Getting Started:

Introduction:

Welcome back to the second part of the ultimate Django course. In the first part of this course, we built the data model and the admin interface for an e-commerce application. In this part, we're gonna continue our journey and build a restful API for this application. So by the end of this course, we'll have a complete production grade backend for an imaginary online store. To take this course, you should have watched the first part because there I covered a lot of fundamental and important concepts that every Django developer must know. So if you missed the first part, you might find this course a bit challenging. Also, just like the first part, this course is very hands-on, so I want you to code along with me and complete all the exercises.

This is literally the only way to truly learn, understand, and master Django. I'm Mosh Hamadani, and I'm going to be your instructor in this course. Now, let's jump in and get started.

Setting Up the Project:

Alright, the first thing we're going to do is set up our project. This is the same project we completed in the first part of this course. So even if you fully completed the first part, I still want you to follow this video and set up this project from scratch because I want to make sure we are working on the exact same code and database so there are no surprises later on. Okay? So here I'm using DataGrip or you can use your favorite database management tool. We're going to use this tool to create a new database for the second part of this course. So we can easily go back and forth between these two parts. So let's open a new query console and type create database storefront two.

Okay. Now we have a new empty database. So we need to open our project and run our migrations. So Django will automatically create our database tables for us. Now, right below this video, you will find a zip file where You can find all the source code for each section as well as a database seed file. So go to the code folder and open the first section. Here's our starter project. Here we have storefront2, which is the project for this part. Technically, this is the completed project in the first part, but I've renamed the folder to storefront2 so we can separate these two projects. So simply open this with VS Code. Now we need to configure our database settings. so press command and t on mac or control and t on windows and jump to the databases section of our settings module.

So as you can see, our engine is MySQL. Our database is called Storefront. We should change this to storefront 2. now change the user and password to whatever you have configured for mysql on your machine, okay? Now save the changes and open a terminal window. First way to install our application dependencies. So we type pip and install All right, this is going to create a new virtual environment and in that virtual environment is going to install our application dependencies All right. Now we need to activate our virtual environment and to do that. We're gonna run pip and shell so pip and shell Good now we are inside the virtual environment so we can run Python and manage the PI migrate All right Django created our database tables for us beautiful now we need to populate these tables with some data So back to data grip first.

I'm going to open the services panel and close all the existing sessions So close all the sessions good now I'm going to open a new query console for storefront to Now back to our zip file let's go to the data folder and open this seed file in data grip all right we have a number of errors because data grip doesn't know which database we're going to run these statements on top of. So, from the session drop down let's select console 23 and here we're going to select storefront 2. Okay? Now, we're going to select all these statements and run them in one go. Alright, beautiful, let's verify that everything works successfully. So, I'm going to go to the product table, first we need to refresh, okay, and then we're going to go to store, underline product.

So here we have a thousand products, beautiful. The final step is to start our web server. So, back in VS Go, we're going to run python, manage.py, run server. Now, because I don't have any other projects listening on port 8000, our web server started at this port. On your machine, you might still be running the project for the first part of this course. So make sure to stop that web server and then start this web server. Okay. Now let's go to this address. So we don't currently have anything in the root of this project. That's totally fine. We're going to take care of that later in this course. Now, let's go to the admin panel. Now because we have a brand new database,

in this database, we don't have an admin user. So back in vs Code, we need to create a new super user. So let's open a new terminal window and run python manage.pi create super user all right we get this weird error because the virtual environment for this project is not activated. This happens from time to time in vs Code. So here we should run pband shell. Okay, the virtual environment is activated, so one more time. Python, manage.py, create superuser. I'm going to call this user admin, and use my email. And for the password, I'm going to use 123123. Now, it's complaining that the password is not secure. I'm going to bypass validation. Now we have a superuser, so we can use that to log into the admin interface.

Alright, this is the admin interface we built in the first part of this course. So we can go to the collections page. Here we have all these collections and we can see the number of products in each collection. Beautiful. So our project is ready. From the next section, we're going to start building an API for this project.

Building RESTful APIs with Django REST Framework:

Introduction:

Alright, we're going to start this course with an introduction to RESTful APIs. So in this section, we'll be talking about RESTful APIs, what they are, and how they work. Then we'll install Django REST Framework, which is the most popular framework for building web APIs with Django. Once we do that, then we'll talk about some of the fundamental concepts such as creating API views, creating serializers, serializing and deserializing models, and much, much more. So by the end of this section, we'll have a basic API for managing our products and collections. I'm really excited about this section. I hope you are too. So let's jump in and get started.

What are RESTful APIs:

So in the first part of this course, we implemented the admin panel for managing our data. But this interface is only supposed to be used by admins. So we need a way to expose our data to clients like a web app running inside a browser or a mobile app running on a mobile device. And this is where API is coming to the picture. API is short for application programming interface. Building an API is essentially building an interface that client apps can use to get or save the data. It's like building a remote control with a bunch of buttons where each button provides certain functionality. Similarly, our API is going to have a bunch of endpoints for different purposes. So we can have an endpoint for getting the list of products as well as creating, updating, and deleting them.

We can have other endpoints for managing our shopping carts, orders, and so on. Client apps can send requests to these endpoints to get or save products, orders, shopping carts, and so on. So that's the basics of APIs. But what makes an API RESTful? Well, REST is short for Representational State Transfer. Yeah, I know, it's one of those funny computer science terms that only some PhD students understand. In fact, this concept was part of the thesis of a PhD student back in 2000. In practical terms, REST defines a bunch of rules for clients and servers to communicate over the web. Following these rules help us build systems that are fast, scalable, reliable, easy to understand, and easy to change. An API that confirms to these rules is called RESTful.

Now, what are these rules? Well, you can read an entire book on RESTful rules and concepts, but practically speaking, there are three concepts that everyone should know. And these are resources, resource representations, and HTTP methods. We'll talk about these over the next few lessons.

Resources:

The first key concept you need to understand in RESTful APIs is the concept of resources. A resource in a RESTful API is like an object in our application, like product, collection, shopping cart, and so on. These resources are available on the web and client applications can access them using a URL. Now, do you remember what URL is short for? It's short for Uniform Resource Locator. So it's a way to locate a resource on the web. It's basically a web address. So assuming our website is hosted at moshbuy.com, we can access all products using moshbuy.com slash products. Now, if you want to access an individual resource, like an individual product, we append its ID to the URL. Now, a resource may contain other resources.

For example, a given product can have one or more reviews. So using this URL, we can get all the reviews for product number one. Or if you want to get a specific review, we can reference it using its ID. Now, as a general rule of thumb, we shouldn't nest our resources too deep, two levels is fine. But beyond that, our URLs get complex and unmanageable. Now, do you see a pattern or consistency across these URLs? This pattern is one of the attributes of restful API's. So if you simply follow this pattern, our API will be familiar and easy to understand to others. So this is all about resources. Next, we're going to talk about resource representations.

Resource Representations:

So we can identify a resource using its URL. Now, when we hit that URL, the server is going to return that resource in a certain format or representation. It may return it as HTML, XML, which is an old format for representing data or JSON, which is a modern format and essentially replaces XML. We'll talk about it in a minute. But what is important here is that none of these are the internal representation of a resource on the server. In other words, on the server, We identify a resource, like a product, using an object or an instance of a Python class. But when we return this object to the client, we're going to represent it as HTML, XML, or JSON. Because these are the formats that clients understand.

Now, REST doesn't dictate what format or representation we should use. It's entirely up to us. We may support one or multiple representations. If we support multiple representations, the client should tell the server what representation it needs when asking for data. Now what is JSON? Well, JSON stands for JavaScript Object Notation. It's a notation we use for representing objects in JavaScript. Here's an example of a product represented using JSON. So in JSON, we represent an object using a pair of curly braces. Inside the braces, we can have a bunch of key value pairs or properties. So here we have name, age, isOnline, and so on. Now, note that the keys are always strings, so they should be surrounded with double quotes. The values, on the other hand, can be anything.

So in this example, the value of the name property is a string, the age is a number, isOnline is a Boolean, employer is an object, so it can have its own properties, and interests is an array. In this case, it's an empty array, but in JSON, we can have an array of anything. We can have an array of numbers, strings, and so on. So this is JSON, and Throughout this course, we'll use this notation to send or receive data from the server. Next, we're going to talk about HTTP methods.

HTTP Methods:

So when building a RESTful API, we expose one or more endpoints for clients. Now each endpoint may support various kinds of operations. Some endpoints may only allow reading data while others may allow modifying data as well. And this is where HTTP methods come in the picture. Using HTTP methods, the client can tell the server what it wants to do with a resource. So HTTP defines methods like get for getting a resource or a collection of resources. We have post for creating a resource put for updating it, patch for updating part of it, like a subset of properties, and delete for deleting a resource. So let's go through a few scenarios. Let's say we want to create a product. In our client app, we should send a post request to the products endpoint so the server knows we want to create a product.

But where is the product we want to create? We're going to insert it as a JSON object in the body of the request. So the server will read this product and create it. Now let's say we want to update product number one. first we need to ask are we going to update all properties or just a subset of them if you want to update all properties we should send a put request otherwise we should send a patch request to this url note that this url is referencing a specific product and not the products collection now similar to creating a product we should include a product object in the body of the request so the server will extract this product from the request and update it accordingly now to delete a product

we should send a delete request to the server. Again, note that this URL is referencing a specific product. Now in this case, we don't have anything in the body of the request because all the server needs to delete a product is its ID, which is included in the URL anyway. Now don't worry if all of this is new to you because we're going to go through this over and over throughout the course. Alright, that's enough theory for now. Next we're going to install Django REST framework and start building our first API.

Installing Django REST Framework:

Now we're going to install Django rest framework to build a restful API. This is a separate framework that sits on top of Django and makes it incredibly easier to build restful api's So here in the terminal window, we should run pipenv install Django rest framework All right Django rest framework is installed. Now we need to add it in the list of installed apps, so let's jump to to the install apps section of the settings module, I would like to add it right after django apps. So before the apps that we have created in this project. Okay. So rest underline framework. All right. That's all we had to do to install django rest framework. Next, we're going to talk about creating api views.

Creating API Views:

All right, now let's see how we can create an endpoint like store slash products. So if you send a request to this endpoint, we should see all the products in our database. So back to our project, let's go to the store app and open the views module. In this module, we should create a view function. Do you remember what a view function is? It's a function that takes a request and returns a response. So we're going to create a function called product list that takes a request. I hear we should return a response object. So on the top, from Django, that HTTP, we should import the HTTP response class. Now we get these warnings, because the current Python interpreter is not the same as the one that we have in our virtual environment.

Let me show you how to solve this problem. So we open a new terminal window and type pip n shell. Now we can see the path to our virtual environment. So on a Mac computer, it starts with slash users. on Windows, it's probably c drive backslash users, whatever. So we're going to grab this path up to the bin folder, okay? Let's copy that. Now, on the top, under the view menu, go to the command palette and search for python interpreter, okay? Now, we're going to select the first item, enter interpreter path. And over here, we're going to paste the address that we copied and at the end, we have to append slash python 3. so now vs code knows that we're using that particular python interpreter inside the virtual environment for this project, okay?

So we imported the http response class. Now we're going to return http response. And in the body of the response, we're going to write okay. For now, we're going to keep things simple. Later on, we'll get all the products and return them in the response. So the next step is to map this view function to a url pattern. So back to our project. Look, in the store app, currently we don't have a URLs module. So let's add a new file called urls.py. Now we created a URLs module earlier in the first part of the course. So I'm going to copy some code from that module. That module is inside the playground app. So here's the URLs module. Let's copy all the code and paste it in our new module.

Now we need to make a couple of changes here. instead of hello, we're going to use products. So if you have a request to the products endpoint, that request should be handled by which function? Product list that exists in the views module that we have imported on the top. Okay. Now, this urls module belongs to the store app. So we need to import it in the main urls module. So back to our project. We have another folder here, storefront. And this is where we have the main or the root URLs module. So let's open this file. And over here, we have a URL pattern for the playground app. I'm going to duplicate this line and change playground to store. So if the URL of the request starts with store, it should be handled by store that URLs module.

Okay, now let's test our implementation after this point. So back to the browser. Enter. Okay, our endpoint is working. Beautiful. So this is how we can create a view function in Django. We create a function that takes a request and returns an instance of the http response class in Django. So in django we have two classes, HTTP request and http response. But Django REST framework also comes with its own requests and response classes. These classes are simpler and more powerful than the ones that come in Django. Let me show you how to use them. So first on the top, from rest framework, the decorators, we're going to import the API view decorator. Now, if we apply this decorator to our view function like this, the request object that we receive here will be an instance of the request class that comes with the rest framework.

So this will replace the request object in Django with the newer request object that comes with the rest framework, which is simpler and more powerful. Now similarly, we're going to replace this old http response object with the newer one that comes with the rest framework. So, first, we're going to import the class from rest framework dot response. We're going to import the response class. Next, we're going to replace http response with response. So I made two simple changes here. First, I applied this decorator, and second, I replaced http response with response. Now, With these two simple changes, if we hit this URL one more time, we get this beautiful page which is called the Browsable API. This Browsable API makes it incredibly easy to test our API endpoints in the browser.

But note that we only see this if we hit our endpoints in the browser. If a client app, like a mobile app, hits our endpoints, it's not going to see the Browsable API. It's only going to see the data in the response. Let me simulate that. So if we click on this dropdown and select JSON, look, This is all a client app is going to see, okay? Now, back to this page. What do we have here? Well, on the top, we have a heading that is generated based on the name of our view function, product list. Next, we have information about our request. So we received a get request at this endpoint. And over here, we can see information about the response. So the status of the response was 200, which means okay.

Allow tells us What HTTP methods are supported at this endpoint? So we have options which we can use to see what operations or what methods are available, and get, which we are using right here. Content type tells us the type of content in the response. So that is application slash JSON. And Very is used for caching, which we'll talk about in the future. And here's the body of the response. Okay, so now let's create one more API view. So back to our code and i'm going to create another view function for seeing the details of a product. So, product detail takes a request. Now we're going to apply the api view decorator and return a simple response. Okay. Next, we need to map this to a url pattern.

So, back to our urls module. Let's duplicate this line. And in the second pattern, we're going to add a parameter. So we're going to type a pair of angle brackets and give our parameter a name, followed by a forward slash. Now, if we get a request at this endpoint, we want our second view function to be called, product detail. Now, let's go to this view function and give it the ID parameter. ID. And it's better to return the ID in the response so we can verify that everything is working. Now, let's go to products slash one. Alright, we see one in the response, beautiful. But what if we type a non numeric ID like A? Our endpoint still works? But technically this shouldn't be allowed because our product IDs are integers.

So, back to our URLs module, we can apply a converter to this parameter. So right before that, we're going to type int colon. Now, if we hit this endpoint one more time, we get an error because our products endpoint only accepts integer values for the ID parameter. So, let's pass one and Everything works beautifully. Okay, we're done with this lesson. Next, we're going to talk about serializers.

Creating Serializers:

So we have created an API view. Now, instead of including the OK message in the response, we want to include the list of products, right? So we need a way to convert product objects to JSON objects. How are we going to do that? Well, in Django REST framework, we have a class called JSON renderer. This class has a method called render that takes a dictionary object and returns a JSON object. So if you convert a product object to a Python dictionary, we can pass it to this method and get a JSON object. And this is where serializers come into the picture. A serializer is an object that knows how to convert a model instance like a product object to a python dictionary. Let me show you how to create a serializer actor project here in the store folder, we're going to add a new file called serializers.py.

Now, from res framework, we're going to import the serializers module. Next, we should create a class called product serializer and And this class should inherit the serializer class that is defined in this module. So serializers.serializer. Now, we need to decide what fields of the product class we want to serialize. What fields we want to include in a Python dictionary. So let's quickly go to the product class. Look, our product class has all these fields, but what we return from our API doesn't necessarily need to have all these fields. Because what we have here is the internal representation of a product But what we return from our API is the external representation of a product. Sometimes you might have sensitive information here, we don't want to expose them to the outside world.

That is why we need two separate representations of a product, an internal representation and an external representation. Now in this demo, I only want to return the idea of a product as well as the title and unit price. So back to this class, here we should define three fields exactly like how we define fields in a model. So ID equals serializers dot integer field. So look, this is exactly like defining a field in a model. Now if you go to Django REST framework dot org, under API guide, serializer fields, you can see all types of fields that are available to us. So we have Boolean fields, string fields, like char field, email field, and so on. We also have numeric fields, daytime fields, and so on.

So pretty much everything that we have for creating a model is also available here. Now, all these fields have a bunch of core arguments that are available everywhere, like read only, write only, required, which is set to true by default. We can also assign default values and so on. But certain fields, depending on their type, also support additional arguments. Again, exactly like how we define a model. So back to the code, let's define another field called title. We're going to set this to serializers.charField. And here I'm going to set maxLength. to 255. Now, you might be wondering why we need to specify that here. Because later, we'll use this serializer when receiving data from our API. Let's say we want to create a product object, we want to make sure that the product object that is sent to our API is a valid object.

We'll talk about that later in the section. Next, we need to define unit price. And by the way, we don't have to call this unit price, we can call this price. Because this is a completely separate object from the product object that we defined earlier. So the name of the fields don't even have to match, okay? But for consistency, I'm going to call this unit price. And here we're going to set this to serializers.decimalfield. The decimal fields have two arguments that we need to set. Max digits, I'm going to set it to six, and decimal places to two, okay? So this is how we define a serializer. Now that we have a serializer, we can use it to convert a product object to a Python dictionary.

And that's what I'm going to show you next.

Serializing Objects:

Now that we have a serializer, let's see how we can use it to convert a product object to a JSON object and then include it in the response. First, on the top, we need to import a couple of classes. From the models module in the current folder, we're going to import the product class. And from the serializers module in the current folder, we're going to import product serializer. So note that here I'm using a period to indicate the current folder. Okay? Now, we're going to modify this view function. So we're going to get the product with this id and then include it in the response. So here we're going to say product dot objects that get we set pk to id and get a product object.

Next, we should create a serializer and give it this product object. So serializer equals product serializer. And here we pass our product object. Now, the moment we create this serializer, this serializer will convert our product object to a dictionary. And we can get that dictionary from serializer.data. So instead of including the ID in the response, we're going to include serializer.data. Now in this implementation, I'm not using a JSON render that I told you about in the previous lesson, because all that magic will happen under the hood. So at some point, Django is going to create a JSON render object and give it this dictionary. The JSON renderer will then convert that dictionary to a JSON object, and then that JSON object will end up in the response.

All of that is hidden from us, okay? Now, back to the browser. Let's hit product slash one. Okay, so here's our product object. Beautiful. Now, note that here, unit price is rendered as a string, even though this is a decimal field. Why is this happening? Well, this is one of the default settings in Django REST framework, but we can easily overwrite it. So, back to the project. Let's go to the settings module. Now, at the end, we're going to define a new setting. REST underline framework. All of this should be in uppercase, okay? Now, we set this to a new object. And in this object, we're going to add a key called course underline decimal underline to underline string. Make sure to spell it properly, okay?

We're going to set this to false. And now, if you refresh this page, We can see that unit price is rendered as a decimal value. Great. Now, what if we ask for a product that doesn't exist? So product number 0. We get an exception, and this is not good. Because one of the conventions of RESTful APIs is that if an object doesn't exist, we should return a response, and the status of the response should be 404. So 404 is a standard HTTP code that means not found. So back to our project, we need to wrap this code inside a try block. so try, then we indent these lines and catch an exception of type product dot does not exist. Now here we're going to return a response.

In the response, we don't need to put anything. We just want to set the status to 404. So status equals 404. Now, if we refresh this page, instead of an exception, we get a response. And as we can see, the status of the response is 404, okay? Now, 404 is such a famous code that pretty much every developer knows, but generally speaking, we should avoid magic numbers like this. It's better to use constants because they make our code more readable. So on the top, from REST framework, I'm going to import the status module. In this module, we have a bunch of constants for various HTTP status codes. So we're going to replace 404 with status dot http 404 not found. See, our code is immediately more readable.

That's better. So this is how we can handle the situation where a product doesn't exist. But repeating this pattern every time is time consuming. That's where we use django shortcuts. So on the top, look, we have a statement here from django.shortcuts for importing the render function. But render is not used here. Instead, we're going to import get object or 404. This function wraps this logic. The try block followed by an exception handler and returning a response with this status code. So we can get rid of these two lines as well as our try block. Now, instead of calling product.objects.get, we're going to use our shortcut function. So get object or 404. Here we need to pass two arguments. The first argument is the type of our model, which is product.

And the second is our lookup parameter. So pk equals id. Now, back to the browser, refresh. Just like before, we get a response with the status of 404. But this time we also have a message in the response. Detail not found. So we're done with this view function. Now let's modify this other view function. So here we're going to get all products. So product.objects.all. As you know, this returns a query set. We get that query set. Now, just like before, we create a serializer, product serializer. Now, in this other view function, we give this serializer a product object, but we can also give our serializer a query set. So, we pass the query set here. Now, we should also set manage true, so the serializer knows that it should iterate over this query set and convert,

each product object to dictionary, okay? Now, just like before, instead of returning okay, we're going to return serializer.data. Now, back to the browser. Let's hit the product sign point and here are all the products in our database. Beautiful.

Creating Custom Serializer Fields:

So I told you that the objects that we return from our API don't necessarily have to look like the objects in our application. Now, the main reason behind this is that our data models are really implementation details of our application. The implementation may change in the future. We may add new fields in the product class. We may rename existing fields or even delete them. We don't want these changes to be exposed to the outside world. Why? Well, once again, think of a remote control. Inside this remote control, we have some technology and This technology may change from one version to another. The manufacturer may replace some of the parts to make the production cheaper or more reliable. But those changes are hidden from us consumers.

So the buttons on the outside don't change because those buttons represent the interface of a remote control and we as consumers are dependent on that interface. Now similarly, our API represents the interface of our application. So we should try to keep it as stable as possible. Otherwise, existing clients may break. so if you want to change our API, we have to properly study the impact of the change and potentially provide different versions of our API. That's a topic for the future. Back to our product standpoint. Now as you can see, here we are missing a few fields of the product class. Let's say this was a deliberate decision. We decided not to include those fields here. Now similarly, we can decide to add new fields here that don't exist in the product class.

Let me show you. So back to our product serializer, let's add new field here called price with tax. We're going to set it to serializers dot serializer method field. So that means we're going to define a method here and that method is going to return the value for this field. So let's define a method. We can call it anything we want. So calculate tax but we should give it two parameters. Self and the product being serialized. Now here we can return product dot unit price times 1.1. Now note that when i typed a period here, we didn't get intellisense because obviously python doesn't know the type of the product object. So this is where we use type annotations. So if we annotate this with product, now when we type a period, we can see all members of the product class.

Okay? Now that we have this new method, we need to reference it over here. So we set method name to a string. Calculate. And by the way, let's enable auto saving so we don't have to constantly save the changes. So under the file menu, let's select auto save. Okay, that makes our job easier. Now refresh. Alright, we got an exception, unsupported operand types for multiplication. Basically, the reason this is happening is because we're multiplying a decimal value by float value, because the unit price of a product is defined as a decimal. But this number in Python is by default a float. So we need to convert it to a decimal before we can multiply it by the unit price. So on the top, from the decimal module of Python, we're going to import the decimal class.

And then over here, we're going to wrap this number with a decimal object. Good. Refresh. Okay, beautiful. Now we have this new field price with tax. Great. Now we can also rename fields. So let's say for consistency, we want to rename unit price to price. So let's rename unit price to price. Okay, now refresh. We get another exception saying product object has no attribute price. The reason we're saying this is because by default Django assumes that we have a field by this name in the product class. So it will read the value of this field from that target field. Now, because we have a mismatch here. We need to tell Django where to look in the product class. So, all our fill types Have an argument called source.

We can set this to unit price. Now, refresh, and the problem is solved. Great. So as we can see, our internal and external representation of these product objects are starting to evolve differently.

Serializing Relationships:

Alright, now let's talk about serializing relationships. So when returning a product, we can include a related object like a collection. There are a few ways we can do this. Let me show you. So first we're going to import the collection class from the models module. Now over here we're going to define a new field called collection. We're going to set it to serializers dot primary key related field. So with this we can include the primary key or the idea of each collection in a product object. But first we have to set an argument called query set. We should set it to query set for looking up collections. What is it? Collection that objects that are pretty simple. Now, we're not getting anything here.

So let's open up the terminal window and debug. So here's the terminal. Look at this error. relational field must provide a query set argument. But I already set the query set argument. this happens sometimes because the change detection mechanism behind this doesn't work. So we have to stop the web server with ctrl and c and then restart it. Okay, no errors. So refresh. Great. Now in each product, we can see the idea of the collection. Beautiful. That's one way to serialize a relationship. Another way is to return a collection as a string. So we can return the name of each collection here. So back to our serializer, instead of a primary key related field, we're going to use a string related field. So Django will convert each collection to a string object and return it here.

Now what is the string representation of a collection? Let's find out. So let's jump to the collection class. So earlier in the first part of this course, we overrode the string method. So here we're returning self.title. Now take a look. So our page is taking forever to load. Can you tell why? Let's find out. So using Django debug toolbar, we're going to look at the queries behind this page. So look, we have a thousand extra queries here. What are those queries? So if you pay close attention, for each product, we have an extra query to read its collection. So here we're selecting all these fields of the collection where collection ID equals four. The reason this is happening is because by default, we have lazy loading here.

So when we read product objects, their collections are not loaded. And because we need to access the title of each collection, for each product, Django is sending an extra query to retrieve the collection of that product. So to solve this problem, we need to load products and their collections together. Now, where should we do this? Well, let's go back to our views. So this is the view for getting the list of products. And over here, We have a query set for retrieving products. Now, before calling the all method, we have to call select related and pass collection. Okay? Now, take a look. Refresh. The problem is solved. Great. So now, each collection is converted to a string. That's another way. What approach you choose is purely personal preference.

Different people have different preferences for different models. Another way to serialize a relationship is by including a nested object. So we can include a collection object here. Let me show you. Back to our serializer. First, we're going to create a collection serializer. So class collection serializer. Just like before, nothing new so far. So we're going to define two fields here. ID, which is going to be an integer field. And title, which is going to be a char field. So serializers.charField. Pretty simple. Now, to include a nested object, all we have to do is use our collection serializer here. So we set collection to a collection serializer object. Now take a look. So now each collection is rendered as an object. Beautiful. Now there is one more way to serialize a relationship.

Instead of including an object here, we can include a hyperlink to an endpoint for viewing that collection. now this one has a few more steps, so pay close attention. Back to the product serializer, first we have to set the collection field to serializers dot hyperlink related field. Now, here we need to set two arguments. The first one is query set. We set it to collection dot objects dot all. The second one is view name. What is this? This argument is used for generating hyperlinks you'll see that in a second. So let's set that to collection dash detail. Currently we don't have a view by this name, so we need to create it in the URLs module. So let's jump to the URLs module of the store app.

Here I'm going to define a new route and say if you have a request to collections forward slash integer ID, we want this request to be handled by a view function called collection underline detail. Currently we don't have this view function, So let's go ahead and create it. We're going to go to our views module and down the bottom, we're going to create a new view function. So collection underline detail with two parameters, request and ID. Now for now, I just want to return a response of okay. Later we can come back and retrieve that collection, serialize it and return it. Let's not worry about it. Now we're going to decorate this with API view. Good. So back to our URLs module. We're mapping this endpoint to this view function.

Now we can give this mapping a name. And this name is the name that I used over here. Collection dash detail. So back to the URLs module. We're going to set name to collection dash detail. Now let's test our implementation up to this point. So refresh. We don't see anything. So once again, let's open up the terminal window. and see what is going on here. The view name argument is required. Once again, my changes are not detected. So we're going to stop the web server and restart it. Good. No errors here. So let's refresh this page all right we get an exception saying hyperlink related field requires the request in serializer context. So we need to pass our request object to our serializer because the request contains information about URLs.

So the exception is giving us some hint. It's saying add context equals this object when instantiating the serializer. So back to our views module, this is the view function for retrieving all products. And this is where we are initializing our product serializer. Now using the context object, we can give this serializer extra stuff. So here we're going to set context to dictionary. In this dictionary, we're going to add a key called request. And the value is this request object that we have. interview function. Okay, so we set it to request. Good. Now, let's refresh. Okay, we get a different kind of error saying could have resolved URL for hyperlink relationship using view name collection detail, you may have failed to include the related model in your API, or incorrectly configured the lookup field attribute on this field.

So much going on here. Basically, the reason we're seeing this error is because Django REST framework expects a certain convention in our URL. In this URL, our parameter should be called PK. So Django REST framework will read the value of pk and use that to look up a collection. So because we have changed this parameter to pk here, we have to go to this view function and rename id to pk as well. Now refresh and take a look. Now each collection is rendered as a hyperlink. So if we click on this link, we'll go to the endpoint for viewing details of that collection. Like currently, we haven't implemented this view, but we can always come back and fix it. So to recap, there are four ways to serialize a relationship.

We can use primary keys, string values, nested objects, and hyperlinks.

Model Serializers:

So you might be thinking that so far we have been repeating ourselves. Look, all these fields we have here, they also exist in our model classes. So our collections have ID and title, our products have ID, title, price, collection, and so on. So there are two places where we are defining these fields and their validation rules. So if tomorrow we decide to change the validation rules for, let's say the title of our products, there are two places where we need to change that rule. One in the serializer and also in the product class. There must be a better way, right? So this is where we use model serializers. Using the model serializer class, we can quickly create a serializer without all this duplication.

So here in the product serializer, we're going to change the base class to model serializer. Now temporarily, I'm going to comment out all these fields. Instead, we're going to create a meta class. So class meta. Here we should set two attributes. Model, product, and fields to an array or tuples of fields in the product class we want to include here. So we're going to include id, title, unit price, and collection. Now I know this is slightly different from the fields we have defined here, so over here we have price instead of unit price, and we also have price with tax, but we'll talk about that in a second. What I want to point out here is that we don't need to redefine all these fields here.

we can have Django REST framework look up the definition of these fields in the product class and automatically create a product serializer for us. So, back to the products endpoint. Let's refresh. We get the same result as before. Now, for collection, which is a related field, we have a primary key related field. So by default, model serializers use primary key related fields. If this is not what you want, you can always override it. So, we can bring back the collection field that we defined in the previous lesson. And with this, we can convert each collection to a hyperlink. So take a look. Now we have hyperlinks, but i prefer to stick to the defaults. So let's get rid of this field. Good. Now what about unit price?

Once again, we can bring back the price field and then mention that over here. So if this field Exists and the product class Django risk framework is going to use that otherwise is going to look for that field in this class Okay. Now let's refresh So instead of unit price we have price again. I prefer to stick to conventions and not repeat myself So let's change this to unit price and then we can get rid of this line as well Similarly, we don't need the ID and title fields. So let's delete them as well. We only want to include a price with tax because this field doesn't exist in the product class. So let's bring that in and include it over here. Price with tax.

Now the order matters. So I'm adding this field right after unit price and that means in this json objects price with tax also comes after unit price, okay? So this is how we can use model serializers. Now before we finish this lesson, There is one more thing I want to highlight here that is a bad practice you should avoid. Here, instead of setting fields to an array, we can set it to a special string like this. Double underscore all double underscore. That means all fields in the product class. Now why is this a bad practice? Because we don't want to automatically include every field in our product class. If tomorrow we decide to add a new field in the product class, that field will automatically be exposed to the outside world.

And this is against all the concepts I talked about earlier. We want to separate the external and internal representation of a product. Maybe that field includes some sensitive information we don't want to expose to the outside world. So do not use all. This is for lazy developers. You don't want to be one of them. So always explicitly specify your fields. Like this, okay? Now let's apply the same change to the collection serializer. So we're going to have it inherit model serializer Now we can get rid of these fields and define a meta class. We set the model to collection and fields to ID and title. Okay. Let's refresh and make sure everything works. Beautiful.

Deserializing Objects:

All right, now let's talk about deserializing objects. Deserialization is the opposite of serialization and it happens when we receive data from the client. So let's say the client wants to create a new product. To do this, it should send a POST request to the product's endpoint and in the body of the request, it should include a product object. Now on the server, we have to read the data in the body of the request and deserialize it so we get a product object and store it in the database. Let's see how this works. So back to our product list view function. First, we need to pass an argument to this API view decorator. We have to pass an array of strings that specify the HTTP methods with support at this endpoint.

So get and post for creating data. Now previously, we didn't have to do this because get is supported by default. But now because we're also going to support the post method, we have to explicitly pass this array here. Okay. Now in this function, we have to say, if request that method equals get, we're going to execute this logic. Elif request that method equals post. We're going to do something else. And this is where deserialization happens. So we're going to read the data in the body of the request and deserialize it. So first we need to create a serializer object. Now previously we passed a product object here or a query set. And this is where serialization happened. But to deserialize data, we have to set the data keyword argument to request.data.

So if we set this argument, product serializer is going to deserialize this data. And then the data is going to be available in an attribute called serializer.validatedData. But before we can access this attribute, first we have to validate the data. And that's the topic for the next lesson. So for now, Let's just comment this out and return a simple response and say, okay. Now, back to the browser, refresh. If you look down the bottom, you see a box for sending content to the server. So this is one of the benefits of the browsable API. Now that we're supporting the post method at this endpoint, we get this box in the browsable API. It makes it much easier to test our API endpoints. So for now, I'm going to pass an empty object, post,

And we get this response. Great. So next we're going to talk about validating data.

Data Validation:

Now let's talk about data validation. So I told you that before we can access the validated data attribute. First, we have to validate the data. Otherwise, we're going to get an exception. Let's see this in action. So I'm going to bring this line back in and down the bottom, post an empty object to the server. Look, we get an exception saying you must call is valid before accessing validated data. So back to the code. Here, after we create a serializer, we're gonna say if serializer is valid, then we're gonna execute this logic. Otherwise, we're gonna return the list of errors to the client. So we're gonna return a response. In the body of the response, we're going to include the errors. So serializer.errors.

And here we should set the status to 400, which means bad request. That means the client didn't send valid data to the server. So status equals status dot HTTP, 400, bad request. And if you remember, the status is a module that we imported from the rest framework. So look on the top, from rest framework, import status, okay? Now take a look. Refresh. So once again, we're going to post an empty object, and we get these validation errors. Title, this field is required, unit price, and collection. So for each field, we have an array of validation errors, okay? Now, there is another way to write this code that is cleaner and more concise. Here we can pass a keyword argument called raiseException. We can set it to true and then we don't need this if else block.

So let's get rid of the if statement and the else clause. We only have the happy path. So if we get invalid data, Django REST framework is automatically going to return a response with the status of 400 And it's going to include validation errors in the body of the response, just like before. So let's verify it one more time. Refresh. Something's not working. So let's bring up the terminal. Unexpected indent. Again, I believe that the changes are not detected. So let's restart the server. Great. So refresh and post one more object to the server. Perfect. The same result as before, but less code. Beautiful. Now let's talk about the validated data attribute. For now, I just want to print this on the terminal.

So let's print it. And test this one more time. So this time I want to pass a valid product object. So a product with three keys. Title. Note that I'm wrapping title in double quotes. Otherwise, our JSON object is not going to be considered valid. So we're going to set this to a or whatever. Next, we're going to set unit price to one and collection to one. Now let's post this. Okay, we got this response. Now, let's look at the terminal window. Now look over here. This is our validated data attribute. So it's an order dictionary with three key value pairs, title, which holds a string, unit price, which holds a decimal value, and collection, which holds a collection object. So do you see what happened here?

As part of deserializing the request data, Django REST framework automatically retrieved a collection with the ID that we specified. So now we have a collection object here, okay? So this is the validated data attribute. Now in the next lesson, we're going to save this in the database. For now, let's not worry about it. One more thing I want to cover in this lesson is validation at the object level. There are situations where validating the request data involves comparing multiple fields. For example, think of the scenario where a user registers. So we have username field as well as password and confirm password. We want to make sure that these two fields are equal, right? Now, with our current implementation, we cannot achieve this because our validation rules come from the definition of model fields.

So if we need anything extra, we need to overwrite the validate method in our serializer. So back to our product serializer, here we can define the validate method. This is actually defined in the base class, the model serializer class, but we are overriding it here. So this validate should have two parameters, self and data, which is a dictionary. Now here we can say something like if data of password doesn't equal data of confirm password, then we're going to return serializers that validation error and here we can pass an error message like passwords do not match. And of course, that doesn't make sense in the context of our products, but this is just an example. Otherwise, if the data is valid, we're going to return it.

So in our validate method, we should either return a validation error or the actual data, which is a dictionary okay now in this case, we don't really need this method, so i'm going to delete it, okay? So back to our view. We validated the data. The next step is to save the data in the database. And that's what we're going to do next.

Saving Objects:

Now let's talk about saving data. So our product serializer is inheriting the model serializer class. Now this model serializer has a save method that we can use for creating or updating a product. So back to our view, right after we validate the data, then we can call serializer.save. Now technically we don't need to touch the validated data attribute because the save method has some logic for extracting data from this dictionary to create or update date a product. So, let's delete this line and test our implementation up to this point. So, back to the product's endpoint. I'm going to paste a product object here with three keys. Title, unit price, and collection. Post. Alright. We get an exception saying column inventory cannot be null.

Because inventory is one of the required fields in the product class and we haven't supplied a value for this field in our request body. So let's quickly look at the product class. What fields do we have here? Title, slug, that is also required and we should include it in the body of the request. Description is optional because we have set null to true. We have unit price. We should also include inventory. Then we have last update, which is optional because it's automatically set. We have collection and promotions is also optional because it's a many to many field. so if you don't supply a value, it's going to be set to an empty list. So let's go back to our product serializer and include two more fields here.

One is slug and the other is inventory. Now technically we could also include description and in fact we should do that because when retrieving products we want to read their description as well. So let's add that here. So one more time let's try to create a product. So back to the product standpoint, I'm going to post a product object like this. Take a look. We got okay in the response. So this product is created. See how simple it was, we didn't have to do anything. Now there are situations where we want to override how a product is created. Perhaps we want to set some special fields or associate a product with another object in the database. So back to our serializer class. Here we can override the create method.

def create okay so this method takes the validated data dictionary so over here we can create a product object and unpack this dictionary validate the data then we can set those special fields whatever they are next we're going to save that product and finally we're going to return it from this method so create is also one of the methods that exist in the base model serializer class It's called by the save method if we try to create a new product. Similarly, we have another method for updating a product. So, if you want to override how a product is updated, then we can override this method and in this method we have two parameters. Instance, which is a product object and validated data. So here we can say instance dot unit price or some other special field

We can read that from validated data like this. Then we're going to save the instance and finally we're going to return it. In this case we don't really need to do this. We can rely on Django REST framework to automatically set all these fields for us. So all I wanted to point out here is that the save method will call one of these methods depending on the state of the serializer. So for now let's delete these methods. We're done with creating a product, let's see how we can update a product. So, back to our view, now, for updating a product, we should modify this other view function. Because to update a product, we should send a put or a patch request to a particular product, like the product with the id 1.

So, back to this view function, here we're going to pass an array of our HTTP methods, so get and, we can use put for updating all properties, and patch for updating a subset of them now it's entirely up to us what methods we want to support here sometimes we only want to allow a subset of properties to be updated in that case we should only support the patch method not the put method now in this demo i'm going to keep put and remove patch just for simplicity now just like before we need to check the request method so if request that method equals get then we're going to return this object elif request.method equals put, then we need to deserialize the data, validate it, and save the product object in the database.

So here we're going to create a serializer, product serializer. Now, just like before, we're going to set data to request.data, so deserialization happens, but in addition to this, we should also pass a product instance. if we pass a product instance, this serializer will try to update the attributes of that product using the data in the request. So where is that product? Well, we're reading that over here. So I'm going to move this line outside of this block. So we have this product object in both these statements. So over here, we're going to pass our product. And in this case, let me try to save this serializer this serializer will call the update method because we're instantiating it with an existing product and some data to be serialized.

Okay? So we have the serializer. Next we call serializer is valid and we set raise exception to true. Next we call the save method and finally return a response. Now what is going to response? Serializer.data. Now this is something that I forgot to implement when creating a product. So we were returning the OK message, but the restful convention is to return the product that was created. So once again, we're going to return serializer.data and the status of the response should be 201, which means object created. So we set this to status.http 201 created. Again, this is another restful convention. When your endpoint creates a new resource, it should return a response with the status of 201. Okay? So, let's test this real quick.

Back to the products endpoint. Once again, I'm going to create a new product. Post. Great. So we have a response with the status of 201 and here's the product object that we created. Now note that this product object automatically includes the ID that was generated on the server. Okay? Now let's try to update a product. So I'm going to go to products slash 2. Now to save time, i'm going to copy this object, put it in the request body, and make a few small changes. First, we're going to remove the ID. Then, for the title, we're going to add a plus sign here so we know it's updated. Also, we should remove price with tax because it's a calculated read-only field. So it's not going to get updated.

Now let's send a put request to the server. Great. So, we got a response with the status of 200 and Here's the product object, and here's the updated title.

Deleting Objects:

The last thing we're going to talk about in this section is deleting a product. We're going to implement this operation in the product detail view function, because this is where we work with a particular product. So first, we need to add the delete method right here. Now, note that the moment we add this method here, if you go back to the browser and refresh, we see a delete button here. Beautiful. I totally love this browsable API. It makes it much easier to test our API endpoints. Now let's implement this operation. So Back to our view function. We need another if statement. So elif request.method equals delete. Then we're going to delete the product that we retrieved earlier. So that is super easy. We're going to say product.delete.

And then we're going to return an empty response with a status of HTTP 204, which means no content. Again, this is one of the conventions of restful APIs. So quite often, when we delete a resource, we return an empty response with the status of 204. Now, this is not written in stone, so in your application, you might prefer to return the object that was deleted. That is totally fine. Now let's test this. So, back to the browser. I'm going to delete product number one all right we got an exception saying cannot delete some instances of model product because they're referenced through, protected, foreign keys, order item dot product. Basically, what this exception is saying is that we have order items that are associated with this product.

So we cannot delete this product. Now, in this case, we don't want to show an exception to the user, because to them, it looks like our application blew up. Instead, we want to return a proper response. And in the body of the response, we want to include an error message. So before we delete a product, we should check to see if there are any order items associated with this product. here's one simple way. We can say if product that order item on the line set that count is greater than zero, then we're going to return a response with the status of http 405, which means method not allowed. Now, how do i know this? Well, part of this is experience, but as we move from one project to another, we might face new problems.

So even i don't know sometimes what is the right status code. So I highly encourage you to look at this website httpstatuses.com. On this page, you can see all standard HTTP status codes. So use your judgment to decide what is the best status code to return to the client. Now back to this page, let's refresh and delete this product. Okay, now we get a proper response with the status of 405, which means method not allowed. That is much better. But we can also include an error message in the body of the response. So Over here, we can pass a dictionary, which will be converted to a JSON object. In this dictionary, we can add a key called error. And here we can say product cannot be deleted because it is associated with an order item, something like that.

So let's refresh and delete. Okay, that is much better. Now we have a proper error message that the client can get and show to the user. Beautiful. Now one last thing I want to improve here is the name of this attribute, order item underline set. In the first part of this course, we decided to stick to the conventions but going forward, I'm going to rename these related attributes to more meaningful names. So it would be better if we call this order items as opposed to order item underline set. So we're going to go to our models module in the store app Let's look at the order item class. So this is where we're associating an order item with a product. Now as part of defining this foreign key, we can specify the related name in the product class.

So related name, we're going to set that to order items. Okay, now back to our view function. We're going to change this to order items. That is better.

Exercise: Building the Collections API

Alright, now it's your turn. So as your exercise, I want you to implement the collections endpoint. At this endpoint, we can get all the collections in our database, as well as the number of products in each collection. Pretty useful. Now down below, we can also create a collection, just like how we can create a product. Now we also have another endpoint for working with a particular collection. So we can get that collection, we can delete it and update it. So go ahead and spend 15 to 20 minutes on this exercise when you're done come back, watch the rest of the video. All right, let me walk you through what i have done. It's pretty similar to how we implemented the products endpoint, but there are a few slight changes here.

So we have one view function called collection list for getting a collection and creating a new collection. And of course, we have mapped this one endpoint in our urls module. So here we have a new url pattern. If you have a request, To this end point, that request will be handled by this view function. Now, in this view function, if you have a get request, we're going to get all our collections and annotate them with the number of products in each collection. So this is the part that is different. We talked about annotating objects in the first part of this course. So if you don't remember this concept, go back and watch that lesson in the first part. So we annotate our collections and get a query set.

Then we send that to our collection serializer for serialization and return the serializer data in the response. Again, exactly like how we implemented the products endpoint. Now, as part of this, I had to modify the collection serializer. So here in the list of fields, I added products count, but because the collection class doesn't have this field, I had to define it here in the serializer as an integer field. Otherwise, we'll get an exception, okay? Now, back to our view function. Next, if you have a post request, then just like before, we're going to deserialize the data, we're going to validate it, and then save it in the database. Nothing new here. Now we also have this collection detail view function that we created earlier in the course, I just had to implement it.

So here we're going to get the collection with the given ID. Now as the first argument to this function, I'm passing a query set, the same query set that I used earlier. So We get all our collections and annotate them with the number of products in each collection. Now one more thing that you need to pay attention to here is the name of this attribute, products. So here I overwrote Django's convention for naming related fields. So in our models module, this is where we have defined the product class. Now here we have the collection field. As part of defining this field, I set the related name to products. So now each collection has an attribute called products, okay? Now, like our view function, so we get a collection, then if you have a get request, you serialize the collection and return the data to the client.

Again, exactly like how we retrieve the product. Now, updating a collection is also the same, but we have a slight change in deleting a collection. So here we're checking to see if this collection has any products. So if collection.products.count is greater than zero, we return an error to the client saying this collection cannot be deleted because it includes One or more products. And by the way, You can get the completed code in the zip file that I gave you at the beginning of the course. Just go to the second section and look at the finish folder. So this is my implementation of the collections endpoint. Now as part of implementing this, I realized that I had to make a slight change in the admin module as well.

So over here, this is where we are defining the Collection admin class. In this class. We have a method called get query set and And this is where we're annotating our collections with their number of products. So over here, previously we had product. Now we're using products because I changed the name of the related field. That's pretty much it. So I hope you finish this exercise successfully. If not, don't worry, you're still learning. As you practice and write more code, everything becomes simpler and essentially becomes second nature. So we're done with this section. In the next section, we're going to talk about advanced API concepts. So I will see you in the next section.

Advanced API Concepts:

Introduction:

Welcome back to another section of the Ultimate Django course. In this section, we'll be talking about advanced API concepts that help us build APIs faster with less code. We'll be talking about class-based views, generic views, view sets, routers, as well as searching, filtering, and pagination. So by the end of this section, our products endpoint will be in a much better shape and our code, as you will see, will be cleaner and more concise. I'm super excited about this section. I hope you are too, so Let's jump in and get started.

Class-based Views:

All the views we have created so far have been function-based views. But Django REST framework also supports class-based views, which make our code cleaner and more concise. Plus, they provide a lot of reuse opportunities. So as we go through this section, you will appreciate them more and more. So let me show you how to convert this view function to a class-based view. First, we go on the top and from REST framework, that views module, we're going to import the API view class. This is the base class for all class based views. So now we're going to define a class called product list and have this class inherit API view. Now note that in terms of naming I'm following Python's naming convention for naming classes.

So here we're using Pascal case where we're capitalizing the first letter of every word. In contrast for function based views we're using a different convention. We're using lowercase letters and we're separating multiple words using an underscore. Okay? Now in this class, we're going to define two methods. A get method for handling a get request and a post method for handling a post request. So the incoming request gets automatically dispatched to one of these methods depending on its method. So we define a method called get with two parameters, self and request. Now in this method, we're going to add the code that we have in this block. So let's grab all this code and move it right here. And of course, we should fix the annotation.

Good. So this is our get method. Now we need a method for handling a post request. So define post with two parameters. And then we're going to move these few lines to our new method. Okay. We're done with this view function. We don't need it anymore. So the first benefit of a class-based view is that Here we don't have those if statements. You know, as our application gets more complex, we end up with many nested if statements, and they're really ugly. So the first benefit of class-based views is that they help us write cleaner code. But there's more to that. We're going to learn about them as we go through this section. Now, in order to use this class, we have to go to our URLs module.

Since we don't have this view function anymore, we're going to reference our productless class. Now, this class has a method called asView. When we call this method, this method will convert this class to a regular function-based view, okay? So at the end of the day, there is a function under the hood that gets called, but when writing our code, we use a class, and that means we get access to all object-oriented programming features. Now, before we move on to the next lesson, I want you to convert this view function to a class-based view. That's super easy, so spend a couple minutes on this, then come back and see my solution. All right, here's what I've done. I've created a new class called product detail with three methods, get, put, and delete.

Now, the only thing I want you to pay attention to here is that previously, We had this line for getting the product outside of our if statements. But now, because we don't have those if statements, We have to add this line in every method to get this product and do something with it, Okay, That's the only change. Everything else is the same. now. Let's test our implementation. So, back to the browser, refresh. We got an exception saying, get got an unexpected keyword argument ID. Why is this happening? Well, back to our view. Look, we're sending pk to ID. But where is id defined well i forgot to include it as a parameter to the get method. Because when registering this route, look over here, we have the id parameter.

So our get, put, and delete methods should also have this parameter. So let's add that here as well. And one more time. Okay, now, beautiful. Our endpoint is working, so let's move on to the next lesson.

Mixins:

So I told you that class based views provide a lot of reuse opportunities. Let's talk about that for a few minutes. So look at the implementation of the get method of the productless view. There are three things we're doing here. First, we're creating a query set, then we're creating a serializer and giving it that query set. And finally, we are returning a response with a serialized data. If you have paid close attention, we have the exact same pattern when listing our collections. So let's quickly jump to our collection list view. Look over here. Once again, we're creating query set, then we're creating a serializer and giving it that query set. And finally, we are returning serialized data in the response. There are only two differences in these implementations.

Can you tell what they are? All right, the first difference is how we're creating a query set. So in this view function, this is how you're creating a query set. In the product list view, the logic for creating this query set is different. Now, what is the second difference? The second difference is the serializer we are using for serializing data. So here we are using the collection serializer. In the product list view, we're using the product serializer, right? Everything else is exactly the same. Now we have the same situation when creating a resource. So when creating a collection, first we create a serializer to deserialize the data. Then we validate the incoming data. Next, we save the data. And finally, we return a response with the serialized data.

We have the same pattern when creating a product. Now this is where mix-ins come into the picture. A mix-in is a class that encapsulates some pattern of code like this. So on the top, from rest framework, that mix-ins, let's import the list model mix-in as well as the create model mix-in. We're going to look at the implementations of these classes. So if you're using a Mac, hold down the command key. If you're on Windows, hold down the control key and then click on list model mixing. So this is a class with a single method called list. Now in this method, we have the logic for listing a bunch of models. So first, we're creating a query set. Now here we have a call to filter query set, don't worry about this, we'll talk about filtering and pagination later.

If you leave out the details, you can see that here we're creating a query set, then we're creating a serializer with that query set. And finally, we're returning the serialized data. This is very similar to the pattern we had in our code, right? Now, let's look at the implementation of the other mixin. So you can hold down shift command and O on Mac or shift control and O on Windows to see all the symbols in this module. So we're going to look at create model mixin. Now in this class, we have a create method that encapsulates the logic for creating a resource. So first, we're creating a serializer, then you're validating the incoming data. Next, we have a call to perform create method, which is another method in this class, and this is where we call serializer.save.

Next, we're getting the success headers to include in the response, and finally, we're returning a response with the serialized data, and the status of the response is 201. Again, very similar to how we implemented the logic for creating a product or a collection. So in Django REST framework, we have various mixins for performing different kinds of operations on a resource. So here on Django REST framework website, under API guide, look at generic fields. Here on the left, you can see all the available mixins. So we looked at list model mixin as well as create model mixin. Similarly, we have a mixin for retrieving a model or getting a single instance of a model. And we also have two mixins for updating and destroying or deleting a model.

Now, before we move on to the next lesson and see how we can use these mixins, I want you to spend a couple minutes and study the implementations of these mixins. So, here in the mixins module of Django REST framework, spend a couple minutes and study the implementations of other mixins.

Generic Views:

you learn about mixins. Now, most of the time, we are not going to use these mixins directly. Instead, we're going to use concrete classes that combine one or more mixins. We call these classes generic views. So here we have a class called list create api view that combines two mixins. List model mixin and create model mixin. Let's look at the implementation of this class real quick. So, back to our views module, from rest framework, the generics, let's import list create api view. Now, look, this class has multiple parents. List model mixin, create model mixin, and generic api view, which is the base class for all generic views. Let's have a look at this class real quick. So this class provides a bunch of methods that we're going to override in our custom views.

For example, here we have get query set for creating a query set object, and get serializer class for specifying the type of serializer we want to use in our view. Remember, the only difference we had between our product list and collection list views were in the query set and serializer classes, right? So, back to this generic view. This generic view combines two mixins and provides two methods. A get method and a post method. The get method simply delegates to the list method of the current object. Now where is the list method? Well, we inherited from the list model mixin. Because this mixin provides a list method for listing a bunch of objects, right? Now similarly, the post method simply delegates to the create method of the current object, which we have inherited from the create model mixin.

So a generic view is a concrete class that combines one or more mixins and provides handler methods like get, post, put, and delete. So this was list create API view. Now we have another generic view called list API view that only has the listing functionality. So if we use this view, we can only list our resources. We cannot create them. There is no create functionality in this generic view Okay, now similarly we have retrieve API view, which only provides retrieving functionality, But we have other combinations like retrieve update, retrieve destroy, as well as retrieve update and destroy. Okay, so so let's see how we can use these generic views. Back to the product list class. We're going to have this class inherit list create api view.

Now we need to override two methods, get query set and get serializer context. So define get query set. Now here we should return this expression. This is how we get a product query set. So let's return that right here. Next we need to override get serializer class. Now here we're going to return product serializer. Just a class, not an object. We're not creating an object here. Now if you pay close attention, while creating this product serializer, we need to pass a context object. We have another method for that that is defined in the generic API view class. So here we can override get serializer context. And here we can return our context object. What is that? It's a dictionary that contains the request object.

So here I'm going to return a dictionary. Give it the request. Now where is the request? We can get it from the current object. So self.request. Now with this new implementation, we can delete these two methods. Because all this logic is already implemented for us, right? So delete. And this is the end result. This is much cleaner and more concise than what we previously had. But we can make it even more concise. so in the generic api view class, we have two attributes called query set and serializer class. If you want to have some logic for creating a query set or a serializer, we can implement that logic in these methods. But if we don't have any special logic and we simply want to return an expression or a class, we can use these attributes.

So here i can set query set to this expression. That is simpler and more concise than defining a method. This method is useful if you want to have some logic, some condition for creating a query set. Maybe we want to check the current user and depending on the current user and their permissions, we want to provide different query sets. Similarly, we might need some logic for specifying the serializer class. Perhaps different users or different roles can have different serializer classes. In this particular case, we don't have any logic, so it's easier to use The serializer class attribute. We set it to product serializer. Okay, so we can delete these two methods. We're only going to keep this method because we don't have an attribute for specifying the serializer context.

It's only possible by overriding this method. okay? Now, back to the browser. Let's refresh. So we have the same functionality as before. We can see all of our products, but let me show you something really cool. Down the bottom, We have a form for creating a product. or if you want to use JSON, we can switch to the raw data tab and over here we have a sample of a product object that we can send to the server. Previously, before we started using a generic view, we didn't have this sample. We only had an empty content box. So this is another benefit of generic views. As your exercise, I want you to convert this view function to a generic view. Spend a few minutes on this, then come back and see my solution.

Alright, here's my implementation of the collection list view. As you can see, it's very simple. I've set query set to this expression that we had before and serializer class to collection serializer. Now we can see all collections in our database. But there's something tricky here. Down the bottom, when creating a collection, our JSON object has two properties title and products count. We don't want to specify products count when creating a collection. So If I just pass the title here and remove this property, look, we get an error when posting this to the server. We get a bad request saying products count is a required field. How can we solve this problem? Well, here in the collection serializer class, when defining this field, you want to mark it as read only because it's not used for creating or updating a collection.

So here we pass a keyword argument called read underline only. We set it to true. Now, Let's refresh this page. Down the bottom, we no longer have products count. So we can successfully create a new collection. Beautiful.

Customizing Generic Views:

There are situations where a generic view may not quite work for us. So let's see how we can customize it. Here's an example. Our product detail view provides three operations, get, put, and delete. Now we have a generic view that provides all these operations. So to use it here, we need to change the base class of this view to retrieve, update, destroy, API view. Now we need to set two attributes just like before. The first one is query set, which we set to what? Product.objects.all. And the second is serializer class, which we set to product serializer. Now with this change, we can delete the get method because this logic is completely implemented in the retrieve mixing. So delete. Similarly, we can delete the put method because once again, this logic, this pattern is completely implemented in the update mixing.

Delete. But look at the delete method. Here we have some logic that is specific to our application. So none of the mixins in Django REST framework know about products, order items, and their count. This is specific to our application. So here we need to override the delete method that we have inherited from this class. Let's have a look at this class real quick. So this class has four methods. Get, put, patch, and delete. we're using these three methods exactly as they are, but we're overriding this method and replacing its implementation with this other implementation, okay? This is how we can customize a generic view. Now let's test our implementation. So back to the browser, refresh. We get an exception saying expected view product detail to be called with a url keyword argument named PK.

Fix your url conf or set the lookup field attribute on the view correctly. What is happening here? Well, back to our URLs module, look at this URL pattern. In this pattern, we call this parameter ID, but our generic view expects it to be called PK. So to solve this problem, we can change this to PK and now refresh. Everything works beautiful. But let's imagine that we have a strong reason for calling this ID. How can we solve this problem? Well, back to our view. Here we have an attribute. called lookup field for specifying the name of the url parameter. If we set this to ID, the problem disappears, okay? Now, in this case, I prefer to stick to the default conventions in django res framework, so we can remove one extra line and call this parameter pk, and this way we're consistent with this other url pattern.

That's better. So, let's refresh. Now let's try to delete product number one. Okay, we got a familiar error. Delete got an unexpected keyword argument pk. Can you tell why this is happening? Pause the video and think about it for a few seconds. Here's the answer. We call this parameter pk, but in our delete method, we have a parameter called id. So we need to rename this to pk as well. Okay, now let's refresh and delete this product one more time. Okay, we got an error saying this product cannot be deleted, because it's associated with an order item. Beautiful. So we're done here. Now as an exercise, I want you to rewrite the collection detail view using a generic view. It's super easy. It's going to take you only a couple of minutes.

Alright, here's what I've done. I've created a class called collection detail that inherits retrieve, update destroy API view. Here offset query set to this expression that we had earlier and serializer class to collection serializer. Now just like the product detail view, we're overriding the delete method. So using the PK parameter, we look up this collection, then check to see if the collection has any product. If it does, we return an error. Otherwise, we delete the collection and return an empty response. And also, in the URLs module, over here, I called collection detail dot as view, pretty simple. So we're done here, let's move on to the next lesson.

ViewSets:

let's talk about view sets. So currently we have two views for managing our products. We have the product list view for listing and creating products, and we also have product detail for getting, updating, and deleting products. Now if you pay close attention, you can see that we have some duplication across these classes. For example, in both these classes, we have set serializer class to product serializer. Now look at our query set. Our query sets are very similar, but slightly different. in this view, we are eager loading products with our collection. But this is actually a mistake on my end. We added this earlier when we wanted to show the title of each collection when retrieving products. But we don't need this anymore. Because when retrieving products, we're just displaying the idea of each collection.

So there is no need to eager load products with their collection. So, if we remove the call to this method, now our query sets become identical. So we have more duplication across these classes. And this is where we use view sets. Using a view set, we can combine the logic for multiple related views inside a single class. That's why it's called a view set. It's a set of related views. So let's see how we can use them. On the top, from REST framework, that view set, we're going to import the model view set class. Now let's have a quick look at the implementation of this class. So this class has multiple base classes. We have all these mixins that you're familiar with, as well as generic view set.

Let's have a look at this class. So this class also has multiple base classes, view set mixin and generic API view, which you're familiar with. So everything you have learned about generic views also exist in generic view sets. So here we have those attributes like query set and serializer class and everything else you have learned so far. So back to our view, let's see how we can combine the logic for these two views using a view set. We're going to create a new class called product view set and look at the naming convention. So we have the name of our resource followed by view set. Now this class should inherit model view set. Now, because this is also a generic view, here we have query set and serializer class attributes.

So I'm going to move these two lines up here. Now, once again, because this is a generic API view, we also have this method, getSerializerContext. So let's move it right here. Now our product list class is empty. So we're going to delete it. Now in product detail, once again, we have query set and serializer class. We already moved this line, so delete. We just need to move this method for deleting a product because here we have some custom logic. So let's move this as well. Okay. Now we don't need this class anymore. So we are left with product view set that combines the logic for multiple views. So we have a single class for implementing the products endpoint. Using this single class, we can list our products, we can create them, update them and delete them.

This is the benefit of using view set. So this is our product view set, but our application is broken now. Because in our URLs module, we're still referencing these two views, product list and product detail, which are gone. so in the next lesson, we're going to talk about routers, and then you will see how we can use a router to create the routes for view set but before we get there, as an exercise, I want you to create a view set for our collections. So create a new class called collection view set for combining these two views all right here's my implementation. So we have a new class called collection view set that inherits model view set in this class i've set query set and

Serializer class attributes. And I've overridden the delete method exactly like before. Nothing new here. But I want to highlight something. Using this model view set if we inherit from model view set we can perform all kinds of operations on a resource. We can list that resource, we can create it updating and so on. But what if we don't want to have right operations? What if we don't want to be able to create a resource or update it or delete it? Well, we have another class in the view set module called read only Model view set so we can import this Now if we inherit from this class, we can only perform read operations We can list all collections or retrieve a single one, but we're not going to be able to create update or delete a collection Okay, so just be aware of that for now.

We don't need it. So I'm gonna use the model view set class here Alright, we're done with this lesson. Next. We're going to talk about routers.

Routers:

All right, let's talk about routers. So when we use view sets, we're not going to explicitly register these URL patterns. That's the job of a router. So we register our view set with a router, and the router will take care of generating these URL patterns for us. Let me show you how this works. So first on the top, from REST framework, the routers, we're going to import the simple router class. Next, we create a router object and register our view sets with this router. So we call router.register. Like here we need to pass two arguments. The first one is prefix, which is the value we are using right here. That's the name of our endpoint. So products without a forward slash, just the name of our endpoint, okay?

Now the second argument is our view set. So views.productViewSet. So with this line, we're saying that the product's endpoint should be managed by the product view set. Similarly, we can register our collection view set here so collections endpoint should be managed by collection view set now we can get all these URL patterns from router that URLs so if we're going any further, I just want to print this on a terminal so you see what happens under the hood. So from P print module of Python P print is short for pretty printing. It's a nicer way to print things on the terminal. so from this module, we're going to import the p print function now instead of print, we're going to use p print okay now because our application is currently broken, I'm going to temporarily comment out these lines.

Now, here in the terminal, look, we have an array of four url pattern objects. Here's the first pattern. This is for our products endpoint where we can get all products or create a new product. Now, this carrot we have here, represents the beginning of a string and the dollar sign represents the end of the string. So what we have here is called a regular expression. So here's our first pattern and the name of this pattern or this route is product dash list. So this name is generated based on the prefix that we specified. Again, another convention in django rest framework. Now look at our second url pattern. We have product slash and here's our pk parameter. Once again, all these extra characters represent a regular expression which specified the format of this PK parameter.

If you're not familiar with regular expressions, don't let that scare you. It doesn't really matter. Now the name of this pattern is product dash detail. Okay. Now we have two more patterns for our collections endpoint. So one for retrieving all collections and another one for working with a particular collection. Okay. So back to this module. Currently, we don't have any explicit patterns in this array. So we can simply set URL patterns to router.urls. As simple as that. Now, refresh. Look, our API is working, beautiful. So this is how we can use routers. Now, what if we have some specific patterns in this array? Well, then we don't want to set URL patterns to router.urls. So instead, we're going to create a new pattern here.

Path, for the route we're going to pass, an empty string. And here we're going to call the include function. Remember the include function we talked about before. With this, we can import routes from somewhere else. So we're going to include router dot URLs. In this array, we might have other paths or URL patterns for specific purposes. Okay. Now, in this case, we don't really have this scenario. So I'm going to simplify the code and set URL patterns to router that URLs. Okay. Now let's remove the call to p print and remove it from here as well. So this is how routers work. Now in this module, we also have another router called default router. If we use this router, we get two additional features.

So back to the browser. If we go to store, we get this page, which is called our API route. So over here, we can see various endpoints that are available to us. so we have the products endpoint at this address and the collections endpoint at this address now don't confuse this with the documentation this is not the documentation of your api because here we cannot see what operations are available at each endpoint we don't know if we can create a product or not we don't know if we can update it or not all we see is the url of endpoint also we don't know anything about the shape of our request and response objects so this is just an api route We'll talk about documentation later in the course.

Okay. Now the second feature we get here is that if you go to store slash products, and then append dot JSON at the end, we see our data in JSON format. So if a client application send a request to this address, they get all data in JSON. That's another additional feature. Hey, while reviewing this video, I realized I made a mistake and forgot to change something in this lesson. Actually, I discovered this way later in the course. Right now I'm in the middle of section 7 and I just discovered this issue in this lesson you were watching. So what is the issue? Well, back to our product standpoint. Look, we have this delete button here, but this button should only appear when we are looking at a specific product, like product number 1.

But here we are looking at the list of products, so it doesn't make sense to have this button here. But why is this happening? Well, earlier, when we combined our list and detail views into this view set, I forgot to change the delete method to destroy. Let me show you what I mean. So let's look at the implementation of model view set. So here we have nothing, we're just combining these mixins. Now let's look at the destroy mixin. Now in this mixin, we have this destroy method. And over here, we have a call to get object, this returns the object we're looking at like a specific product. So we get that object, Then we have a call to perform destroy, which is another method in this mixing.

And over here, we delete the instance. And finally, we return a response. This is very similar to our implementation of the delete method. The only difference is this validation logic. So instead of overriding the delete method, we should override the destroy method. Now earlier, we overwrote the delete method because before using this view set, we were using retrieve Update, destroy api view that in this class, we have a delete method that simply delegates to the destroy method. So earlier, we didn't have this problem. The problem happened after we combined our list and detail views into a view set, okay? So, let's fix the problem real quick. First, we're going to overwrite the destroy method. Def, destroy. Now, over here, we have a call.

to the destroy method of the super or the base class. Before doing that, we want to add our validation logic. So I'm going to grab these two lines and move them up here. Now here we need our product object, but I don't want to retrieve it from the database again, because in our destroy method, look, where is that in the straight model mixin. When we call get object, that product is retrieved from the database. So I don't want to retrieve it twice. Instead, I'm going to change our validation logic and rewrite it like this. So we can say if order item that objects that filter where product ID equals. Now from this keyword arguments, we can read the PK parameter. So keyword arguments of PK.

So we apply a filter and then we count the objects. If the count is greater than zero, then we return a response with the error. Otherwise, we call the destroy method of the base class, and this is where that product gets deleted. So let's delete this method. We don't need it anymore. And test our implementation so refresh all right the delete button is gone. But if you look at a particular product, the delete button is back. So let's try to delete product number one. We get this error just like before. So now we need to apply the same change in our collection view set. And I'll leave it to you as an exercise.

Building the Reviews API:

All right, now we're gonna take our API to the next level and introduce reviews. So a given product can have reviews, and we should be able to access an individual review like this. So because we have nested resources, we need to talk about nested routers. But before we do that, first we need to build our model. Now there are three steps we need to follow here. First we need to create a model class, then we need to create a migration, and finally we need to apply that migration. So let's do this together real quick. back to VS Code. Let's go to the models module of the store app. Here we're going to define a new class called review, which should extend models.model. Now in this class, we're going to define four fields.

The product that this review is for, which should be a foreign key to the product model. And here we're going to set undelete to models.cascade. So if we delete a product, all its reviews are deleted automatically. Now I would also like to set related name to reviews. So in the product class, we'll have an attribute called reviews. Next we need to define name. This is the name of the person leaving a review. So we're going to use a char field with a max length of 255. Next we define description. This is the actual review. And here we're going to use a text field so we don't have limitations. So people can leave longer reviews. And finally we define date. which is going to be a date field, and here we're going to set auto now add to true, so when we create a review object, this field gets automatically populated, okay?

So here's our model class, now we need to create a migration. So here in the terminal we run python manage.py make migrations. Alright, look at the name of this migration file. The name doesn't specify the kind of change we have applied, because as you can see We have some leftover changes from before and we forgot to create a migration. So as a best practice, whenever you make any changes to your models, always create a migration. Otherwise, you will end up with a big migration file that contains changes across multiple models. Now in this case, I don't want to worry about this, so let's go ahead and run this migration. So, Python, manage.py, migrate. Beautiful. We're done with the model, now let's work on the API.

now once again, there are three steps we need to follow here. First, we need to create a serializer class. Then we need to create a view. And finally, we need to create a route. Either using a router, if you're using view sets, or by explicitly registering a url pattern object. So, let's go to the serializers module and create a new class. Review serializer. This should extend serializers.model serializer. Now here we're going to define a meta class, and in this class we're going to set two attributes. Model, which we set to review, and fields, which we set to an array of strings. ID, date, name, description, and product. Okay? We have an indentation problem, so let's fix that. Good. So here's our serializer. Now we need to create a view.

So let's go to the views module. Okay? Here we're going to define a new class called review view set that extends model view set. So instead of defining two separate views, one for listing reviews and the other for working with an individual review, we're using a view set that combines all operations for view sets inside a single class. Okay? Now here we're going to set query set to review.objects.all and serializer class to review serializer. Okay? Now the final step. We need to register routes. And this is where we use nested routers. We'll talk about that next.

Nested Routers:

So we have this project on GitHub called DRF, or Django REST framework, nested routers. On this page, you can see the instructions for using this library. So first we install it using pip or pipenv. And then under quick start, you can see a real example. So we have domains and a given domain can have named servers. So here we have nested resources and that's why we need to use nested routers. So to implement this, first we need to create a simple router, just like before, and register the parent resource. So you're registering domains and mapping them to domain viewset. Nothing new so far. Now this is a new part. Once we do that, then we create a nested simple router. So just like we have simple router, we also have nested simple router that is defined in the routers module of this library.

So we create a nested simple router and give it three arguments. The first one is the parent router that we created earlier. The second one is the parent prefix, which is domains, or in our example, it's going to be products. And the third argument is the lookup parameter. So this is set to domain because in this URL, look, in the last URL, we have two parameters. The first one is domain underline pk, the second is pk. So the value we specify here as the lookup parameter will be used as a prefix for this parameter, okay? So we create a nested simple router, we call that domains router, and then on this router we register the child resource. So that is name servers, we're mapping this to name server view set, and finally we set base name to this value.

Now this value will be used for generating the name of our URL patterns. Remember, when we use a router to register a route, that router generates Two url patterns one is called list. The other one is called detail So this value is specified here will be used as a prefix for list and detail views. Okay So we create two routers here and then we include the route Across both these routers inside our URL patterns array. So let's go ahead and convert this example to products and reviews First we're going to go in the terminal and run peep and install drf dash nested dash routers now let's go to the urls module on the top from rest framework nested we're going to import the routers module now in this module we have a bunch of router classes so we have default router that replaces the default router that we imported from the rest framework we also have nested default router now similarly we have

simple router and nested simple router. So we're going to use the default router that comes with this new library so routers the default router and that means we're not going to use the default router that comes with django rest framework. So we can delete this line. Good. So we create the parent router. Now we need to create the child router. So we say routers dot nested default router here we have to pass three arguments, the parent router, the parent prefix, which is product, and our lookup parameter. So lookup equals product. So that means we're going to have a parameter called product underline PK in our route. Okay. So we create a nested default router and store it in an object called products underline router.

Now on this router, we're going to register our child resource. So products router, the register, here we specify the prefix, which is reviews. Then our view set, which is views dot review view set. And finally, base name, which is used as a prefix for generating the name of url patterns. So that's going to be product dash reviews. So our routes are going to be called product reviews list or detail. Okay? This is just a prefix. Now that we have two routers, we can combine the URLs, both these routers and include them in the URL patterns object. Now, alternatively, if we have set URL patterns to an array and in this array we have some explicit routes, we can use the include function to include the route across both these routers, okay?

So, let's test our implementation. Over here, let's go to products one slash reviews. Great. Currently, we don't have any reviews. So let's post a review object here. I'm going to set the name to mosh description to first review and product to one. Now technically, we don't need product here, we can read it from the URL. But for now, let's not worry about it. So post beautiful, we have a new object. So the status of the response is 201. And here's the ID of our review. So if you refresh this page, now we see the review that is assigned to this product. Now we can go to this review and see all the details. We can delete this review or update it. Everything is working.

So this is how we can use nested routers. Now let's see how we can improve this implementation. So let's go to our reviews endpoint and create another review. So we don't want to pass the product ID here. We want to read it from the URL. So back to the review serializer class. Let's remove the product field from this array and see what happens. So refresh. Now, when creating a review, we don't have to specify a product. But let's see what happens when we post an object to the server. We get an error saying column product ID cannot be null. Why is this happening? Well, back to our serializer. You know that this serializer has a save method that would either create a review or update it.

Now when creating a review, it would take these values, and use them to set various fields of a review object. Now, we don't have a product ID here, and that's why we're getting that exception. But how can we solve this problem? Well, back to our view set. In this view class, we have access to URL parameters. So we can read the product ID from the URL, and using a context object, we can pass it to the serializer. Remember, we use a context object to provide additional data to a serializer. So here we're going to overwrite and the get serializer context method and return a dictionary in this dictionary we're going to add a key called product ID and we're going to set it to self that keyboard arguments this is a dictionary that contains our URL parameters so from here we're going to extract product underline PK remember I told you that our URL has two parameters product PK and PK so we pass this dictionary to our serializer and

Now, in our serializer, we're going to override the create method for creating a review. So instead of relying on the default implementation that would simply get these values and set various fields in a review object, instead we're going to provide our own implementation. So first we're going to read product ID from self.context, which is dictionary. From here we're going to extract product ID. And then we're going to say review.objects that create. So here we can pass multiple key value pairs. We're going to set product ID to this value. And then we're going to unpack the validated data dictionary that we receive here. So we create a product object and then return it from this method. Okay, let's test our implementation. So refresh, and we're going to create a new review, a and a post

Beautiful. Now we have a new review with ID2. Now here in our database, look at the review table. Here's our review. And as you can see, this review is assigned to product number one. Great. Now, what if we go to product number two slash reviews? We still see this review. This should not be here. So let's see how we can solve this problem. Back to our view. Look, we have set our query set to review.objects.all. So all reviews are returned. no matter what product we're looking at. So here we need to apply a filter, but there's a problem. If we call the filter method and set product ID to self, that keyword, look, we don't have access to self here. So instead of setting the query set attribute, we need to override the get query set method.

So delete this and override get query set. Here we're going to return review.objects.filter Product ID equals now here. We have access to self or current object. So we say self that keyword arguments of product underline PK just like before now Let's test our implementation. So refresh we don't have any reviews for product number two But if you go to product number one slash reviews, we see this review here beautiful.

Filtering:

So currently, when we hit the product standpoint, we get all products on our database. But what if you want to filter these products, let's say filter them by a specific collection, we should be able to pass a query string parameter here, like collection ID. If we set this to one, we should only see products in this collection. So let's see how we can implement this. Back to our product view set class. Look, we have set query set to product that objects that are here we need to apply a filter. But in the previous lesson, you saw that we cannot call the filter method here. So we need to delete this line and overwrite the get query set method. Now here we say product.objects.filter where collection ID equals, now to read query string parameters, we have to go to self.request.query underline params.

This is a dictionary, so from here we can read collection ID. But what if we don't have a collection ID? This code is not going to work. So the proper way to implement this method is like this. First, we define a query set and set it to product.objects.all. Then we try to read collection ID from query string. So we define a variable collection ID and set it to self.request.queryPrams of collection ID. Now, if collection ID is not none, this is where we apply the filter. So we get the query set that we defined earlier. And on this query set, we call the filter method and say collection id should equal collection ID. Now here we get a new query set. So we'll use that to reset our previous query set.

And finally, we return this query set. Now let's test our implementation. So back in the browser, refresh. Our API is not loading, so let's look at the terminal window. Look, we have an error saying base name argument not specified and could not automatically determine the name from the view set as it doesn't have a query set attribute. So what this error is saying is that because we removed the query set attribute here, Django REST framework cannot figure out the base name. Remember the base name? Let's go to the URLs module. So earlier, with this nested router, We set the base name while registering this route. And I told you that this base name is used to generate the name of our URL patterns.

Now, by default, Django REST framework uses the query set attribute to figure out the base name. But because we deleted the query set attribute and now we have a method, Django REST framework cannot figure out what the base name should be called based on this logic. It's too complex for it. So we have to explicitly specify the base name. Back to the URLs module. This is where you're registering the product standpoint. So over here, as the third argument, we set base name to product. So with this, we'll have two URL patterns, products list and products detail. This is just a prefix, right? So back to the browser, refresh, it's still not working. So back in the terminal, looks like my change is not picked up.

So I'm going to stop the web server and restart it. Okay, and refresh. All right. Now we have a multi-value dictionary key error. Collection ID. The reason we're getting this error is because currently we don't have a collection ID here. So if you pass a Collection ID, the error will go away. And now we only see products in collection number one, right? So back to our view set. The reason we saw that error is because of how we are written this line. I told you that query prams is a dictionary and Now here we are assuming that our dictionary always has a key by this name. But what if we don't? This syntax is not going to work. That's why we get that error.

So instead of using square brackets, we have to call the get method. The get method returns none, if we don't have a key by this name. Now here we can also optionally supply a default value if we want to. In this case, we don't need it. So collection ID should be none. So this line will not get executed. Now, Back over here, let's remove this parameter. Now we see all products, and if we apply a collection ID, we only see products in a particular collection. Beautiful.

Generic Filtering:

So we implemented basic filtering. But what if, in addition to collection, we want to filter our products by another field? Then this logic is going to get more complicated. This is where we can use generic filtering. So we're going to use a third-party library called Django Filter so we can easily filter any models by any fields. We don't have to hand code this filtering logic. Let me show you. So here in the terminal, first we're going to install django-filter. Next we need to add this in the list of installed apps. So we go to installed apps and Add that right here. Django underline filters. So the name of the app is different from the name of the library. The library was Django dash filter.

The name of the app is Django underline filters. Pay close attention. Now that we have said this, we go back to our view set. On the top, from Django, underline filters, dot, VS Code is not picking up this app, so the workaround is to close this project. Close folder, and reopen it. Okay, so we have django filters, and then we go to rest framework, and import django filter backend. This backend gives us generic filtering. Now that we have this, we go to our view set, and We set filter backends to an array with Django filter backend. With this backend, all we have to do is specify what fields we want to use for filtering. So we set filter set underline fields to an array of fields.

In this case, collection underline ID. Now we can completely remove our filtering logic and bring back our query set attribute. So product, .objects.all. That's all we have to do. So we specify our filtering backend and the name of fields for filtering. Now, because we closed this project and reopened it, we have to restart our web server. So python, run server. Great. Now, back to the browser. Refresh. We see all products. Beautiful. But if you specify a collection ID here, so collection ID equals one. Now we only see products. in this collection. But there is more to this. With this filtering backend, we get this new button here called filters. So we can easily select a collection here and see products in that collection.

That's very handy. Again, another reason I love this browsable API. Now let's take filtering to the next level. What if you want to filter by unit price? So back to our view set, we can add unit price here. But there is a problem with this. Let's refresh. and look at our filters, if we set unit price to 10, we only see products whose unit price is 10. This doesn't really make sense, because when filtering products by price, we want to find products whose price is in a given range, greater than some value or less than some value. So this is where we need to implement a custom filter. Now that really goes outside the scope of this course, you need to look at the documentation of Django filter library.

So if you just google Django filter, You can find that page here Django dash filter dot read the docs that I own on this page You can find all details about creating a custom filter, but let me show you an example real quick So here in the store app, I'm gonna add a new file called filters now from Django Underline filters that rest framework. I'm going to import the filter set class Now we create a class called product filter which should extend filter set. And in this class, we define a meta class. Here we set the model to product, which we should import. It's not picking up, so we have to manually import it from the models module in the current folder. Okay?

So we set the model and then the field to dictionary. So instead of using an array, we use a dictionary because here for each field, we can specify how the filtering should be done. So for collection ID, We're going to use exact filtering, so equality comparison, but for unit price, We're going to use, well actually here. We need to pass an array. Okay, so for unit price, We're going to use less than and greater than. This is a special language that this library understands. So for more details, you really need to look at their documentation. Now that we have a filter class, we go back to our view set. So instead of using filter set underline fields, we set filter set underline class to product filter.

So all our filtering is encapsulated inside this class. Now, back to the browser, refresh, look at our filters, so we can filter by collection. I'm going to pick all products, but i want to filter by unit price. So I want to make sure the price is less than 10, but it doesn't really make sense. It would be better if we swapped the order of these fields. So greater than some value and less than another value so Back to our product filter. I'm gonna add greater than first and less than second. Now, Let's refresh and look at our filters. So, all products whose unit price is greater than 10 and less than 20. There you go. Now we see these products, so this one is $19, this other one is $16, and so on.

Now look at query string parameters, so we have collection ID and unit price double underscore greater than and unit price double underscore less than. So this library takes care of reading all these query string parameters and filtering our products by these values. It's incredibly powerful. So we're done here. Next we're going to talk about searching.

Searching:

Now, what if you want to find products by their title or description? This is where we use searching. So searching is for text-based fields. Let's see how we can implement it. It's super easy. So back to our views module. First, on the top, from resframework.filters, we import the search filter class. This is another filter backend. Now, I see we have a bunch of unused stuff here, so let's clean up this code real quick. First, on the top, we don't need to import collections. I don't know how it ended up up there, so Delete. We don't need http response, so delete this line. Also, we don't need these generic views anymore because now we are using view sets. So, delete. We don't need to import api view decorator because now we are using class based views and similarly we don't need to import api view and read only model view set.

Okay? That is better. Now, back to our view set. Here in the filter backends array, we're going to add search filter, and we're going to set search fields to the list of fields we want to use for searching. So title and description. Now, we can also reference fields in related classes. For example, we can say collection double underscore title. In this case, it doesn't really make sense, so I want to search only in title and description. Now, back to the browser. Let's refresh. Okay, so look at our filters. Now we have a search box here, so if we search for coffee, we find any products that have coffee in their title and description. And as you can see, our search is case insensitive, so even though I typed coffee in lowercase, we can still see this product.

Now we can also type multiple search words, so over here we can say coffee space 10 ounce, or we can separate them using a comma, either works. So if we search, Now we can only see products that have both these keywords in their title or description. So this is all about searching. Next, we're going to talk about sorting data.

Sorting:

All right, now let's see how we can sort data. So from the same module that is restframework.filters, we're going to import another filtering backend that is ordering filter. Now I know the name is a little bit weird because ordering has nothing to do with filtering, but it is what it is. So let's accept it and move on. So back in our view set, in the list of filter backends, we're going to add ordering filter. And just like how we specified our search fields, here we can specify our ordering fields. So we set it to the list of fields we want to use for sorting, like unit price and last update. Now back to the browser, refresh. Let's look at our filters. Now down here, we have various ways to sort our data.

So let's sort by unit price in ascending order. Okay, now we can see this product is only one dollar and so on. Now look over here. we have a query string parameter called ordering. Now if you want to order by unit price in descending order, we prefix this with a minus. Now the most expensive product is on top. Now we can also sort by multiple fields, so we can sort by unit price in descending order and by last update in ascending order. That also works. Now what is interesting here is that from this API. So even though we are not returning it here, we can still use it for sorting data. This is all about sorting. Next, we're going to talk about pagination.

Pagination:

Alright, now let's talk about pagination. So in our views module, from resframework.pagination, we're going to import the page number pagination class. So using this class, we can paginate data using a page number. So we can go to page 1, 2, 3, and so on. And by the way, if you haven't noticed, I like to sort these import statements because this makes our code more readable. So first, I like to bring stuff from Django, followed by stuff from resframework, and then stuff from our application. There's just one tiny problem here. And that is the first statement here. So from store that filters, we're importing product filter. This is part of our application. So I like to list this at the end. And also, we don't need to add the application name.

So dot filters, that is better. Now we're going to use this class in our view set. So here we said pagination class. And once again, look, I tried to sort these attributes as much as possible. So we set this to page number pagination. now we need to specify the page size. For that, we have to go to our settings module. So, settings. Now here in the settings for rest framework, we have to set page size to, let's say, 10. Now, back to the browser. Note that the moment i refresh, the result we get here is going to be different. So, take a look. Now, instead of an array of products, we get an object with these properties. So we have count, which is the total number of products in our database.

We have next, which is a link to the next page. So we go to products endpoint, and here we have a query string parameter that says page equals to. Now because we're on the first page, previous is null, otherwise this would be a link to the previous page. We also have results, which is an array of products, okay? So now we can go to the second page, and we see different result. Now this pagination is currently only available in the products endpoint. So if you go to the collections endpoint, we don't have pagination. Now if you want to have pagination everywhere, we can set it globally in our settings module. So here we set another setting called default underline pagination underline class. We're going to set that to rest underline framework dot pagination dot page number pagination.

Now with this, we don't need to specify the pagination class per view or per view set. So, back to our views module, we can delete this line. Now, back to the browser, let's refresh. We have pagination for our collections and our products. Beautiful. Now we have another pagination class called limit offset pagination. So instead of using a page number, we use a limit and an offset value. Let me show you. So, back to the settings module, let's change the default pagination class to limit offset pagination. Now back to the browser. Let's refresh. So look, our query string parameters have changed. Now instead of a page number, we have limit and offset. So for the next page, we're going to take 10 products and we're going to skip 10 products.

This is another way to paginate data. But quite often, we use page number pagination. So back to the settings module. I don't want to use this default pagination class everywhere. So I'm going to delete this line. Back to our views. I want to bring it only in our products view set. So we set pagination class to page number pagination. So now we have pagination only product view set. However, here in the terminal, we get a warning saying you have specified a default page size setting without specifying a default pagination class. Because in our settings module, we removed the default pagination class, but we left the page size setting. So that's why we're getting this warning. Now there are two ways around this. One way is to suppress the warning, which I personally don't like.

So we have to do some research and find a specific error code and include that in the REST framework settings. So we have to suppress that error code or that warning code. I don't like that approach. The other approach is to create a custom pagination class and set the page size there. Let me show you how we can do this. So here in the store app, we're going to add a new file called pagination.py. Now, from REST framework, We're going to import page number pagination class. Now here we define a custom pagination class. We can call it default pagination that inherits page number pagination. And here we set page size to 10. Now back to our settings module. We're going to remove this setting and back to our view.

Instead of page number pagination, we're going to use default pagination. So we no longer need page number pagination in this module. So on the top, let's remove this line. Good. Now back in the terminal, the warning is gone. So we're done with this section. In the next section, we're going to build a shopping cart API from scratch. So I will see you in the next section.

Designing and Implementing a Shopping Cart API:

Introduction:

Welcome back to another section of the ultimate Django course. In this section, we're going to put everything together and build a shopping cart API from A to Z. This is a fantastic opportunity for you to practice everything you have learned so far. So, let's jump in and get started.

Designing a Shopping Cart API:

All right, let's talk about designing our API endpoints. So what operations do we need to support here? Well, our clients should be able to create a cart, add items to a cart, update the quantity of an item, remove an item from a cart, get a cart with all its items, and delete a cart. Now here's what I want you to do. Grab a piece of paper, and based on what you have learned in the course so far, figure out what endpoints we need to support these operations. For each operation, specify what kind of HTTP requests we should send to the server, Is it going to be a get request, a post request or what? Also specify the URL endpoint as well as what we're going to send in the request and what we will receive in the response.

So spend a few minutes on this, then come back, watch the rest of the video. All right, for creating a cart, we're going to send a post request to the cart's endpoint. Now note that the body of the request is empty here because as I told you before, our carts are anonymous. So we don't want to force users to log in before they can add items to their shopping cart. So when creating a cart, we're not going to send someone's user ID or customer ID to get a cart back. So we send a post request to this endpoint and we get a cart object from the server. Now this cart object has a unique identifier that we're going to save on the client for subsequent requests.

So when the user adds an item to their shopping cart, we're going to send the cart ID back to the server. We'll talk about that in a minute. Now for getting a cart, we're going to send a get request to this other endpoint. And obviously we get a cart object back. And similarly for deleting a cart, we're going to send a delete request to this endpoint. Pretty straightforward. Now let's talk about cart items. For adding an item to a cart, we're going to send a post request to this endpoint. So this endpoint represents the items of a particular shopping cart. So here we have the shopping cart ID. Now in the body of the request, we're only going to send two attributes, product ID and quantity.

Because we have the cart ID in the URL, so all we need to send to the server is product ID and quantity. So we send this request and we get the item that was created. Now this item also has a unique identifier that will use for subsequent requests. For example, when updating a cart, we're going to send a patch request to this other endpoint. Now here we have two URL parameters. The first one is the cart ID and the second one is the item ID. Okay. Now in the body of the request, we're going to send the quantity and in the response, we'll get the updated quantity. So here we're going to support only patch requests, not put requests. Because we put we can replace an entire object.

But in this case, we're not going to replace an entire item, we just want to update its quantity. That's why we use the patch method here. And finally, for deleting an item, we send a delete request to this endpoint. Again, very simple. So if you put all this together, essentially, we have four new endpoints, two for carts and two for cart items, we can implement the first two endpoints using a class called cart view set, and the other endpoints using a class called cart item view set. But before we implement these, first we need to revisit our data model. We'll talk about that next.

Revisiting the Data Model:

All right, before we build our API, first we need to address a couple of issues in our data model. So let's look at our cart model. Here we have a single field that is created at. But as you know, Django automatically gives each model a primary key field, and the type of this field is an integer. So that means in our API, we're going to have a URL like this, cart slash one. Now there's a problem here. The problem is that a hacker can easily guess someone else's cart ID and send a request to this endpoint to mess with that cart. Because it's really easy to guess these numbers, 1, 2, 3, 4, and so on, right? So to solve this problem, we're going to use a GUID, or a Globally Unique Identifier, which is a long 32-character string.

It's going to make it much harder for a hacker to guess that string. So, back to our model, we need to redefine the primary key field. id equals models. UUID field, or we can say UID, however you prefer. Here we're going to set primary key to true and give this a default value. So when we create a cart object, Django automatically assigns the ID field a u8 now on the top, From the u8 module of Python, we need to import the u8 for function now back over here We're gonna set default to u8 for note that I'm not calling this function. I'm simply passing a reference to it and If we call the function, at the time we create a migration, a GUID will be generated and hard-coded in our migration file.

Let me show you what I mean. So here in the terminal, let's run Python, manage.py, make migrations. All right, look at this migration file. So over here, look, we have a GUID, and this is hard-coded in our migration file. We don't want to use the same value for every shopping cart. So that's why we shouldn't call that function. So Let's delete this migration file. Well, back over here, we're not going to call this function. We just pass a reference to it. Now let's recreate this migration. Good. Now take a look. In this new migration, we don't have a hard-coded GUID. Look, default is set to a reference to UID function of Python. Now, before we run this migration, I want to talk about the implications of making this change.

So back to our database. Look at our cart table. Look, here's the primary key and the type of this key is big integer which takes 8 bytes in MySQL. If we convert this to a GUID, we're going to store 32 bytes here. So the new field or the new key is 24 bytes or 3 times larger. Now look at the cart item table. Here we also have a reference to the cart ID. So for each record in this table, we're going to store 24 bytes extra data. Now is this going to be an issue? Well, it really depends. People have different opinions about using GUIDs in the database. Some people are totally against using GUIDs because they take extra space and performance-wise, they're a bit slower than integers.

But let's do some real assessment here. Let's imagine in this table we have 1 million records. So let's pull up the calculator and do some basic calculations. So 1 million records times 24 bytes. That is the extra space in bytes. So we're going to divide this by 1024. This is the value in kilobytes. And one more time, divide by 1024. So we're going to store 22 additional megabytes of data. Is this an issue? In my opinion, no. These days, disk space is so cheap. And 22 megabytes for 1 million records is really nothing. Also, take into account that this is really a temporary table. It's not going to grow indefinitely. So as people place orders, we're going to move these records to our order and order item tables.

In those tables, we're not going to use GUIDs, we're just going to use integers. Because the orders API that we're going to build later in the course is going to be secure, it's not going to be open to anonymous users. So a client has to authenticate and be authorized to access a particular order. We'll talk about that later. But back to this story, this table cart item is not going to grow indefinitely. So for 1 million records, we're going to store 22 additional megabytes of data, not a big deal. And also take into account that here in the cart table, we have this field, and with this we can keep track of abandoned carts. So if some people add items to the shopping cart and then forget about it, every now and then we can clean up this table.

So we can run a job and delete all cars that are, let's say, more than three months old. So this table is not going to grow quite large. Now what about performance? Yes, in theory, looking up a great key is slower than an energy key, but these days our servers are so powerful and Also, all these database engines are highly optimized. So I don't think we should do any kind of optimization here before we do a proper test and make sure this is going to be an issue. As I always say in my courses, premature optimization is the root of all evils. So don't assume right from the get go that this solution is going to be slow and then make it complicated.

But in case you're curious, let me briefly talk about an alternative optimized solution. The other solution is to keep this structure exactly as is. So we're going to use a big integer key here. But in this table, we're also going to have an additional column, we can call that unique ID or cart ID, whatever. In that column, we're going to store a grid. So with this change, we're not going to store a bunch of grids. In the cart item table, we're only going to have integers. It's right here. But because each cart has a grid in our API layer, instead of a number, we're going to use grids. But internally, we're going to translate that grid to a number. So that's going to complicate our queries a little bit.

That's why I'm totally against premature optimization. You always have to pay cost for these optimizations. Your code is going to get more complicated. There is a chance you introduce new bugs. And also, you may end up with other performance problems. So we're going to keep this change and run our migration. So back in the terminal, Python, manage.py, migrate. Beautiful. Now, there are two more changes we need to apply here for the cart item model. The first change is that here we have a foreign key, but I want to assign a related name to items. So that means in our cart model, we're going to have a field called items. It's nicer than cart item underline set. Okay? So that's one change. The other change is that in the first part of the course, I forgot to apply a unique constraint here.

Because in this table, we want to make sure we only have a single instance of a product in a shopping cart. So if the client adds the same product to the same cart multiple times, instead of creating multiple records, we should only increase the quantity. Okay? So using a unique constraint, we can make sure that there are no duplicate records for the same product in the same cart. So how do we do that? Using a meta class. So class meta. And here we set unique together to a list of lists. Because we can have multiple unique constraints on different fields. For example, we can have a constraint on cart and product, but we can have another constraint on two or three other fields.

That's why we have a list of lists. Now, we don't need this here, so let's simplify the code. Good. So, I saved the changes, that's why the screen jumped a little bit. Now let's create another migration. So Python, manage.py, make migrations. Now, let's run it. Python manage.py migrate. Beautiful. So our data model is updated. Now back in the database, let's refresh. So look, in our cart table, the ID column is now a char of 32, a string of 32 characters. And also, we have a unique constraint on these two columns, cart ID and product ID. So our data model is in a good shape. From the next lesson, we're going to build our API step by step.

Creating a Cart:

Let's see how we can create a cart. Now, I highly encourage you to pause this video and treat this as an exercise because you have all the knowledge to implement this on your own. So spend a few minutes and implement creating a cart, nothing more, because we want to implement these operations Step by step together. Okay. All right. Here's my solution. Now, whenever we want to implement an API, first, we build a serializer, then a view and a route. So let's go to our serializers module and Over here, we're going to create a new class called cart serializer. They should extend serializers dot model serializer. Now here we're going to define a meta class. We set model to cart and fields to ID and created at that technically we don't need to return this to the client because this is only used on the server.

This is for maintenance. So I don't see a value in returning this to the client. So let's simplify this. We only return the ID. So this is our serializer. Now the view. So let's go to the views module and create a new class called cart view set. Now here's the tricky part. We're not going to inherit from model view set because this class provides all operations. List, retrieve, create, update, and delete. But look at the operations we need to support for our cards. We need to be able to create a card, get a card, and delete it. So here we don't have a list operation. So we cannot send a get request to the carts endpoint and retrieve all carts. Otherwise, all our cart ideas are going to be exposed to the outside.

We don't want to do that. And similarly, we don't have an update operation. So we cannot send a patch or put request because it doesn't really make sense to update a cart. We only update the quantity of a cart item, not a cart. So instead of extending model view set, we want to create a custom view set. How can we do that? Well? Let's look at the implementation of this class real quick. Look, there is nothing here and we just have the pass keyword. So this class, as I told you before, is nothing but a combination of all these mixins with a generic view set Now here we don't need the update and list operations. So we're going to create a custom view set, but I'm not going to add retrieve and destroy mixins yet.

We're going to add them as we go through this section. So back to our views module. First, we go on the top, and from rest framework that makes sense we're going to import create model mixin, and Here in the view set module. We're going to import the generic view set class now back to our view. So, first we add create model mixin, and then generic view set Now here we set the query set to cart that objects that all and and serializer class to cart serializer. We're done with the view. Next, we need to register a route. So we go to our urls module. Here's our default router. So on this router, we're going to register a new endpoint called carts and map it to views.cartViewSet.

Good. Let's test our implementation up to this point. So let's go to our api root. Now we have The cart's endpoint, beautiful. Now at this endpoint, we don't support a get operation. So look, if we send a get request to this endpoint, we get HTTP 405, which means method not allowed. But we can post to this endpoint. So over here, we can post an object. Now there's a tiny issue here. ID is set to null, but we don't want to send the ID to the server. We want to send an empty object. How can we solve this problem? Well, what you see here comes from our serializer. back to cart serializer. Look, we have listed id here. That is why we saw this field in the content box.

Now here we want to declare this field as a read only so we don't have to send it to the server. We're only going to read it from the server. So we're going to set id to serializers dot do it field and we said read only to true. Now if you didn't know this part, that's totally fine. Remember, you're still learning. So even though i said you have all the knowledge to implement this on your own, didn't expect you to implement it to perfection, because you're still learning. Now, back to the browser, refresh. Now we have an empty object here, so let's post it to the server, and we get this new cart with this ID. Beautiful. So we're done with the first operation, next we're going to implement getting a cart.

Getting a Cart:

All right, now we're going to implement getting a cart. Now, once again, I want you to implement this on your own first and then come back, see my solution. But before we get started, let me tell you what exactly I want you to build. So we created a cart. Now, if we grab the cart ID, add in the URL and send a get request here, we should get a cart with all its items. Now, currently we don't have a way to add items to a shopping cart. So I want you to go into database and manually populate the cart item table. So add three items to the shopping cart, and then as part of implementing this operation, return all those items in the shopping cart.

Now each item should include a product object, as well as a total price, which is unit price of the product times quantity. Now the cart object itself should also include the total price. Now I have to tell you, this exercise is a little bit more tricky than the previous one. So it's going to take 15 to 20 minutes of your time, maybe a little bit longer, you might get stuck, you might get errors. Don't worry, don't get disappointed. Just do your research and do your best to solve this problem on your own. Because in the real world, I'm not next to you to help you out. So you have to learn how to solve problems on your own. So spend 15 to 20 minutes on this, then come back see my solution.

Okay, we get this cart instance, beautiful. But currently, we are not returning our cart items. So back to our cart serializer. Let's add the items field here. Now let me quickly point out something here. Let's go to our cart model. look at our cart item. Earlier, I added related name here to items. So in our cart model, now we have a field called items. That is why we can add this field over here, okay? Now, let's refresh. So we get an empty array. So now let's go to our database and add a few items. So let's open up the cart table. I'm going to copy the cart ID. Now let's go to the cart item table. I'm going to add three records here.

So for the first one, I'm going to set the quantity to 10. Cart ID to this value that I copied and product one for a second 120 and 230 and three. Now I'm going to submit my changes. Whoops, we got an error looks like we don't have one of these products. I think earlier I deleted product number two. So let's use product number four here. Now, let's resubmit. Great. So back to the browser. refresh all right now we have item IDs, but we want to return the actual cart item objects. So here we need a serializer for our cart items. So back to our serializers module, let's define a new serializer. Cart item serializer. They should also extend model serializer. Here we define a meta class, set the model to cart item, and field to

ID, product, and quantity. Now, we need to use this in our cart serializer. So we have to move its definition before cart serializer, okay? Now we're going to explicitly define the items field and set it to cart item serializer with many equals true. Let's see what happens. So refresh. Beautiful. Now, we see the actual cart items. But you want to do the same thing with our products. So instead of returning the product ID, you want to return an actual product object. How can we do that? Very easy. Back to cart item serializer, we should redefine the product field. So this is going to be a product serializer object. Now, let's refresh. Okay, so we get this product objects. But look, We're getting all fields of product objects that we defined earlier.

So we have title, description, slug, inventory, and so on. What if we don't want to return all these fields when getting a cart? Perhaps in the future, we're going to add additional fields in our product serializer, so we don't want all these fields end up here. Let me show you how to solve this problem. We can create another serializer for serializing a product in a shopping cart item. So instead of using this product serializer, we can use another serializer. So let's define, Another serializer. We can call that cart item Product serializer or if you don't like this name, we can call this simple product serializer and potentially reuse this in other situations where you want to return basic information about a product, like its ID, title, and unit price.

Nothing more. So, this should also be a model serializer We define our meta class as usual, Set the model to Product, fields to just the basic fields we need. So id, title, and unit, underline price. And here we're going to use a simple product serializer. Now take a look. Refresh. Okay, that is much better. So next we need to add the total price for each item. That is going to be a calculated field. So back to our cart item serializer. Here we're going to add another field called total price and And this is going to be a calculated field. So we said total price to serializers dot serializer method field. Okay. Now earlier I told you that with these fields we can follow a particular convention for defining the method that returns the value for this field.

So that method should be called get underline followed by the name of the field. So total price. Which takes self and a cart item. now here we're going to return cart item dot quantity now look because we don't have IntelliSense, we can annotate this with cart item and now we get IntelliSense. So we're going to multiply quantity by cart item dot product dot unit price okay let's see what happens up to this point so refresh all right now we have the total price. So quantity, which is 10, times unit price of four dollars comes down to forty dollars beautiful now we also need the total price for our cart so back to our cart serializer once again we're going to add a new field called total price which is going to be a calculated field so serializers dot serializer method field we define our method get total price which takes self and a cart now here i'm going to write

a list comprehension. Here's the syntax. We add square brackets, and here we say item for item in collection. Now what is our collection here? That is cart.items.all. Because cart.items returns a manager object, so using all, we get a query set which returns all these items. Now for each item, instead of returning the item, we want to return item.quantity times unit price. So item.quantity times item that product that unit price okay so with this expression, we get a list of totals. Now we need to sum all of this up. So we're going to return a sum of this list. Okay? Let's test our implementation. Back over here. Refresh. Now look at the total price for the cart. Beautiful. Now one last thing before we finish this lesson.

Let's open up django debug toolbar. So currently we have 13 queries to produce this result. That doesn't sound right to me. So let's see what is going on here. First, we're getting a cart. Then we're getting the items for this card then for each product, we have an extra query to read the attributes of that product, like title and unit price. This is where we need to use eager loading. So when retrieving a cart, you want to eager load that cart with its items and products. Where should we do that? In our view set. So back to our view set. This is where we have the query set for retrieving our cart. So here we're going to call prefetch related because a cart can have multiple items.

And that's why we use the prefetch method for foreign keys where we have a single related object. Instead of prefetch related, we use select related. Okay, so we want to prefetch a cart with its items. Now for each item, we also want to preload the product. So here we add double underscore product. Now, back to the browser, refresh, look. Currently, we have only seven queries. So the first few, as you know, are always there. Now, for getting a cart, we have a query to retrieve the cart. Then we have another query to retrieve the items in the cart and one to retrieve all products referenced in this cart. So we don't have a separate query per item. We're done with this operation. I know that was quite a lot of work.

It's totally fine if you couldn't finish it. I think that was probably the most complicated operation we had to build in this section. So going forward, the other operations are going to be simpler.

Deleting a Cart:

All right, now we're going to implement deleting a cart. This one is super easy and you can knock it out in a couple of minutes. So do the exercise and then come back, see my solution. All right, let's get started. So currently we don't support delete methods at this end point. That's why we don't have a delete button here. But adding that button is as simple as adding another mix into our view set. So back to our cart view set, currently we're supporting the create and retrieve operations. So we're going to add another mix in here called destroy model. mixin. Now it's also better to reformat our code. So we can add his class names like this. That's better. Now let's refresh. All right, we have a delete button.

But I'm not going to delete this cart. We have some data here, we want to get back to it. So let's go to our cart endpoint and create a new cart. Oh, and by the way, look, now that we have added the items field, that field appeared here when creating cart. So let's quickly solve this problem. Back to our cart serializer, we're going to mark this field as read only, you're only going to return it from the server. Okay, so read only equals true. Now refresh. All right. It's gone. Beautiful. So let's create a new cart. All right. So this is our new cart object, our items array is empty and total price is zero. So let's go to this cart. And Deleted.

Okay. The card is deleted. So if we refresh this page, we get a 404 error. Beautiful. So that was our delete operation. Next, we're going to implement getting card items.

Getting Cart Items:

Alright, we're going to implement getting cart items. So for this cart, if we go to the items endpoint, we should see all items in this cart. So the difference between this endpoint and the previous one is that here we don't have cart specific properties. So we don't have the ID of the cart or the total price. We only have an array of cart items, okay? Now in this cart, we have an item with ID of three. If we add that here, we can retrieve that single cart item. So this is pretty easy. It's only going to take a few minutes of your time. Do this and then come back and see my implementation. Alright, back to our views module. Here we're going to create a new class.

A new view set. We're going to call that cart item view set. And have it extend model view set. Because here we support all operations. We can list all items in a cart. We can retrieve a single item. And we can also add, update, Delete an item. So we support all operations and it's much easier to inherit from model view set. Now here we set query set to cart item that objects, but we don't want to retrieve all cart items. We want to filter by cart ID. So instead of setting this attribute We're gonna override the get query set method. So we return cart item that objects that filter that cart ID should equal here we're going to extract cart id as a url parameter from self that keyword argument of cart underline pk.

We have done this before. Now, we set the serializer class attribute to cart item serializer. So we have built a view set. Now we need to register the route. So we go to our urls module. So here's our default router. Now because we have nested resources, we need to use a nested router here exactly like how we build products and reviews. So we're going to create a nested default router. So we say routers.nestedDefaultRouter. We pass the parent router, the parent prefix, which is cards, and the lookup parameter, which is card. So once we set this, we have a URL parameter called card.pk. That's how we extract it this url parameter right here. Okay? So we said look up to cart. And we get a new router.

We call that cards router. Now on this router, cards router, we register a new endpoint called items. And we map it to views dot cart item view set. Okay? And finally we set base name to cart dash items so we'll have two routes. One is called cart items list and the other is called cart items detail. Okay. Now that we have this router, finally, we need to include its URLs in the URL patterns list. So append carts router dot URLs. Good. Now let's test our implementation. So we have this cart. Let's go to the items endpoint. All right, we see all items in this cart, but we don't have other cart properties like cart ID and Total price. Okay. Now, what if we go to a single item?

So in this cart, we have an item with the ID of three. If you add that here, we see the cart item beautiful. But if you access an item that doesn't exist in this cart, let's say item 30. We get a 404 error. Beautiful. All right, we're almost done. Let's just make sure that we don't have any extra queries under this API. So let's look at the items on this cart. Okay. Now in Django debug toolbar, let's look at our SQL queries. So look, we have one query for loading cart items and three extra queries for reading the product referenced by each item. This is unnecessary. So here we can use eager loading. Back to our view set. Let's break this down into multiple lines.

So we add a backslash. So on this line, we filter and then we call select related. product. Okay, now, let's refresh. Now we get five queries. And if you pay close attention, you can see that we have an inner join between cart item and product tables. Beautiful. So we're done with this step. Next, we're going to talk about adding cart items.

Adding a Cart Item:

Alright, we're going to talk about adding a cart item. Now this one is a little bit more tricky, so I don't expect you to do it as an exercise. So just code along with me. So we're looking at the items for a particular shopping cart. Now down the bottom, look at the object that we have to pass to the server to add a product to this cart. So we have two properties here. Product, which is an object, and quantity. Now what you see here is generated based off of our serializer. The same serializer we have used return this result. But in this case, why do we have to pass a product object here? It doesn't really make sense. Because to add a product to a shopping cart, all we need is the product ID and quantity now here is one way to solve this problem.

We can go to our cart item serializer and mark the product field as read only. With that, back to the browser, let's refresh. So product is gone, now we only have quantity. Then, Back to our serializer, we can add product ID as another field here. But this is ugly. It's redundant. Also, when updating a cart item, we don't want to send the product ID either. We only want to send the quantity. So in that case, product ID is also going to be read-only, which is not possible because we either mark a field as read-only or not. And that applies to all scenarios. So what we need here is a different object for adding an item to a shopping cart. so we need a different serializer here.

So let's delete this ugly product id from here and remove this as well. Instead, we're going to create a new serializer. So down the bottom, I'm going to create a class called add cart item serializer this is going to be a model serializer. We define our meta class. We set the model to cart item and fields to ID, because we're going to return that in the response. And product ID. And quantity. Now, back to our view. In our cart item view set, we don't want to hard code cart item serializer. We want to dynamically return a serializer class depending on the request method. So, we're going to override get serializer class. Here we say if self that request that method equals post all in uppercase, then we're going to return add cart item serializer.

Otherwise, we're going to return cart item serializer. Okay, let's see what happens after this point. So back to our API, let's refresh, we get the same result as before. So now we don't have that product object in this box. But where is product ID? Well, even though our cart item model has a product ID attribute, This attribute is generated dynamically at runtime. It's not a field we can reference here. So we have to explicitly define this field. We have to say product ID is an integer field. So sterilizers dot integer field. Now let's refresh one more time. So this is the object we should send to the server. Beautiful. Now let's implement the saving part. This one is a little bit tricky because when we add the same product to the same cart multiple times,

We don't want to create multiple cart item records. We want to update the quantity of an existing item. So in this serializer, we cannot rely on the default implementation of the save method that comes from model serializer. We have to reimplement the save method based on the requirements of our application. So let's override the save method. Now here we need to get the product ID and quantity. Where do we get that? Well, earlier I told you that behind the scene, there is a call to serializer dot is valid when the data gets validated, then we can get it from an attribute called validated data which is a dictionary. So currently we are inside our serializer class, so we say self that validated data that's our dictionary.

From here, we can read the product ID that we received from the client and store it here. And similarly, we can read the quantity. Okay? Now what about the cart ID? Cart ID is not in the request. It's available in the URL. But in the serializer, we don't have access to URL parameters. So just like before, we have to go back to our view, get the URL parameter, and using a context object, pass it to the serializer. So back to our view. Here we should overwrite get serializer context and return a dictionary with one key value pair, cart ID. And we set that to self.keywordArguments of cartUnderlinePK. Okay? Now, back to our serializer. We can read the cartId from self.context of cartUnderlineId. Great.

Now the saving logic. So we're going to start with cartItem.objects and get a cartItem with two attributes. Its cartId should equal our cartId and the product id should equal our product ID. So we get a cart item. Now, if there is no such a card item, this line is going to throw an exception. So we need to wrap this inside a try catch block so we say try. Let me bring this down, okay? Except cart item does not exist. So if we get here, that means we have to create a new item. Otherwise, if we are here, that means we're updating an existing item. Okay? So what are we going to do here? We're going to set cartItem.quantity plus equal the quantity that we read earlier.

Then we save the item. Now, what if this item doesn't exist? Well, here we're going to call cartItem.objects.create and give it the cart ID as well as the product ID and quantity. Now here we can say Product ID equals product ID and quantity equals quantity, but this is a little bit redundant So we can unpack self that validated data dictionary. This is simpler So we create an object now the tricky part. Let's look at the default implementation of the same method. So let's look at the code for model serializer Now in this module hold down shift command and O on Mac or shift control and O and Windows. With this we can find the symbols defined in this module. So let's look at our save method.

We have two implementations. One in base serializer, the other in list serializer. Base serializer is the parent of model serializer. So let's look at this one. Now don't worry about all these details. I just want to highlight something very small. So first we have a bunch of assertions to validate the data. Next we have our validated data dictionary. Here we have this logic saying if self that instance is not known then we're going to update some record. Otherwise, we're going to create a new record. Now, either way, we're setting self that instance So the object that is updated should be returned from this method and stored in this attribute. Now similarly, the object that is created should be returned here and stored in self that instance and finally self that instance is returned and

from the save method. So we have to follow the same pattern so all these building blocks can talk to each other properly. So back to our serializer. In our try catch block, here we're saving our cart item so we say self.instance equals cart item. Now if we create a new cart item, we set self.instance to the object that is returned from the create method. And finally, at the end, we return self.instance. Let's test our implementation. So I'm going to start with a brand new cart. So we go to our cart list and create a new cart. Great. Now, let's go to the items for this cart. And down here, set product ID to one and quantity to 10. Let's post this. Okay, you got a new cart item with this ID beautiful.

Now what if we post the same product, but a different quality or it can be the same. Now, post. Here's the new quantity, 20. So we don't have duplicate records in our database. So if you refresh this page, in this cart, we only have a single item with this quantity. Beautiful. We're almost done. We have tested the valid values. Now we should test invalid values. What if we pass an invalid product? Product 0 What do you think is going to happen? Our application blows up. So we need to prevent this and return a meaningful error to the client. Earlier in the course, we talked about data validation in serializers. So we can either validate the entire object that is passed here or individual fields.

To validate individual fields, we have to follow a specific convention. We define a method called validate, underline, followed by the field name. In this case, product ID. Now here we're gonna have two parameters, self and the value we are validating that is a product ID. So here we can write some logic and say, if not product that objects that filter You're gonna set the PK to this value. That's the product ID. If this doesn't exist, then you're gonna raise serializers that validation error and Here we can say, no product with the given ID, was found. Otherwise, we return the value. This is the same pattern we followed for validating the entire object. So we either raise a validation error or return the valid value.

Now, let's test this. So refresh. One more time, let's pass product zero with quantity one Post. Okay. Now we have a validation error associated with this field. Beautiful. Now what about quantity? What if we pass an invalid quantity like zero or negative one? Well, zero worked, which is not a big deal. But what if we pass negative one here, we're getting this error saying, ensure the value is greater than or equal to zero. Now, where is this coming from, from the definition of our model, because in our serializer, we're simply referencing the quantity field of our cart item model. So let's look at our model. We have defined quantity as a positive small integer field. That's why it doesn't accept negative values. But if you want to take this to the next level and ensure quantity is always greater than zero, here we can set validators To a list of validator objects.

So here I'm going to use min value of validator We talked about this in the first part of the course. So we can pass a validator object here and set the minimum value to 1 Now, refresh down the bottom and Let's set product ID to one and quantity to zero. Okay, we got a different error saying the quantity should be at least one. Beautiful. So we're done with this step. Next, we're going to talk about updating a cart item.

Updating a Cart Item:

Alright, with every step, our API is evolving and getting better and better. So now let's implement updating a cart item. So we're looking at this particular item in this particular cart. Now down below, we can see the object that we have to pass to the server to update a cart item. But once again, we don't want to pass this entire object, it doesn't really make sense. We only want to update the quantity. So we're going to use the same technique we used in the previous lesson. And that's your exercise for this lesson. So create a custom serializer for updating a cart item. It's super easy. It's only going to take a couple of minutes of your time. All right, here's my implementation. So we're going to define a new class called update cart item serializer, which is a model serializer.

Here we define a meta class, set the model to cart item, and fields to only quantity. Now, back to our view set. Over here, where we determine the serializer class, we need to add another if statement and say, if request that method equals patch, then we're going to return update cart item serializer. So we're not going to allow put requests here because we only want to update a single property of a cart item object. But how can we prevent put requests? Super easy. Here we have an attribute called http method names. we can set it to the list of methods that we allow at this endpoint. So we're going to allow get, post, patch, and delete. Now, back to the browser. Note that when i refresh, the put button is going to disappear.

So take a look. Look, now we only have patch, and this is the object we're going to send to the server. So currently the quantity is 20. Let's change that to 10 and patch it. Here's the updated quantity. Let's refresh this page. And here is the updated quantity as well. Beautiful. The final step is deleting a cart item.

Deleting a Cart Item:

Alright, now the final step, we want to delete a cart item. As you can see, this functionality is already implemented for us. Because our cart item view set extends the model view set class. And as you have learned, this class provides all operations. And also over here, you're allowing delete requests. So there's really nothing we have to do here, we just want to test this out and make sure it really works. But before we do that, I want to point out that these method names here have to be in lowercase. So if I change this to a capital delete, look, refresh, it disappears. So hopefully you didn't have any problems in the previous lesson. Now, let's test this functionality. So I'm going to delete this.

Okay, we've got a 204 response. Beautiful. Now if you refresh this page, we get a 404 error. So this resource is deleted from the server.

Django Authentication System:

Introduction:

Welcome back to another section of the ultimate Django course. In this section, you will learn about Django's authentication system for identifying users. So we'll start off by talking about how the authentication system works under the hood, so you understand all its building blocks. We'll then talk about the user model and how to customize it to fit the requirements of your application. So as part of this, we'll look at extending the user model and creating profiles. Next, we'll talk about permissions and groups for controlling users access. This is a short but exciting section, so let's jump in and get started.

Django Authentication System:

So every Django application comes with a full-featured and flexible authentication system. We briefly talked about this before, but let's dive in and take a closer look at this authentication system. So let's jump to the list of installed apps. Now look, every Django project contains this app, django.contrib.auth. This is the authentication system. So using the system, we can identify users, we can allow them to log in, log out, change their password, and so on. So in this app, we have a bunch of models like user, group, and permission. And obviously we have tables for persisting these models. So back to our database, look, these are the tables for the auth app. So we have group, permission, user, and their combinations. Now as an example, let's look at the user table.

So in this table, we have the user ID for each user. We have the password, which is stored in the encrypted format. We have the last login date and time. is supervisor tells us if this user has all privileges. So if you set this to one, this user can do everything in this application okay you also have the username, first name, last name, and email that is staff tells us if this user can log into the admin area. So we can create additional users and give them access to the admin area. But we can control the level of access they have to this area. We'll talk about that later in this section. We also have is active and date joined pretty straightforward. Now we can easily customize this table and add additional columns.

And that's what we're going to talk about in the next lesson. But before we do so, let's go back to the settings module. In this module, we have another section called middleware. And middleware is a function that takes a request and either passes that request to the next middleware or returns a response. So you know that in Django, when we receive a request, at some point, that request is going to be passed to a view. Now, during this time, Django is going to run that request through these middleware functions in order. Now, each function can take the request and add something to it, or it can return a response. If it returns a response, the next middleware function is not executed, okay? Now, why am I saying this?

Because here we have a middleware function called authentication middleware. The job of this middleware is to read the user information from the request and set the user attribute on the request object. Let me show you what I mean. So let's go to one of our views. So here in the collection view set in the delete method, this request object at runtime is going to have an attribute called user that this is either set to an instance of the anonymous user class, or an actual user object. And this is the job of the authentication middleware. So this middleware reads the user information from the request and sets this attribute, okay. So this is the basics of the authentication system. Next, we're going to talk about a couple of strategies for customizing the user model.

Customizing the User Model:

So the user table in Django authentication system is pretty good. But sometimes we need to store additional data about a user. And here we have two options. One option is to use inheritance to extend the user model. So we can create another model called app user. And this model should extend the user model in Django. The other option is to create a profile. So we can create a profile model. And in this model, we'll add a one to one link to the user model. So in this scenario, we're not using inheritance, we're using composition. So the profile model is composed of a user model. Okay. Now, what are the practical differences? And when should we use which approach? Well, in terms of the database, with the first approach, we'll end up extending the user table.

So any extra attributes that we add in our custom user model will end up in this table. With the second approach, we're not going to extend this table, we'll have a separate table and And that table will have a foreign key to the user table. So practically speaking, we should use the first approach only for storing attributes related to authentication. Anything that impacts the authentication process. Anything extra that is not related to authentication should really go in a profile table. Examples are the user's birth date, address, and so on. So with this approach, we allow each app to have a different concept of a user's profile. For example, in the sales app, The customer model represents the user's profile, whereas in the HR app, the employee model represents the user's profile.

And similarly, in the training app, we can have a model called student, which is another user profile. So each app can have a different concept of a user's profile. Also, take into account that we cannot use the first approach in the middle of a project. It's a little bit tricky, and I'm going to talk about that in the next lesson. So most of the time, we'll use the second approach to customize the user model. So that's the theory. Over the next two lessons, we'll explore these two approaches in detail.

Extending the User Model:

Alright, let's talk about extending the user model. So back to our database, look at the user table. If you pay close attention, you can see that here we have a unique constraint on the username column. So we cannot have multiple users with the same username. Great. But what about email? Currently, there is no unique constraint on the email column. So we can have multiple users with the same email. Now let's imagine that in the future, we want to allow our users to log in with their email. So here we need to apply a unique constraint the email column. And this is a valid use case for extending the user model because we are changing something related to authentication. Okay? So, back to our project.

We need to create a new user model. But where should we add that model? We don't want to add it in the store app. Because this app is for building an online store. It has nothing to do with authentication. What we're trying to solve here is something very specific to this project. Now look at this other app. Store Custom. We built this in the first part of the course. And here we combine features from different apps. So the code that we write here is very specific to this project. And this is the right place to add our new custom user model. But before doing so, I want to change the name of this app to something more meaningful. Let's rename this to core.

Because this is essentially the core of this project. Okay. Now, we need to go to the apps module and rename this class to core config. So press f2 to rename this class properly. Okay. Next, we need to change the name here to core. And then we need to go in the list of installed apps and change store custom to core. And I prefer to add this after our reusable apps. So first, we have all these reusable apps followed by the core of this project. Okay. Now, back to this app in the models module, Here, we need to create a new class called user. And this should extend the abstract user in the authentication system. So on the top, from Django, that country, that auth that models, we should import the abstract user class.

In the previous lesson, I made a mistake, I said, we should extend the user class, but technically, we should extend the abstract user class. Okay. So we extend this class. And here we redefine the email field. So we say email is models that email field. And here we apply a unique constraint. Beautiful. Now we need to tell Django that we're going to use this class instead of the user class in the authentication system. So we go to the settings module. And here we need to define a new setting. We can add it anywhere in this file, it doesn't really matter. Here I'm going to add it at the end. The setting is us underline user underline model. make sure to spell it properly, otherwise things are not going to work.

So we set this to core dot user now here in the terminal, we get an error related to this class, liked item. The error is saying field defines a relation with the model auth.user, which has been swapped out. So we are seeing this error because we swapped out the built-in user model in the auth app with our custom user model. So let's jump to the liked item class. look over here. We have a foreign key to the user model defined in auth.models. So one way to solve this problem is by using our custom user model. So we can say from core.models import user. But there is a problem with this approach. The problem is that now this app, the likes app, is dependent on the user model in the core app.

So it's no longer an independent reusable app. We can no longer distribute it without distributing the core app. But the core app is not supposed to be reusable. It implements features very specific to this project. So we don't want to explicitly import this user class here. Instead, we're going to go to django.conf and import the settings module. Now from the settings module, we can read the auth user model. So we say settings.auth, underline user, underline model. Again, make sure to spell it properly. So back to the terminal. Now we get a different error saying dependency on app with no migrations. You're seeing this error because our light app is indirectly dependent on the core app. Because here we have a new custom user model.

But currently, there's no migration to create a table for this model. So let's open a new terminal window and say Python manager pi make migrations. Okay, We have a new migration, now let's run it. So python manage.py migrate. Now we get a different error saying migration admin initial is applied before its dependency core initial on database default. So the reason this is happening is because we decided to swap out the user model in the middle of the project. So this migration, the first migration in the admin module is dependent on the user model. And later on, we decided to change that user model to a different user model defined in this migration. So this is why we cannot easily extend and swap out the user model in the middle of the project.

So as a best practice, you should always create a custom user model at the beginning of your project, even if there is no requirement to change the authentication flow in the future. So you just create an empty class using the pass keyword, and this will ensure that later on, if you want to replace this class, you're not going to have any problems, okay? but what can we do now? Well, let's bring this back. The only way we can solve this problem is by dropping and recreating our database. So that's like the nuclear bomb solution. We don't want to do that. But in this case, we have no other choices. So back to data grip, let's open a new query console and say, drop database storefront to semicolon.

And here we add another statement, create database storefront to. Now we select both of them and execute in one go. Okay, now back to the terminal. Let's migrate our database one more time. Okay, this time all migrations were applied successfully. Now back to the first terminal window. Our web server is running beautiful, But there are a couple of issues we need to fix in the admin panel. But we cannot log into the admin panel now because we created our database from scratch. So we need to create a new super user. Back to the terminal and let's run Python, manage.py, create super user. So this is admin, and i'm going to give it some fake email. It's on password. Okay, now let's log into the admin.

So here on the admin panel, look, we no longer have the users page under the authentication app. So in the core app, we need to go to the admin module and register an admin model for managing our users. So we define a new class called userAdmin, and this should extend the userAdmin in the auth app so we get all functionality already implemented for us. So on the top, from django.contrib.auth.admin, we're going to import userAdmin. But because we have a name clash here, we can give this an alias. We can rename it to baseUserAdmin. Now we have this class extend, base user admin. And For now, let's just add the past keyword. We don't want to customize it yet. All we want to do is register this with the user model.

Now, which user model? The one that we defined in this app. So, from the models module in the current folder, we're going to import the user model. Now, back in the admin, refresh. So here's the core app where we can access all our users. Beautiful. Now, let's add a new user. So currently, we have username, password and confirm password. But we also need to capture email because we applied a unique constraint on the email field. And if we don't supply value for the email, we'll end up with an empty string in the database. So next time we create another user, that user will also have a blank email and we get a duplicate record error. So back to our user admin, let's look at the implementation of base user admin.

So in this class, we have an attribute called add field set. These are the fields that we see when registering a new user. So we have username and two passwords. So let's grab this code and add it to our new admin model class. Now in the list of fields, we also want to include email. And I want to include first and last names, even though this is not compulsory. But by capturing them here, you won't end up with blank values in the first and last name columns. so let's add first name and last name. Now the final step. Refresh. So we have email, and as you can see, all these fields displayed in bold are required, but first and last names are optional.

Beautiful. So let's create a new user called john smith and give it a password. One more time. John Smith at domain.com and john Smith. Save. Okay, we successfully registered a new user. Now back in the database, look, the user table is no longer part of the auth app. It's now part of the core app. And in this table, we have this new user that represents John Smith. So let's quickly recap. To extend the user model, first we create a new model that extends abstract user. Then in the settings module, we set auth user model to our custom user model. And from this point onward, we never reference the user model directly. Instead, we use settings.authUserModel.

Creating User Profiles:

Alright, let's talk about user profiles. So this customer model we have defined here is essentially the profile of a user in the sales app. The only thing that is missing is the link to the user model. So let's define a new field here. User, this is going to be a one to one field. But here we don't want to reference the user model in Django or in the core app, because the sales app is supposed to be a reusable app. So we should be able to reuse it in any project no matter what model is used for representing users. In other words, we don't want to link this model to the built-in user model in Django or a custom user model. We want to link this to the auth user model of the containing project.

So we need to import the settings module and reference auth user model. So on the top, from django.conf or configuration, we're going to import the settings module. Now back to where we were, Here, we're going to reference settings that auth user model. So by default, this setting is set to the user model in Django. But in this project, we changed it to the user model in the core app. Now for on delete, we're going to enable cascading. So if we delete a user, the associated customer record is also deleted automatically. So models that cascade. Now, in this model, we have a bunch of redundant fields. So we have first name, last name, and email. All these fields exist in the user model. So we need to delete them from here.

Now we need to make a few more changes. Because we removed the first and last name fields, here in the stir method, we need to reference the user object. And also, when ordering customers, you want to use user double underscore first name and last name. Okay, save. Back in the terminal, we have a bunch of errors in customer admin model. The first error is saying the value of ordering refers to first name, which is not an attribute of customer. So let's go to the customer admin model. All right, look, here's the ordering attribute. So once again, we need to reference the user field. And for this reason, when loading customers, you want to eager load them with their users Otherwise, for each customer, a separate query is going to be sent to the database.

So we're going to set list select related to user. Okay. Now, back in the terminal, we have two more errors. This one is complaining about the list display attribute. So take a look. Here's list display. But in this case, we cannot use the user field. This syntax is not supported at the time of recording this video. So the workaround is to define a method called first name. in the customer model, and in that method will return user that first name so back to our model, here we're going to define a new method first name and in this method we return self that user that first name and similarly we need another method for the last name last name okay good now back in the terminal, we don't have any more errors, so now let's create a migration to update the database.

So, a new terminal window, Python, manage.py, make migrations. Alright, we get a warning saying you're trying to add a non-nullable field user to customer without a default. Why are we getting this? Well, back to our database, look at the customer table. We're trying to add a new column to this table, and this column doesn't accept null values. So what's going to happen to existing customers? That's why we need to supply a default value. Now here we have two choices. We can provide a one of default value, or we can quit and add the default value in our model. We don't want to hard code a default value in our model, because otherwise, every time we create a new customer, that customer will be associated with a specific user.

We don't want that. So here I'm going to select the first option and supply a one of default value, I'm going to use one, which is the idea of the admin user. So let's go ahead with that. All right, here's our migration. So let's apply it to our database. Python, manage.py, migrate. Beautiful. Now back to our database. Let's refresh and look at the customer table. So now we have the user ID, which is a foreign key to the user table. And also first name, last name, and email columns are removed from this table. Beautiful. Now back in the admin, let's make sure everything works out to this point. So you go to the customers page. Currently, we don't have a customer. So let's add a new customer.

I'm going to supply some value for the phone and membership. Now here we have a drop down list for our users. So we can associate a customer with an existing user. Now in a real world scenario, we want to replace this drop down list with an auto complete box, but I'm going to leave it up to you as an additional exercise. So let's associate this with admin user. Good. Now currently, we don't have the first name and last name for the admin. So let's add another customer and associate this with John Smith. We know that John Smith has a first and a last name. So save. And here we can see the first and last name of this customer coming from the associated user model.

Beautiful. But there is a problem here. Look, we cannot sort this table by first or last names. We can only sort it by membership and orders. To fix this problem, back to our customer model, we need to apply the admin.displayDecorator here. So on the top, from django.contrib, let's import the admin module. Now, back to where we were, here we're going to use the display decorator, and set the field we want to use for sorting. So ordering equals user double underscore first name. And one more time, for the last name method. Alright, now, back to the admin refresh and now we can sort by first and last name beautiful so let's quickly recap to define a user profile we simply create a profile model in this case customer and in the profile model we add a one-to-one field with the user model or more accurately with the auth user model of the containing project this makes our apps more reusable so they're not coupled to a specific user model implementation.

Groups and Permissions:

Alright, now let's talk about groups and permissions. A group is a collection of permissions. So instead of assigning users a bunch of permissions on an ad hoc basis, we can add them to one or more groups and each group can contain a bunch of permissions. So in this application, currently we don't have any groups. So let's add a new group called customer service. Now in this box, we can see all permissions available in our application. So let's filter by customer. Now look, in the store app, for the customer model, we have these permissions. We can add a customer, change it, delete it, and view it. So every time we create a model and migrate our database, Django automatically creates these permissions for us.

Let me show you where they are stored. So back to the database, look in the auth app, we have this permission table. Let's take a look here. So down the bottom, look at the last record. Can view user. So because we created a custom user model, Django created this permission for us. Now this permission has a unique identifier, a name, a descriptive name. It also has a code name. This is also a unique identifier, but it's a string that we can reference in the code. Now here, you can see the content type ID column. Remember content type ID? So in this database, we have a table called Django content type, which specifies all models in our application. For example, in the core app,

we have a model called user. And this is the idea of this model 19. So over here, this permission is associated with this model. Okay. Now back to the admin. So let's select all these permissions, and add them over here. Now for customer service, we also want to assign permissions to manage orders. So this filter by order. Again, we have permissions for adding, changing, deleting and viewing orders, as well as order items. so let's select all of these and add them then save now let's add john smith to the customer service group so back to the home page we go to the users here's john smith now first we need to mark him as a staff member so we can log into the admin area now over here we can see all available groups so i'm going to select customer service now optionally we can assign john

bunch of permissions explicitly. I don't want to do that, so let's go ahead and save the changes. Now I'm going to log out and log back in as John Smith. Look, now John has access to the admin panel, but he can only manage customers and orders. So this is how groups and permissions work. In the next lesson, I'm going to show you how to create custom permissions.

Creating Custom Permissions:

the last thing we're going to talk about in this section is creating custom permissions. But why do we need to create custom permissions? Well, sometimes we have operations that are not quite about creating, updating or deleting data. For example, think of canceling an order, it's a special kind of update. So by canceling an order, we don't want to delete it, we want to change the status to canceled. Now let's say we want to give some users the ability to cancel an order but not updated. This is where we need custom permissions. And they're super easy to create. So let's go to the order class. To create a custom permission, here we need to add a meta class so class meta here we set permissions to a list of tuples.

Each tuple represents a permission. So here's one tuple. Here we need to add two values. The first one is the code name. Cancel order. So this is a unique identifier that we use in code. The other is a description can cancel order. Okay. Now that we have modified our model, we need to create a migration and run it. So here in the terminal Python managed by make migrations. Beautiful. And finally, Python managed by migrate. Lovely. Now, back to the database. Let's open the permission table. Look at the last record can cancel order. Beautiful. So now we can log into the admin. and give someone this particular permission. So I'm currently logged in as John Smith. Let's log out and log back in as admin.

All right. Now let's change the permissions for John Smith on this page. Over here. Let's search for cancel order. There you go. So we can assign this particular permission to John Smith. But what about the act of canceling an order? Well, that's the topic for the next section. So in the next section, we're going to apply these permissions to our API endpoints to make them secure. So if you're not here, I will see you in the next section.

Securing APIs:

Introduction:

Welcome back to another section of the ultimate Django course. In this section, you're going to learn how to secure your API endpoints using permissions. So we'll start off by talking about token-based authentication, which is the de facto standard for authenticating users with RESTful APIs. Then we'll add authentication endpoints to our API and allow users to register, log in, log out, and so on. And finally, we'll apply a bunch of permissions to some of our API endpoints so they are not accessible by just anyone. So now let's jump in and get started.

Token-based Authentication:

So as I said earlier, token-based authentication is the de facto standard for authenticating users with RESTful APIs. And here's how it works. Let's say a new user is going to use our application. So first they need to register. So on their machine, the client app is going to send a request to the user's endpoint. Here on the server, we're going to capture their username, password, name, email, and other fields, and create a new account for them. Next, the user needs to log in. So the client app needs to send a request to the authentication endpoint, and it should pass the user's credentials, meaning username and password, to the server. Now, on the server, we're going to validate the user's credentials. If they're not valid, we're going to return an error.

Otherwise, we're going to return a token. This token is like a temporary key we're going to give to the client to access protected resources. So the client is going to store this locally, and next time it needs to access protected resources, it's going to send it to the server. for example, let's say the user wants to look at their profile. So the client needs to send a request to this endpoint and it should pass the token in the request header. Now, once again, on the server, we're going to read the token. We're going to validate it. If it's valid and not expired, we're going to give access to this resource. Otherwise, we're going to return an error. This is the big picture. Now, over the next few lessons, you're going to see this process in action.

Adding the Authentication Endpoints:

So you know that Django comes with a full-fledged authentication system, but this system doesn't include an API layer. So we don't have any endpoints for users to register, login, and so on. We only have a bunch of models and database tables. Now we can build this API layer by hand, but that's pretty tedious and repetitive. We don't want to repeat it in every project. So this is where we use a fantastic library called Joeser. Joeser is the RESTful implementation of Django authentication system. It provides a bunch of views for user registration, login, logout, password reset, and so on. So head over to josir.readthedocs.io. Now let's go to getting started. On this page, you can see all endpoints that Josir is going to add to our backend.

So we have an endpoint for managing users. We have another endpoint for getting the current user and so on. Now let's go to the installation page. So I'm going to show you the installation steps in this video. But because this is a third party library, I don't know what's going to happen to it. In the next version, the installation steps might be different. So always refer to their latest documentation for the most up to date installation instructions. So first we need to install Joeser. So back to the terminal, pipenv install Joeser. All right, good. Now, there are two more libraries we need to install. For now, I'm going to skip them. We'll get back here later. Now for configuration, we need to add Joeser and the list of installed apps.

So let's go to the list of install apps and add Joeser right here. I'm going to add it after the rest framework. So Joeser. So these are all third party libraries. And then we have the apps in our application. Next, we need to register a URL pattern in the URLs module. So we're going to go to the main or the root URLs module in the storefront folder. So this is where we're including routes from different apps. Over here, we're going to add another pattern for the authentication endpoints. We're going to delegate all these requests to josser.urls. Now, josser relies on an authentication backend or an authentication engine to do the actual authentication because josser is just an API layer, a bunch of views, serializers, and routes.

So we need an authentication engine to do the actual work. And here we have two choices. We can use token-based authentication built in Django REST framework or JSON web token authentication implemented in a separate library. Now, what is the difference? Well, token-based authentication built in Django REST framework uses a database table to store tokens. So every time we receive a request on a server to validate the token, this backend is going to go to the database to make sure this is a valid token. so that's going to incur a database call for every request. In contrast, JSON web token authentication doesn't need a database. The way these tokens are structured completely eliminates a database call because every token has a digital signature and on the server, we can use that signature to ensure this is a valid token.

So back to the documentation, let's go to the authentication backends page. Now here we have instructions for token based authentication that comes with rest framework. as well as instructions for json web token authentication. So, let's follow these steps. To use this backend, first we need to install a separate library that is not included on this page. So, back to the installation page under getting Started. Look at this library over here. Django REST Framework Simple JWT, which is short for json web Token. So, first we need to install this. So, pipenv install this. All right, good. Now let's go to the authentication backends one more time and look at the instructions for JSON web token authentication. So here in the REST framework setting, we need to add this setting, default authentication classes.

So copy. Now let's go to the REST framework setting and add that setting right here. So we are using this class, JWT authentication. as our authentication engine. Next, we need to add a setting that is specific to this library we just installed. Simple JWT. So let's copy this and paste it somewhere here. With this setting, we're specifying the prefix that should be included in the request header. So for sending the authentication token to the server, we're going to prefix the token with JWT. You'll see that shortly. So back to the documentation. I believe there's one more step. We need to include one more URL pattern in the main URLs module. So let's go to the URLs module. I'm going to duplicate this line and include joser.urls.jwt.

Now here in the terminal, we have a weird error saying ellipsis object has no attribute rsplit. I don't know what's going on, so I'm going to pause the video and troubleshoot this. All right, this is what happened. When copying this code from their documentation, I forgot to leave this out. So this is the danger of copy pasting code. So delete. Now, back in the terminal, no errors, beautiful. So under documentation, you saw that Joeser adds these endpoints. But all these endpoints are prefixed with auth. So back to our project, let's go to auth slash users. So this endpoint is working. But currently, we cannot access it, because we get a 401 error, which means unauthorized, because this is a protected endpoint. Not everyone can access this endpoint.

It's not open to anonymous users. So to access this endpoint, we need to pass a JSON web token in the request header, and we'll do that soon. So now that we have set up our authentication endpoint, next, we're going to see how we can register new users.

Registering Users:

All right, let's see how we can register a new user. So we have the users endpoint for managing users. And as you saw in the previous lesson, we cannot send a get request to this endpoint without authenticating first. But look at the allow header. Here we also support post requests, and post requests are open to anonymous users so they can register. So to register a new user, the client app should send a post request to this endpoint. Now down below look, We have a form for registering a new user. So I'm going to set the email to user one at domain.com. Give it a username and a simple password. Now take a look. All right, we get a bunch of errors for our password being too short, too common, and entirely numeric.

Where are these coming from? Well, let's go to our settings module. Here we have a setting called auth password validators. Here we have a bunch of validators. So we have user attributes similarity validator. This ensures that your password is not the same as your username. We have another validator that checks the minimum length. We have another validator to ensure the password is not a common phrase. And one more validator that checks to see if your password is purely numeric. So these are various validators available to us and we can always customize them. So back to this form. Let's create a new user one more time. So user1 at domain.com user1 and for the password I'm going to say I love Django. Now post.

Great. So we have a new user with this ID. But what about the first and last names? What if you want to capture them as part of registration? Well, you know that the content of this box is based off of a serializer. So Joeser has a serializer for deserializing this data. So if you want to capture additional fields, we need a custom serializer. So let's go to Joeser's documentation. On the top, search for serializers. On the settings page. So look, here we have a dictionary of various serializers used in Joeser. For example, for creating a new user, this is the default serializer. But using the settings module, we can easily replace this with a custom serializer. So back to our project. And the first question is, where are we going to add this serializer, we don't add it in the store app, for the same reason that we didn't want to customize the user model in this app.

Because this requirement is very specific to this project, how users should register on this website. So we're going to add this in the core app. So let's add a new file called serializers.py. Now, from Joeser, The serializers, we're going to import user create serializer. Next, we need to create a class called user create serializer. But because we have a name clash, I'm going to give this class an alias. So as base user create serializer. And of course, we could call it anything. Now, our custom serializer should extend this serializer. So we get all that functionality and add something extra to it. So let's look at the implementation of the serializer. So we have a bunch of code, look at the definition of the meta class.

Here we have the fields attribute set to this expression. The result of this expression is a tuple of three values, username, password, and email. These are the fields that we saw in this box. So back to our serializer, let's create a meta class here. Now we want this meta class to inherit everything in a meta class, in this class. So, base user create serializer dot meta. The only thing we want to overwrite is the fields attribute. So I'm going to set this to a list or we can also use a tuple. I'm going to include id so we can return it from the server. Username, password, email, first name, and last name. Okay? Now technically we could avoid extending this base class and also set the user attribute here.

But this is a better practice to inherit everything in the base class and overwrite specific parts. Because in the next version, the implementation of the middle class might change. So we want to inherit all that functionality and change some bits, okay? So now we have a serializer. Next, we need to register it in the settings module. So let's go to the settings module. Down the bottom, let's create a new setting. That's Joeser. We set it to a dictionary here. We add a key called serializers Again, make sure to spell it properly so in this dictionary, we're gonna add user underline create and by the way, I Got that from the documentation. So here we need to add a key value pair The key is user create and the value is the path to our custom serializer.

So we set user create to core that serializers dot user create serializer. Okay, let's test our implementation. So back to this page, refresh. Now take a look. So we have all these fields here. Beautiful. Let's create another user user two. With I love Django as the password. User two at domain.com. And it's going to be Joe Smith. Post. Lovely. We have user number four. You also see the username, email, first name and last name here. Now, back to our serializer. What if we want to include some profile data here as well? Let's say, birth date. So birth date is not part of the user table, it's part of the profile table or the customer table. So here's the thing, if you add that field here, birth date, because this is not a field in the user model, we have to explicitly define it here.

So we can say, birth date is a date field. So on the top, From rest framework, we import serializers. And here we say birth date is serializers dot date field. And we can make this optional. Now while reviewing this video, I realized I made a mistake. I shouldn't have set this field to read only because we want to capture it from the client. So I don't know what I was thinking. But that was a mistake. And I'm sorry about that. Now, to save this, we need to override the save method and change how this data is saved. So first, we create a user record, and then we create a profile record. But this is not the right way to implement this, because this serializer is purely responsible for deserializing user data and creating a user record, nothing more.

Sure, we can give it extra fields to save, but this is like a chef who also comes out of the kitchen to take orders. It's certainly doable, but is this the right way to manage a restaurant? No. In a proper restaurant, every person, every role should do one thing and do it well. So the chef is purely responsible for cooking, whereas the server is purely responsible for taking orders. We want to build software like this. So our software components should have a single responsibility. When you start mixing responsibilities, that's when your code starts to get ugly, it becomes spaghetti. Everything will be dependent on everything else and it will be very hard or impossible to make changes without breaking a ton of things.

It has nothing to do with object-oriented programming or a specific framework or language. It's how you've written your code. If you pay attention to these little details and properly separate responsibilities in your software, trust me, the end result is always going to look beautiful and easy to maintain. So what should we do here? Am I saying that we should not capture user's birth date on the registration form? No, I'm not saying that. So on the client, on the front end, we're going to have a registration form and that form might include several fields. Some of these belong to the user model. The others belong to the profile model like the customer model. Now, to save all these fields, the client should first send a request to the user's endpoint to create a user account, and then it should send a separate request to the profile endpoint to store those additional profile-related fields.

Yes, we're going to have two separate calls to the server as opposed to one call, but our endpoints are properly separated. And when we properly separate these endpoints, they become more reusable and easier to maintain. We don't want these endpoints to step on each other's toes. For the same reason, We don't want people working in our organization stepping on each other's toes. So everyone, every endpoint should have one and only one responsibility. So I'm going to remove the birthdate field from here and make this class simpler. Good. Next, we're going to talk about implementing the profile API.

Building the Profile API:

Now look at the endpoints provided by Joeser. As you can see, all these endpoints are for managing users and authentication. So these last few are only for authentication. So Joeser doesn't have any endpoints for user profiles. And that makes perfect sense because profiles are specific to apps. So Joeser doesn't know we have an app called sales. And in that app, we have a profile called customer. All is concerned about is the authentication of users, nothing more. So we need to build the customer API or the profile API ourselves, and that's super easy. So let's do that real quick. Back to VS Code. Now here's a question. Where should we implement that API? In the store app. Because this is where we have defined the concept of a customer.

So the API belongs to this app as well. So in this app, we start with a serializer. So we define a new serializer. Customer serializer. extends model serializer. Here we define a meta class. We set the model to customer and fields to. What fields do we need here? Well, let's have a quick look at the customer model. So we have phone, birthdate, membership, and user. So back to our serializer, I'm going to include ID of the customer, user ID, phone, birthdate, and membership. So our serializer is ready. Next we need a view set. So let's go to the views module in the same app and create a new view set. So customer view set. Now this can be a model view set. But model view set is not a good choice here because we don't want to support all operations at this endpoint.

For example, we don't want to list customers. That is something that we only need on the admin panel. So a web or a mobile app talking to our API doesn't really need a list of customers. So what operations do we need here? We should be able to create a customer, retrieve a customer and update it. Also, we don't want to delete a customer because when we delete a user, the customer record gets deleted automatically. So here I'm going to create a custom view set using a bunch of mixins. Create model mixin, retrieve model mixin and update model mixin. and on top of all these, we're going to add generic view set. Like this. So, here we're going to set query set to customer.objects.all and serializer class to customer serializer.

Pretty simple. Now the last step. We need to register the route. So, let's go to the urls module of the store app. And here we're going to add one more endpoint called customers and we're going to map it to customer view set. Now, back to the browser. Let's go to the store app. So here we have a new endpoint for managing our customers. Now, as you can see, the get method is not allowed because we don't want to allow the client to retrieve the list of all customers, okay? But we can create a new customer. Now here we have phone, birthdate, and membership, but user id is missing. So, back to our serializer. Even though in the customer model we have the user id attribute,

This attribute is created dynamically at runtime. So we need to explicitly define it here. So user ID equals serializers dot integer field. Okay, now, let's refresh. So here we have the user ID, but technically, we don't need this field here. Because later in this section, we're going to protect this API endpoint. So only authenticated users can call this endpoint. So the client is going to send a token to the server. And on the server, we can extract the user ID from the token. So technically, we don't need to pass the user ID and the request body. But for now, let's just go with the solution because we haven't implemented security at this endpoint. So let's create a profile for one of our users.

In my database, here's the last user I created, user number four. So I'm going to set that here. I'm going to set the phone to 1234, birthday to, let's say, 2000-01-01. membership to be for bronze. Now post. Beautiful. Now we have a customer record for this user. And here we have the phone birth date and membership. Now back in the database, let's make sure we have this record in the customer table as well. So open it. Here's the new customer record that we just created for user number four. So our customer or profile API is ready. Next we're going to talk about authenticating users.

Logging In:

All right, let's talk about authenticating users. So once again, on this page, you can see we have a bunch of endpoints for authenticating users. So we have these two endpoints for token-based authentication that comes with Django REST framework, and three more endpoints for JSON web token authentication. And I told you that the difference between these two authentication backends is that with token-based authentication, validating a token requires a database call. Whereas with json web tokens, we don't need an extra database call, okay? So we're going to go to jwt slash create to create a new token. And this is essentially the login endpoint. So let's go to auth slash jwt slash create. Now, you can see that the get method is not allowed at this endpoint, but we can post to this endpoint.

So over here, let's pass a username and password. So user1 with an invalid password. Post. Okay. We get an error saying, no active account found with the given credentials. And the error code is 401, which means unauthorized. This is a very important status code, so remember it for the future. Now, once again, let's log in with valid credentials. So user one with, I love Django. All right. Now we get two tokens. We have an access token, and a refresh token. An access token is a short-lived token that we use for calling secure API endpoints. But when this expires, we use the refresh token to get a new access token. So by default, the refresh token is valid for one day, whereas the access token is valid for five minutes.

And we can easily override these settings. So if you Google Django REST Framework Simple JWT, over here, you can find the documentation of this library at readthedocs.io. So this is the second library that we installed. This is our authentication backend. So let's go to the settings. Now look, here we have a couple of settings, access token lifetime, which is set to five minutes and refresh token lifetime, which is set to one day. Now these default values are pretty good in my opinion, but in this course, I'm going to change the lifetime of the access token From five minutes to one day because I don't want the token to expire in the middle of my recordings. So I'm gonna copy These couple of lines now let's go to the settings module and Add this setting at the end and of course, we need to change the lifetime to one day.

So days equals one Now we need to import the time Delta function so on the top From DateTime module, we're going to import time delta. Okay. So I'm going to log in one more time and store this token somewhere so I can reuse it in future lessons. So one more time. Let's pass the user a password. User one with I love Django. Okay, we got a new pair of tokens. Beautiful. Now I'm going to copy this entire line. Let me show you something. Here in VS code, let's open a new file and paste this line. Now let's trim this. So I'm going to remove all this stuff up to this double quote. So this is the beginning of our token. Make sure there is no space before the token.

Okay. Now at the end, there is another double quote. Let's delete that as well. So store this somewhere on your machine so we can reuse it in future lessons. So this is how we allow users to log in. We simply send a post request to this endpoint and pass users credentials. Then we get a pair of tokens. Now we need to store this on the client. Now that's really outside the scope of this course because this course is about backend development and front-end development is a completely different beast. So depending on what framework you're using to build the front-end, the implementation is different. For web apps, we use JavaScript to store the token inside browser's local storage. For mobile apps, depending on the framework you use, the implementations are different.

But all mobile platforms provide some kind of local storage. So we're going to store the tokens in the local storage and send them in future requests. We'll talk about that later in this section. But what about logging out? Well, to log out the user, all we have to do is remove the tokens from the client. Nothing else. So there's no endpoint we're going to call to log the user out. Because these tokens are not stored in a database. That's why we're using JSON web tokens. In the next lesson, I'm going to demystify that for you.

Inspecting a JSON Web Token:

All right, let's demystify JSON web tokens. So head over to JWT.io. This is the official website for JSON web tokens. So on this website, you can read about them and see how they work. We also have a debugger, which we're going to look at in a minute, and a bunch of libraries for generating and validating JSON web tokens for different platforms. So here we're using simple JWT, which is for Django. But if you're building your backend using a different platform like Node or ASP.NET, then there are other libraries you can use to generate and validate json web Tokens. So what is this debugger? Well, look here on the left side, we can paste a json web token and see decoded on the right side.

So I'm going to paste my json web Token. Now on the right side, you can see we have a header, a payload, and a signature. And these parts are color coded in the encoded json web Token. so the header is actually a json object. That is why we call these tokens json web tokens. Because this token is essentially a json object that is encoded in some way. So the header is a json object with two properties. Type, which is JWT. And algorithm, which is HS256. What it is, it doesn't really matter. I don't even know. Now the second part is the payload. Once again, here we have a json object with four properties. Token type is access. because this is an access token.

But if this was a refresh token, we would see refresh here, then we have the expiration date time. And this value represents one day from now. Next, we have JTI, which is a unique identifier for this token. And finally, we have my user ID here. Where did that come from? Well, when we log in our authentication backend, or our authentication engine validates our credentials, if they're valid, it's going to retrieve our user account. And then it will grab our user ID and puts it in the JSON web token. Okay. So this is the header and the payload. Now the most important part of this token is the digital signature in blue. This digital signature is generated based off of the header and the payload.

So if you make any changes in the payload, the signature has to be regenerated. So if a hacker grabs your JSON web token somehow, and then brings it to this debugger and tries to change the user ID to someone else's user ID, like one, Now the signature has to be regenerated. If they don't regenerate the signature and send this token to the server, the server is going to say, hey, this is an invalid token. We didn't generate this because this signature doesn't match our signature. It's like a fake passport or a fake driver's license. But you might be wondering if a hacker can regenerate the signature. In theory, yes, but that's only possible if the hacker gets access to the server. That is where we have a secret for generating digital signatures.

So over here, you can see the formula for generating the signature, we grab the header and the payload. And using a secret, we generate a digital signature. Now this signature is only stored on the server. So unless a hacker gets access to our server, they cannot regenerate this digital signature. And whatever changes they apply to the payload essentially becomes invalid. So this is how JSON Web Tokens work.

Refreshing Tokens:

Let's do a quick recap. So I told you that when we log in, we get two tokens, an access token that is valid for five minutes by default, and a refresh token that is valid for one day by default. Now, if the client needs to access a protected API endpoint, it needs to send the access token to the server in the request header. And I'm going to show you that later in this section. But if the token is expired, the server is going to respond with a 401 error meaning unauthorized. So at this point, the client needs to call the refresh endpoint using the refresh token to get a new access token. With this new access token, then it can call protected endpoints.

So back to the list of endpoints. Look, we have an endpoint for refreshing a json web token. So let's see how this works. So let's go to jw slash refresh. Now, over here, we have to pass our refresh token. So I'm going to paste my refresh token, and then post. Okay, now we have a new access token that is valid for five minutes by default. But in my implementation, I changed this to one day. So this is going to be valid for the rest of the course.

Getting the Current User:

All right, you learn how to allow users to register and login. Now let's see how we can get the current user. So we have an endpoint for that. That is users slash me. So let's hit this endpoint and see what happens. Us users me. So we get a 401 error, which means unauthorized, because we haven't supplied our authentication credentials. So here we need to pass our access token to call this endpoint. Now for that, I'm going to use a Chrome extension called mod modify header. I don't know if this is available for other browsers, but whatever browser you use, I'm pretty sure there are plugins that allow you to modify your HTTP request headers. So let's add this to Chrome real quick. Alright, now, let's pin it to the toolbar.

So we click on the extensions button and pin this to the toolbar. Okay, it's right here. Now, here we can add a new request header. So I'm going to set authorization. And by the way, here, we should get auto complete, I'm not sure why it's not working on this machine. So for the value, we have to type JWT as a prefix, followed by our access token. Now you might be wondering why we're using JWT here. Well, remember the setting that we said earlier, I told you that with this setting, you're saying that JWT should be a prefix for setting the authorization header. Okay. So We type that. Now, back to this page. Let's refresh is still not working. So I'm going to pause the recording and troubleshoot.

Alright, here's the reason I realized that I made a mistake and I redefined this setting over here. So this new definition is overwriting the previous definition. So we need to merge these two definitions into one. So I'm going to cut this line and add it right here. Okay, Save. And now let's refresh. So now we can access this protected endpoint. And we can also see our email ID and username. But what if you want to include the first and last names as well? Well, we need a custom serializer. So how about you pause the video and do this as an exercise? Alright, here's my solution. So back to Joeser documentation, let's search for serializers. Here on the settings page, as you can see, we have a serializer for getting the current user.

So to include the first and last names, we just need to create a custom serializer. So back to the serializers module of the core app, let's import user serializer and give it an alias like base user serializer. And then we need to create a new class called user serializer that extends base user serializer now over here, just like before, we're going to create a meta class that extends base user serializer dot meta and here we redefine the fields attributes. So we're going to return id, username, email, first name, and last name. Now the final step, we need to register this in the settings module. So over here, I'm going to duplicate this line and change this to current user and the serializer to user serializer.

Pretty simple. Now let's refresh this page. So we see ID, username, email, first name and last name. Beautiful. Now in this case, first and last names are empty because we didn't supply them while registering user one. So if we try to log in as user two and get a new access token, we'll see the first and last names of this user, which is Joe Smith. So this is how we can get the current user. Next, we're going to talk about getting the current users profile. That one thing I forgot to mention is that once you're done with this section, make sure to remove the authorization header here. Because otherwise, this extension will send this header to every website you visit. And this can totally screw the authentication process on other websites.

other websites that use JSON web tokens. So you will end up sending an invalid token to those websites and perhaps you're not going to be able to log into them anymore. So when you're done with this section, make sure to remove this header here.

Getting Current User’s profile:

Alright, now let's see how we can get or update current users profile. So we're going to add a new endpoint like this store slash customers slash me. So back to our customer view set. Here we're going to define a new method called me with two parameters self and request. Now this method is more accurately called an action. So all these methods that we have here in this view set or responding to requests are called action. So we have the create method. that we inherit from this mixin. We also have the retrieve method that we inherit from the retrieve mixin and so on. All these methods are called actions. So in this case, we're defining a custom action and we need to decorate it with the action decorator in rest framework.

So on the top from rest framework, that decorators, we're going to import the action decorator. Now look, we have quite a bunch of import statements on the top and this is making this file a little bit ugly. So later on, I will show you how to refactor this and make it cleaner. For now, let's not worry about it and just focus on our task. So back to where we were. We're going to decorate this with the action decorator. Now here we need to set the detail argument. If we set this to false, that means this action is available on the list view. So it's going to be available here customers slash me because customers is a list. Now, if we set this to true, the action is gonna be available on the detail view.

So we'll go to a specific customer and then access this endpoint. In this case, we want the action to be available on the list, so we set detail to false. Now, before going further, let's just return a simple response and make sure our plumbing is working. So let's hit customers slash me. Beautiful, we get okay. now the next step is to retrieve the customer and return it to the client. So you saw that a json web token contains a user id in its payload. Now how can we get that? Well, earlier we talked about the authentication middleware, remember? Let's quickly jump to the middleware section of the settings module. So here we have this authentication middleware. The job of this middleware is to inspect the incoming request and if there is information about the user,

is going to retrieve that user from the database and attach it to the request object. So back to our view set, every request has a user attribute. If the user is not locked in, this is going to be set to an instance of the anonymous user class. Otherwise, this is going to be a user object. So instead of returning, okay, let's return request that user that ID. Now refresh. So here's my ID beautiful. So now we can easily retrieve a customer with this user ID and return it to the client. So here we say customer.objects.get where user ID equals request.user.id. We get a customer object. Then we need to serialize it. So we create a serializer, a customer serializer, and give it this object.

And finally, we return serializer.data. Take a look. All right, here's my profile or customer record beautiful that currently we cannot update this profile because we don't have a box over here and also Look at the allow header. We can only send these requests to the sand point So now we need to enable put requests. So back to our action Here we need to set another argument called methods so we want to support get and put okay, so with that First we need to check the request method. So if request that method equals get, then we're going to execute this logic. Elif request that method equals put, we're going to do something else. So here we're going to start with a serializer, customer serializer, you want to start with an existing customer.

So I'm going to move this line out of this block. So this customer object is available everywhere in this method. So we give the serializer, our customer object and the request data. So data equals request data. Now one thing I forgot here is what if this user doesn't have a customer record. So instead of calling the get method here, we need to call get or create. With this, we're not going to end up with an exception. Okay. So here we have a serializer. Next, we need to validate the incoming data. So serializer is valid. And we raise any exceptions. Next, we call serializer that save. And finally return a response with serializer that data exactly like what we have done before many, many times.

So back to the browser, let's refresh. Alright, we got an exception saying tuple object has no attribute user ID. So I made a mistake here. Look, this method, it doesn't return a customer object, It returns a tuple with two values. The first value is a customer object. The second value is a boolean that tells us if this object was created or not. So here we're going to take that tuple and unpack it immediately to get this customer object. So I'm wrapping this in parentheses and adding a second variable created. So we are unpacking that tuple right here. Now let's refresh one more time. Okay. So here's my customer record. Now using this form, we can update these attributes. But look, we don't want to change the user ID here, because we don't want to associate this profile with someone else's account.

So user ID should be read only. So let's go to customer serializer. And over here, we're going to mark user ID as read only. Good. Take a look. All right, now we can only update phone, birthdate and membership. So I'm going to change phone to 111, birthday to 112000, and membership to gold. Put, lovely. So here's the updated object. Let's refresh and make sure we still get the right data. Okay, everything is working. So now we can get or update current user's profile, but we need to apply a permission to this endpoint so it's only accessible by authenticated users. We'll talk about that next.

Applying Permissions:

All right, let's talk about permissions. So here on Django REST framework website, under API guide, let's go to permissions. So here we have a bunch of permission classes, like allow any which is applied globally by default. And that's why we can access all API endpoints without logging in. We also have is authenticated is admin user is authenticated or read only. And we can also create our own custom permission classes. I'm going to talk about that in the next lesson. But for now, let's see how we can use these permission classes. One way is to apply them globally. So here in the settings module, and the settings for the rest framework, just like we have default authentication classes, we can also add default permission classes.

So we set this to a tuple or a list, I prefer a list. So we don't need that extra comma if we have only one item. Here we add a permission class as a string. So we can say rest framework. dot permissions, dot, the default value is allow any, but we can change this to is authenticated. Now, all of our API endpoints are closed to anonymous users. So if you open a private browser window, and go to store slash products, look, we have to authenticate in order to see the list of products. So we can apply one or more permission classes globally. And then if you want to, we can always override these permissions on specific views. so for now, I'm going to delete this setting from here.

Now, let's go to our customer view set. Over here, we can set permission classes, again, to a list of one or more permission classes. Now, why do we need a list here? Well, we can supply multiple permission classes here. And if any of them fails, then the client will not be able to access this view. Okay. So let's go to the top. And from rest framework, rest framework permissions, we're going to import is authenticated. Now, back over here, I'm going to pass that and this list is authenticated. So now all actions in this view set are closed to anonymous users. But once again, if we want to, we can override this permission in this particular action. So here we can set another argument called permission classes.

Again, we set it to a list of permission classes. We don't want to do this. So let's remove that. Now, what if you want to have different permissions for different actions like create and retrieve? Let's say we want anyone to be able to retrieve a customer object, but only authenticated users or admin users can update a customer object. Well, for that we need to override a method that we inherit in this view set called get permissions. So here we can check to see if self the request that method equals get, then we can return a list of allow any. Now one thing I need to emphasize here is that we should return a list of objects, not classes. So this is the difference between this method and this attribute.

Here we're setting this attribute to a list of permission classes. But with this method, we should return a list of permission objects. So if we don't add these brackets here, you're going to get a weird exceptional runtime. So we allow unrestricted access to the get method. But for anything else, we want the user to be authenticated. So is authenticated. Okay, now, back to my private browser window, let's go to store slash customers slash two. So I can read this customer, but look, I cannot update it. So this is how permissions work in Django rest framework, pretty simple. In the next lesson, I'm going to show you how to create your own custom permission classes.

Applying Custom Permissions:

All right, let's talk about creating custom permissions. So I'm looking at our products endpoint. Now, currently we don't have any products in the database because earlier we had to drop and recreate our database. But what matters here is that this endpoint is currently open to anyone, including anonymous users. So if I go to a private window and refresh, look over here, I can create a new product. I can even update or delete products. Certainly we don't want this to happen. We want only admin users to be able to modify products, but anyone, including anonymous users, should be able to retrieve the list of products. Now, back to Django REST Framework's documentation. Look here on the permissions page. We don't currently have a class called isAdmin or readOnly.

We only have isAuthenticated or readOnly. So this is a situation where we need to create a custom permission, and that's super easy. So back in VS Code, here in the store app, let's add and your file called permissions.py. Now on the top from rest framework that permissions, I'm going to import two classes. One is is authenticated and the other is base permission. Let's look at the implementation of this class to get some ideas. So this class extends base permission. So this is the base class for all permissions in Django rest framework. Now here we have a single method. has permission that returns a boolean value and here we have a simple condition. If request that user is set and this user is authenticated, then we return true.

So earlier i told you that if the user is not logged in, request that user is set to an instance of the anonymous user class. In that class, if the read is authenticated, we get false, okay? So this is the implementation of this permission class. let's look at another example is admin user here we are saying if the request that user is set and this user is a staff member meaning they can log into the admin panel then we're going to return true so now that you understand how permissions work let's create our own custom permission class so back over here we don't need to import is authenticated anymore i just wanted to show you how it works so we create a new class called is admin or read only

that extends base permission. Here we override has permission. Now we check to see if request.method equals get, then we're going to return true. So anyone can access the target view. Otherwise, we're going to return true if both these conditions evaluate to true. So here we call the bool function and say, if request.user is set and request.user is staff. That means this is an admin user, I can execute other operations. This is how we can create a custom permission class. Now there is a tiny issue here. And that is with the current implementation, even head or option requests require the user to be an admin, we certainly don't want that to be the case. So better way to write this condition is like this, we can check to see if this method is in

the list of safe request methods. So, from rest framework, we import the permissions module. Here we have a constant called save methods. Let's look at this. So here we have get, head, and options. Okay? So if request.method is in the list of safe methods return true. Otherwise, we have this condition here. Now let's consolidate these two import statements. So, i'm going to remove the second one and here we say permissions dot base permission good now let's apply this to our product view set so we go to product view set and over here we set permission classes to is admin or read only now let's test this so back to my private browser window let's refresh Now I can see the list of products, but I cannot change or add a product.

Now in contrast, in this other window where I've logged in as John Smith, and I'm looking at the products endpoint, look, I can create a new product. Great. Now let's go to our API route. So we secured our products endpoint. Let's apply the same change to our collections endpoint. Let's go to collection view set and set permission classes. to is admin or read only. Good. Now, what about shopping carts? We don't want to apply any permissions here, because anyone including anonymous users should be able to create a shopping cart. So we're done here. And finally, with customers, we want to make sure that only authenticated users can access this endpoint. So let's go to customer view set. All right, look, we applied this permission over here.

And that's good. But I just noticed a problem. Earlier, we used these mixins here, create, retrieve, and update, because i assume that using this API, we don't want to list or delete our customers. I assume that we want to perform these tasks via the admin panel. But let's change the requirements a little bit. Let's make this endpoint consistent with our products and collections endpoints. So we want to allow all operations, but we want to restrict them only to admin users. So only admins can manage customers, but any authenticated users should be able to access their profile. So instead of these individual mixins, we're going to use model view set as the base class. So all operations are available. Then we're going to change the permission class to is admin user, which is defined in rest framework.

And then we're going to overwrite this permission for this particular action. So here we set permission classes to is authenticated. So now, if you open a private browsing window and hit the customer's endpoint, look, we don't have access to do anything. But if I open a normal browsing window, and currently I'm logged in with John Smith. So let's go to customers endpoint. Now because John Smith is a staff member, so he's an admin. And now we can see all customers, we can also create a new customer and update existing ones. But also, I can access my current customer record. That is better, because now all our endpoints are behaving consistently.

Applying Model Permissions:

So with our current implementation, only admin users can manage customers via the customer's endpoint. But if you remember, earlier in the course, we created a group called customer service. Now, what if you want to allow these people to manage customers via our API as well? This is where we use Django model permissions. So back to our view set, here we're going to use another permission class called Django model permissions. When we apply this permission, the user has to be authenticated, and they should have the relevant model permissions. So I'm currently logged in as John Smith. And if you remember, early in the course, I put John Smith in the customer service group. So now I should be able to access the customers endpoint.

So look, I can create a new customer. Or if I go to a particular customer, I can delete or update that customer. but now let's remove john from the customer service group and see what happens. So, back to the admin. Let's go to users. And then pick john Smith. Now I'm going to remove the customer service group from here all right save the changes. Okay. Back to this window. Let's refresh. So I can still view this customer, but look, the delete button has disappeared and Also, I cannot update this customer. So I can only view data, but not modified. Let's see why this is happening. So back to the code. Let's look at the implementation of this class. So in this class, we have a map or a dictionary of permissions.

So look for get options and head, we don't have any permissions. But to send a post request, the user should have the add permission. So here we have app label, which is going to be changeDynamicLoudRuntime.addUnderlineModernly. So this is the code name for the permission. So back to the database, look at the permission table. Here we have permissions like addCustomer, changeCustomer, deleteCustomer, and viewCustomer. So what you see here is the mapping between HTTP methods and relevant permissions. So similarly, to send a put or a patch request, the user needs to have the change permission. Now, to solve this problem and prevent people outside of the customer service group from viewing data, we need to create a custom permission class and extend Django model permissions.

So here in the store app, let's go to the permissions module and create a new permission class. We can call this full Django model permissions. This should extend Django model permissions. Now, here we're going to define a constructor. Then we're going to set self that perms map of get to let's go back to this class and copy a bit of code as an example. So I'm going to grab this permission, then paste it over here and change add to view. So to send a get request, the user should have the view permission. Okay. Now, we're going to use this permission in our customer view set. So full Django model permissions. Now back to the browser. Let's refresh. Now I no longer have access to this endpoint.

Now look at the status code of the response. 403, which means forbidden. So even though I sent my authentication credentials, because I don't have access to this resource, I got a 403 error, okay? Now we have another similar class called Django model permissions or a non read-only This is exactly like Django model permissions, but anonymous users will have read only access to data. So back to my private window, let's go to the customer's endpoint. Now look, I have read only access to data, but I cannot change anything. So this is how model permissions work. But for this application, this is a bit of overengineering, we don't really need that level of complexity. But I thought to include this in the course, in case you have scenarios like that at work that you want to implement with Django.

For this application, we want to limit customer management operations only to admin users so is admin user that is good for us.

Applying Custom Model Permissions:

The last thing we're going to talk about in this section is applying custom model permissions. So in the previous section, we created a custom model permission called cancel order. Now let's see how we can apply these custom model permissions to our API endpoints. Now for this lesson, because currently we don't have the orders API, I'm going to create a custom permission for our customer model. So let's go to the customer model. And here in the meta class, We should set permissions to a list of tuples. Each tuple should include two values. The first one is the code name for the permission, like view history. And this is the value we reference in code. You'll see that in a minute. The second value is a description for the permission can view history.

We often use this convention. Okay. Now that we have modified the model, we need to create a migration and run it. And as a result, this permission will be stored in the database. So Python, manage.py, make migrations. Good. And then migrate. Lovely. Next, we need to go to our customer view set and create a custom action. So let's go to customer view set. That just like we have the me action, we're going to create the history action for viewing the history of a particular customer. So we define an action called history with history. three parameters, self request, and PK, because this is for a particular customer. Now we need to decorate this with the action decorator. And here we need to set detail to true.

Because once again, this is for a particular customer. Now before going further, let's just return a simple response. Okay. Now, we're looking at customer number one, let's go to the history endpoint. Okay, we get that response. Beautiful. So now let's apply the permission. for that, we need to create a new permission class. So here in the store app, let's go to the permissions module and create a new permission class. So class, view, customer, history, permission. Now, all permission classes that we create should derive from base permission, okay? Now, here we override has permission, just like before, but our condition is a little bit different from what you have seen earlier. So here we're going to return request that user that has perm. So this user object has a method called has perm.

And here we pass the code name for the permission. So we start with the app name, which is store and dot, and then the permission name view underlying history. If this returns true, then the user is going to have permission and they will be able to access the history. So the final step is to go to our customer view set. And decorate this custom action with our new permission class. So we set permission classes to view customer history permission, okay? Now back to the browser, refresh. All right, look, because John currently doesn't have this permission, he cannot access this endpoint. So back to the admin panel, let's refresh and give John this explicit permission. So view history. Alright, I'm going to add this permission here.

But remember, this is just for this demo. As a best practice, you should never apply ad hoc permissions, because then over time, it becomes really hard to see who has what permissions, always create groups, add permissions to your groups, and then add users to groups. This way, you can easily filter users by groups to see who has what kind of permissions. Let me show you real quick. So first, I'm going to save the changes. Okay, now, back in the list of users, look on the right side, We have a filter in my group, because currently we have one group in the system. So if we select customer service, now we can see anyone with the customer service group. Currently, no one. But we don't have this filtering by permissions.

We can only filter by groups. That's why you should avoid ad hoc permissions, okay? But anyway, now John Smith has the view history permission. So back to this other tab, and now we get the okay message. Beautiful.

Designing and Building the Orders API:

Introduction:

Welcome back to another section of the ultimate Django course. In this section, we'll take our application to the next level and build the orders API. Once again, this is a great opportunity for you to get your hands dirty in code and practice a lot of stuff you have learned so far. So let's jump in and get started.

Designing the API:

Alright, let's spend a couple of minutes and plan out what we're going to build in this section. So we're going to build a new endpoint for managing orders. Now to create an order, we simply send a post request to this endpoint. Now all we need to include in the request body is the cart ID, because we can extract the current user's ID from the JSON web token included in the request header. So that means this endpoint should only be open to authenticated users. Okay, so using cart ID and user ID, we can create an order object, and then returning to the client. Now we should also support get requests at this endpoint. But what the user sees depends on their permission. If I'm a regular user, I should only be able to see my own orders.

But if I'm an admin, I should be able to see all orders in the database. Now we should also be able to retrieve a specific order. And once again, here we have the same policy. So I should only be able to retrieve my own orders unless I'm an admin. Now potentially, we can also support patch and delete requests here for updating the status of an order or deleting it. So starting from the next lesson, we're going to build this endpoint step by step.

Getting the Orders:

All right, the first thing we're going to implement here is getting the orders. Because creating an order is a bit more complex, so I would prefer to start with something simple. So we have a foundation, and then we'll work on creating an order. And this is how I want you to write code. Always start simple, always build a foundation, and then gradually build things up step by step. So this one is pretty easy. We're going to build a new endpoint. When we hit this endpoint, we want to see all orders in the database. So at this point, we don't care about permissions or any other aspects of this endpoint. So when we hit this endpoint, we get all attributes of the order. We also have order items.

And each order item includes a nested product object. So you can do this on your own, spend five to 10 minutes on this, then come back see my solution. Alright, here's my solution. So just like always, we start with a serializer. So here in the serializers module of the store app, we're going to create a new class order serializer. which is a model serializer, we create our meta class, set the model to order, and fields to, let's go to the order class and see what fields we need here. So, placed at, payment status, and customer. So, just like always, we start with ID, then customer, placed at, and payment status. Okay? Next, we need a view set. So, we go to the views module of the store app and create a new class, order view set.

Now I'm going to have this extend the model view set class so we get all operations. Here we set query set to order that objects.all and serializer class to order serializer. Pretty simple. Now the final step. We go to the urls module of the same app and register a new endpoint for managing the orders. Nothing new so far. So with that, we can hit this endpoint. Good. Now currently, we don't have any orders. So let's go to the database and manually create one to make sure everything works properly. So back to the database. Because we dropped and recreated this database, our database is currently empty, and we don't have any collections or products. So let's open a new query console and bring our seed file right here.

Now, we select the schema, we set it to storefront two, we select all the statements and execute them. Good. So we have a bunch of products. Now, let's create an order with a bunch of order items. So here's the order table. So let's add any order. Place that I'm going to set this to 2021 September 1, P for payment status and three for customer ID. let's submit the changes. Good. Now we have order number two, because previously i created an order and then deleted it. That's why the id is two on my machine, but on your machine is going to be one. Now let's create a bunch of order items. So in order item table, we add a couple of rows. For quantity, I'm going to use 10.

10 for unit price, order 2 product 1. now 20, 20, order 2 and product 2. Okay? Submit the changes. Good. Now back to our endpoint. Let's refresh. So we have order number two placed by customer three with these other attributes. Beautiful. One thing that is missing is our order items. So back to our serializers module. Let's create another serializer. Order item serializer. We define our meta class, set the model to order item and fields to let's double check. So here we have order, product, quantity, and unit price. So I'm going to include ID of the order item, followed by product, unit price, and quantity. So I'm not including the order here because we're going to use this serializer inside our order serializer. So here we also include the items field and set items.

to order item serializer with many equals true. Okay, now save and back in the browser, refresh. Alright, we got an error saying order object has no attribute items. So let's go to the order item class. This is where we have a foreign key to the order model. So we need to set related name to items. Okay, now refresh. All right, we can see our order items beautiful. But let's change product to a nested object. So this way, we can return all critical information about each product. So the client doesn't have to send additional requests for each product in the order. So here in order item serializer, we set product to simple product serializer. Remember, we defined this earlier in the course. With this serializer, we only serialize the critical information about a product.

So refresh. Now, each product is a nested object. Beautiful. I think this API is in good shape. Next, we need to talk about permissions.

Applying Permissions:

Alright, let's talk about applying permissions. So our new endpoint is currently open to everyone, including anonymous users. So if I go to my private window and hit this endpoint, I can still see this order. Certainly not good. So the first thing we need to do here is applying a permission class. So here we're going to set permission classes to isAuthenticated. Now, back in the private window, refresh. I cannot see this order. Beautiful. Now, back to this normal window. I'm currently logged in as John Smith. And I'm looking at this order for customer number three. But this is a different customer. This is not John Smith. Let me show you. So back to our database. Let's look at the customer table. Here's customer number three.

This record is associated with user number four. Now who is user number four? Let's find out. So you go to the user table. User four is this user ID user two, which is Joe Smith. completely different account. Now, in this case, John Smith is an admin user, so he should be able to see all orders in the database. But if John is not an admin, he should only see his own orders. Currently, we haven't implemented this rule. So, back to our view set. Instead of setting the query set attribute here, we're gonna override the get query set method now here we say if self that request and user in staff, meaning admin, then we're going to return order.objects.all. So all orders in the database.

Otherwise, we want to apply a filter and only retrieve orders for a specific customer. So order.objects.filter where customer id equals. Now here we need to find the customer id of the current user because the customer id is not included in the json web token. So from user ID, we need to calculate customer model, we're going to get the customer with user ID equals self that request that user that ID. And here we're going to retrieve a complete customer object. But all we need is the customer ID. So we can apply a bit of optimization here, and only pick the ID field. Now, this is technically premature optimization, because we haven't identified a performance problem, and we're fixing it ahead of time. But this is really tiny.

It's not making our code more complicated. So I think this is still a good practice here. So we get the customer ID. And then pass it over here. Now look in this code, we have a couple of references to self that request that user. So we can simplify this code a little bit, we can make it cleaner and say, user equals self that request that user. And then replace both these expressions with user. That makes our code a little bit cleaner. Okay. So we apply a filter and then return the result. Okay. Now let's see what we get. So let's hit the send point. Our website is not working. So back in the terminal, we have an error saying base name argument not specified.

So because we removed the query set attribute from this view set, and now we're overriding the get query set method, Django REST framework cannot figure out the base name for our endpoint. So we go to our URLs module. And just like before, where we set the base name over here, we're going to set the base name for this endpoint. Now, do you remember why we need to set this? This is for generating the name of views. So we're going to have two views orders less than orders data, okay. But here we only specify the prefix. Now, let's refresh. still not working because the changes are not picked up so i'm going to restart the web server good no errors so refresh lovely so i can still see this order because i'm an admin user but let's go to the admin and pull up john's account and remove his admin status now after we save the changes john should not be able to view this order there you go

Now there is a tiny issue in our implementation if the current user doesn't have a customer record or a profile This line or more accurately this method is going to throw an exception. Let me show you so Back to the database. Let's go to the customer table. This is the record for John Smith. So I'm gonna delete this record and Submit the changes Good now, let's hit this endpoint one more time There you go. Customer matching query does not exist. Because earlier I told you that the get method expects one record in the database. If you have zero records or more than one record matching this criteria, we're going to get an exception. So here we're going to change this to get or create.

Now you know that this method returns a tuple with two values. The first value is the object we are reading. The second is a Boolean that indicates if a record was created or not. So, back to the browser now refresh all right the error is gone. Beautiful. Now, even though using this method we solve this problem, here we are violating an important principle in programming, and that's called command query separation which means our methods or our functions should either be commands and change the state of the system, or they should be queries, which means they should only return data. They should only answer questions. They should not change the state or the data in the system. In this case, this method get query set is purely for getting data.

But as part of getting data, we're changing the state of the system. So if we hit this endpoint for reading orders of a customer, we'll end up creating a customer record. So this is the violation of the command query separation principle. When you violate this principle, your software becomes unpredictable, you'll end up with weird results, weird side effects. So later in this section, I will show you a better technique for solving this problem. For now, we're good. Next, we're going to talk about creating an order.

Creating an Order:

Earlier I told you that when creating an order, all we need to send to the server is the cart ID. So we cannot use the order serializer because the object that we're representing here, look at these fields, this object has a completely different shape from the object we need to send to the server. So we need a new serializer. So in the same module let's create a new class called create order serializer. Now here we're not going to use model serializer because we're not going to use a meta class that is based on the order model. Why? Because cart ID is not a field in the order class. So instead of using the model serializer, we're going to use the base serializer and that means we don't need this meta class.

We just want to explicitly define a field here. Cart ID is a UID field. Now in the serializer, we need to overwrite the save method. Because the logic for saving an order is very specific. It's not something we want Django to generate for us. So we have to go to the shopping cart table, we have to grab all cart items, move them to the order items table, and then delete the shopping cart. So this is a very specific logic, we need to implement it by hand. Now for now, I don't want to do anything here. I just want to print this cart ID and the user ID on the terminal. So to get the cart ID, we have to go to self.validated data of cart

ID. Now what about the user ID? Well, here we don't have access to the request object because we are inside a serializer. So we have to go to our view set and using a context object past the user ID here. So back to our view set. Let's override get serializer context. Here we return a dictionary with user ID, and we set it to self that request that user dot ID. Pretty simple. Now, back to the save method, we're going to print self the context of user ID. Okay. Now the next step is to use this serializer in our view set. So instead of hard coding this particular serializer, we're going to override get serializer class. And here we say if self that request that method equals post, then we're going to return create

order serializer. Otherwise, we're going to return order serializer. Let's test our implementation after this point. So back to the browser refresh. Now we can post a cart ID here. But currently, we don't have a cart. So let's go to our API route. Let's go to the cart standpoint and create a new cart. So here's a cart ID. I'm going to copy this. Now, back to the orders endpoint. Let's send this to the server. Okay? So we got this back because this is a field in our serializer. Now here in the terminal, look, we have the cart id and the user ID. So with these two pieces of information, we can create an order object. So back to our serializer, we're going to say order the objects that create now what do we need to pass when creating an order?

Take a look. So here we have place that. which is set automatically. you have payment status, which has a default value. And customer. So the only field that we need to set is customer. So, here we need to set customer to now once again, we need to get the customer object for this user so customer the objects that get or create where user ID equals self context of user ID. Now, in this case, this is not a violation of the command query separation principle, because with the save method, we're changing the state of the system. So this method is acting as a command, not a query. The violation was in other scenario where we had a query method. But in that query method, we change the state of the system.

Okay. So here we get or create a customer object. We set it here. And then using this customer, we create any order. Okay. Now, one more thing here is that this method returns a topple. So let's unpack it. Okay, good. Now let's test our implementation before going further. So I'm going to refresh and post this form one more time. Good. No errors. Now we should have an order object. So back to the database. Let's look at the order table. There you go. We have a new order for this customer customer number four, which is John Smith. Great. So the next step is storing Order items. We'll do that next.

Creating Order Items:

Now for creating order items, first we need to get items in this cart, and then for each cart item, we need to create an order item and save it in the database. So first we're gonna say, cart item.object.filter, where cart ID equals self.validatedData of cart ID. Here we get a list of cart items, but more accurately we get a query set. but next, when we iterate this query set, we get a collection, right? So now let's remove these two print statements. We don't need them anymore. Now, here we need to convert cart items to order items. And for that, I'm going to use a list comprehension. Do you remember the syntax? Item for item in collection. What is our collection here? Cart items.

So we get each item, and now we want to convert each item to an order item. Okay? now here, using keyword arguments, we're going to initialize this order item object. So, first we need to set the order to the order object that we just created. So, let's store that right here, and then initialize the order item like this. Next we need to set the product to item dot product. And that means, when retrieving these cart items, we need to eager load them with their product, otherwise, for each cart item, we're going to end up sending an extra query to read the product of that item. So here we're going to say, select related product. Now this line is getting a little bit too long, so let's break it down into multiple lines.

That is better. Good. So we have a bunch of card items. Now, let's move on. So we set the product. Next we set the unit price of the product at the time of placing the order. So item.product. that unit price. And final quantity to item dot quantity. So we get a list of order items. Now we need to save them. But before doing so, I would prefer to reformat this code because currently, all these brackets are all over the place. And it's not really clear what we're doing here. So I would prefer to put this on a new line. So we can clearly see that we're creating a list here, then I would like to indent these few lines. So we can see

these are the fields of the order item. Next, I'm going to align the right parenthesis with order item. And finally, I put this right bracket over here. So now we have a clean structure in our code. Lovely. Now we need to save these order items. Now we're not going to iterate over them and save them all individually because that would be too many queries to the database. So we're going to insert them in bulk. And for that, we have a method. Order item. that objects that bulk create, and then we pass our order items. Let's test our implementation. So this cart currently doesn't have any items. So we need to create a new cart. Let's create a new cart. Here's the cart ID.

So let's go to cart slash this slash items, and add a couple of products to this cart. Park one with quantity of 10 and product two with quantity of 20. I always use this pattern because it makes it easier to see the data and ensure we're saving the right data. Okay, so now we have a couple of products in this cart. If we refresh this endpoint, let's hit it over here. Look, we have two items in the shopping cart. Beautiful. So now let's go to our API route, and then orders and submit. this card ID. Alright, no errors here. Now, back to the database. Let's look at the order item table. So here we have two items, quantity of 10 unit price for is the order ID and product ID.

So one and 10, two and 20. Now don't worry, if the IDs here are different on your machine. Because again, I was doing some practice before recording this video. That's why we have five and six here. But on your machine, you should see one and two. So we store the order items. Now finally, we need to delete the shopping cart. So we say cart.objects.filter where PK equals. Now once again, we need to extract cart ID from the validated data dictionary. So I would prefer to extract this expression and store it in a separate variable called cart ID so we can reference it multiple times. So on the top, cart ID equals this expression, okay? Now, over here, we set PK to cart ID, and then delete that cart.

Now there's a problem in this implementation. Look, first we create an order, then we create a bunch of order items, and finally we delete the cart. So we have multiple updates in the database. Now what if our database server goes offline in the middle of one of these updates? We'll end up in an inconsistent state. So some of the data will be missing. This is where we use a transaction. Using a transaction, we can treat a block of code as atomic, like an atom. So either every code in that block should be executed, or if something goes wrong, all the saved changes should be rolled back. This is the benefit of using transactions. So on the top, from Django dot DB, we import the transaction module.

Now let's reorganize these statements. So first we import stuff from Python, then from Django, Next, from rest framework, and finally, from our application, okay? Now here we don't need to prefix store, we can simply reference the models module in the current folder. Now, back over here, we're going to wrap this entire code in the save method inside a transaction. So we say with transaction.atomic, this function returns the transaction object that we pass to the width block. Now I wanted to indent all the code, and put it inside the width block. Okay, let's test our implementation one more time. Now refresh. All right, no errors. So back to the database. Let's look at the cart table. Our second shopping cart that had a bunch of items is now gone.

Beautiful.

Returning the Created Order:

Now you might have noticed that when we post this cart ID to the server, instead of an order object, we get the cart ID back. This is not how we design our API. So let me show you how to solve this problem. First, you need to understand why this happened so we come up with the right solution. So back to our order view set, let's go to the definition of model view set and then look at create model mixin. So in the create method, first we get a serializer And this comes from our view set. So in our view set, we either set the serializer class attribute or we override the get serializer class. So we get a serializer, then we validate data, we save the changes.

And finally, using the same serializer, we return the response back to the client. Now, in this case, our serializer has only one field cart ID. And that is why what we send to the server is exactly what we get back. Now to solve this problem and return an order object, we need to create a different serializer and then we need to override the create method because we cannot rely on this implementation so we need to provide our own implementation and there we're going to use two different serializers we're going to use one serializer to deserialize the data and get the cart id and we'll use another serializer to return the order back to the client so back to our view here we're going to override the create method so

we're going to create a serializer, which is create order serializer, and give it the request data. Next, we validate the data, exactly like the create model mixing. So raise exception should be set to true. Next, we save the changes. Now we're going to create another serializer. So we reset this to order serializer and give it our order object. but where is the order object? Well, we should return it from the save method. So let's go to our serializers module. And here in the save method, at the end, we're going to return our order object. Okay? Now, back to our view. So we get this order, then we give it to this other serializer. And finally, we return a response with serializer dot data okay so

Let's test this implementation. All right, we got an exception, but there is no meaningful message. All we see is user ID. So let's look at the stack trace where this exception was thrown. So look in our serializer at this line, we have this code. User ID equals self.context of user ID. So back to the create method. When creating this serializer, we need to give it the context object so we have access to the user ID. so over here, we can set context to, here we can call this method in this class get serializer context but now this method is really unnecessary. We can simply grab this dictionary and set it right here. This way there is no distraction in our code. What we need is right in front of us, okay?

So let's remove this unnecessary method. This was only useful if we wanted to rely on the create model mixing that we inherit in this class. In this case, we're not relying on that mixing, we're implementing the create method from scratch. So we set the data and context. Now, back to the browser. Let's refresh. All right, now we get a new order object with this ID.

Data Validation:

So we implemented and tested the happy path where we receive valid data. But let's see what happens if we receive invalid data. So there are two scenarios we need to consider here. One is that we receive an invalid cart, a cart that doesn't exist. And the other is we receive an empty cart, a cart with no items. In that case, we don't want to create an empty order without any items, it doesn't really make sense. So now that we have deleted this cart, let's see what happens if we post this one more time to the server. so nothing happens but if you look at the order table you can see we're creating all these extra orders that don't have any items so look at our implementation in the save method we always create an order right from the get-go whether the given card exists or not this is not right so first we need to validate the cart id and for that we can create a validate method so we define a method that starts with validate followed by underline and then the name of the field cart id

now here we need two parameters, self and cart ID. So first we want to make sure that this cart exists. So we can say if cart dot objects dot filter where pk equals cart ID, that exists. So if this card doesn't exist, then we're going to raise a validation error. So raise sterilizers dot validation error and here we can see no cart with the given ID, was found. Otherwise, we're going to return the cart ID as the valid value. Let's test this scenario before going further. Again, step by step. So I'm going to post this one more time. Alright, now we get an error saying no cart with a given ID was found. And the status code is 400, which means bad request.

So the client sent invalid data. Beautiful. Now the second scenario, we're going to send an empty cart to the server. So for that first, we need to create an empty cart. Post Okay, here's an empty cart. Now, let's go to the orders endpoint and send this to the server. Again, we didn't get any errors, but we got an extra order without any items. So to solve that, we go back to our validate method and add another if statement. So if cart item, that objects that filter where cart id equals cart ID, that count is zero, so if you have no items in this card then we're going to raise another validation error. And here we're going to say the card is empty. Okay?

Now, back to the browser, let's create another shopping cart, because the previous cart was deleted. So brand new empty card, we're going to send it to the orders endpoint. All right, now we get this beautiful validation error saying the cart is empty. So this is how I want you to write code. Always do things step by step. Don't try to do too many things. Always focus on a single task. First build the happy path, then improve it, then refactor your code, then add data validation and so on.

Revisting the Permissions:

All right, let's revisit our permissions. So let's go to a specific order that I've placed. Order number four. Look, I can delete and update this order. But I don't believe this function should be available to all authenticated users. We want to restrict this only to admins. So back to our view set. Instead of setting the permission classes here, you're going to overwrite get permissions. Now here we can say if self.request.method. equals put or patch or delete. But we're going to end up with a long if statement. So here's a better way. We can say, if the method is in this list, meaning put, patch, and delete, then we're going to return is admin user otherwise we're going to return is authenticated and make sure you're returning a list of objects, not permission classes, okay?

Now here we can also argue that We don't want to support put requests because we're not going to update all properties of an existing order. We want to update a subset of them, like payment status. So let's remove put from here and then restrict the HTTP methods. So we set HTTP method names to get, patch, delete. Now we can also enable head and options for some clients to see what operations are available at this endpoint. and by the way, make sure to spell this in lowercase, otherwise it's not going to work. So, one more time, let's refresh this page. The delete button is gone, and i cannot update this order.

Updating an Order:

Alright, let's talk about updating an order. So first we need to make John Smith an admin user. So here in the admin, let's pull up John's account and make him an admin. Save. Good. Now refresh. So now we have access to delete and update functions. But we don't want to patch all these properties. For now, we only want to patch payment status. So everything else should be read-only. We don't want to change the customer of an order. We don't want to change when the order was placed, as well as the items, these attributes should not be touched. Now here we have two ways to implement this. Look at our order serializer class. One option is to redefine all these fields over here and mark them as read only except payment status.

But I don't like this approach because with this will end up polluting this class. And also in the future, if you add new fields in the order class, we have to remember to come back here and redefine those fields and mark them as read only. So this is not a very solid solution. A better approach is to create a custom serializer for updating an order, just like how we created this serializer for creating an order. So let's define a new serializer called update order serializer. This should be a model serializer. Because we're going to define a meta class, set the model to order and select a subset of fields. So we want to rely on the default implementation of model serializer for saving the changes.

So what field do we want to allow here payment status okay now back to our view set let's look at this method get serializer class so we need another condition, elif self.request.method equals patch. Then we're going to return update, order, serializer. Now, take a look, refresh. We only have payment status, so let's change it to C for complete. Patch, good. This is the updated property.

Signals:

Alright, let's talk about signals. In Django, we use signals to decouple our apps and prevent them from stepping on each other's toes. Let's talk about them using a real example. So let's go to the order view set. And look at the implementation of the get query set method. So earlier, I told you that here we have a violation of the command query separation principle. Because this is a query method. So we call it to get some data, but we're potentially changing the state or the data in the system. This is a violation of the command query separation principle, because our methods should either be commands and change the state of the system or queries. So they should return data, but not both. So how can we solve this problem?

Well, we should always think about the underlying cause. The reason we have to use this method over here is because it is possible that at the time this method is called, we don't have a customer associated with this user. but what if when someone registers, we automatically create a customer record for that user? Well, we can go to user, create, serializer. And in this serializer, we can overwrite the save method. So first we call the save method of the base class. This creates and returns a user object. So we store it over here. And then we can say customer, the objects that create user equals user. Pretty simple, right? But if you remember, Earlier, I didn't like the idea of touching customer profiles in the serializer.

So I told you that I didn't want to include the birthdate field here. Because with this approach, as our registration form gets more complex, this user creates serializer has to worry about more aspects of the application. So it's going to become more responsible for many other things. This is like a chef that has so many responsibilities that in this particular case, this is only one extra line of code and This is really not the end of the world, but let me show you another way to solve this problem. This is where we can use signals. So in Django, our models have a bunch of signals or notifications that are fired at different times. For example, we have pre-save that is fired before a model is saved.

We also have post-save, which is fired after a model is saved. And we also have pre-delete and post-delete. So in our application, we can listen to these notifications and do something. so here's our current implementation. In the core app, where we have user create serializer, you're responsible for two things, creating a user and a customer. Now, what if we go in the store app and listen to the post save signal of the user model? So we're going to get notified when a user registers, and then we can create a customer record right there. So with this approach, we have shifted the responsibility of creating a customer from the core app to the store app. So the responsibilities are better distributed across various apps.

So let's see how we can put this in action. So back to user and create serializer. Let's delete the save method and also remove the birth date field. Okay, now in the store app, let's add a new file called signals.py. Here we should define a function called create customer for new user look a very descriptive name. Now this function should have two parameters sender, which is the class sending a notification or a signal and our keyword arguments. In our keyword arguments, we have a key called created. This is a Boolean. So we can check to see if a new model instance is created. If so, this is where we're going to create a customer. So we say customer dot, it's not loading. So from the models module, we're going to import the customer model.

And then here we say customer that objects that create user equals that to get the instance, we're going to go to our keyword arguments and pick the instance, and tense, okay, as simple as that. So now we have this function, we need to tell Django that this function should be called when a user model is saved. So first, we need to import a decorator from Django dispatch, we're going to import the receiver decorator, then we applied on this function. Here we need to pass two arguments. The first one is the signal we're interested in, in this case, post save. So we need to import it from Django dot DB dot models dot signals. We import post save. Then we pass that over here.

Now we don't want to listen to the post save signal of every model in this project. We're only interested in the post save event of the user model. So here we set the sender to that we don't want to reference the user model defined in the core app, because it's going to add a dependency from the store app to the core app. Look, currently we are in the store app. And this user model is defined in the core app. So with this implementation, our store app is going to be dependent on the core app. And with this, it's no longer going to be a self contained app that we can independently deploy and reuse in other projects. so everywhere we want to take the store app, we should also take the core app.

And this doesn't make sense, because the core app is very specific to this project. It's not something we want to reuse somewhere else. So just like before, from django.conf, we import our settings, and then we reference settings dot auth user model okay so we have to find this function, and django knows that it should call it every time a user model is saved. but this code is not executed unless we import it somewhere. Where? Well, in this app, in the store app, we go to the apps module, and in this config class, we override the ready method. This method is called when this app is ready. It's initialized. So here we import store dot signals all right now i'm going to create a new user called user3.

So post. We didn't get any errors. Now we should have a customer record for this user. So here in the database, let's look at our user table. This is the new user on my machine, user three, and its ID is six. So now let's look at the customer table. We should have a record for the user with ID six beautiful. So back to the get query set method where we had a problem with our new implementation. Now that we are using signals, we no longer have to worry about Creating a customer. So here we simply call get and Instead of a topple we get the customer ID. Okay. Now we need to apply the same change in a couple more places So let's search for get or create in this project.

We have two more references one is in the serializers module This is the same method of create order serializer. So Here we also call get And we get a customer object. Good. And there is one more instance. In the views module, here, in the me action, once again, we call get and get a customer object. Good. Next, we're going to talk about creating custom signals.

Creating Custom Signals:

So you learn that every model in Django fires a bunch of signals at different times. Now we can also create our own custom signals. For example, every time we receive an order in the store app, we can fire a signal like order created. With this, other apps that are interested in this event can subscribe to this signal and get notified. So let's see how this works. Here in the store app, let's look at the signals module one more time. This function that we defined in the previous lesson is called signal handler. Now in this module, we can also define a signal, but it's best to separate our signals from handlers. So in this app, in the store app, I'm gonna add a new folder called signals.

And then I'm gonna move signals to this module and rename it to handlers. So now in this module, we only have signal handlers. Now there's an issue here. The models module is not in the current folder, so we need to type the full path, store.models. Good. Now, we need to go to the apps module and load store.signals.handlers. Good. Now, in this folder, we're going to add an init module. And this is where we define our signals. So from django.dispatch, we import the signal class. and then we create a signal object. We call it order created like this. So a signal is simply an instance of the signal class. Now we need to fire the signal when an order is created. And for that, we're going to go where?

To create order serializer. So here in the save method, after we create our order and remove the shopping cart, this is where we're going to fire our signal. So order on the line Created. It's not being picked up. So on the top, From the signals module, we're going to import order create it now back over here. Now this object has a couple of methods for sending a signal. We have sand and sand robust. The difference is that with sand if one of the receivers fails and throws an exception, the other receivers are not notified. So here I'm going to use sand robust. Now, here we need to pass an argument called sender. This is the class that is sending this signal. So let's go back to our handlers module.

Earlier, when we defined this function, we gave it two parameters, sender and keyword arguments. So now when calling a handler, we need to supply the sender argument, okay? So here we need to get the current class from self.class. This is a magic attribute that returns the class of the current instance. Now, optionally, we can supply additional data with our signal. In this case, the order that was created. So we set order to order and pass it as a keyword argument. Good. So now in the store app, every time we create an order, we fire this signal. With this in place, we can go to the core app and receive the signal exactly like what we did in the previous lesson. So here in the core app, let's add a new folder called signals.

then we create the handlers module. Now, from store that signals we're going to import order created. Next, we define a handler function. We can call it whatever we want. I'm going to call it on order creator then we give it two parameters, sender and keyword arguments, exactly like before. Next, we need to decorate it with the receiver decorator. So from Django, the dispatch, we import receiver and apply it over here. Now, which signal do we want to receive? Order created. Good. Now, for now, let's just print the order that we receive with this signal. So, print keyword arguments of order. This is the keyword argument that we passed when firing this signal. Okay? Now, the final step. We need to load this module when this app is ready.

So, we go to the apps module and override the ready method. Now here we import core dot signals dot handlers. Here at the orders endpoint, I want to create a new order. But looks like the form for posting a cart ID is missing. Because earlier I made a mistake. In the order view set class where I specified our HTTP methods, I forgot to include the post method. So now refresh, Here's a form. So I'm going to pass this cart ID that I created before recording this video. Post no errors here. I hear the terminal look, this is the order object that we received in the core. So using signals, we can decouple our apps and prevent them from stepping on each other's toes.

So in our current implementation, the store app fires the order created event, and the core app simply gets the order and prints it on a terminal. The interesting part about this model is that the store app doesn't care what happens in the core app after this order is created. Similarly, we can have many other apps that are interested in this event and every time an order is created, those apps will get notified and do something that is relevant in their domain. This is the beauty of signals.