this way, if something goes wrong, we don't end up with an open cursor. But you know that try finally blocks are a little bit verbose.

So it's much easier to use a cursor using a with statement. So let's delete these lines and say with connection that cursor. This returns a cursor object. So using the as keyword, we give it an identifier, like cursor. And now we can use this cursor inside the width block without having to explicitly close it. So the cursor is always going to get closed, even if there is an exception. Okay. So this is how we can execute raw SQL queries. Now one last thing before we finish this lesson, we also have another method here for executing stored procedures. So we can call a stored procedure like get customers and give it a bunch of parameters one to a whatever. This is much better and cleaner than

writing your SQL queries in the middle of your Python code. So you encapsulate your SQL queries inside a stored procedure and then call that in Python. Now that really goes outside the scope of this course, but I covered it in detail in my SQL course. So we're done with this section. In the next section, we're going to talk about the admin interface.

The Admin Site:

Introduction:

Welcome back to another section of the ultimate Django course. In this section, which is the last section of the first part of this series, we're gonna talk about setting up the admin interface for managing our data. We'll talk about various ways to customize the admin interface, how to add computed columns, load related objects, add searching and filtering, implement custom actions, add data validations and more. So by the end of this section, we'll have a full fledged admin interface for managing our data. I'm super excited about this section I hope you are too, so let's jump in and get started.

Setting Up the Admin Site:

Alright, the first thing we're going to do in this section is setting up the admin app. So every Django project comes with an admin interface, which we can access at this address slash admin. So here we have our login screen. Now to create a new user, we have to open a terminal window and run Python manage the pi create super user. Okay, now it's asking for username, I'm going to use admin. Next we need to add email address, so programming with mosh at gmail.com. And for password, I'm going to use 1 2 3 1 2 3 and repeat now it's complaining that the password is too short and it's not secure. But for this lesson, I'm not going to worry about this. I'm going to bypass the validation by pressing Y. Okay, great.

Now we have an admin user. So let's login now we get this error complaining about the session app and because the admin app is dependent on the session app and earlier in the course we removed this app by accident. So we need to add it back in the list of installed apps. So back to our project, let's open up the settings module. Here in the list of installed apps, we're going to add django.contrib.sessions and the order doesn't really matter. So save, now we need to run the migrate command to generate the tables for this app. So back to the terminal, Python, manage.pi migrate. Great. Now, back to our database. Let's refresh. Take a look. We have a new table here called django session.

And this table is used for temporarily storing some user data. So currently it's empty because we haven't logged in the admin panel. But once we log in, we're going to see a new record in this table. So, back to the browser. Let's refresh. Good. So this is our admin interface where we can manage our groups and users. So currently we have a single user in this database. Let me zoom out a little bit so we can see things more clearly. So we have this admin user with this email. We have a bunch of filters on the right side. We're going to talk about them soon. We also have the list of groups which is currently empty. So these users and groups are stored in the tables of the auth app.

So back to our database. Look, the auth app has a bunch of tables for managing groups, permissions, and users. So let's have a quick look at the user table. Here we have one record. That is for the admin user. The password is stored in encrypted format. And here we can see the last login date and time. We can see that this user is a super user, which means it has all the permissions. Here's the username. We don't have the first name and last name. We can set that in the admin interface later. We can see the email. and here we can see is staff which means this is an admin user for this website now if you forget your password you can always reset it in the terminal so back to the terminal we run python manage.py change password admin here we can enter a new password but i'm going to leave it out so ctrl and c to exit okay now back to the browser let's go to the home page of our admin panel so here on the top we can see django administration

Let's see how we can change this to something more meaningful. Now back to our project, let's open the storefront folder and go to the URLs module. Now in this module, all the URLs of the admin app are hooked up at this URL. So any URL that starts with admin is going to be routed to the admin app, okay? Now, because the admin module is imported here on the top, this is a good place to apply a bit of customization for the admin app. so we can say admin.site.site underline header, we can set it to store front admin, now save and refresh so we change the header on the top, great, we can also change the label here, for example we can set admin.site.index title to let's say admin, now save and refresh, so the title is changed as well

So now that our admin panel is ready, next we're going to register our models so we can manage them here.

Registering Models:

Now let's see how we can register our models so we can manage them in the admin site. So back to our project, let's go to the store app. Every Django app has this module called admin, and this is where we write all the code for customizing the administration panel for that app. So the first thing we need to do is registering our models for the admin site. So the admin module is imported on the top. Here we're going to say admin.site.register. Let's say we want to register the collection model. So on the top, from the current folder, we're going to import the models module. And then here we can pass models.collection. So back to the admin panel, refresh. Now look, we have the store app and currently only one of our models is registered in this app.

So we can look at the list of collections, we can add a new collection or change one of the existing ones. So let's look at our collections. Now what you see here is the default representation of a model object. So we're going to change this and instead we want to show the title of each collection. So back to our project. Let's go to the collection class. Now here's a question for you. How can we change the string representation of an object in Python? We override the magic stir method. So every Python object has this magic method that is called when that object is converted to a string. So here we type def double underscore stir double underscore and press enter. So this code is automatically created.

Now what you see here is type annotation. I talked about this earlier in the course. This is basically saying that this method returns a string object. Inside this method we have the default implementation which returns a string like this. Now we're going to change the default implementation and instead we're going to return self.title. Now say take a look. Beautiful. So now we can see all the collections. The first collection doesn't have a name and this happened in the previous section when we were updating data. So there we made a mistake and ended up storing an empty string for the collection title. So let's select this and delete it. Yes, I'm sure. Good. Now, there is a problem here. Our collections are not sorted.

So let me show you how to sort them. Back to our collection class. Here we're going to define a meta class for specifying the default ordering of collection objects. So we create a new class called meta, we talked about this earlier in the course, and here we said ordering to a list of fields. So here we want to sort the collections by their title, so we add that here. Good. Now take a look. Beautiful. All collections are sorted alphabetically. Now as your exercise, I want you to repeat the exact same steps and register the product model so we can manage our products in the admin panel. Alright, here's my solution. So here in the admin module, I registered the product model. And here in the product class, I overwrote the stir method to return the title of each product.

And I also implemented this meta class for ordering the products by their title. So now in the admin side, we can see the list of products. And as you can see, our products are sorted alphabetically. Great. So next, I'm going to show you how to customize the list page.

Customizing the List Page:

Now let's see how we can customize the list page. So I'm going to show you how to add new columns here, how to make them editable, how to change the number of items and so on. So back to our admin module. First we need to create a new class called product admin. We could call it anything, but by convention we use the name of our model like product followed by the word admin. Now with this class we can specify how we want to view or edit our products. So this class should inherit from admin.modelAdmin. Now in this class, we can set a bunch of attributes to customize the list page. For example, we can set list display to the list of fields we want to display, like title and unit price.

Now that we have this new class, we need to pass it while registering the product model. So we say product admin. This is one way, but there is also a shorter way. We can use the register decorator on this class. So we say admin, register and here we pass models that product so you're saying that this class is the admin model for the product class. And with this, we don't need the last line anymore. So that is shorter. Now, back to the browser. Refresh. So look, we have a new column where we can see the price of each product. Beautiful. Now we can sort by this column as well. Now we have to sort columns. First, our products are sorted by unit price and

and then by title. If you don't want to sort by title, you can just click on this icon where we can see a cross. Good. So now our products are sorted by unit price in ascending order. We can also change the sort order. Beautiful. Now imagine if you wanted to code all of this by hand. It would probably take you a couple weeks if not longer. So let's do some more customization. Another attribute we can set is list editable and With this, we can specify the fields that can be edited on the list page. For example, if we set this to unit price and refresh, now we can edit the price of a bunch of products in one go. For example, we can change this to 90 and this one to 95.

Now down the bottom, we have the save button. So we save two products in one go. Now back to the code, we can also set list per page. to let's say 10. And with this, we're going to see 10 products on each page. So save and refresh. Look, now we have 10 products and we have this beautiful pagination on the bottom. So customizing the list page is really simple. Now, if you want to see the complete list of options you can set here, just google django model admin. On this page, on the right side, over here, look, model admin options. Here you can see the complete list of options you can customize. As we go through the section, I'm going to show you many of these, but we don't have time for all of these.

So I leave it up to you to read about the other options. All right, now it's your turn. Set up this page where we can view our customers. So here we have three columns, first name, last name, and membership. We can change the membership of each customer right here. Very handy. Again, imagine if you wanted to code all of this by hand. That would take you a really long time, right? So build this page. over here, we have 10 items per page, and the default sort order is first name followed by the last name. So spend a couple minutes on this, then come back to my solution. All right, here's what i've done. I've created a new class called customer admin. In this class, I've set list display to first name, last name, and membership.

I've set list editable to membership, ordering to first name followed by the last name, and list per page to 10. Very, very easy. Next we're going to talk about adding computed columns.

Adding Computed Columns:

Alright, let's see how we can add a computed column to the list of products. So here I've added a new column called inventory status. If the inventory of a product is less than 10, here we see low, otherwise we see okay. So let's see how we can implement this. So back to our product admin class. In the list of fields we want to display, we don't want to add inventory because otherwise we would see a number in this list. We don't want this. We want to get this number and Depending on this value. We want to return a string. So here in this class, We're going to define a method called inventory status with two parameters, self and product. So we want to take a product, and depending on its inventory, you want to return an inventory status.

So we can say if product that inventory is less than 10 We're gonna return low. Otherwise, we're going to return okay. Now, instead of inventory, we're going to display inventory status. Take a look. So, this is what we get. Beautiful. But this column is not sortable by default because Django doesn't know how to sort the content of this column. So, to implement sorting, we need to apply the admin, the display decorator to this method. And here we specify the field that should be used for sorting the data in this column. so we said ordering to inventory which is one of the fields of the product model so back to the browser refresh now we have sorting here we can sort in ascending or descending order beautiful.

Selecting Related Objects:

Alright, let's talk about loading related objects. So, here in the list of products, I want to add a new column to show the collection of each product. So, back to the product admin class, here in the list of fields we want to show, let's add the collection field. Now, because collection is a related field, Django will show the string representation of a collection here. So, back to the collection class, remember, earlier in the section, we overwrote the str method of this class. So when showing the products, for each product, Django will call this method to get the title of the collection of that product. Okay? So back to the browser. Refresh. Here's the collection column. Beautiful. Now let's open up Django debug toolbar and look at our queries.

So look, we have seven queries on this page. Now, don't worry about all these queries. Just look at the last one. So over here, we're selecting all these columns from the product table. followed by these columns from the collection table, and we have an inner joint between the product and collection tables. Beautiful. Now, what if we don't want to show the string representation of a collection? In other words, what if we want to show a particular field in this model? Unfortunately, here we cannot use the double underscore notation to reference a particular field like the title field. Now you might say, Mosh, what is the point of this? We could see the title of the collection just by typing collection. So here we get the string representation of a collection.

But imagine we have a special field here we want to show on this page. Let me show you how to do this. First I'm going to remove this extra underscore. So I'm going to define a method by this name. Define collection title with two parameters. Self and product. Because we're rendering a bunch of products. So we're going to get a product and return product.collection.title Now, back to the browser, refresh, let's hide this. So our collection column is here, beautiful, but let's look at our queries one more time. Now this time we have 17 queries. So we have 10 extra queries, because for each product, Django is sending an extra query to the database to read the collection of that product. So in the previous section where we talked about query sets, you learned that.

We have a method called select related, for eager loading or preloading the related fields. Now here we have a special attribute for the same purpose. That is called list select related. We set this to the list of fields we want to eager load. In this case, collection. Now, back to the browser. Let's look at our queries. Now we're down to seven queries. Beautiful. So this is how we can load related objects on the list page. Now as your exercise, I want you to set up the order page where we can see our orders and their customers. So we'll spend a couple of minutes on this, then come back see my solution. All right, let me show you what I have done. So I created the order admin class to show three columns, ID, place add, and customer.

And also in the customer class, I overwrote the star method. So here we are returning an F string for combining the first name with the last name. I also implemented a meta class for setting the default ordering of our customers. So now anywhere we have a list of customers, our customers will be sorted by their first name, followed by their last name. With this now we have a new page called orders. And over here we can see all the orders, the date and time they were placed and their customer.

Overriding the Base QuerySet:

Sometimes we need to override the base query set used for rendering a list page. For example, here in the list of collections, let's say we want to add a new column to show the number of products in each collection. So back to our admin module, this is how we're registering the collection model. Currently, we're not specifying an admin model. So here we need to create a new class called collection admin, which should inherit from admin.modelAdmin. And we should decorate this class with admin.admin. Register. Now here we set list display to title and products count. Now our collections don't have a field called products count, so we need to treat this like a computed field. So we define a new method called products count with two parameters, self and collection.

And here we return collection dot products count. But again, our collection objects don't have a field by this name. so this is where we need to overwrite the query set on this page and annotate our collections with the number of their products. So every model admin has a method called get query set which we can overwrite. So here we have a couple of errors because the generated code includes type annotations and these classes are not imported on the top. So here we have two choices if you're on a mac you can press command and period or if you're on windows you can press control and period to import this class on the top. Alternatively, if you don't care about type annotations, you can just remove this.

Okay, so this is the get query set method with the default implementation. So we're going to the base class and calling the get query set. Now we don't want to return this query set immediately. We want to annotate it with products count. Now here we're going to use the count object for counting the number of products. So let's press enter. to automatically import this. Good. As the argument, we're going to pass product. Okay. Now, back to the page. Refresh. And here we can see the number of products in each collection. Beautiful. But look, this column is not sortable. Because Django doesn't know what field to use for sorting this column. So, to solve this problem, we need to decorate this method with admin.display.

And here we said ordering to the name of the field that should be used for sorting. In this case, products count. Now, refresh, and now we can sort our collections by the number of their products. Beautiful.

Providing Links to Other Pages:

Alright, now let's see how we can add links here. So when we click on these links, we can see the products in each collection. So, back to the products count method. Instead of returning a number, we should return a string containing an HTML link. Now to generate an HTML link, we should import a utility function on the top. So on the top, from django.utils.html, we should import the format HTML function. Now let me show you a cool shortcut. Under the go menu, look at the shortcut for the back command. It's control and hyphen. So if we press control and hyphen, we jump back to where we were. That's pretty handy. So here we're going to call format HTML and give it a format string.

Now this is where we need to type a bit of HTML code. In HTML, we can represent a link using the anchor element with two tags, the opening and the closing tag. Now in between these tags, we can type the link text. In this case, I want to show the number of products. So here we type a pair of braces, and we're going to replace the braces with the number of products. So we're going to pass this expression as the second argument to the format HTML function, okay? Now, for the actual link, we go to the opening tag and set the href attribute to a string like this. For now, let's just send our users to google.com, okay? And then return this value.

Let's test our implementation up to this point. So refresh. We get these links. Beautiful. Let's click on a link. We go to Google. Now let's see how we can send our users to the products page. So here's the products page. Look at the URL. That is admin slash store slash product. We don't want to hard code this URL in our code because this URL can potentially change in the future. So we should ask Django to give us the URL of this page. And to do that, we have to import another utility function. So back to the code from Django, that URLs, we're going to import the reverse function. Now, let's go back to our method. Here we're going to call the reverse function and give it a special argument.

We type admin colon. Now here we need to type the name of our app, followed by the model, followed by the page. So what app are we working on? It's the store app. What is the target model? is the product model. Because we want to send the user to the product list page, right? Now, what is the target page? It's called change list. So, this list of products is called change list, okay? Now, let's test our implementation up to this point. So, we call the reverse function, get the URL, and replace google with it. So, here we type another pair of braces, and because this is the first placeholder, As the second argument to this function, we should pass the URL. Now, back to the browser, let's go to our collections, and click on one of these links.

Great. We go to the list of products, but there is no filter applied here. To apply a filter, we need to append a query string to the URL. So we type a question mark, followed by collection, double underscore ID, and a value. So we need to add this part dynamically. So back to the code, after the call to the reverse function, we're going to add a question mark that indicates the beginning of a query string. Now because we're going to type a long string, I would prefer to wrap this expression in parentheses so we can break it down into multiple lines. So look, we have reverse on one line, then on the next line we have plus question mark, and then plus This is where we need to generate the query string parameters.

And for that we're going to use another utility function. So on the top From Django that utils that HTML module. We're going to import the URL encode function Okay. Now back to everywhere. We're gonna call this function and Give it a dictionary because a query string can contain multiple key value pairs. So that's why we use a dictionary here now the key we're gonna add is collection double underscore id. And the value is collection, which is this collection that we get in this method. So that is collection dot id. Now this returns a number. We should convert this to a string, so we wrap it with a string function. Okay? That's pretty much all we have to do, so let's test our implementation.

Back to the collections page. Let's look at the beauty collection. Now we only see the products in the beauty collection. Let's look at another collection, like cleaning. Okay, beautiful. Now here's your exercise. Go to the customers page and add a new column for viewing the orders of each customer.

Adding Search to the List Page:

Alright, let's see how we can add searching to the customers page. So back to the customer admin class, we're going to set search fields to the list of fields we want to use for searching, like first name and last name. Now, I prefer to sort these attributes, because this makes our code more readable. So I'm going to move list per page up. And the rest looks good. Alright, now, back to the browser, refresh, we get this beautiful search box. So if we search for cat, we see anyone who has cat in their first or last name, okay? Now, what if we search for M? Look at the first customer, Adele Amar. It doesn't make sense to see Adele Amar here, because when we search for M, we expect to see people whose first or last name start with M. So, back to the code.

Here we need to use one of our lookup types. Remember, we talked about lookup types in the previous section. So, after our field, we type two underscores, followed by a lookup type, like starts with. And similarly, over here. Okay? Now, back to the browser, refresh. This time we don't see any customers, because this search is case sensitive. So if we change this to an uppercase M, now our search works. So, to solve this problem, we need to use I starts with. I is short for insensitive, okay? And one more time here. Now, refresh, our search works. If we change the uppercase M to a lowercase m, our search still works. Beautiful. So this is all about searching. Next, we're going to talk about filtering.

Adding Filtering to the List Page:

All right, let's see how we can implement filtering on the products page. So let's say we want to filter our products by their collection and last update. So back to the product admin class, here we should set a new attribute called list filter to the list of fields we want to use for filtering. So collection and last update. Now take a look. Good. So I'm going to collapse the left panel. Great. Now look on the right side, we have this beautiful filtering panel. where we can select a collection, and now we only see the products in that collection. Very powerful. Also down below we have filtering by last update, so we have any date, today, past seven days, this month, and this year.

So we got all these capabilities by writing only one line of code. Imagine if you wanted to code all of this by hand. That would take you a long time, right? Now, we can also create our own custom filters. So let's say we want to add a filter here to see only products with low inventory. So back to the admin module. Here we should create a new class called inventory filter. We could call it anything doesn't really matter. But this class should extend admin that simple list filter. Now here we should set a couple of attributes. One of them is title. And this title will appear over here after by. So I'm going to set that to inventory. And the second attribute is parameter name.

This parameter name will be used in the query string. so when we selected this collection, you can see in our query string, we have a parameter called collection ID. So here we can set parameter name to inventory or anything we want. Now, there are two methods we need to implement here. One of them is lookups. And with this method, we can specify what items should appear here. So once again, we have a couple errors because of type annotations. If you don't care about them, you can just remove them. so i'm going to remove them so we have less code on the screen so we can see things clearly. Now, here we're going to return a list of tuples. So each tuple represents one of the filters here.

Okay? So here's our tuple. In each tuple we should have two values. The first value is the actual value we use for filtering. This can be anything. Here I'm going to use less than 10. the second value is a human readable description. So we can say low and low will appear Over here. Okay. So this is our first item. We could add as many items as we want. But I'm going to stick to one filter here. So this is the first method. The second method is query set. And this is where we implement the filtering logic. So once again, I'm going to remove type annotations. All right. Now, in this method, we're going to write some logic like this. So if self dot value,

This returns the selected filter. If this equals less than 10, which is the value that we defined over here. So as a best practice, we should store this in a variable where we can reference multiple times. But I'm going to leave that for now so we don't get distracted. So if the user has selected this filter, then we're going to take this query set and call the filter method on it. Now look, we don't have IntelliSense here because Python doesn't know the type of this object. So this is the benefit of using type annotations. If we decorated this parameter with its type, which is query set, now look, when we type dot, we can see all methods of the query set class. So here we're going to call the filter method and get products with inventory less than 10.

And of course we should return the new query set. So here's our custom filter. Now to use it, we simply type the name of the class in this list. So here we say inventory filter. Take a look. So here's our new filter by inventory. This is the title that we set in our class. Now, if we select low, we only see the products with low inventory.

Creating Custom Actions:

Alright, let's talk about defining custom actions. So every list page comes with a delete action for deleting multiple objects in one go. But we can also extend this list and register our own custom actions. For example, let's say we want to define a custom action for clearing the inventory of a bunch of products in one go. So we want to set their inventory to zero. Let me show you how to implement this. So, back to the product admin class. Here we're going to define a new method called clear inventory. And of course, we could call it anything. But here we need three parameters. Self, request, which represents the current HTTP request, and query set, which contains the objects the user has selected. Now we should decorate this method with admin.action and set the description to the text we want to appear in the dropdown list.

So clear inventory. Now in this method we can do anything we want for updating objects. So we can take this query set and call this update method. And here we set inventory to 0. So this will immediately update the database and then return the number of updated records. So we can get that value and store it in a variable like updated count. Now to show a message to the user, we call self.messageUser. So every model admin contains this method for showing a message to the user. So we call this method and give it two arguments. The first argument is the request object that we get here. The second argument is the message we want to show to the user. So here I'm going to use an F string and say updated count products were successfully updated.

So we defined a custom action. Now we go to the top over here where we set the attributes of this class, we set actions to the list of actions we want to show to the user. In this case, clear inventory. So we pass the name of the method as a string. Okay? Now, back to our page. Now, look. We have a new action here, clear inventory. So I'm going to select the first two products and clear their inventory. Go. Great. So we get this beautiful message saying two products were successfully updated and as you can see, the inventory of these products is now low. Great. Now here we can also show error messages. So when calling the message user method, we can specify the type of the message as the third argument.

So first we go to the top and from django.contrib we import the messages module. Now we go back to where we were and pass a third argument here. So I'm going to say messages dot now here we should have different type of messages. So we have error, we have debug, info, success, and so on. So I'm going to show an error message. Once again, let's select a couple of products and clear their inventory. Okay, this time we see an error message. So this is how we can define custom actions.

Customizing Forms:

So we talked a lot about customizing the list page. Now let's see how we can customize the form for adding or updating models. So let's add a new product. We get this beautiful form that is automatically generated based on the definition of the product model. Again, very, very powerful. Without Django, if we used a different framework for every model, we would have to create a form like this from scratch. And that would be a lot of work. So let's see how we can customize this form. So back to the product admin class. here we can set fields to the list of fields we want to show to the user. So if we set this to title and slug, now we only see these two fields.

But of course, this doesn't make sense because our product has a bunch of other required fields that we need to fill out, like the unit price and inventory. So if we don't show them to the user, we cannot create a new product. Okay? Now, we have another attribute called exclude, which is the opposite of fields. So with this we can exclude certain fields like promotions. Now, take a look. So the promotions field is no longer here. We also have read only fields read only fields so if you make the title read only now when we refresh, it appears down below and it's read only again it doesn't really make sense here, but i want you to know that these options are available to you.

Now let's remove this and refresh. So let's talk about the slug field. Let's say we want to create a new product called brown shoes. It would be nice if the slug field was auto-populated like this, right? So let's see how we can implement this. Back to our code. Here we can set pre-populated fields to a dictionary. In this dictionary, we can specify how each field in this form can get pre-populated. So we can pre-populate the slug field with the title field. So here we're passing a list, which means we can add multiple fields here, and Django will combine the value of these fields to pre-populate the slug field. Okay? Remove. Now take a look. So refresh. As we type brown shoes, Django uses a bit of JavaScript to pre-populate this field.

But note that if we type something here, and then go back to update the title, the slug field doesn't get changed. So it only gets pre-populated if we haven't touched it. Now look at the collection field. Here we have a dropdown list and this makes perfect sense because we have a limited number of collections. But if you had hundreds or thousands of collections, showing a dropdown list would have a couple of issues. For starters, it would be a terrible experience for the user because the user would see a dropdown list with a hundred items. And also, in terms of performance, getting all these objects from the server and rendering them on the webpage would have a bit of overhead. So let me show you how to convert this

an autocomplete field. So back to our code. Here we can set autocomplete fields to collection. Now before going back to the browser, let's look at our terminal. Here we get an error saying collection admin must define search fields, because it's referenced by product admin that autocomplete fields. The reason we see this error is because Django doesn't know how to search for collections. So we need to set this attribute in the collection admin class. So let's go to collection admin. Okay. And here we set search fields to title because we want to search for our collections by their title. Now refresh, take a look. So here we have an autocomplete field. So we only see 10 collections and other collections are hidden. So we can search for them

by typing a search query here. So every time we type a character, Django is going to send a request to the server to get the collections whose titles match our search phrase. Okay. So these are a few ways to customize this form. But again, if you want to find all the options available to you, just search for Django model admin. On this page, we talked about this earlier, look at model admin options. On this page, you can find all possible ways to customize the lists and forms. Now here's your exercise. Look at the page for adding a new order. On this form, we have two fields, payment status and customer. But our order class also has another field called placed at. Why do you think that field is not visible here?

Pause the video and think about it for a few seconds. Here's the answer. Let's look at the order class. So here's the definition of the place at field. As you can see, we have marked this field to get auto-populated by Django. So if you remove this argument and refresh this page, now we get two fields for setting the date and time. We can select today's date and the current time, or we can go to a calendar and select a particular date. Again, very, very powerful. We're getting all these features with very little code. So let's put this back over here and refresh. Alright. So look at the customer field. Currently, we have a drop down list, which is a very bad experience, because this drop down list is populated with 1000 customers.

So this is a great opportunity to use an autocomplete field. So pause the video and convert this field to an autocomplete field.

Adding Data Validation:

Let's talk about data validation. So by default, our forms include basic data validation logic. For example, because we have defined all these fields as required, meaning none of them are nullable, if we try to save this form without supplying any values, we get all these validation errors. Beautiful. We didn't have to write a single line of code to get this functionality, right? Now let's go through a few validation scenarios and see how we can customize the logic here. First, I'm going to make the description field nullable. back to the product class, let's make the description field nullable. Okay? Now we need to create a migration and apply it to our database. So, python manage.py make migrations, good, and then migrate. Beautiful. So, let's go back to the browser and refresh this page one more time.

So, look, we still get this error saying this field is required even though we made the description field nullable. Why is this happening? Well, back to our description field, this keyword argument only applies to our database. So to make this field optional in the admin interface, we should set another argument called blank to true. Now, let's refresh. Okay, this time we don't get a validation error for the description field. Great. Now, what about unit price? Well, here we cannot type any characters like ABC. Because the type of this field is decimal and this input field only accepts numbers. So that's a great starting point. But what if they set the price to 0 or negative 1 the validation error goes away? Look, we don't get a validation error, but this is not a valid value for the unit price.

So back to our models module. On the top, from Django that core that valid errors We're going to import min value validator In this module, we have a bunch of built-in validators. If you want to see all these built-in validators, just google Django validators. On this page, you can see all these built-in validators. So we have regex validator for validating using regular expressions. We have email validator that is automatically applied to our email fields. We also have url validator that is applied to url fields and so on. Now down below, you can also see we have max value validator, min value, max length, min length, and so on. So here we imported min value validator. Now, back to everywhere. Here's the unit price field.

Let's break this down into multiple lines. So our code is cleaner. Now we set validators to a list of validator objects. So here we're going to create a min value validator object. And in the constructor, we're going to specify the minimum value, let's say one. Now optionally, we can also specify message if you want to I'm gonna stick to the default message. So, back to the browser let's refresh all right now we get this default message saying ensure this value is greater than or equal to one. Great. Now similarly, we could apply the same validation logic to the inventory field, but I'm gonna leave it to you as an exercise. The only thing I'm gonna do here is making promotions optional. Because when adding a product, I don't necessarily want to apply a promotion to it.

So back to the product class, here's the promotions field. So we should set blank to true. Now refresh. The promotions field is no longer required, great.

Editing Children Using Inlines:

So currently we can create a new order, but there is no way to manage the items for an order. So let me show you how we can edit them right here. So back to our code, here's our order admin class. Before this class, I'm going to create a new class called order item inline. Now this class should inherit from admin dot either tabular inline or stacked inline. We're going to talk about the differences in a minute. So let's go with tabular inline. now in this class, we set the model to models dot order item okay now here in order admin we should introduce this class as an inline. So first, I want to sort these attributes. And over here, we're going to set inlines to order item inline okay now take a look.

So below the order, we have this beautiful table where we can manage the items for this order. So our product is currently a drop down list, we want to convert this to an autocomplete field, so back to our code, this inline class we have over here, indirectly inherits from model admin, so all the attributes you have learned about so far, also apply here, so here we're going to set autocomplete fields to product, now take a look, so now we have an autocomplete field, beautiful, so we can select the product, set its quantity, as well as the unit price, we can delete them, the reason you see three rows here is because these are three placeholders. If you don't want to see these placeholders, just set extra to zero.

Now take a look. So by default, we don't have any rows here. We can add any order item, set it right here. We can also set the minimum and maximum number of items for our order. So over here, we set minimum to one and maximum to ten. Now take a look. we should always specify one item, otherwise we're going to get a validation error. So this is a tabular inline. What we have here is a table of rows and columns. Now in contrast, we can replace this with stacked inline. And now each item will be represented as a separate form. So this is how we can edit children using inlines.

Using Generic Relations:

Now let's talk about using generic relationships. So here in our product form, we want to add a new section down here for managing the tags. So back to our code, first we should go to the tags app, and here in the admin module, we should register our tag model so we can manage our tags in the admin interface. So here we say admin.site.register. Now we need to import the tag class on the top. So from the models module in the current folder, we're going to import the tag class and then register it right here. Now refresh. Let's go to the home page of the admin interface. Take a look. We have the tags app and over here we can manage all of our tags.

Now currently we cannot see the label of each tag, so we need to override the stir method in the tag class and that's really easy. So let's go to the tag class and override the stir method. Now here we're going to return self dot label. Okay, refresh. Beautiful. So we can manage our tags. Now using inlines, we can manage the tags in our product form. So let's open up the admin module in the store app. Here's our product admin class. Before this class, we should create an inline class for managing a tag. So we create a class called tag inline. Now this class should inherit from a special class called generic tabular inline. So So on the top, from django.contrib.contenttypes.admin, we should import generic.

We can either import generic stack inline or generic tabular inline. I'm going to go with generic tabular inline because that's going to give us a kind of more compact interface. Okay, now let's go back to where we were. So this class should inherit generic tabular inline. So this is a tabular inline, but we use this with our generic objects. So in this class, we should set model to tagged item. Okay. Now that we have this inline class, we're going to go to our product admin and set inlines to tag inline. And of course, I prefer to sort these. So let's move this right here. Good. Let's see what we get off to this point. So back to the homepage. Now, in the store app, we add a new product.

Okay. So over here, we have a form for adding tags. Now, just like before, we have a dropdown list, which doesn't really make sense for tags because we can have hundreds or thousands of tags. So we're going to convert this to an autocomplete field, which is pretty easy. Here in the tag inline class, we set autocomplete fields to tag. Now take a look. All right, we should have an error. So here in the terminal, django is complaining that model admin must define search fields because it's referenced by tag inline dot autocomplete fields. So once again, Django doesn't know how to search for tags. So let's go back to the admin module in the tags app. Now here, we need to create an admin class for our tag model.

So class tag admin should inherit from admin.modeladmin and in this class we set search fields to label. Now we should decorate this with admin.register. Good. So refresh. Okay, the error is gone. Now we have an autocomplete field for looking up tags. Beautiful. So this is how we can use generic relationships in our forms. But there is a problem in our implementation and we're going to talk about that in the next lesson.

Extending Pluggable Apps:

The last thing we're going to talk about in this section is extending pluggable apps. So here in the admin module of the store app, look on the first line, we're importing the tagged item class. So that means our store app has a dependency to the tags app. So we cannot build and deploy it independently. Remember, earlier in the course, I told you that ideally our apps should be self-contained so we can easily plug them into new projects. So here we need to decouple the store app from the tags app. But how? That's actually very easy. So here are the tags and store apps. You want to build and deploy them independently of each other. So none of these apps should know anything about each other.

Okay. Now in this project, we're going to create a new app, which we can call store custom. And this is the customization of the store app, which knows about both these apps. This is very specific to our project. So we're not going to deploy this app for others to reuse. Okay. So here in the terminal, let's run Python. Manage.py, start app, store, custom. Good. Now, we have this new app, and in this app, in the admin module, we're going to combine features from the two pluggable apps. So, from store.admin, we're going to import product admin, and from tags that model we're going to import the tagged item class. Okay? Now why do we have an error here? That should be models. Okay. Good.

So back to the admin module of the store app. Let's jump to the tag inline class. So this class should be moved to our new app because this is where you're referencing the tagged item class. So cut it from here and paste it right here. Now we need to import this class on the top. So from Django.contrib .content types.admin, we're going to import generic tabular inline. Great. No errors up to this point. Now we need to create a new product admin which extends the generic product admin that comes with our reusable app. And in this new implementation, we're going to reference the tag inline class. So, we create a new class called custom product admin which extends product admin. This is the class that comes with our pluggable app.

Now over here we set inlines to tag inline. As simple as that. Okay? Now that we have a new product admin, we need to unregister the old one and register the new one. So here we say admin dot site dot unregister product and then we're going to call admin dot site dot register product with this new custom product admin. Okay? Now the final step. We need to register this app in the list of installed apps. So, command and t, let's jump to installed apps. Good. Now, so here's the generic implementation right after we're going to add store custom. Okay? Now let's see what happens. So, refresh. Looks like we have an error. So, back to the terminal, we have an error saying tag inline is not defined.

So I forgot to remove this from the old product admin. So let's jump to product admin in the store app. And over here, we're going to remove this line because we implemented this in our custom store app. So delete. And that also means on the top, we no longer need to import tag item. So our store app has no dependency to the tags app and we also don't need to import this generic class. So delete, good, and refresh. Take a look. So here's the product form. Now we have replaced the admin interface for this app using our custom store app. So that means if we go back to the settings module and remove this new app, we see the generic implementation of the product form without the tags.

So this is the interface we're going to build and deploy independently. Anyone can create a new custom app to customize this interface. Now, let's bring this app back. Beautiful.

What’s next?

We have reached the end of this course. First and foremost, I wanna say thank you for allowing me to be your instructor and following along all the way through. I hope you have learned a lot and I'm gonna see you again in my future courses. This course is the first of a series. In the next part, we'll talk about the advanced Django concepts such as building RESTful APIs, authentication and authorization, logging, caching, sending emails, deployment and more. So if you're serious about learning Django and want to become a backend developer, don't miss the next part. And if you enjoyed this course, please support me by telling others about my coding school, CodeWithMash.com. I really appreciate your support and send you my best wishes.

Once again, thank you so much for your support and I hope to see you again in the next part.