Getting started:

Welcome back to the ultimate Django course. In the first two parts of this series, we built a complete restful API for an imaginary online store. In this part, we're going to continue our journey and talk about advanced concepts such as uploading files, sending emails, running background tasks, automated testing, performance testing, caching, and much, much more. So by the end of this course, you're going to be able to build production grade Django applications with confidence. To take this course you should have watched the first two parts because I've covered so many concepts and shortcuts that I'm not going to repeat in this part. Otherwise, you will feel this course is too advanced for you. I've been working really hard on this series and I'm so excited to be your instructor once again.

Now, let's jump in and get started.

Setting Up the Project:

Alright, let's talk about setting up our project. This lesson is super important, so please follow all the instructions I'm going to show you, even though I know you already have the project on your machine. So first, open data grip, or your favorite database management tool, and create a new database called storefront 3. So we're going to create a separate database for this part of the course. Alright, now look below this video, you will find a zip file that contains all the code we're going to write in this course, just like before. so we're going to go to the code folder. So here we have the code for each section. We're going to go to the first section and open the start folder. So here's our project.

This is almost what we had at the end of the second part but there is something extra here and i'm going to talk about that in a minute. So let's open this with vs Code. Now the first thing you need to do is changing the database password. So let's go to our database settings. Now over here, change the password to whatever you use to log into mysql on your machine. Now we need to install our project dependencies. So, we open a terminal window and run pipenv install. Alright, our project dependencies are successfully installed. Now we need to activate our virtual environment. And for that, we type pipenv shell. Beautiful. Now, we need to tell VS code to use the python interpreter installed in this virtual environment.

to do that, first we have to grab this path. So let's copy this. Then we click over here. Now on the top, we select enter, interpret the path, and paste the path that we just copied. Now at the end, we need to replace activate with python all right now the next step is to run our migrations so python manage.py migrate all right All our database tables are created, beautiful. Now we need to populate these tables. Now in the previous part, I gave you a separate SQL file that you opened in DataGrip or your favorite database management tool. In this part, we're going to take a different approach. So I've created a custom command for populating the database. So if you run python manage.py without any arguments, we can see all commands that are available to us.

So over here, you can see that in the store app, we have a command called seeddb. Let me show you how I've implemented this, it's really simple. So here in the store app, we have a new folder called management, and inside this folder we have another folder called commands. This is where we add our custom commands. So Django automatically looks for any custom commands inside this folder. So over here we have a file called seed underline db that indicates the name of our custom command. Now in this file, we have a class that extends base command that is defined in django.core.management.base. Now in this class, we set the help attribute to this description, and we override the handle method. So first we print populating the database, then we get the current directory, and using os.path.join, join it with seed.sql.

So in this folder, we have our seed file that I gave you separately in the previous part of the course. Now it's part of our project, and using this custom command, we're going to execute this sql file. So, we compute the full path to our sql file, and then using the path class, we read the entire text in this file, which gives us our sql instructions. Then we open a connection to our database and execute this sql statement. Very simple. So, back to the terminal, let's run python manage.py seed underline db. Alright, our database is populated, Now we need to create a super user, so python manage.py create super user, I'm going to call this admin, give it some fake email, put some password, alright, we have an admin user, the final step is to run our web server.

So python manage.py run server. Alright, so let's head over to this address, here's what we get, because currently we don't have a home page. But we're going to add that later in this course. So let's go to store slash collections and make sure our collections exist. There you go. So we have all these collections that we had in the previous part of the course. We can also go to store slash products. So our product table is populated with a thousand products and here we have pagination that we implemented in the previous part. Beautiful. Now the final thing is login into admin, so let's go to the admin, and login with our admin account, beautiful. So everything is working as expected, but if you encounter any errors on your machine, please use our forums, that is forum at codewithmosh.com, unfortunately I'm not able to troubleshoot issues on your machine, you need to help each other or use google or stack overflow to troubleshoot errors.

Uploading Files:

Introduction:

Welcome back to the ultimate Django course. In this section, we're going to talk about uploading files. So together, we'll build a RESTful API that client apps can call for uploading images. It's going to be a lot of fun, so let's jump in and get started.

Managing Media Files:

the first thing we need to do to implement the upload feature is deciding where these user uploaded files should be stored and how we're going to serve them. So let's create a new folder here in the root called media. You could also call it upload, it doesn't really matter. But in Django, media refers to user uploaded files. It's a term you see often in Django documentation or blog posts. So let's imagine all user uploaded files are going to end up in this folder. Now we need to tell Django about this folder. So let's go to our settings module, here we have a setting called static url, static in django world refers to static files of our application like css, javascript, images and so on.

These are the files that are bundled with our application. Media refers to user uploaded files. So right here, because these concepts are closely related, we should define a setting called media url, and this is the end point at which we want to expose our media or user uploaded files. So we set it to slash media slash. Make sure to end it with a forward slash. Now we should also tell Django where these media files are stored in the file system. And for that we're going to set media root to the full path to the media directory. So first we need to go on the top and import the OS module, then back to where we were, here we're going to call os.path.join In this module we have a setting or a variable called base dir that represents the current directory, we're going to get that and append media to it.

Now to test this, I'm going to add an image to our media directory, let's imagine this image is uploaded by a user. So let's see if we can successfully serve it. So I'm going to grab this image, it's a dog image, you can put anything, it doesn't really matter. Now we need to go to our urls module and define a route for serving these media files. So, we go to the urls module of the storefront folder, and on the top, from django.conf, we import the settings object, also from django.conf.urls.static, we import the static function. Now, over here, we're going to concatenate this list with the result of the static function. So we call this function and give it two arguments. media url and media root.

So the first argument is settings.media underline url, and the second argument is a keyword argument called document underline root, we set that to settings.media underline root. Now, you don't really need to memorize any of these things, you can always get back to this lesson or find this stuff on Django documentation. So with this new line, we're telling Django that we want to expose an endpoint that is defined over here, that is slash media, And any requests that go to this endpoint should be routed to the file system at this address. Okay? Now this strategy is good for development. For production, we're going to take a different strategy which we'll talk about later in the future. So a better way to write this code is like this.

Instead of modifying this list right here, we're going to write a condition. If settings.debug is true, then we're going to take URL patterns and add something to it. this is better. So in development, debug is turned on, but in production it's turned off. Now let's test our implementation. So, let's go to the browser, and head over to media.jpg, there you go. So, we can successfully serve media files, starting from the next lesson, we'll implement the upload feature.

Adding Images to Products:

So let's imagine that each product can have zero or more images. So we need to make a small change to our data model and add a one to many relationship between product and a new model called product image. So let's go to the models module of the store app. Here's our product model. Right after this class, I'm going to define a new class. Technically we could define it anywhere in this file after the product class, but because this class is highly related to the product class, I would prefer to define it right after it. So class product image, which should extend models.model. Now in this class we need two attributes. The first one is our product, which is a foreign key to the product model.

Now here I'm going to set undelete to models.cascade. So if we delete a product, all its images are also deleted. Now I also want to set related name to images and As you know, this makes our queries a little bit cleaner and easier to read. Now the second field we need is the actual image. So image is models image field. So here we have image field, we also have file field. The difference between these two types is that image field validates the uploaded image and ensures it's a valid image. It also has properties that only apply to images like width and height. So if you want to allow the user to upload images, you should use image fields, but for any other types of files like

document, PDFs and so on, you should use file field. Now here we need to set upload to to a path like store slash images. Now there are a couple of things I need to clarify here. The first thing is that with this implementation, we are not going to store images in the database because that's going to make our database super big and our queries are going to be also slow. So for performance reasons, we're going to store images in our file system and their path in the database okay now Now, this path that we have specified here is relative to our media root. So in the previous lesson, we set media root to the media folder over here. So when the user uploads an image for a product, that image is going to be stored here inside a folder called store slash images.

Now, because we have used image field here, we need to add a library called pillow. So here in the terminal, we run pipenv install pillow. Alright, lovely. Now the final step is to create a migration and run it. So python manage.py make migrations, I don't know why my autocomplete is not working, but anyhow, so let's run this migration. Lovely. So our data model is ready, next we're going to build an API for uploading images.

Building an API to Upload Images:

in this lesson we're going to build an endpoint like this, products slash one slash images slash one. So here we have nested resources and as you know we're going to use a nested router to implement this endpoint. A quick question for you, what building blocks do we need for building an API? We talked about this several times before. We need a serializer, a view, and a router. So let's implement this one by one. So first we go to the serializers module of the store app. and create a new class called product image serializer which extends serializers.model serializer. Here we define a meta class. We set the model to product image and fields to id and image. So we don't want to return the product id here because it's already available in the url.

So look, when we hit let's say products slash one slash images We don't want to return the product id in the response because it's already available in the url. Okay? So that is why we're only returning these two fields. So this is our serializer. They expect it to implement a view. So we go to the views module of the store app and define a new class called product image view set. This extends model view set. Now here we set serializer class to product image serializer. and query set to product image dot objects dot all. Now there's a problem here, we don't want to return all images or all product images in the database, we only want to return the images for a particular product.

So instead of setting this attribute, we're going to override the get query set method, and here we return product image dot objects dot filter we set product id to, here we need to get the product id from the url, how do we do that? We have done this several times before, self. keyword argument of, now what is the name of this url parameter? Well, you know that, in nested routes, here we have two url parameters, the first one is going to be called product underline pk, and the second one is going to be called pk, okay? So, back over here, we're going to grab product underline pk. So here's our view set, now let's define a route. So we go to the urls module of the store app, now here we have a router for our products endpoint, earlier we registered the reviews endpoint over here, so let's register a new endpoint.

We say products router dot register images, we bind this to views dot product image view set, and because we have overridden the get query set method, here we have to set the base name. So we set it to product dash images. Nothing new so far. So let's test our implementation. So let's go to store slash products slash one slash images. Alright, so here in this html form we can upload an image. So I'm going to upload our dog image. Alright, we get an error saying column product id cannot be null. Can you tell why we're getting this error? At this level, you should be able to answer this question. You should be able to troubleshoot this on your own. So pause the video and think about it for a minute.

Here's the answer. So, let's get back here. In this form, we're only supplying the image, not the product id. Or if you look at the raw data, in this JSON object, we only have the image property, not the product id. but we don't want to include the product id in this object or in this form because it's already available in the url. So when creating a product image object we should extract the product id from the url and use it to save the product image. So back to our view set, in this view set we have access to our url parameters just like how we used product pk to filter product images. So we're going to extract product pk one more time and using a context object, we're going to pass that to our serializer.

Then, in the serializer, we're going to grab that from the context and use it to create a product image object. Pretty simple. So, here we define get serializer context and we return a dictionary with one key value pair, product id, and we set it to self, that keyword argument of product on the line pk. Good. Now, let's go to our serializer. In this class, we should override the create method. So, we get the product id from self that context of product ID. And then, instead of relying on the default implementation, we're going to explicitly create the product image object. So, we say product image that objects that create first we set product id to this product ID. and then we pass all the validated data stuff.

As simple as that. So let's test our implementation. Back to the browser, I'm going to select our dog image one more time, post, beautiful. Now we have this image stored at this address, and the ID of this image is 1. Now look at the media folder, here we have store slash images, and in this folder we have two images. We have two because I uploaded one image, before recording this video. So the same image uploaded twice, and look, the second time I uploaded this image, Django automatically appended these characters to the file name, so we don't end up overwriting an existing file.

Returning Images from the API:

implemented the uploading feature. Now let's say the client wants to render the list of products. So it's going to hit the products endpoint. But look, currently we don't have the images here and we don't want the client to repeatedly hit separate endpoints for getting the images of each product. That's way too expensive and slow. So when returning the list of products, we want to return their images as well. And that's very easy to implement. So first we go to our product serializer class Here are the fields that we're currently returning. We're going to add a new field called images and We define it right here. So images we set it to product image serializer and Because the product can have multiple images here.

We said many to true Now we have an error because product image serializer is defined after product serializer. So we have to go to this class product image serializer grab this code and put it right before product serializer. So cut and then we go to product serializer and paste it right here. Like this. Okay? So we set many to true. We should also set read only to true because when creating a product we don't want to pass multiple images. We only want to pass properties related to a product object. Now back to the browser. Let's refresh. So each product now has images property. Now the first product we get here is product number 648. So let's go to this product. Just to test this, let's add an image real quick.

Good. So let's go to the products endpoint one more time. Now look, this product has one image with this id and here's the address of this image. So if you click it, we can see our dog. Beautiful. Now there's one issue here. Let's open Django debug toolbar and look at the SQL tab. So we have 14 queries and many of these queries, as you can see, are duplicate. Because for each product, Django is now going to the database to fetch its images. So we need to eager load our products with their images. And that's super easy. We go to our product view set. This is where we are fetching our products. So here we're going to call prefetch related images. Okay, now let's test it one more time, refresh, this time we only have 6 queries.

Next we're going to talk about validation.

Validating Uploaded Files:

Alright, let's talk about validation. So here in the product image class, look at the image field. Earlier I told you that. This image field validates the uploaded file and raises an error if it's not a valid image. So if you try to upload a document or a PDF, we're going to get an error. And that's why we had to install pillow. Because pillow is an image processing library for Python. So under the hood, this image field class uses pillow for validating the incoming image. Okay? But what about the file size? Well, currently there is no built in validation for that, so we need to create it from scratch, but it's really easy. So here in the store app, let's add a new file called validators.py, here we're going to define a validator function like this, we can call it validate file size, give it a file, now here we define a variable called max size in kilobytes, I'm going to say to 15, because the doc image that I have is 80 kilobytes,

So with this we can see the error when uploading the dog image. Now here we can say if file.size is greater than max size in kilobytes times 1024, then we need to raise a validation error. So we go on the top and from django.core.exceptions we import the validation error class and over here we raise a validation error object and here we can pass a format string like this. we can say files cannot be larger than if we pass an argument, which is going to be max size in kilobytes, followed by kb. Okay? So we have a validator function. Now, we're going to go back to our model and over here, we're going to set a keyword argument called validators. We can set it to a list of one or more validators.

So here I'm going to use validate file size. Let's test this, so we're going to go to product slash one slash images and upload our dog image one more time. There you go. Here's our validation error saying files cannot be larger than 50 kilobytes. Beautiful. So this is all about validating the file size. Now in Django we can also validate the file extension and that is useful or actually essential if you're using a file field. So If you're using a file field, perhaps you want to restrict the client to uploading, let's say pdfs or document files. With images we don't have to do that because the image field does that under the hood. So for this demo, let's change this to file field, just for a second.

And use the built in validator for file extensions. So on the top, look, we're importing main value validator from this module. So in this module we have another class called file extension validator. Now, back to our model, I'm going to temporarily remove this validator, so we can upload our doc image, and we don't get an error about the file size. So here we create a file extension validator object, and set allowed extensions to a list of extensions that we support. Let's say pdf, just the word pdf, so we don't need a period, okay? So, let's test this. I'm going to repost this form, Now look, we get a different error saying file extension jpg is not allowed. Allowed extensions are pdf. So this is how validation works.

Now let's remove this and bring back our validator function. So validate file size. And we should change this to image field. So we're done with this step. Next we're going to set up our client application for uploading images.

Setting Up the Client App:

Alright, in this lesson we're going to set up a very basic client for uploading files. So if you look at the zip file that I gave you at the beginning of the course, here in the second section, uploading files, you will find a folder called client app. This is a very basic JavaScript project for uploading files, but you don't need to know JavaScript to take this lesson. Just follow along with me, it's really easy to set up this project. I've created this project for full stack developers who want to see how everything comes together, both the client side and the server side. So, We're going to open this in a new instance of VS Code. So here I have a new instance of VS Code.

Now I'm going to drag and drop this right here. Alright, to run this project we need Node. I believe these days everybody has Node on their machine, but if not, head over to Node.js.org and download the latest version. So download and install Node is really easy. Now back to VS Code, we open a terminal window. Let's see what version of Node I'm running on this machine. I'm running Node version 17. Now, there are two steps you need to follow to start this project. First we need to install our project dependencies. So we type npm install. Alright, all our project dependencies are installed, but we get a bunch of warnings, ignore them, they don't really matter at this point. So, the next step is to run npm start.

This starts a web server at localhost on port 8001. So let's open this in our browser, we get this simple page with an input field for uploading images. But this feature currently doesn't work because there is a problem that we're going to talk about in the next lesson.

Enabling CORS:

Alright, let's see what happens if we try to upload an image using our client app. So I'm going to select our dog image, and upload it. We get an error saying, could not reach the server. Why is this happening? Well, this is a common security measure that is implemented in all browsers. It's called CORS, which is short for cross origin resource sharing. I know, it's a mouthful. But in practical terms, this policy prevents an app hosted on one domain from sending a request to an app hosted on another domain. So in this case, we have this client app hosted on port 8001, trying to send a request to our app hosted on port 8000. So even though both these apps are hosted on the same machine, Chrome sees them hosted on different domains.

And that's where the course policy kicks in and prevents this request from reaching the server. So to solve this problem, we need to go to our backend and do a bit of configuration. So requests coming from this address are not blocked. And for that we're going to use a library called Django course headers. So if you google this term, you'll find this GitHub page. On this page you can find the installation instructions. It's very easy. First we install Django course headers, then we add it in the list of install apps, next we register a middleware, and as you can see here, this middleware should be placed as high as possible. And finally, Using one of these settings, we specify what origins or what domains can send requests to our backend.

So we can use this setting to specify a list of IP addresses or domain names. We can use this other setting to use a regular expression or we can use this setting to allow all origins, which is not a good idea. So let's go ahead and install this library. Back to VS Code, pipenv install django-cores-headers. Alright, we're done with the first step, now we need to go to the list of installed apps, and register an app called cores headers. We can put it anywhere, it doesn't really matter, but I prefer to put all these Django apps, these are the built in apps, I prefer to put them first, then third party apps, and finally our own apps. Next we need to register a middleware.

And for that, it's best to get back to the documentation and copy this long string, so we don't have any typos. So here in the list of our middleware, I'm going to add this first. Okay? So we're done with this step, finally we need to specify what origins can send requests to our backend. So we have a setting for that called cores allowed origins. So let's copy this, now somewhere in this settings module, it doesn't really matter where, but I prefer to put it right here. after internal ips because what we said here is kind of similar. So we set this to a list of strings specifying domain names or ip addresses. So HTTP localhost port 8001 and i prefer to use the numeric representation using an ip address.

127.001 represents localhost. So we're done here. Now let's try uploading our file. So back to our client app. Upload. we get our validation error because this file is larger than 50 kilobytes. So let's go to our validate file size function and increase the limit to 500 kilobytes and upload this one more time. So we successfully uploaded the image and if you notice, we had a beautiful progress bar appearing for a second. Now if you want to see it more clearly, you can upload a larger file and also apply throttling. Let me show you what I mean. So right click somewhere and go to inspect, this is chrome developer tools, you're probably familiar with it, let's go to the network tab, over here we can apply throttling to simulate a slow connection.

So I'm going to select slow 3G, and upload this file one more time. There you go. Beautiful. Now one more thing I want to show you here is the list of requests sent to the backend. So I'm going to undock this, into a separate window so you can see clearly. So look, the moment I pressed the upload button, we had two requests sent to the server. The first one is an options request, the second one is a post request for uploading the file. Let's inspect the options request here on the headers tab, let's look at the response headers. So here we have a header called access control allow origin, and this is set to local host port 8001. This is coming from the library that we just installed.

So we put this header in the response and return it to the client. The client will read this header and find out that it's allowed to send a request to our backend. That is why we have another request, which is a post request for uploading the file. So this is how course works. Now before we move on, let's make sure to remove throttling because everything we do in Chrome going forward will be slow. So let's remove throttling. Good. Alright, we are done. Let's move on to the next lesson.

Managing Images in the Admin:

the last thing we're going to talk about in this section is managing product images in the admin interface. So let's go to the list of products and pick one of our products. Look, there's currently no way to manage our product images here. So similar to tags, you want to show our product images right here. Now do you remember how we implemented this section? Let's do a quick refresher because we did this in the first part of the course. So back to vs Code, let's search for product admin, so we have two instances, we have one class that is defined in the store app, and another one that is defined in the core app. The first one is the reusable part, so any project that uses the store app is going to use that admin class.

The other one that is defined in the core app is the part that is very specific to this project. It basically combines features from different apps. So let's take a quick look here, look, in this class we have set inlines to a list, and in this list we have tag inline, which is defined right here. So to show the product images, first we need to define an inline class, we can call that product image inline, and then we need to register it here. But I have a question for you. Where do you think is the right place to define this inline class? We don't want to define it here, because this is the core app, this is very specific to this project. We should define it in the store app, because the product image class is defined in the store app, so any project

that uses this app should also have the capability to manage product images in the admin interface. So let's go to the product admin class. Right before the product admin class, we're going to define an inline class. So class product image inline. This should extend admin tabular inline. Now over here, we set model to product image. Okay. now we need to register this in the admin class, so right here, we define a new attribute called inlines, and set it to a list, and here we add product image inline, now there's a problem here, let's go back to our custom product admin, look, this class extends our product admin, and here we are overwriting the inlines attribute, so whatever we set there is going to be lost here, so in the core app,

we should also add product image inline in this list. Now let's see what happens, so back to the admin, refresh, okay, in this section we have the list of product images, but currently we only see the path. Wouldn't that be nicer if we could see a thumbnail of each image? Let's see how we can implement this. Back to our product image inline class, here we need to set an attribute called read only fields to a list, and in this list we add thumbnail. Now thumbnail is not one of the fields of the product image class, so we need to define it as a method here. So def thumbnail, we give it two parameters. The first one is self, because this is a method of this class.

The second parameter is an instance of the product image class. So we're going to take that product image and convert it to an HTML image. Now, a product can have 0, or more images, so we need to account for that here. So we can say if instance.image.name is not empty, remember, this instance has two fields, product and image. So we are referencing that image field here. So if we do have an image, we're going to return an HTML image tag like this. Now if you don't know much HTML, don't worry, just follow along, make sure to type everything as I show you. So here we return an image tag like this, and give it a source attribute. Now we should set the source of this image to the URL of our image.

So we're going to convert this to a formatted string, and here we're going to render instance.image.url. Okay? Now otherwise, if we don't have an image, we're just going to return an empty string. Let's see what happens. So refresh, alright, we don't see the actual image, we see the image tag rendered as a string. So to solve that, we need to import a function from Django. So on the top, look, we already have that function here. From Django.utils.html, we are importing the format.html function. So, back to where we were, we're going to pass this as an argument to the format.html function. So, format.html like this. Okay? Take a look. Alright. Now we have the image, but this image is way too big. We want to show a thumbnail.

So this is where we use CSS. So using CSS, we apply a bunch of attributes to this image. So back to our HTML tag, here we are setting the source attribute. We should also set the class attribute to a class called thumbnail. So this is a CSS class that we're going to define in a minute. So save the changes. Now, back to our project. Here in the store app, we're going to create a new folder called static. This is a special folder for Django. So when our project starts, Django web server goes through every installed app, and it will collect all the static files from this folder. So in this folder, we're going to add a file called styles.css. So this is a static asset.

It's part of our project. Other static assets can be images and javascript files. We'll talk about them later in the course. So here we're going to define a CSS class like this. Period. thumbnail, with braces, and inside braces we add a bunch of key value pairs. So we set width to 100px and height to 100px. Make sure to type it exactly as I'm showing you here. Now, back to the admin, refresh, we don't see any changes, because this admin page doesn't know that we need to import a CSS file on this page. So we need to go to our product admin class, the original one that we have defined, in the store app, so in this class we're going to define a media class, which is an inner class, so after the last method, we're going to define an inner class called media, just like we have a meta class, here we have a media class, now with this media class, we can specify the static assets that should be loaded on the product admin page, so we can load css or javascript files, so here we set css to a dictionary,

in this dictionary we add one key value pair, the key is going to be all, and I'll explain what that means in a second, the value is going to be a list of CSS files, so here we add styles.css, now what is all? Well, in CSS we have the concept of media type, so we have screen, and with this we can apply styles that only apply to screens, we also have print, with this we can define styles that will only be applied when printing a page, but when we use all these styles when we apply it everywhere. So this is how we can load our style sheet or our CSS file on the product admin page. Now let's take a look. Refresh, there you go.

Good. But if you pay close attention, you can see that our dog image is a little bit squashed. It's out of proportion. Because it's a vertical rectangular shape and we have converted it to a square that is 100 pixels wide. To solve this problem, we need to go back to our CSS class and apply another property here called object fit, we set it to cover. Now if you want to learn more about CSS, I have a comprehensive three part course on HTML and CSS, I cover all these concepts in detail. So, let's move on, now refresh, okay, the image is looking good, now there's one problem in this implementation. I told you that, when our project starts, Django web server is going to look at the static folder of every installed app, and it's going to collect their files.

that means, if we have another file with the same name in another app, one of these files is going to overwrite the other file. So this is why we need to namespace these files. So inside the static folder, we create a new folder called store, and then we move our CSS file right here. Okay? Now with this change, we need to go back to our product admin class, and in this media class, we need to change the path for a style sheet, so we prefix it with store. Like this. Let's make sure everything still works. Beautiful. So this is how we can manage product images in the admin interface.

Sending emails:

Introduction:

Welcome back to another section of the ultimate Django course. In this section, we'll be talking about sending emails from your Django applications. So, let's jump right in and get started.

Setting up a Fake SMTP Server:

To send emails, we need an SMTP server. Now in case you don't know, SMTP is short for simple mail transfer protocol. So an SMTP server is software that knows how to send and receive emails. Now in the real world, we have to use a real SMTP server, and quite often that costs us money. We'll talk about that later when we get to production. For now, as part of our development, we need to set up a fake SMTP server. The one that I personally love is called SMTP for Dev. So if you Google this, you will find this GitHub page. Now let's look at the installation instructions. Right here. So there are a few ways to install SMTP4DEV. If you have .NET Core version 3.1 or greater, you can install it using this command and then start it like this.

If you don't have .NET, you can run it using Docker and this is my preferred approach. If you don't know what Docker is, Docker is a platform for shipping and running software. It's one of those things that I believe everybody should know these days, and quite often when you go for job interviews, it's one of those skills that they want you to know. If you want to learn Docker, I have a comprehensive 6 hour course where I teach you everything about Docker so you understand what it is and how it really works. So using Docker, we can run SMTP for dev using a single command. So if you don't have Docker on your machine, head over to docker.com slash product slash docker dash desktop

and download the latest version for your operating system. Now, back to this page, I'm going to copy this command, back to VS Code, I've opened a new terminal window, let's run this. So we don't have this image SMTP4Dev on my machine, and now Docker is going to download this image from Docker Hub, which is a repository for Docker images. So just like we have GitHub for source code, we have Docker Hub for Docker images. So now we have an SMTP server on this machine, which we're going to use in the next lesson for sending emails. But the beautiful thing about this fake SMTP server is that it also gives us an administrative panel. So if you go to localhost port 3000, we can see this administrative panel.

This is like your outlook or mail client. So as we send emails in our application, those emails will appear here. So this is a great way to test sending emails. So go ahead and set this up and I will see you in the next lesson.

Configuring the Email Backend:

Alright, now we need to configure our email backend. An email backend is essentially an engine that is responsible for sending emails. So in Django we have a few different email backends, we have an SMTP backend, which is the default one, and this backend uses an SMTP server to send emails. We also have a console backend, so if we use this backend, the emails that we send will appear in the console or terminal window. We also have another backend for writing emails to a file, we have another backend for writing emails to local memory, and a dummy backend that does nothing. Now out of these, most of the time we use the SMTP backend, but sometimes you might want to use the console backend.

So let me show you how you can configure the backend. So here in the settings module, we set email underline backend to django.core.mail.backend and of course you don't have to memorize any of these things, you can always find this in the documentation. So backends. Now here we have a module for smtp backend, and in this module we have a class called email backend. If you want to use the console backend you just change the name of the module. We have another module for writing emails to a file. So in this case we are going to use smtp backend which is the default one. So technically we don't even have to set this setting, but I'm going to put it here just for reference.

Now when using an smtp server, we should also set a few other settings. so we should set email underline host to the address or smtp server, in this case localhost, because this is where we're running our smtp server. We should also set our host user, now this fake smtp server doesn't have a user and password, so we're going to leave that empty. The next one is host password, again, I'm typing this for clarity, in this case we don't have to set the settings. Now in production, we don't want to store the password here in the setting file, because that password is going to get checked into git and everyone can find that. So we're going to use environment variables to load the password.

Again, we'll talk about that in the future when we get to production. Now, we should also set the port. By default, SMTP servers run on port 25, but because this is a fake smtp server, it runs on port 25 25 now optionally we can also set default from email to, let's say, from at moshpy.com. So this is how we can configure our email backend. Next I'm going to show you how to send emails in Django.

Sending Emails:

With everything in place, now let's see how we can send emails. So let's go to our playground app and open the views module. So in this view, before rendering this template, let's send an email. First we go on the top and from django.core.mail, we import a couple of mail functions. So look, we have a few functions here, mail admins for sending email to admins. We also have mail managers, as well as send mail and send mass mail. The difference between the last two functions is that sendMaskMail opens a single connection and sends all these emails and then closes the connection, but sendMail opens a new connection for each email. So for performance reasons, if you have a large number of emails or if you have honestly more than one email, you should always use sendMaskMail.

So in this lesson, I'm going to import sendMail and mailAdmins. Now to send an email, we call sendMail and give it a bunch of arguments. The first one is the subject, the second one is the message, the third one is the from email, and with this we can overwrite the default from email that we configured in the previous lesson, so we can say info at moshpy.com. The fourth argument is the list of recipients. So here we pass a bunch of email addresses, let's say bob at moshpy.com. Now, this function will throw an exception if an attacker tries to modify email headers and control the to and from fields. So you'll probably come across those fake emails that pretend to be from Google or YouTube or your bank, this is the situation I'm talking about.

So to prevent such attacks, first we need to import a class called bad header error, and then we should always wrap this call inside a try catch block. So let me say try, and then accept bad header error. Now in this case we should return an error to the client and say bad header error or whatever, in this lesson I'm just going to pass, otherwise we should return a message and say the email was successfully sent. So this is how we can send emails. Now let's test this, so here's our playground slash hello endpoint, I'm going to refresh, okay, now let's look at our smtp for dev admin panel, which is on localhost for 3000, there you go, we have an email, and you can see the subject, and you can see the body, pretty straightforward.

Now let's talk about sending email to site admins. So this time, instead of this function, I'm going to call mail admins. Now the signature of this function is a little bit different. First we specify the subject, then the message. We can also supply an HTML message, and this is also available in the send mail function as well. So what we put here can be HTML content, but what we put in the second argument is just plain text content. In this case I'm going to use the same message. Now for this to work, we need to configure our site admins. So this function is not going to go in your database, it's not going to look at the user table and find the site admins.

So we need to go to our settings module, and here we define our site admins using a key called admins. We set this to a list of tuples. So here's the first one. Each tuple should have two values, a name and an email. So we can say mosh, and the email is Let's say admin at moshpy.com. So let's test this as well. Back over here, refresh, and here's the second email sent to admin at moshpy.com. Now if you pay close attention, the second email is a little bit different from the first email. Look at the message, compare it with the first email. So this is plain text content, but in the second email we have HTML content. So if we go to the parts tab,

look here we have multiple parts, we have plain text content as well as HTML content. So, male clients that support HTML will show HTML content, and other clients that don't support HTML will display text content. Now honestly, I don't know if this is still applicable these days, I think all male clients support HTML content, but who knows, maybe there is a device with limited display capabilities, in that case, you want to send both text and HTML emails.

Attaching Files:

Now, what if you want to attach something to our email? Well, for that we need to use the email message class. So all these functions that we talked about in the previous lesson, like send mail or mail admins, internally they use the email message class. So in this lesson, instead of relying on these functions, I'm going to import the email message class and create an email message object. So once again here we can set the subject, we can also set the message, the from email, the list of recipients, exactly like before, so let's say john at moshpy.com, so we create an email message object, now this message has a method called attach file, so we call this method and give it a path relative to our project directory, so let me show you what I mean, so here's our project, you start from here and then type the path to a file, so in the playground app,

let's create a new folder called static. And here i'm going to create another folder called images. And then i'm going to drag and drop my dog image over here. So there you go all right now here we type playground slash static slash images slash dog the jpg. Okay? Now finally to send this, we type message dot send. So the send mail function is basically a shortcut to all this, okay? Now let's test this, so back in the browser, refresh, alright, now here we have a new email with this subject, and look we have an attachment, and in this email we have our dog image. Let's verify that it works, there you go, beautiful. So if you want to have more control over your emails, like if you want to attach something, or if you want to use BCC or CC features, then you have to use the email message class.

Sending Templated Emails:

So in Django we can send emails either using the email message class or one of the shortcut functions. Now in both examples I've shown you so far, our email message was just one word. In reality, that's never the case. Quite often our emails have long text and we also want to dynamically insert data into the email body. To do that we use a library called Django templated mail. It's basically a thin wrapper around Django's mailing functionality, but it allows us to store our email messages in template files. Let me show you how it works. So first, we install django-templated-mail Alright, now we need to create an email template. So, back to our playground app, here in the templates folder, we can create a new folder called emails, and here we add a template called hello.html.

Now in this template file, we can supply the subject of our email as well as its text and HTML body. So first we define a block using this syntax and we call that block subject. Now whenever we open a block, we should also close it. So we close it like this, okay? Now in between these tags, we can type the subject of our email. So this is a long subject. Obviously, we don't want to hard code these long strings in our code. that just makes our code really hard to maintain. So this is the benefit of these template files. Now similarly we have other blocks for creating the text and HTML body. So we can create this block, text body, and of course we should end it like this, and we can also supply another block for HTML content, so HTML body, and end it like this.

Now this is optional, you can totally leave out the text body, so let's work on html body, here we can type anything we want, we can add an html h1 element and say hello, and say my name is, now to dynamically insert content, we use the same syntax that we have in Django templates. So we're going to pass a variable called name, and we'll set the value in our code. Now let me show you something, if I save the changes, VS code completely messes up our template, so let me show you a workaround. i'm going to undo the changes. Now, let's go to the file menu and enable auto save. So, with this, as we make changes, VS Code automatically saves them, but it doesn't mess up the formatting of our template.

However, if i explicitly save this file, again, everything gets screwed up. Okay? So that's a little tip. Now, let's go back to our view. Now, instead of this email message, we're going to use another class defined in this library that we just installed. So from templated email.mail, we import base email message. Now this class extends the email message class in Django, so it has all these features you have learned, for example, it has a method for attaching files, sending emails and so on. But the constructor is a little bit different, so it doesn't have these arguments. So let's see how we can use it. So I'm going to create a message, base email message, And here we set a couple of keyword arguments.

The first one is template name, which we set to hello.html. And the second one is context. This is a context object that we use to pass data to our template. So we can set this to a dictionary with one key value pair, the key is name. That's the variable that we expect in our template, right? And we can set the value here. So now we have a message and then we can send it. Now once again, as a best practice, we should always namespace our templates. Now I just realized that I made a mistake, so this template file is actually part of the emails folder, so we need to type the relative path, so emails forward slash hello.html, okay? Now, let's test this, so back over here, refresh, alright, we got an error saying send missing one required positional argument, and that is two.

So, the send method here is a little bit different, here we should pass a list of recipients, so let's say john at moshpy.com. ok, refresh, No errors. Now take a look, so this is our long subject and in the email body we have html content so we have a heading and here's my name dynamically rendered in the email template.

Running Background Tasks:

Introduction:

Welcome back to another section of the ultimate Django course. In this section, we'll be talking about running background tasks in Django applications. So let's jump in and get started.

Introduction to Celery:

In almost every application, we have resource intensive tasks like processing images and videos, generating reports, sending emails, running machine learning models, and so on. Now, we don't want to run these tasks inside the process that runs our application, because if that process is busy, it can't continue responding to client requests. So we should keep this process as free as possible, and anything else that takes time, we should offload it to another process. In other words, we should run it in the background. So here is a real example. When a user uploads a video, we don't want to process that video inside the main application process and have the user wait until we're done. Instead, we're going to kick off the video processing task in the background and quickly get back to the user and say, hey, we're processing your video.

When we're done, we can send the user a notification and say, your video is processed and ready. Okay, great. So how can we do this? That's where we use Celery. No, not the vegetable, but the tool that we can get from celeryproject.org. With Celery, we can start several workers to execute tasks in the background. So whenever we want to execute a long running task, we send it to a queue that all these workers are watching. Each worker will pick a task from this queue, execute it, and then becomes available for the next task. So with this model, we can execute many tasks in parallel. If our system is overloaded, we can easily scale our application by adding more workers. The beautiful thing about this model is that

these workers don't impact our main application process. So if a task is delayed or fails, our main application process is not affected. So it can continue serving clients. Now with Celery, we can also schedule periodic tasks. For example, we can configure Celery to run a special task every hour or every Monday at 9am. So Celery is awesome, and in this section, you're going to learn how to use it to run one-off or periodic tasks in the background.

Message Brokers:

So you learn that our application communicates with salary workers through a queue. You can think of a queue as a pipe between different applications. So messages go in this queue and get processed in order. Now this message queue is part of some kind of software we call message broker. Now the first time I heard this term, I didn't quite get it because English is my second language. So for my international students, let me make this super simple for you. In English, broker means middleman. So let's say you want to buy insurance. There's so many insurance companies out there. So instead of you talking to each of them and getting a quote, you talk to an insurance broker who knows about these companies and their offerings.

Based on what you're looking for, the broker will connect you with one of these companies, okay? Now in the software world, we have message brokers that play a very similar role. They're responsible for passing messages between applications in a reliable way. So if application A wants to send a message to application B, it uses a message broker. Now if the target application, in this case, application B is unavailable, the broker will keep the message and retry later. So it will guarantee the delivery of messages from A to B. Now, what if the message broker itself becomes unavailable? Well, for that we can set up a cluster of message brokers, so if one broker goes offline, we have other brokers that can route messages from A to B. So in a nutshell, we use message brokers to reliably deliver messages between applications.

And that's why we need a message broker here, so our Django application can reliably pass messages to Celery workers. Now, there's so many different message brokers out there, but the two most popular ones for Django applications are Redis and RapidMQ. Now technically, Redis is not a real message broker. It's an in-memory data store. So we can use it as a database, as a cache, but also as a message broker. RapidMQ on the other hand, is a real enterprise-level message broker. So it has many capabilities that Redis doesn't provide. But of course, that comes with a cost. It's more complex. So in this course, we're going to use Redis because it's pretty easy to set up and later in the course, we'll use it to implement caching, which is an optimization technique we'll talk about later.

So we're going to use Redis both as a message broker and also as a cache. Now for your applications, I also recommend you to start with Redis because it's really easy to get up and running. So don't over engineer, don't assume that you should use RabbitMQ right from the get go because it's more powerful. Start with Redis and if you have a valid reason that Redis is not meeting your requirements, you can always easily switch to RabbitMQ later on. So next we're going to talk about installing Redis.

Installing Redis:

Alright, let's talk about installing Redis. Now, we can always get Redis from their website, but the easiest way to run Redis on your machine is using Docker. So earlier I told you to download and install Docker. Don't say, no mosh, I don't want to learn another tool. Docker is something you really need to learn. So assuming you have Docker on your machine, open up a new terminal window and run Docker, run dash D, meaning detached. So we're going to run Redis in the detached mode, meaning in the background. Then using dash P, we're going to specify port mapping. So we're going to run redis inside a docker container. Now this container is technically a process on your machine, just like other processes you're currently running.

But this process is a little bit different. It's inside an isolated environment. As an analogy, you can think of it as a lightweight virtual machine. So we're going to run redis inside a lightweight virtual machine, but for us to access that virtual machine, we need to specify a port mapping. We need to map a port on localhost to port on that virtual machine or container, so we can send traffic to that docker container. So we're going to map port 6379 of localhost to port 6379 of that docker container. And this is the standard port that redis listens on. And finally we specify the image which is redis. So let's go ahead, now on this machine I don't have the redis image, so docker is downloading it from docker Hub.

All right, now redis is running inside a container and over here you can see the container ID. So let's verify it. We type docker ps to list the running containers. Take a look. So let me resize this window so you can see clearly. All right, so we have a container by this id and this is the short version of that container ID. Inside this container or lightweight virtual machine, we are running an image called redis. Now look at the status column, unfortunately it's not visible in my recording window. So over here it says status up 50 seconds ago. So that means this container is currently up and running. Now in the next column we have ports, where we can see port mapping, so let me resize this window one more time.

Alright, here's our port mapping, so we have mapped port 6379 of localhost, so all the zeros mean localhost. So we have mapped this port to port 6379 of this container. So now we're running Redis on this machine, we should also install Redis as a dependency of our Django project. So pipenv install Redis. Alright, we're done with this step, now we're ready to install Celery.

Celery and Windows:

unfortunately celery has dropped support for Windows since version 4. Sorry, my windows friends, but that's the reality. So if you're on Windows, you have to run your Django project inside a Linux environment. And for that, you need to use WSL or Windows subsystem for Linux. So on Windows, you can easily run a Linux environment that's gonna have its own file system. Don't be intimidated. If you have never run Linux, don't worry, it's super easy. So I've created a PDF for you that contains all the instructions and to run your project inside a Linux environment. Just be aware that this is probably going to take about half an hour to one hour or even longer, so before going forward, stop right here, follow these steps, and then move on to the next lesson.

Setting Up Celery:

Now we're ready to install Celery. So open up a new terminal window and run pipenv install Celery. Good. So Celery is installed. Now we need to configure it. And for that we're going to create a new module called Celery. So let's go to our project. Here in the storefront folder, let's add a new file called Celery.py. Now, in this file there are a few things we need to do. First we need to import the os module, and set an environment variable called django settings module. So we say os.environment that set default, as the first argument we pass django underline settings underline module, make sure to spell this properly, and as the second argument, we specify the path to our settings module. So that is storefront dot settings.

So we're setting this environment variable to storefront dot settings. and this references our settings module inside the storefront folder, okay? Now, on the top we should also import the Celery class, so from Celery module we import the Celery class. Next we create a Celery instance, so Celery is Celery, we give it a name, like storefront, next we need to specify where Celery can find configuration variables, so we call Celery, config from object. Now here we pass a string like this, django.conf colon settings. That means we're going to go in this module, django.conf and load the settings object. That is why we have a colon here, but a dot here. Okay? So that's our first argument. The second argument is a keyword argument called namespace.

If we set this to celery, that means all our configuration settings should start with celery. So, let's go to our settings module, and down the end, define a new setting. So, salary underline broker underline url. Again, make sure to spell it properly. We're going to set this to our redis server. So, that is redis colon two forward slashes local host port 6379. That is the port that we specified when running redis inside a docker container. Finally, we add forward slash 1. That is the name of our database. So we can have 1, 2, 3 and so on. So by convention we go with 1. Okay? So we have a setting and now Celery knows how to find all these settings. Okay? Next we call Celery.auto discover tasks.

So in the next lesson we're going to create a task and that task is going to be in a tasks module. So by calling this method, we're instructing Celery to automatically discover all these tasks. Okay? So this is how we configure a celery now we need to load this module inside the init module of the current package. Because otherwise, Python is not going to execute this code. So, in this folder, storefront, we have the init module. We're going to import our celery module right here. So we can say, from period Celery, import Celery. Okay? So if you don't import this module, Python is not going to see this code. Okay? Now the last step. We need to start a celery worker process. So we open a new terminal window and run celery dash A. Next we specify our project, that is storefront.

Next we specify the type of process we want to start, in this case worker. And finally for debugging and testing, we set log level to info. Okay. Take a look. So down the end you can see celery is ready. And over here we have this warning saying using settings.debug leads to memory leaks. So this is because currently we are in our development environment and debug is turned on, in production we're not going to have this warning. So don't worry about this. Now if you scroll up, you can see some information about Celery. So this is the name of our Celery app, storefront, colon, whatever. For transport we have Redis, so Celery is connected to this Redis instance. Now in front of concurrency, we have 8

that is the number of CPU cores on this machine I'm using. So we have 8 Celery workers that are ready to pick up tasks. Okay? So Celery is up and running, in the next lesson I'm going to show you how to create and execute a task. But before we get there, I want to emphasize something. So far, we had to open two extra terminal windows. One for running Redis using Docker, and the other for running a Celery worker process. So that means every time you want to start your project, you have to open three terminal windows. One to run the web server, another to run Redis, and one more to run Celery. This is where Docker comes to the rescue. So instead of manually opening these terminal windows and running these commands, we can use Docker to start our project using a single command.

I'll talk about that at the end of this course. So for now, don't worry about it. I just wanted to give you another benefit of using Docker. So next we're going to talk about creating and executing a task.

Creating and Executing Tasks:

Alright, let's see how we can execute a long running task using Celery. So we're going to go to our playground app and add a new module called tasks.py. So this is the module, this is one of the modules that we instruct at Celery to automatically discover, okay? Now in this module let's define a function called notify customers that takes a message. So let's imagine we're going to call this function and as part of this we're going to send 10,000 emails to our customers. This is going to be a long running task. So we don't want to hold the main process. This is where we use salary. So for this demo, let's print sending 10,000 emails. We can also print the message that we receive.

Now to simulate a long running task, I'm going to put this function into sleep. So from the time module, let's import the sleep function. Here we call sleep and put this in sleep for 10 seconds. And finally we print another message saying, emails were successfully sent. Okay? Now to execute this with Celery, we need to decorate it with one of the Celery decorators. Now there are two ways to do this. First I'm going to show you the approach that most tutorials show you, but there is a problem with this approach, so then I'll show you a better way. So earlier we created this Celery module in the storefront folder, right? So let's load it right here. We can say from storefront, that celery import the celery object.

That is the celery instance that we created in this module. So we're loading the celery object, right? Now we can decorate this function with celery dot task. So this is what most tutorials show you. But there is a problem here. The problem is that with this approach, our playground app is being dependent or coupled to the storefront folder. So it's no longer going to be an independent reusable app so So anytime we want to deploy and reuse this app in another project, we should also take the storefront folder, which is very specific to this project, right? So let me show you a better way. A better way is to go to the salary module of the salary library that we installed, and import the shared task function.

Now we can apply that as a decorator right here. Okay? So this is how we define a task. Now to execute it, let's go to the views module, of the playground app. So now i'm going to delete this code for sending an email. And that means we don't need these import statements. So instead we're going to go to the tasks module of the current folder and import notify customers. So here we call notify customers dot delay. That is the tricky part. So we're not going to call this function directly instead. We call the delay method on it. Okay? And then we pass our argument, the message we wanted to send. Let's say hello. So let's test this. Let's go to the browser and hit the send point.

Good. So look, we got the response immediately, even though the notify customers function took 10 seconds to execute. But it's actually executing in the background. So back to VS Code, here's the terminal window for running Celery. We have an error saying receive unregistered task of type notify customers. The reason we're seeing this is because we started Celery before implementing this function. So when Celery started, this task was not part of our project, so even though we enabled auto-discovering of tasks, Celery was not aware of this task. That's why it's saying this task is unregistered. So to solve this problem, we need to stop this process by pressing ctrl and c. Okay, now we restart it, so Celery-a storefront worker. So now let's hit this endpoint one more time, now back to salary window, look, our notify customer task started, and these two yellow lines are the messages that we printed, so sending 10,000 emails and hello.

Now after 10 seconds, our task completed. So this is the beauty of salary, with salary we can execute long running tasks in the background. So when we hit this endpoint, we didn't have to wait 10 seconds for the emails to be sent, we got the response immediately. So our main application process is now free to serve other client requests. Now let me show you something really interesting. So in this window, I'm going to stop our salary worker. Okay? So our application is running, but our salary workers are offline. Now let's see what happens if we hit this endpoint one more time. Look, we don't get any errors because everything is happening in the background. So our application sent a message to our broker, in this case, redis,

But because our workers are offline, Redis could not deliver the message to them. So the moment our workers become available, Redis is going to retry sending this message. Let's see this in action. So, back to the salary window. Let's start our workers one more time. Alright, now look, we immediately got this message saying task notify customer started and here are our messages. And finally after 10 seconds, our task completed. So this is why we use message brokers. Using message brokers, we can reliably send messages from one application to another.

Scheduling Periodic Tasks:

Sometimes we need to schedule a task to run at a certain time or periodically, like every Monday at 10 a.m. This is useful for generating periodic reports, sending emails, running maintenance jobs, and so on. For that, we use Celery Beat, which is a task scheduler. So just like your heart is beating, Celery Beat is constantly beating and kicking off tasks. The actual execution of this task, though, is still done by Celery workers. So Celery Beat is a process that acts as a manager or an orchestrator. So let's see how we can schedule a periodic task. first. We go to our settings module and Define a new setting called celery on the light bead on the line schedule again Make sure to spell this properly.

Otherwise things are not going to work. So we set this to a dictionary, and here we define our tasks. So I'm going to define a task called notify customers. and we could call this anything, but for consistency, it's better to use the same name as our task function. Now we set this to a dictionary. Here we should specify our task. So it's a task to the full path to our task function. So we go in the playground app in the tasks module and reference notify customers. Next we need to specify the schedule. Now there are a couple of ways to set this. We can set this to a number like five and that means every five seconds. So Saturday beat is going to kick off this task every five seconds.

Or if you want to use minutes, we can say 15 times 60, every 15 minutes. Now, if you want to have more control over the schedule, you need to use the crontab object. So on the top, from salary.schedules, we import crontab. Now back to our schedule, we can set this to a crontab object like this, crontab. We can set day of week to 1, meaning Monday. hour to 7, minute to 30. That means this task should be executed every Monday at 7.30. Here's another way to use a crontab. So we can set minute to a string like this. Asterisk slash 15. That means every 15 minutes. Now if you want more examples of using crontabs, you need to look at Celery documentation on configuring Celery beat.

So for this demo I'm going to use 5 seconds, so we can see things clearly. Now if our task function takes arguments, we can specify them here. We set this to a list or a tuple. I'm going to pass hello world. Now optionally if our task function takes keyword arguments, we can pass them here as well. So we set this to a dictionary of a bunch of key value pairs. Now we don't really need this here, so delete. So this is how we configure salary beat schedule. Now we need to start celery beat process. Back in the window where we run our web server, you can see our web server is crashed, so let's restart it. Okay, no errors here. Now we open a new terminal window and run celery dash A. We specify our project, which is storefront, and then the type of process, which is beat.

So previously we started the worker process, now we start the beat process. Okay? Take a look. So Celery beat has started. and as we can see, it's connected to our broker at this address. Now, back in the celery window, look, every five seconds, Celery Beat is kicking off our notify customer tasks. So this is how we can schedule periodic tasks with celery Beat.

Monitoring Celery Tasks:

Alright, the last thing we're going to talk about in this section is monitoring celery tasks. And for that, we're going to use a great tool called floor. No, it's not flower, but you can call it flower if you prefer to. So here in the terminal, let's install floor. Alright, floor is installed. Now we need to start the floor process. So celery dash a storefront floor. Okay, floor is started and we can access it at localhost port 55 55 So on the top you can see the number of active jobs or tasks as well as process jobs failed jobs and so on Down below we can see our workers. So we have one worker It's currently online and it has two active tasks. It has processed eight tasks and so on now if you go to the tasks tab You can see the history of tasks.

So here's the name of our task and as you can see, each task has a unique identifier. We can see the status of each task, the arguments we pass to it, the date and time it started, finish and so on. It's pretty straightforward. Doesn't really need any more explanation. We also have the broker page where we can see our broker. So we're connected to redis instance. In this broker, we have a queue by this name. So this is the queue that is receiving our messages and routing them to salary workers. So that brings us to the end of this section. In the next section, we're going to talk about my favorite topic and that is automated testing.

Automated Testing:

Introduction:

Welcome back to another section of the ultimate Django course. In this section, we'll be talking about automated testing. Now a lot of people find automated testing confusing, so I'm going to simplify it for you and give you a practical recipe you can follow to test your applications. Now, let's jump in and get started.

What is Automated Testing?

let's start the section with a quick introduction to automated testing and its benefits. So, over the last 10 hours or so, we have built a few endpoints for managing collections, products, carts and orders. Now, each of these endpoints support various operations. We can create objects, we can update them, delete them, and so on. Now, some of these operations involve a bunch of rules. For example, you know that only admin users are allowed to modify products. So if an anonymous user or someone who's not an admin tries to modify a product, they should get an error, right? Now here's the thing, we can manually test these endpoints in the browser, but as our application gets more complex, the time we need to manually test these functions increases exponentially.

Plus, over time, we're going to forget about various rules hidden behind these endpoints, unless we document them in a script that a tester can use to test various scenarios. But again, testing all these scenarios over and over going to get extremely time-consuming this is where automated testing comes to the rescue. With automated testing, we can write code to test our endpoints and their business rules. So we write that code once and run it over and over. Every time we change our software or every time you want to deploy it, we run hundreds or thousands of tests within seconds and see if you have accidentally broken something or not. So automated testing allows us to write better code and release it with more confidence.

Now every so often you come across people like our superstar coder, yeah, that John Smith guy who hates automated testing and thinks it's a waste of time. I feel him. Because automated testing is like a double edged sword. If you do it the right way, it can be extremely valuable. But if you do it the wrong way, well, you're in trouble. These tests get in the way, they slow you down and just make you frustrated. The reality is that there's a lot of bad information and poor practices about automated testing out there. Just like John Smith, I also wasted a lot of time writing useless tests. So it took me quite a while to really understand the right way to write tests. We'll talk about that more in the next lesson.

Test Behaviours, Not Implementations:

One of the reasons a lot of people fail with automated testing is that they test implementations, not behaviors. Let me explain what I mean. Let's say we want to test a microwave. We press the start button and observe the monitor. We expect the monitor to show a timer running for one minute, right? So this is how we expect a microwave to behave and that's how we test it. We don't open up a microwave and test the electrical signal going in and out of every transistor just to see if the microwave is working or not. Testing software is the same. we should test how the software behaves, not how it's implemented. Because the implementation may change over time. I see a lot of people testing individual building blocks on an API like models, views, routers, serializers and so on.

Over time, your implementation may change. You may replace a function based view with a class based generic view and later on, you may replace that view with a view set. Similarly, you may split your model into two different models or combine two models into one. These are implementation details that may change. If you write tests for these details, your tests will break as you change the implementation. Then you have to spend a whole lot of time fixing or rewriting these tests. That's why a lot of people fail when it comes to automated testing. So remember, your tests should test the behavior, not the implementation. So they should test how an API behaves, not how it's implemented. Let's put this in practice. Let's say we want to test the creation of a collection.

so we send a post request to the collections endpoint now how does this endpoint behave that's what we need to test so here we have a few scenarios if the client is not authenticated we expect a 401 response meaning unauthorized if the client is authenticated but the current user is not an admin then we expect a 403 response meaning forbidden so the current user should not have access to this function now if the current user is an admin but the request doesn't include a collection name then we expect what? A 400 response, meaning bad request. And also, the response body should include an error message for the collection name. And finally, if the current user is an admin and our request includes a collection name, then we expect a 200 response and the idea of the new collection should be included in the response.

So this is how our collections endpoint should behave when it comes to creating a collection. We can test this behavior manually by hand in the browser, or we can automate it using code. Obviously, if you automate it, we can run all of our tests in a matter of seconds and see if you have broken something or not. So I want to emphasize this one more time. Your tests should test the behavior and not the implementation. Next we're going to talk about testing frameworks.

Tooling:

Alright, let's talk about tooling. Just like we need a framework for building an API, we also need a framework for writing tests. So a test framework gives us a structure for writing tests, as well as a program to run our tests and give us a report. Now, there are so many different testing frameworks for Python applications, but the two most popular ones are unit test, which is built into Python itself, and pytest, which we have to install separately. Now, different people love different tools, but in my opinion and a lot of other people's opinion, pytest is a better framework, because it has more features, it has a huge community, it has a lot of plugins, but more importantly, it has less boilerplate. So our tests will be shorter and more concise.

Here's a side by side comparison of the same test written with unit test on top and pytest on bottom. You can obviously see that tests written with pytest are shorter, cleaner, and more concise. So, here in the terminal, first we're going to install pytest, pipenv install dash dash dev, meaning development, so we're going to install pytest as a development dependency. So it's not a dependency that we're going to deploy with our application to work. We need it only for development, okay? So pytest. Alright, now let's go to our pip file. So take a look. Here are the packages that our application needs. Now we also have div packages and these are the packages that we use for development. And pytest is right here.

Now to test Django applications with pytest, we should also install one of the pytest plugins for Django. So pipenv install dash dash dev pytest dash Django. So we have installed pytest, next we're going to write our first test.

Your First Test:

Alright, let's see how we can write our first test. So we're going to go in the store app, and create a new folder called tests. The name of the folder is important, make sure it's tests in plural form, because that's what pytest looks for. Of course we can configure pytest to look for something else, but it's best to stick to the conventions, okay? Now in this folder, we're going to add a new file, the name of the file should start with test underline. Now we want to test the collections endpoint, so we type collections here. Alright, now to define a test, we create a function, the name of the function should start with test on the line. Again, this is one of the conventions, if you don't follow this convention, pytest is not going to pick up your test.

Okay? Now, it's really important that your test function clearly identifies what behavior you're testing. I see a lot of people creating test functions like this. Test on the line collections. What does this mean? What scenario are we testing here? It's unclear. Or I have seen people creating test functions like test underline create. Again, it's not obvious what scenario are we testing. Here's a real example. Let's say the scenario we're testing is that if the user is not authenticated, we should get a 401 error. So, let's type that here. If user is not authenticated, actually we can make this shorter. Instead of not authenticated, we can use the word anonymous. Just a little bit shorter and reduces the noise in our test. Then this should return 401.

So now the test name clearly identifies what behavior we're testing. Now in this module we're going to have several tests, many of them are for creating a collection, others are for deleting a collection, updating a collection, and so on. So while not necessary, it's best to organize these tests by use case. And for that we create a class called test create collection. Again, the name of the class should start with test, otherwise pytest is not going to recognize it. Now let's move this inside this class and add self. Okay? Now, every test should have three parts. This is what we refer to as triple a, which is short for arrange, act, and assert. In the arrange part, we prepare the system under test.

So this is where we create objects, or we put our database in an initial state and so on. Now in this case, we just want to create a collection, so the arrange part is really empty, we don't have anything here. Now for the act part, this is where we kick off the behavior we want to test. In this case, this is where we send a request to the server. Now, you don't need to type these comments, I'm just adding them for clarity, okay? So, we need to send a request to the server, and for that, we need to import the API client class. So, from REST framework, .test, we import API client. Now, we create a client object. Now, this class has a bunch of methods like get,

post, put, delete, and so on. So we're going to send a post request to slash store, slash collections, make sure to end it with a forward slash, otherwise you're going to get an error. Now, in the request body, we want to include an object. So let's add a dictionary here. We're going to give this collection a title like A. The name of the collection doesn't really matter, so that's why I'm using A here. We could type a long collection name like beauty products, but that doesn't really add any value, it just makes our test noisy. So let's simplify this to a, so we send a request to the server, and we get a response. Now the final part of the test is the assertion part.

So this is where we check to see if the behavior we expect happens or not. So in this case we expect to get a 401 response from the server. So using the assert statement, we can validate this. Here we need to type a boolean expression. So this response object has a property called status on an encode, we check to see if this is 401. So first we go on the top, and from rest framework, we import the status module, now over here, we check to see if this is status.http 401. Okay? So this is how we write tests. Every test you write, no matter what framework or language you use, should follow the same structure. So we have arrange, act, and assert. Now, we don't need these comments anymore, so I'm just going to simplify this test.

Okay? Next I'm going to show you how to run this test.

Running Tests:

Alright, we have written our first test, now let's run it. To do that, first we need to tell pytest where our settings module is. And for that we have to create a configuration file. So here in the root folder, we're going to create a new file called pytest.ini. Now here we type pytest, and then we say django underlying settings module to storefront.settings. Again, as I always emphasize, pay close attention to spelling. So this is our configuration file, now we go into terminal and run pytest. Okay, we have one passing test, beautiful. Now as you have seen, it's easy to write tests, but we have to make sure that these tests are testing the right thing and they're telling the truth. How do we know this test is telling the truth?

We don't want lying tests, right? So let me show you how we can ensure that this test is testing the right thing. We're going to comment out the line of code that is responsible for making this test pass. Let me show you what I mean. So back to our test, here we're testing that the status code of the response is 401. So let's go to our collection view set, collection view set, now look at this class, what line of code is generating for that 401 response? It's this line of code, right? So if we don't apply a permission, anyone can create a collection. So now that we have commented out this line, let's run our test and see what happens. If the test still passes,

that is a lying test, and we don't want any of those tests. So back to the terminal, let's run pytest one more time. Alright, now we have a failure, let's look at the error message. It's saying database access not allowed, use DjangoDB mark. So what is happening is that because we don't have a permission, our API endpoint is trying to create a collection, so it needs database access, and by default, pytest prevents that. So we need to mark or decorate our test function with db decorator. So back to our test module, first we import pytest, then we decorate this method with pytest.mark.django underline db. Now we can apply this decorator to individual methods, but that's repetitive, so it's better to apply it to the class, so all methods in this class will inherit this decorator.

Now back to the terminal, let's run pytest one more time. okay, we have a failure, now look at the error, so you can see our test method, this is the method that is failed, now look at the assertion, we expected a 401 response, but we got 201, which means object created, so now we are 100% sure that this test was testing the right thing, so back to our view, let's bring this line back, and run our test one more time, now we have a passing test, beautiful, so with this test, we're testing that our endpoint is behaving properly. Our test knows nothing about how it's implemented. So it knows nothing about our models, views, routers, and so on. Tomorrow we can change the implementation.

For example, for whatever reason, we may decide to replace this view set with a function based view if you do so, our test will still pass as long as our endpoint behaves the way our test expects, okay? So this is the beauty of writing tests. With these tests, we can ensure that our software is behaves according to its requirements. And more importantly, we don't have to remember all these requirements. We don't have to document them and then go back and forth in outdated documentations. We can simply rely on our tests. We can run them every time we change our code or every time we are ready to release our software. So this is how we run our tests. Now let's look at a few more ways to run our tests.

So if you run pytest, this is going to execute all tests in this project. Now sometimes we may end up with a bunch of failing tests, This is especially true when we have hundreds or thousands of tests. So we change something and then we end up with let's say 50 broken tests. That's overwhelming. So in those cases, we want to isolate our tests. Perhaps we only want to execute tests in a particular directory or in a particular module or in a particular class. So if you want to do that, we type the name of the directory here, let's say store slash tests. This will only execute tests in this directory. Or we can target a particular module like test underline collections.py, we can also target a specific class in this module, for that we need to use double colons.

So then test create collection. And similarly, we can target a specific method in this class by using double colons. So then we type test whatever. Now we can also use dash k to specify a pattern. For example, let's say we only want to run tests that have anonymous in their name. We can do that using the k option.

Skipping Tests:

running tests, sometimes we get a failing test, but let's say fixing that test is going to take some time, and we might be in the middle of something, so we don't want to get distracted. In those cases, we can temporarily skip those failing tests to reduce the noise and finish our work. Then, when we're done, we can come back and fix them. It's really easy. So for this demo, let's say this test is a failing test. We don't want to attend to it, so we can temporarily skip it by applying this decorator. pytest.mark.skip. now here in the terminal we run pytest one more time, look we have one skip test and pytest is not yelling at us saying, hey you have broken this test, yada yada yada, so we can finish our work, then when we're done, we can come back and fix those skip tests.

Continuous Testing:

So there are two ways to run tests. We can run them on demand, like before committing our code to git or before deploying our software, or we can run our tests all the time. This is what we call continuous testing. Now some people love it, some people hate it, because if you're on a slow machine, running all these tests all the time might slow you down. I can't tell you this for sure, I've never had this problem, but I've seen some people complaining about that. I personally love continuous testing. I love to dedicate a separate terminal window for running my tests I can put that on the side or on a separate monitor, and as I'm coding, I can see if I've broken something or not.

So let me show you how to use continuous testing in this project. First we have to install one of pytest plugins. So pipenv install dash dash dev pytest dash watch. Alright, with this plugin installed, instead of manually running pytest every time, we can simply run PTW, which is short for PyTestWatch. Now, this runs all of our tests, and every time we change our code, this is going to rerun our tests. So if you go to our test module and make a small change here, now look, PyTest is rerunning our tests, and now we have a failing test. So, let's fix this. Similarly, if you go to our application code, like our collection view set class, if you make a change here, by commenting out this line, once again, pytest automatically reruns our tests.

So as we are coding, we get real time notification about the status of our tests. I would suggest give it a try, if you don't like it, you can always run pytest manually every now and then. Just remember, as a best practice, you should always run your tests before committing your code to git and before deploying your software.

Running and Debugging Tests in VSCode:

we can also run and debug our tests in vs Code. So here we have the test explorer panel. First we need to configure it to pick up our tests. So click on configure and select pi test then we select the directory that contains our tests. So we're going to select the root directory so this picks up all tests in all apps. So here we have storefront 3 then we have the store app, the tests module, collections module, and here we have the tests for creating a collection. So this is the benefit of grouping our tests inside a class. Now we can see all scenarios we're testing for creating a collection. And if a scenario is missing, we can quickly spot that here. So in this panel we can run an individual test, we can also debug it, which I'll show you in a minute.

We can jump to that test, so here's our test function. We can run all our tests right here. So there you go, we got a green mark, beautiful. Now let's go to this test and create an error, so I'm going to change this boolean expression and rerun this specific test. Okay, now we have a failing test, so look over here on the left, we can see the error message, unfortunately this is not color coded. So what we saw in the terminal window using pytest was better, but here everything is in black and white, so you need to read and understand what's going on here. Now on this bar, we can jump to the previous or the next test failure. We can also close it.

Now one thing that is useful here though is that we can debug this test right in vs Code. So we have a failure. We want to know what is going on here. So first we add a breakpoint. Then we go to the test panel and debug this test. Okay, I'm going to close this window. Now look, our debugger is turned on and we are on this line. So at this level you should know how to use the debugger in VS code, I'm not going to go over it, but you know that we can step over this line, we can look at our variables in the watch window and so on. So when we are done, we can stop the debugger, and remove the breakpoint, let me fix this as well, so this is how we can run and debug our tests in VS code.

Authenticating the User:

Alright, let's see how we can authenticate the user. So in this lesson, we're going to write a test for the scenario where the client is authenticated, but the current user is not an admin. So to save time, I'm going to duplicate this test, and make a few small changes. So if user is not admin, then this should return a what? A 4 or 3 error, meaning forbidden. So the current user doesn't have permission to execute this operation. Now how do we authenticate the user? Very easy. This client object has a method called force authenticate. We call it and set user to an empty dictionary. And then we change the status to HTTP 403 forbidden. Now, let's run our test. Alright, we have two passing tests.

Single or Multiple Assertions:

Sometimes our tests may need multiple assertions. For example, for this lesson, let's write a test for the scenario where the client is authenticated and the current user is an admin, but the data that we post to the server is invalid. So let's duplicate the last test and make a few small changes. So if data is invalid, returns 400. Now, here we need to authenticate the user to a user object that is an admin. To do that, first we need to import the user class on the top. So, from django.contrib.auth.models import the user class. Now, back over here, we're going to set the user to a real user object, and here we set isStaff to true. So this is just an object in memory, it doesn't exist in the database, it doesn't have to be.

As long as we pass a user object here, force authenticate will take care of authenticating that user, okay? Now in this test, our focus is on invalid data, so to emphasize that, I'm going to set the title to an empty string. And then we need to change the status to 400. So this is our first assertion, but we need another assertion for the error message that we expect in the body of the response. So assert response.data, this is a dictionary, so we can access the title property, and check to see that this is not none. So we don't care about the actual error message, because that's implementation detail, the wording might change in the future, we don't want our tests to break if we change an error message.

As long as we have something for the title in the response, we're good, and this test should pass. Now, in this test we have multiple assertions, and this is where some people get a little bit confused, because there is this supposed best practice that a test should have a single assertion. I agree with this principle, because the purpose of this principle is to ensure that a test has a single responsibility. For the same reason that our functions, our methods, our classes should have a single responsibility. But that doesn't mean that we should have a single assert statement in a test method. In this case, we have two assertions, but these assertions are logically related. We're still testing the response that we get from the server.

So we're not testing two separate things. Testing the status code and the body of the response are logically related. So in this case, it's perfectly fine to have multiple assertions in the same test. Now if you want to be dogmatic about the single assertion principle, then we'll have to create another test, we have to repeat all this code, and just change the assertion. That doesn't make sense. We end up writing more code without getting any value. So, your tests should have a single responsibility, they should test a single thing, but testing that single thing might involve multiple assertions, as long as they're logically related. Now let's write another test for the scenario where the data is valid. So again, I'm going to duplicate this, and make a few changes, so if data is valid, then it returns 201, meaning created, so we have an admin user, and we want to send a valid collection name here, now the status code should be 201, and in the body of the request, we should have the id of the new collection, so id,

should be greater than 0. Now, I've seen some people taking another approach here, instead of checking to see that the id is greater than 0, they go in the database and pull the collection with that id to ensure that collection was saved. In theory, that may sound like a reasonable approach, but in practice, I don't agree with that, because if we use the collection model to fetch that collection, our test becomes coupled to our implementation. So if tomorrow we change this model class, the tests that are coupled to that might be affected. the less our tests know about the internals of our system, the more reliable they're going to be. For the same reason that when testing a microwave, we don't care what's happening inside, we just know the interface, we know the buttons, we play with those buttons and expect some behavior.

So in this case, I don't want to use our model to go in the database and pull a collection out. Now, another approach here is to send a get request to the collections endpoint to see if we can get that collection. Again, in theory, that might sound like a reasonable approach, But there is a problem with that as well. The problem is that if the collections endpoint has a bug and cannot return the given collection, this test is going to break even though we could successfully create this collection. So when running our tests, we end up getting a ton of errors. So this is why I prefer just to check the id and ensure that it's greater than 0. If it's greater than 0, we know that all these components have done their job properly and we have a collection with that id stored in the database.

So that's it for this lesson. Now one last thing I want to show you here is that in VS code, if we click on this test name, we can see all scenarios we are testing for creating a collection. So that's very useful. We can easily navigate between these tests. Also, as we write tests for other use cases like getting a collection, deleting it, and so on, we can click on this class name and see all classes or all use cases we have in this module. Currently, we only have test create collection. Alright, now let's run our tests and make sure everything is working. so, you have 4 passing tests, and the test execution took just over 1 second. Imagine if you wanted to test all these scenarios by hand, that would definitely take more than a minute or two, right?

So this is the beauty of writing automated tests.

Fixtures:

Alright, let's talk about a powerful feature in pytest called fixtures. Using fixtures, we can remove duplication in our test code. Let me show you how they work. So look, in every test we have written so far, first we have to create a client object. And that means in every test module, first we have to import the API client class, so we can create a client object in each test. That is repetitive. This is where we use fixtures. So back to our project, here in the tests folder, I'm going to add a new file called conftest.py. This is a special file for pytest. So the fixtures or reusable functions that we define here, pytest will automatically load them without us having to explicitly import this module.

So first we import pytest, then we define a function called apiclient, in this function we import the apiclient class, so from resframework.test we import API client, we could also put this on top, it doesn't really matter. So here we return an API client instance. Now, to make this function a fixture, we apply the fixture decorator to it. So, pytest. fixture. Now this function is a reusable piece of code, and we can add it to each test as a parameter. So, back to our test module, in this test, I'm going to delete this line, and add API client as a parameter to this function. So when pytest tries to execute this test, it will look at our fixtures that we have defined in conf test, it figures out that we have a function by this name, so it will call this function and return its value here.

So in this test, we'll have access to that value as api underline client. So we remove one unnecessary line from this test. We can do the same with other tests, so I'm going to pause the recording and get back in a second. So Alright, now every test is getting api client as a parameter, so we go on the top, we no longer need to import this, so let's remove one unnecessary import statement, that's good. Now the second repetition is in calling this endpoint. So in every test, we have repeated this bit of code. The only difference is in the object that we pass to the server. So sometimes we are passing a valid collection, in other cases we are passing an invalid collection.

Let's see how we can use a fixture to simplify this code. Now, I'm not going to define this fixture in our conf test file, because this is where we define fixtures that we're going to reuse across test modules. Creating a collection is something specific to this module. So, I'm going to define this fixture on top of this file. So, we define a function called create underline collection. Now, this function needs API client, right? So, we need an API client to post a request to the server. So we can add that as a parameter, and once again, pytest will automatically pass that fixture to this function. Now, this function itself is going to be a fixture, so we apply the fixture decorator. Good. So here we're going to call apiclient.post slash store slash collections slash.

Now this is the tricky part. We don't want to hard code a collection here, that's useless, because sometimes we want to pass an invalid collection. Now, if we pass collection as a parameter here, and then replace this object, PyTest will think this is a fixture, so it will try to find it, and because there is no fixture by this name, PyTest is going to give us an error. So to solve this problem, first we remove this parameter from here, then we wrap this line inside an inner function. So here we define an inner function, we can call it whatever we want, I'm going to call it doCreateCollection. Now this function is going to take that collection parameter, okay? And then we move this line inside the inner function, and finally we return our inner function, so doCreateCollection.

So let's see what's going on here. We created a fixture called createCollection. Now in the previous fixture, in the api client fixture, we returned an object, right? In this new fixture, instead of returning an object, we are returning a function. So if we go to one of our tests, like the first test, and add our second fixture, create collection, before pytest runs this test, it's going to call this function, or this fixture, as you know, this returns another function. So over here, we're going to get a function, create collection, and this is the inner function that we are returning over here. So we can call that function and give it a collection. If you have done any JavaScript programming, you know that this is called a closure.

We have the same concept here in Python. So we call this inner function and give it the collection that we have over here. Okay? So with that we can replace the first line The only thing that is missing is the response. So, we go to our inner function, when we send a post request to the server, we return the response. Now we can get that right here. Okay? Now I'm going to pause the recording and apply the same change to every test in this file. Alright, now every test has an extra parameter called create collection. Now look at the first test. Here we don't need the api client parameter, because that call is happening inside this fixture. So, let's remove it. We still need api client in other tests, because that's where we are authenticating users.

Now, authenticating users is again a repetitive step. So we need this in a lot of tests. So here's your exercise. Pause the video and define a global fixture for authenticating users in the conf test module. You'll see my solution next. Alright, here's what I've done. I've created a fixture called authenticated that takes api client. In this picture we return an inner function and this inner function has a keyword argument called is staff which I have initialized to false. So with this I don't have to specify that value in every test. Now in this inner function we simply call force authenticate and this is where we create a user object. So we don't have to import this user class in every test module. So we create a user object and set is staff

to the value that we receive in the inner function. Now back to our test module, so look at this test, we simply call authenticate, and for clarity offset, it's staff to false, but this is not required, so we can remove it, and our test will still pass. Now, I want you to pay close attention to how I have formatted this test. So the first line is the arrange part, this is where we are preparing our system, then we have a line break, next we have the act part, followed by a line break and the assertion. So anyone reading this code can immediately tell what are my arrange, act and assert parts. This makes our code more readable.

Creating Model Instances:

So we're done with the tests for creating a collection, now let's write a test for retrieving a collection. This one is going to be a little bit different from previous tests, so pay close attention. So in this module, I'm going to create a class for organizing the tests for retrieving a collection. So I'm going to call this test retrieve collection. Now there are two scenarios we need to test here. One is that the collection doesn't exist, in that case we expect a 404 error. The other scenario is that the collection exists, so we expect a 200 response, and the collection should be in the body of the response. This is the scenario I'm testing in this lesson. So let's create a method called test if collection exists, returns 200.

Now here we need our API client to send a get request to the server. Now this is the part that I told you is different from previous tests. In the arrange part, we need to create a collection so we can retrieve it later. Because this test should not be dependent on the other test that creates a collection. Because if this test is executed first, it's going to fail. Because there is no collection in the database, right? So our tests should not have any dependency on each other. You should treat each test as if it's the only test in the world. Okay? So how can we create a collection? Here we have two choices. One option is to use our API client to send a post request to our collections endpoint.

But the problem with this approach is that if there is a bug when creating a collection, this line is going to fail, so this test is going to fail as well. even though retrieving a collection might actually work. So with this we're going to get a lot of failures. The other option is to use the collection model. So we say collection.objects, this is not working, so on the top, look, we have the wrong import statement. So from store.models import collection. Back over here, so we say collection.objects.create, give it a title, and this will create a collection. Now, earlier I told you that when writing tests, we should test the behavior and not the implementation. This collection model is part of the implementation, it's not part of the interface of our API.

So in this case, we are violating what I told you earlier, but we don't have a better option, so it's okay to break the rules, because remember, software engineering is not black or white. Sometimes you need to break the rules and make compromises. So, use the collection model to create a collection. Now, what if you wanted to create a product? You know that our product model has a bunch of fields, so initializing all those fields over here would create a bit of noise in this test. Let me show you a better way. This is where we use a fantastic library called model bakery. So here in the terminal, let's install as a development dependency, model underline bakery. Alright, with model bakery installed, now we go back to this module,

on the top, from model bakery, we import the baker module. Now back to our test, we can rewrite the first line, we can say baker.make collection. So with this approach, we don't have to initialize individual properties of this model. This model baker will take care of that for us. So for each field, depending on its type, it will give it a random value. Let me show you. So I'm going to delete this line, and get this collection, and print it on the terminal. So collection, now we're going to access the dictionary property, so we can see individual key value pairs. Now, remember when writing tests, these print statements don't work unless your test fails. So to see this collection, I'm going to deliberately make this test fail.

Now, take a look. Alright, so first we have an error saying database access not allowed because i forgot to apply the django db decorator on our test class. So back over here, I'm going to decorate this class with pytest dot mark dot django underline db. Okay? Now, let's run our test one more time all right here's the object that model bakery has created for us. We have a collection with this id and this title. Now look at the title. It's a long random string. Now if you had other fields here, like a date time, an integer, whatever, model bakery would automatically initialize them to some value. Okay? So with model bakery, we can easily create objects. Now, this will also take care of relationships for us.

For example, if we say, baker that make product, this will automatically create a product and a collection. Because each product should be inserted into one and only one collection. So with this single line, we'll get both a product and a collection. We don't have to explicitly create a collection first before creating a product, okay? Now, sometimes we will have control over the collection for a product. For example, here we have a special keyword argument called underline quantity. We can set it to 10 and this will create 10 products in our database. Now there is a problem here. The problem is that each product is going to have a separate collection. What if you want to put all these products in the same collection?

This is where we need to explicitly set the value of some of the fields here. For example, first we can create a collection, baker.make collection, we'll get a collection object, then we'll use that to set the collection field of the product model. So now, all these 10 products are going to be in the same collection. Okay? So this is the basics of model bakery, if you want to learn more about it, you need to look at their documentation. Let's finish up this test. So, we create a collection, then in the act part, we send a get request to store slash collections, now we need to add the id of the collection here, so I'm going to convert this to a formatted string, and insert collection.id right here.

Now, we get a response, and then make an assertion. So we assert that response, that status code equals status.http 200, and the collection object is in the body of the response. So we say assert response data, earlier I told you that this is a dictionary, so we can access individual properties like id, and compare them with, let's say collection.id. And then we can duplicate this and do the same with the title property. But there is a better way. Instead of comparing individual properties, we can compare the entire data object with another dictionary. So we can say, assert response.data equals this dictionary. So here we add our key value pairs, id is going to be collection.id, and title is going to be collection.title. Let's run our test and see what happens.

Alright, we have a failure, because the status code of the response is 301, but we expect it to 100. Do you know what 301 represents? That's a permanent redirect. So in Django applications, if we send a request to this endpoint, Django will automatically redirect us to this address. So Django always terminates URLs with a forward slash. Now let's run our test one more time. Alright, we get another error saying left contains one more item. So here we have two objects, this is the left object, this is the right object. The error is saying that the left contains one more item or one more key value pair, and that is products count. So, back to our code, let's add products count, and rerun our test.

Okay, we have five passing tests. Now, one last thing I want to talk about here is that when we run our tests, pytest will automatically create a test database for us. So, let's quickly go to our database settings. Look, here we have a database called storefront3. When we run our tests, pytest will automatically create a database called test underline storefront 3. It will create this database at the beginning of executing all the tests, and when all tests are finished, pytest will drop this database. So this way, the data that we create while testing is not going to get mixed up with the data we have in our development database. So that brings us to the end of this section. Now as an exercise, I want you to write all the tests for the collections endpoint

as well as the product standpoint.

Performance Testing:

Introduction:

Welcome back to another section of the ultimate Django course. In this section, we'll be talking about performance testing. So we're going to throw a few hundred users on our application and see how it performs. It's going to be a lot of fun. So let's jump in and get started.

Why Performance Testing:

Just like automated testing, performance testing is something we need to practice while building an application. Because once we go to production, we don't want to get surprises. We don't want to get a wake up call late at night because our application is unresponsive. Now unfortunately, a lot of companies don't do any kind of performance testing or leave it to the last minute just before going to production. And sometimes this gets rushed or canceled because of tight deadlines. But then sooner or later after release, the truth comes out. the hidden performance problems they try to ignore will eventually show up. So remember, you need to run performance tests while building an application. This is especially true if you're building a mission critical system where a slow, unresponsive application can be very costly or disastrous to the business.

For example, in an air traffic control system, downtime is simply not an option. The system should be always up and running and response times should be in an acceptable range. So with performance testing, we can identify and fix these problems before they end up being so costly or disastrous. And that's what I'm going to show you in this section. Now, performance testing is fairly complex and you can read an entire book on it. But in this section, I'm going to teach you the basics and show you a few tools you can use to easily identify most of the performance problems in your applications.

Installing Locust:

Now, there are so many performance testing tools out there, but the one that I personally love is Locust. Because it's really simple, it has a beautiful UI, and we can write our performance tests using Python. So here in the terminal, let's run pipenv install dash dash dev locust. All right, good. Next, I'm going to show you how to write a test script using Python.

Creating a Test Script:

So as part of performance testing, we need to identify the core use cases or the functions that are essential to our business. So in this application, a core use case involves the user browsing the products catalog. Another core use case involves the user registering, signing in, and then signing out. So we identify these core use cases, and then for each use case, we create a test script which specifies the actions the user will take. So in our project, I'm going to create a new folder called Locust Files. the name doesn't really matter, in this folder we're going to add a new file called browse underline products. That's the use case we're going to test. Now, in this module, first from locust, we import the http user class, and then we define a class, we can call it whatever, but this class should extend http user.

Now, let me run a performance test, locust is going to create an instance of this class for each user, and execute the tasks that we define in this class. These tasks include viewing products, viewing particular product details, and adding a product to cart, okay? So for each task we define a separate method. So here's the first one, view underline products. Now to make this method a task, we need to decorate it with the task decorator. So like this. Now, here we're going to send a request to the products endpoint. So this self has a property called client. With this, we can send HTTP requests to the server. So we send a get request to store slash products. Okay? Now, in a real world situation, the user is going to go from one collection to another.

So here we should generate a random value for the collection ID. So from random module, we import randint function, then we call it, and give it a range. As far as I remember, the collections that have products are collections 2 to 6, but these numbers don't really matter. So we generate a collection id, then we need to add that as a query string parameter. So collection id equals, I'm going to convert this to a formatted string, and add collection id right here. Now, when we run these tests, for each url that we have here, we're going to have a separate row in the report. And this is going to make our report too verbose. So here we need to add all these URLs to a particular group to simplify our report.

You will see that in the next lesson. And for that we're going to set the name argument to store slash products. So this is the end point we're testing. And later we'll see if this end point is slower than it should be. Now for you to see, I'm going to break this down into multiple lines. So this was our first task. now let's create a task for viewing a particular product. So, very similar, we're going to define another method called view product. And we should decorate this with task. Now similarly, here we generate a random product id so brand int we have products one to one thousand so let's get around a product ID. Then, we call self that client that get We're going to send a request to this endpoint.

Store slash products. Here we add the product id. And once again, we're going to group all these urls under a particular group. So we can call that store slash products slash id. Meaning we're looking at a particular product. Let me reformat this. Good. So that was the second task. Now we need a task for adding a product to a cart. So, another method, add to cart. Now, for this one, I don't want to generate an integer between 1 and 1000, I want to limit the range, so we end up getting duplicate products in our shopping cart. This way we can see if updating the product quantity is going to have performance issues. So, let's use a small range like 1 to 10, we get a product id, now we call self

.client, .post, now for this one, we're going to send a request to store, slash carts, slash, now here we need a cart id, and this cart id should be generated when a user starts browsing our website. Now for that we have a special method in this class called onStart. Now this is not a task, it's a lifecycle hook. So this gets called every time a new user starts browsing our website. Now in this method, we're going to create a cart for this new user, so we call self.client.post, we send a post request to store slash cart, then we get the response, next we need to call response.json to get the json object in the response, now this is a dictionary, so we can access the id property, now we can store this value in self.cart id, so we can reference it in this other task, okay?

And by the way, I forgot to apply the task decorator. So we add self.cart ID right here. Now as you know, we need to send a post request to the items endpoint. Now, again, we're going to group all these URLs. We can call that group store slash carts slash items. Now for sending data to the server, we set JSON to a dictionary. So here I'm going to set product ID to this product ID. And quantity to 1. Now potentially we can improve our endpoint and assume the quantity of 1 as the default value. So we have defined a bunch of tasks, now we can give each task a weight or a priority, because in a real world situation it's more likely that a user views different products than adding a product to their shopping cart.

So for the first task, view products I'm going to give it a weight of 2, for the second task viewing an individual product, I'm going to give it a weight of 4, which means a user is twice more likely to execute this task than the first task, viewing products in a collection. Now these numbers are arbitrary, you can put any numbers you want. Now, for the third task, adding to cart, I'm going to use 1. So it's less likely that a user adds a product to their shopping cart. Now, we don't want these tasks to be executed immediately after another, because in the real world it takes the user a few seconds or a minute to execute each task. So over here we're going to set an attribute called wait time.

Now to set this we're going to import the between function from locust and here we call between and give it two values, 1 to 5 seconds. So locust is going to randomly wait between 1 to 5 seconds between each task. So when we run this test, for each user we want to simulate Locus is going to create an instance of this class, and it will repetitively call these tasks and apply a wait time after each task. Okay? So this is how we create a test script. Now the beauty of this approach is that our test script is part of our code base. So we're going to commit it to our repository, and every time we change our code, we can rerun our performance tests to see if we have any issues or not.

Next I'm going to show you how to run this test script.

Running a Test Script:

Alright, let's talk about running a test script. Now in this lesson, I want to put a print statement inside each task, so you understand how Locust executes this script. So here we're going to say, view products, then we need another print statement over here, print, view product details, and one more over here. Add to cart. Now, let's open up the terminal window, make sure you're running your web server, then open a new terminal window and run locust-f for specifying the locust file, and this is where we go to our locust files folder and grab browse products module. Alright, now locust is running at this address, localhost port 8089, so let's control click that. Alright, here's locust, on this page we specify the number of users, that is our peak concurrency, for that I'm going to use 1,

next we specify the spawn rate or the number of users that should be started each second again i'm going to use one now for host we specify localhost port 8000 now make sure not to add a forward slash here because in our test script look our urls start with a forward slash so i'm going to remove this now let's start swarming so on this page we can see statistics about the requests sent to the server. So look, we have a few rows, we have post to this endpoint, we have post for creating a cart item, as well as getting all products and getting a particular product. Now this name you see here is the group we specified for grouping related requests. If we didn't apply grouping, we would end up with tons of rows in this table.

Now in this table we can see the number of requests sent to each endpoint, and as you can see we have more requests sent to this particular endpoint because we put a higher weight on this task. We can also see the number of failures, currently zero. We can see the median response time for each request. We can also see 90 percentile. That means 90% of the requests sent to this endpoint got a response in 280 milliseconds or less. We can also see the average, min, and max response time for each request. We can see the average size of each response. So, obviously, this endpoint for getting all the products has a much larger response, and this is by design, so this is not a problem in our application.

We also have current requests per second at each point, as well as the total in this test script. Now we can see this information in a chart as well, and this is getting updated in real time. So Locust has started one user, and this user is constantly browsing our website, executing the tasks we define. So if we go back to our terminal window, we can see what this user is doing. So this user was looking at these product details, then went to a few different collections, next, look at a particular product, now at this point, they added one item to the shopping cart. And as you can see, there is a delay between each task. So this is kind of simulating a real user browsing our website.

Now back to this chart, so here we have total requests per second, and you can see our requests per second are kind of in this range. Then we have response times, Now early on, we have a high response time, I believe that's because our web server is starting, so the following requests are getting almost the same response time. Now, over here we also have a page for failures, so if your web server crashes in the middle of a test, you can see your failures here, that translates to a connection refused failure. We can also see our exceptions, our tasks, and the ratio. We can also download all this data in CSV. Now, I'm going to stop this test. In the next lesson, we're going to do a proper performance test to identify our slow endpoints.

Running a Performance Test:

Alright, now we're going to do a performance test to identify our slow endpoints. So as you can see, I've removed my print statements in this test script. Now in this lesson, I want to deliberately create a performance problem. So let's go to our product view set. Now look, over here, we're eager loading a product with its images. Let's see what happens if we remove the call to prefetch related. So we want to study the impact of this on performance. Now, we should also go to our middleware setting and comment out Django debug toolbar because Django debug toolbar adds a bit of overhead and it's going to mess with our performance report. Now, back to the terminal window. I'm going to stop this process that is running locust and restart it because I want to start from a clean canvas.

No data from the previous test. Okay. now let's go to this address, now on this page, I want to use 500 for the number of users, and 10 for spawn rate. So every second, locust is going to spin up 10 users until we get to 500 users. Then those 500 users will continuously browse our website. Now let's specify the host as well, and start swarming. So it's going to take a while until we get to 500 users, then I'm going to wait another 30 seconds or so until the results become stable, so I'm going to get back in a few seconds. Alright, now I'm going to stop the test, alright, we started with 0 users, and over time, we got more users, now at this point we got 490 users, and then we got 500 users.

Now let's look at our statistics table. The first thing I want to highlight here is that we should not take any of these numbers as absolute values. Treat everything here as relative. So that doesn't mean that when you go to production, the average response time for this endpoint is going to be 179 milliseconds. First of all, we are running this on a development machine with a development web server which is slow. So this web server that comes with Django is meant for development. It doesn't have the performance of a real production ready web server. So if you run this test in a production environment, we're going to see a different result. And also, every time we run this test, again, we're going to see a different result.

So don't treat any of these values as absolute values. Now, in the first class we can see that this endpoint is taking longer to respond than others. So next we'll talk about performance optimization techniques.

Performance Optimization Techniques:

So by running a performance test, we can identify our slow endpoints. Now let's talk about a few optimization techniques. Yeah, most of the time, I would say more than 90% of the time, the issue is either in the query or the database. And here we have a few solutions. The first step is optimizing our Python code, because we're using Django RM to execute queries. So we should make sure that our Python code doesn't translate to costly queries. So we can preload related objects using select related or prefetch related. We can also load only what we need. So using the only method, we can load the fields we are interested in, like title. We also have defer, which is the opposite of only. So if you have a field with a lot of data that we are not going to show to the user, then we can defer it by calling the defer method.

We can also use the values and values list method. You know that with values we get a dictionary, with values list we get a list. Now initializing a Python dictionary or list is cheaper than initializing a Django model. So if you don't need any of the behavior in a Django model like creating, updating, deleting, then you can optimize by using one of these methods. Now, another thing you should watch out for is counting objects. So if you want to count products, this is the right way to do it. We don't want to pull all product objects in memory and then call the len function to count them. This is going to be super expensive. And one last thing is bulk creating or bulk updating.

If you want to create or update multiple objects, it's more efficient to use bulk create or bulk update than creating or updating a bunch of objects in a loop. Because with bulk create, we send one instruction to the database to create multiple records. Now, what if we use these techniques but our query is still slow? Well, then we may want to rewrite that query from scratch. Because the query that Django ORM generates may not be optimal. So if you know SQL well, This is where you may want to rewrite that query from scratch in an optimized manner. Now, what if the query is still slow? Then we need to tune our database by redesigning our tables, adding indexes and so on. I've covered this in more depth in my SQL course in case you're interested.

Now, what if our query is still slow? Well, that's when we're going to store the result in memory. This technique is called caching. So the first time the query gets executed, it's going to be a little bit slow, but then we'll store the result in memory and all subsequent requests will read the data from memory and this is often but not always faster than reading it from a database or network. Just remember, don't assume that caching is always a good strategy because sometimes executing a simple query is faster than reading the result from the cache. Especially in a production environment where you have a separate cache server. So reading the data from the cache incurs a network call. Now, what if we have done all of this and our performance test shows that

after a certain point, let's say a thousand concurrent users, our application fails, so some requests never get a response, or they just take way too long. Well, if we can afford, that's when we need to buy more hardware. So we can upgrade our server to one with a faster CPU and more RAM, but at a certain point, even that's not going to be enough. So then we need to add more servers. And of course, that's going to be more costly. Now if we can't afford that, then we'll have to learn to live with the problem. And that's okay. Not every part of our application should be blazing fast. We should do these optimizations in critical parts that matter the most. If there is a report that admins are going to pull up once every three months, there is no point optimizing those reports.

It's okay, they can wait. It's not the end of the world. We need to spend our time and effort wisely to get some value. Okay? Next we're going to talk about profiling.

Profiling with Silk:

So with locust, we can identify our slow endpoints. Now as I told you in the previous lesson, most of the time, the issue lies in the query or in the database. Now this is where we use another amazing tool to identify the source of the issue. And that is called Django silk. Silk is what we call a profiling tool, meaning we can use it to get an execution profile of our application. So we can see how each function gets executed, what queries are sent to the database, what time is spent on those queries and so on. So if you google Django Silk, you'll find this GitHub repository, Jazzman, let's look at the installation instructions, it's very simple, first we install Django Silk, then we register a middleware and add it in the list of install apps, next, we register a URL pattern, then we run migrate to create database tables, and optionally we collect static files, and don't worry about the last step, we'll talk about it later in the course, this is really unnecessary.

so here in the terminal, let's run pipenv install dash dash dev django dash silk. Alright, silk is installed. Now we need to make a couple of changes in our settings module. First, we need to go to our middleware setting. So here we need to register the middleware for silk. Now the placement of this middleware is important. Let me explain why. So as i told you before, each middleware function takes an incoming request, adds something to it, and then passes it to the next middleware function in the list. For example, our authentication middleware adds the current user to the request, right? Now, a middleware function can choose to process a request. If that happens, the request will not be passed to the next middleware function.

So that would be the end of the request's lifecycle. Now, because silk is going to intercept our requests, we need to make sure that the request gets passed to its middleware. so the request shouldn't get processed by a middleware function placed before silk's middleware, okay? Now in this case, none of our middleware functions process an incoming request, so we can put it anywhere, but I'm going to put it at the end. Now to save time, I'm going to go to their github page and simply copy paste this line, and add it right here. Next we need to add silk in the list of installed apps, so installed apps, I'm going to add that after Joeser, so all third party apps are in one place, and then we have our own apps.

So silk. Okay, now we need to register a route for accessing silk. So let's go to the urls module of the storefront folder. Now I'm going to add that here in this block, because we should use silk only in development and testing. We should never use it in production, because it adds significant overhead to each request. So back to the documentation, let's copy this line as well. and paste it right here. So we simply add a new path starting with silk, and here we're going to include the urls from the silk app. Now in the terminal, we need to run migrations because silk is going to create a new table for storing information about each request. So python manage.py migrate. Alright, so these are the migrations coming with silk.

Now I just realized I made a mistake. Back to our middleware setting. Look, I added silky middleware here and this is always there. So it's going to end up in production as well. Again, we should use silk only in development or testing because it adds a lot of overhead to each request. So here I'm going to write a conditional statement like this. If debug is turned on, then we're going to get middleware and add something to it. So we're going to add this. So, now silk is ready, and we can access it at this address. So this is the dashboard of silk, currently we don't have any data, so now if you start browsing our application, silk is going to intercept every request, and it's going to collect some information for us here.

Now, to simplify things, I don't want to manually browse our application, I don't want to hit each endpoint, so now i want to run another performance test, but instead of simulating 500 users, we can simulate five users, or even one user, just to send some requests to our backend. So, back to this window, let's start a new test with even one user is sufficient. Alright, so this is sending a few requests to our backend and silk is processing these requests in the meantime. Now if you have paid close attention, you can see that our response times are way slower than before. And that's because silk is intercepting every request getting some information and then processing the request. Alright, this is enough, so let's stop this.

Okay, now back to dashboard of silk, refresh, alright, take a look. So we have a few sections here, most time overall, you can clearly see that this endpoint store slash products is a very expensive endpoint. So here we have 13 queries, and that's because we removed the code for eager loading our products with our images. You can also see the time spent overall, as well as the time spent on queries. The second slow endpoint is this one. This is where we send a post request to this endpoint for adding an item to a shopping cart. Now here we only have three queries, so even though silk is saying we spent a lot of time on this request, there's really nothing wrong with this request, I actually tried to troubleshoot it, everything is fine with this operation, the main issue we have here is all these extra queries we have at this endpoint.

So, this is the most time overall section, we also have most time spent in database, so once again we can see our products endpoint here, there is another section, most database queries, there is a lot of information, you can spend your time figuring out how this really works, now in the request page, we can see all requests sent to the server, now by default we see them listed in cards view, we can change that to a table, now then I like to order by number of queries. This way we can quickly find the endpoints where we have all those extra n plus 1 queries. So let's sort, there you go. Now we can clearly see that this endpoint has a lot of issues.

Now we can click on one of these requests to see more information. So here are the query parameters for this request, in this case our collection id was 5, we can see some more information about the request, these are the request headers, we also have response headers, now on the top we have the sql tab, if you go over here, we can see all the queries sent to the database, so you can see we have 13 queries going to this table, product image, and that's because we have no joins, so that's pretty useful, we can also see the execution time of each request, so there is a ton of information here, so using silk we can dig deeper and find the source of the problem,

then we'll optimize, and once we're done, then we go back to the summary page, here we have clear db, we want to clear all the data that silk has collected, so we run another performance test and ensure that those issues are gone. So clear, good, now if you go to the summary page, there is nothing here. So that was a basic introduction to silk, you can learn more about it by reading their documentation or simply playing with this dashboard.

Verifying Optimizations:

with Silk, we identified the source of the issue. We found out that our product standpoint is sending extra queries to the database. So let's go to our product view set and preload our products with their images. So here we call prefetch related with images. Now this is where we need to run another performance test to make sure that this optimization is actually working. We should never make any assumptions. So we're going to run a performance test, but before doing so, need to disable silk, because as I said, silk is going to add a lot of overhead, so our performance report is going to be skewed. So, let's go to the middleware section. Now, even though we're conditionally adding this middleware, right now, because I'm going to run a performance test, we need to comment out these two lines, so silk is not messing up our performance report okay now back to the terminal where you're running locust, we need to stop this process

restart it to remove all the previous data. Okay, now back to the browser. Here we should use the same values we used earlier when we ran our first performance test. So earlier we used 500 users and started 10 users per second. So we're gonna run with the same parameters to ensure we have a fair comparison. Now previously I let the performance test run until we got to 500 users and then I let it run for a little bit longer and So the results were stable. So overall, 7200 requests were sent to the server. So I want to make sure we get to the same point and then stop the test. So let's start swarming. Alright, this time I let the test run for a little bit longer.

So now we have 7800 requests sent to the server. Now let's compare our metrics before and after the optimization. Alright, this is my favorite part. Look at the metrics before and after optimization. So for our products endpoint, previously, 90% of our requests got a response in almost three seconds or less. With our optimization, we reduced that number to 540 milliseconds. So previously, 90% of our users had to wait three seconds to get the list of products. Now they have to wait only half a second. That's an amazing improvement. Now because of this, if you notice, other endpoints are also performing better now. For example, when creating a cart, this metric was 670, we reduced it down to 160. Similarly, when creating cart items, our metric was 710, we reduced it down to 89.

Now look at the overall or the aggregated row. Previously, 90% of our requests got a response in two seconds or less. Now, after the optimization, that value is reduced to 650. milliseconds. That's amazing. Now look at the last column, current request per second. So previously, with this development web server, we were able to handle 120 requests per second, but now after the optimization, we can handle 140 requests per second. So before doing any optimizations, always run a performance test to get your baseline, then optimize, and run another performance test to see if you have really improved anything or not. Sometimes you may assume your optimization is going to make your application faster, but if you run another performance test, you would be surprised that not only didn't you improve anything, but you actually made your application slower.

Stress Testing:

Alright, the last thing we're going to talk about in this section is stress testing. So stress testing is a special type of performance test and we use it to find our upper limits. So we can find at what point our application fails. Knowing the capacity of our application and its execution environment is important, especially if there are times of the year where we expect spikes in traffic. For example, when we have flash sales. So back to the terminal, I'm going to stop locust and restart it. So we start fresh. Now on this page, I'm gonna use a high number of users because the point of this test is to break the application So previously we used 500 for our performance test. I'm gonna use Thousand and see what happens and I'm gonna spin up 10 users per second Now we don't want to test this against our development server.

That's really useless You want to find our applications capacity in a production environment? So we should run this test in an environment that is similar to our production environment in terms of the capacity now Not every company can afford this. I know that. For example, you might have a production environment with 10 servers, So you may not be able to set up a performance testing environment with the same capacity. But running a stress test, at least on one server that is similar to one of our production servers, is still valuable Because it gives us a rough idea of our applications upper limits. Now, for this demo. I have no choice but using our Development server, which is super slow compared to a production server, but let's see what happens.

So, let's start swarming. So I'm going to wait here and see at what point we start to get failures. That is going to be our upper limit. Alright, so I realized that after almost 600 concurrent users, our application started to fail. Alright, look at the chart for total requests per second. Now, this red line that we have over here represents our failures. So at this point, this is where we had 700 users, our application started to fail. So our failure rate per second was 1.5. And over time, this increased to an average of 11 requests per second. Now look at our RPS or request per second metric. Previously, we had an average of about 140 requests per second. After our application started to fail, our RPS dropped down to 10.

That's a terrible situation. But now at least I know the upper limit of my development server. So this is all about stress testing and that brings us to the end of this section. In the next section, we're going to talk about caching, which is an optimization technique.

Caching:

Introduction:

What is Caching?

So in the previous section, we briefly talked about caching. If you have a complex query that takes a while to execute, we can use caching to boost our application's performance. So the first time the query is executed on the database, then we get the result and store it in the memory. Now getting the data from the memory is often, but not always, faster than getting it from a database. So we'll serve subsequent requests using the data in the cache. So the future requests will have a shorter response time and our web and database servers will be freed up to process more requests. So on the surface, this sounds great. But there is a problem with caching. The problem is that if the data in our database gets updated, our cache is not going to get updated.

So the data is going to be stale or out of date until the cache expires. Typically, when we store an object in the cache, we can put an expiration time on it like five minutes or three hours, depending on how frequently the data gets updated. So if the data gets updated frequently, and we should always show the most up-to-date data to the user, then there is really no point storing it in the cache, okay? Now, caching is not limited to the result of database queries. For example, you might have to call a third-party API to get some data. Now, if that API is slow or even becomes unavailable, we can improve our application's performance by storing the result in the cache. We can also run a salary job to update the cache in the background every now and then.

So overall, caching is a great optimization technique, but as I always say, too much of a good thing is a bad thing. So if you cache data aggressively, you will need a lot of memory and that's going to cost you a fair bit. But also, caching may even reduce the performance of your application. So don't assume that just because reading data from the memory is faster than reading it from the disk, then you should always store your data in the cache. Some database queries get executed pretty quickly and your application will respond faster if you simply get the data from the database every time. The only way to know this is by running a performance test before and after optimization. So don't make assumptions without a proper performance test.

I love this old saying, premature optimization is the root of all evil. Alright, next we're going to talk about different types of cache.

Cache Backends:  
Django comes with various types of cache backends or cache engines. So we have local memory, which is the default cache backend and stores data in the same process that runs our Django application. It's good for development, but not a good choice for production. In a production environment, we should use a proper caching server. And for that, we have memcached and Redis. Both of these are enterprise cache servers, so you can choose either. In this course, because we have been using Redis as our message broker, I prefer to use it as our cache backend as well, so we don't have too many dependencies. Now if you're familiar with memcache and prefer to use it in your application, just look at Django's documentation for configuring memcache.

It's pretty straightforward and very similar to what I'm going to show you in this section. Now we can also use our database as a cache, so if you have a complex query that hits multiple tables and takes a while to execute, then we can store the result in the database and quickly pull it out for subsequent requests. This is faster than executing that complex query every time, but it's not as fast as pulling out data from the memory. So if you don't have access to a cache server, then you can use the database backend. And we can also use the file system as a cache, so the result is stored in files. Not something that we use that often, but it's there in Django in case you need it.

So that's all about cache backend. All these backends except Redis come with Django by default. For Redis we have to install a separate library, and we'll do that later in this section.

Simulating a Slow API:

Alright, for this section, we're going to simulate a slow API endpoint. So let's go to the Hello view of the playground app, and make a couple of changes here. Here, we don't care about notify our customers anymore. So did it this line, and this line as well. Instead, we're going to import the requests module built into Python. And using this module, we can send an HTTP request to another service. So here we're going to call requests that get we're going to send a request to HTTPS HTTP bin org slash delay slash to now when we hit this endpoint, this server is going to wait two seconds to respond to us. So this simulates a slow third-party service. Now let's test this. So back in the browser, let's go to playground slash hello.

So look, we're waiting for two seconds and now we got the response. Okay. As I told you in the previous section, whenever you want to optimize, First you need to run a performance test to get your baseline. We'll do that next.

Getting a Baseline Performance Benchmark:

All right, now we're going to run a performance test to collect our baseline. So let's go to the locus files folder and open our test script. We're going to define a new task for hitting our slow API endpoint. So we can define it anywhere. It doesn't really matter. I'm going to call this say hello and decorate it with task. And here we simply call self.client.get. We hit slash playground slash hello. make sure to terminate it with a forward slash. Now, here in the terminal, let's restart locust. Good. Now, on this page, I'm going to use 500 users and spin up 10 users per second. Now let's see what happens when we start swarming. Alright, I'll let this run until we send about 7,000 requests to the server.

Now look at the first row for the hello endpoint. 90% of our users got a response in 2,600 milliseconds or less. And our total aggregated metric is 2,400. So I'm going to keep this report somewhere so after we implement caching, we can compare the metrics.

Installing Redis:

Earlier in the course, we talked about installing Redis. And I told you that the easiest way to run Redis is using Docker. So before going forward, let's make sure that Redis is up and running on our machine. So here we run docker run dash d for the detach mode or running this in the background. Then dash b for port mapping. So we're going to map port 6379 of our local host to 6379 of our Docker container. And the image is Redis. Okay, this is running. Now if you get an error here, that means you're already running Redis in another window. Let me show you. So let's run this one more time. Look, I got this error saying port is already allocated. So now that Redis is running, we're ready to configure caching.

We'll do that next.

Configuring Caching:

Alright, to use Redis as a cache in a Django application, we should also install a library called Django Redis. So, here in the terminal, let's run pipenv install Django-Redis. Good. Now to configure caching, we're going to get back to this GitHub page, so jazzband slash Django-Redis. We're going to grab a piece of code from here. That is our cache configuration. I'm going to copy this. Now, let's go to our settings module and paste this new setting at the end. So we are defining our default cache. As we can see, the backend is Redis cache. The location is the location of our Redis server. So we are pointing this to local host, this port. Now, here we're using database number one, but earlier, we used database number one as a message broker for Celery.

Remember that? So now we need to use a different database for our cache. Alright, so our configuration is ready. Next, I'm going to show you how to use the cache API.

Using the Low-level Cache API:

Alright, now let's talk about using the low-level cache API. So, back to our playground view, from django.core.cache, we're going to import the cache object. Now this object has an API for accessing the cache, so it has methods for getting objects from the cache or storing them in the cache. So, in this view, instead of immediately calling HTTP bin, we're going to check our cache and see if we have the data we're looking for. If we have the data, then we're going to serve this request using data from the cache. Otherwise, we're going to call HTTP bin to get the data and then we'll store the data in the cache for subsequent requests. So, here we're going to say if cache.get now here we need to pass a key for accessing the data we're looking for.

We can call that key anything, I'm going to say HTTP bin underline result. Now if this is none, that means we don't have the data in the cache, so then we're going to call HTTP bin to get the data. So here we get a response, then we call response.json and store the result in data. Next we call cache.set to store the data in the cache. Now here we need to pass a key and a value. So for the key we're going to use the same value as before, and the value is the data. Now we have repeated the key in two places, so if you have a typo our code is not going to work. So a better way is to store the key in one place http bin underline result and then we can replace both these instances with key okay so if we don't have the data in the cache we're going to get it store it otherwise we're going to serve this request using the data from the cache so instead of rendering my name on this template i'm going to render cache.get key okay now back to the browser

The first time we hit this endpoint, our request is going to be slow, it's going to take 2 seconds, but all subsequent requests will be served from the cache, so they will be super fast. Take a look. So, 1 and 2 and we got the result. This is the JSON object that we get from HTTP bin. I'm going to refresh this one more time. Look, we got the result immediately. Now you might be wondering how long the data is going to store in the cache. Well, the default timeout is 5 minutes or 300 seconds, but we can easily change this. So when storing this data in the cache, here we can supply a timeout, let's say 10 times 60, so that's going to be 10 minutes.

Alternatively, instead of repeating this everywhere, we can set the timeout globally. So we go to our cache configuration setting and set the timeout right here. So we set timeout to let's say 10 times 60. Okay? So Using the low level cache API by calling the get and set methods, we have precise control over caching objects. But using this API in every view that needs caching is going to be a little bit tedious. It's going to be repetitive. Every time we have to define a key, then repeat logic like this. So in the next lesson, I'm going to show you a simpler way to cache data.

Caching Views:

Alright, let me show you a better way to cache your views. So instead of repeating this logic, in every view that requires caching, we can use a decorator. So over here, from django.views.decorators.cache, we import cache page decorator. Then we apply it to our view and give it a timeout, let's say 5 minutes. Now with this change, we can get rid of this entire Caching logic, so we don't have to use the low-level API anymore. We simply get our data from HTTP in and Render it on the template. So I'm gonna grab these two lines and delete these few lines. Now, instead of getting the data from the cache, we simply use the data that we retrieved from HTTP bin. So in this implementation, it looks like we have no caching, but the entire caching is controlled using this decorator.

Now how does it work? Well, when we hit this endpoint, the hello endpoint, Django is going to look at our cache and see if there is a key by this name. Now the actual key name is a little bit more complex, it's going to be a long string that Django automatically generates. So the first time we hit this endpoint, Django knows that there is no item by this key in the cache, so it will execute the body of this view. So that's why the first hit is going to be slow. Then, Django is going to store the result of this view or this function in the cache using that key. So the next time we hit this endpoint, the data is going to come from the cache automatically.

The beautiful thing about this approach is that we don't have to worry about any of this low level complexity. Django will take care of it. So, let's test this in the browser. So let's go to playground slash hello. Now the first call is going to be a slow, now we got the result, but if we hit this endpoint one more time, look, we get the result immediately. So caching is working. Now when using caching, there's something you need to be aware of. Let's go back to our view and make a small change here. So instead of passing the data, I'm going to pass my name. Now, let's refresh. Look, we are not seeing my name, because the data is coming from the cache.

So we have to wait until the cache expires in this case, five minutes. Now there are situations where you want to deliberately expire the cache. I'll talk about that later in this section. So don't worry about that. All I want you to know here is that if you make any changes, your changes are not going to be visible until the cache is expired. Okay? Now, here we're using a function-based view, but what if we had a class-based view? This cache page decorator doesn't work on class-based views. For that, we need another decorator. Let me show you. So, let's quickly convert this to a class-based view. Do you remember how to do it? First, we define a class called hello view. This should extend API view in REST framework.

So, from rest framework that views we import the api view class. Then we add it here. Next, we define a get method with two parameters, self and request. And finally, we move our logic to this new method. Okay, now let's get rid of this. Next, we need to register this in our urls module. So let's go to the urls module. of the playground app. And over here, we say, hello view, dot as view. Okay? Let's test our implementation before implementing caching. So back to the browser, refresh, one and two. Okay, we got the response. Now if i hit it again, again, one and two. So every time we hit this endpoint, we have to wait two seconds. Now let's implement caching on this class based view so we cannot apply the cache page decorator here,

going to get an error. So to solve that error, we have to wrap this inside another decorator. So over here, from Django dot utils dot decorators, we import method decorator, then we wrap this with method decorator like this. Okay, now, let's test this one more time. So refresh, Now we see the previous result because we already have a key by this name that is based on this endpoint in our cache. So to recap, for function based views we use cache page decorator, for class based views we use method decorator.

Verifying Optimizations:

implemented caching as an optimization technique. Now let's run another performance test and see what we get. So first, I'm going to restart Locust. Good. Now, on this page, previously we used 500 users and ramped up 10 users per second. Now, before recording this video, I ran a test with these values and I was getting a lot of failures because now our application is so optimized so we get a lot of concurrent users pretty quickly. more than this web server can handle. So then I played with different values, I reduced the number of users to 200, no more errors, but now that means we have to run two tests. One before optimization, one after optimization with the same parameters. This way we can have a fair comparison.

So back to the code, I'm going to comment on this line to disable caching and run a baseline test. Now this is going to take a while So I'm going to pause the recording and then run another test with caching enabled. Then I'm going to come back and compare the result. Alright, here's the end result. So as you can see, I let both these tests run until we had about 2000 requests. This way we have a fair comparison. Now before we implemented caching, 90% of our requests got a response in 2300 milliseconds or less. With caching, we reduced that number to 51. That's a huge improvement. Now, in the last column, look at our current request per second. Previously, we had 62.4. By implementing caching, we slightly improved this number to 64.9.

Managing Redis Cache Content:

Alright, the last thing we're going to talk about in this section is managing Redis cache content. So you know that we're running Redis using Docker. So if you run Docker PS, we can see our running containers. A container, as I told you before, is like a lightweight virtual machine. So imagine we have a lightweight virtual machine with this ID that is running Redis. Now, you want to go in this virtual machine and execute a command. We want to launch Redis command line interface. So we type Docker exec for executing a command, dash it, so we can execute this command in the interactive mode, so we can interact with the terminal window, next we need to type the container id, so on my machine this is what we have, on your machine you're going to have something different, all you have to type here is just the first few letters of the container id, so dfeab is sufficient for me, then we type the command that we want to execute inside this container, so redis dash cli, ok,

So this is ready CLI or Redis command-line interface here. We can send instructions to our ready server For example, you know that we have two databases database. Number one is used for our message broker database Number two is our cache so we can say select two Now you're on database number two. And by the way, Redis databases have numbers. They don't have names So they have one two, I think 16 which is the default so we are on the second database now to see all the keys we type keys space asterisk. So currently there is nothing in my cache because I had a break before the last video. So what I previously had in the cache is expired. So now if you go back to the browser and hit the send point, okay, now Django stored the result of this view in the cache.

Let's verify it. So let's run the last command one more time. There you go. So we have two items in the cache and look at the key. This long key that you see here is generated by Django automatically. So we have cache header and cache page. Now if you want to delete a particular key, you can use the delete command. So we grab the entire key name and we type del followed by the key name. In this case, I don't want to do this. Instead, I want to delete all the keys. And for that we type flush all without a space. Okay? So now the cache is empty. So if we hit this endpoint one more time, again, we have to wait two seconds.

So that brings us to the end of this section. I hope you learned a lot. In the next section, we're going to talk about preparing for production.

Preparing for Production: