Flask: Getting Started

by Reindert-Jan Ekker

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Related

Course Overview

Course Overview

Hey everyone, my name is Reindert-Jan Ekker, and welcome to my course, Getting Started with Flask. I'm a longtime developer and educator, and in this course, I will teach you about creating web applications with Flask. This is a framework that will allow you to become productive and create your first working website in almost no time at all and with almost no code at all. In this course, we're going to create a small but complete web application, and by doing so, we'll cover all of the basic building blocks of a Flask web application. Some of the major topics that we'll cover include creating Flask view functions and mapping them to URLs, using Jinja templates to create HTML pages, creating web forms, and handling user input from those forms. By the end of this course, you'll know all major features of Flask, and you'll be ready to begin writing code for a Flask web application. Before beginning the course, you should be familiar with Python on the basics of HTML. I hope you'll join me on this journey to learn web development with Python and Flask with the Getting Started with Flask course, at Pluralsight.

First Steps

Introducing Flask

Hi, I'm Reindert-Jan Ekker, and welcome to this course about getting started with the Flask microframework. In this module, I'll give you a short introduction of Flask and go over what you should already know to be able to watch this course. Then we'll quickly start coding and create our first web page with Flask. We'll take some time to really understand how Flask serves our pages, or, more specifically, how Flask maps URLs to functions. I'll address some common problems that you might run into, and we'll see a short demo of the Flask debugger. Flask is a framework for writing web applications. It's a microframework, which, as the name suggests, means it's small, clean, and simple. Flask comes with a minimal amount of features, and that means it consists of a minimal amount of code as well. Now, that may sound like a strange choice to you. Why would you choose a library that does as little as possible? In this course, you'll see that Flask apps, especially smaller apps, tend to have very clean and simple code bases. In many cases, you'll be able to build your whole application in a single file, and your code will still be clean and pretty. Flask doesn't force its opinion on you, and with that, I mean that it doesn't tell you to follow certain conventions or architectures. And this makes Flask a very flexible framework to work with. You can just mix and match the extensions you need for the features that you're going to use. For example, if you need a database layer, you choose a database extension, but if you don't need that, you can leave it out. And the same goes for adding authentication, implementing REST, you name it. This is very different from a framework like Django, for example. Django comes with many features, and it's very opinionated. In a Django project, you follow conventions and architectural choices that the Django project team has made for you. Whereas in a Flask project, on the other hand, you're free to make those choices for yourself. Not having so many features also means that the features that are included with Flask tend to be very well documented. And if that's not enough, Flask is one of the more popular Python frameworks, and it has a large community. Most of the time, you'll find the correct answer to your questions on sites like Stack Overflow in no time. So what features does Flask provide? Flask is built on two main components. There's the Jinja 2 template engine for building HTML, which, by the way, is one of the most powerful templating engines for Python, and it's a joy to work with. If you're familiar with Django templates, you'll also find that they're really similar, so there's not a huge learning curve at all. And the other main component is called Werkzeug, which is a German word that means something like a tool, and this provides the HTTP support and does the routing that maps URLs to Python functions. There's also a built-in development server and debugger, and Flask comes with built-in unit testing support. In this course, you'll learn how to use this tiny set of features to build a tiny application. The only part we'll not touch in this course is the unit testing support. That's for another course. Before we start writing code, here's a short overview of things that you should already know before watching this course. First of all, Flask is a Python framework, so you should be familiar with Python. At the very least, watch the introductory Python course and write some simple programs. Although Flask does use some advanced concepts, we'll stick to the basics, so a general understanding of the core language will probably be enough to get you started. You should also know how to use pip to install packages. If you don't know how to do that, please watch my course on Pluralsight called Managing Python Packages and Virtual Environments. By the way, Flask recommends using the latest version of Python 3, so I'm going to assume that you have that installed. Although the demos in this course should work with any more-or-less recent version either of Python 2 or 3. You also should know the basics of web development. That means you should at least know how to write a simple HTML page. It may be helpful if you also know a tiny bit about CSS styling. Although you should be able to follow everything even if you don't. We'll also touch on the HTTP protocol a bit, which is the protocol that enables a browser to ask a server for a web page. But if you don't know about HTTP, don't worry. It's not hard at all, and I'll explain everything you need to know.

Demo: Installing Flask and Starting a Project

Now let's really get started. I'll quickly go over the steps for installing Flask, and then we'll see how to create a simple web application, run it, and inspect the results in a browser. For this course, I will use the community edition of PyCharm as my editor. You should be able to follow this course if you use another editor. I will tell you when you will need to take different steps than I did. In my case, I start by creating a new project, and this gives me the New Project screen. As the location for my project, I choose flask-getting-started under PyCharmProjects. PyCharm will create this folder for me, and if you work with another editor, you might need to create your own folder for your project. PyCharm also automatically creates virtual environments for my projects. If I click on the triangle here, you see that it will create a virtual environment for me in my .virtualenvs folder, and the name of the environment is the same as the name of the project. It's based on my latest installed Python version, which is 3.6. Now, if you're not using PyCharm, you probably want to create your own virtual environment, and if you don't know how to do that, please take a moment to watch the module about that in my course, Managing Python Packages and Virtual Environments. Anyway, I like the default settings, so I'm going to click Create. Now, this is the empty project PyCharm has just created for us, but before we can start writing code, I need to install Flask. To do so, I want to start a terminal in an active virtual environment. Now, the course I mentioned before will show you how to do that in general, but PyCharm actually has a nice shortcut for us. If I click Terminal at the bottom here, it starts a terminal session. And as you can see at the start here, we are already inside our active environment. That means that to install Flask, I only have to say Python -m pip install flask. And this downloads and installs Flask, as well as its dependencies. The most important ones are Jinja, which we will use to write templates, and Werkzeug, which is the online library that handles HTTP requests. Now instead of starting your own project and writing the code yourself, you can also download the demo code from Pluralsight. And the project is also available on GitHub, so you can also clone the GitHub project. It's available at the location shown here. In any case, you will always have to create your own virtual environment and install Flask. I cannot do that for you. If you download the demo code, the project will look slightly different, by the way. Let me show you. As you can see, there are now two extra files here, .gitignore and requirements.txt. The first one is Git specific, and the other one lists the packages that this project depends on. If you download the demo code, you will get these files, but the project will work fine without them. Good, so let's get this

Demo: Creating Your First Web Page with Flask

project started. In this course, I will create a little application that helps me learn a new language by creating so-called flashcards. I'm going to start by creating a Python file called flashcards.py, and I add a few lines of code. What you see here is all we need to host our first simple web page, and that's one of the things that makes Flask so popular and so nice to work with. It takes almost no effort to get started. Before we run and look at the resulting page, let's go over the code. The first line imports the Flask class from the Flask package. Of course, this will only work if you correctly installed Flask. The next line calls the Flask constructor, which will create a global Flask application object. I pass to the constructor the name of the application, and in most simple cases, you can pass dunder name here, which is a special variable containing the name of the current module. So what I pass here is the string flashcards, since that's the name of my Python file. Next, I define a function called welcome that does nothing else than return a string, Welcome to my Flash Cards application. By itself, this is just a function, but the line above it changes it from just a function into something called a view function. Now in case you're not familiar with the syntax starting with an at sign here, we call this a decorator. @ app.route is an attribute of the app object we made in the line before, and we use it to decorate our welcome function. The effect of this is that we're assigning a URL to our function. In this case, the URL we assign is /, which is the so-called root URL of our web application. To see this in effect we need to run our application.

Demo: Running the Flask Development Server

To run our project, we need to start a terminal and make sure we're inside our active virtual environment. There are three steps to take to run our projects. First we have to tell Flask in which module our application can be found. We do this by setting a variable FLASK_APP, and in my case, I do this by saying export FLASK_APP= flashcards.py. Note that I'm not typing the dollar sign. That was already here before I started typing. The text I entered is export FLASK_APP, etc. It's very important here that you don't add spaces around the equals sign. If you add spaces, the command won't work. Now in case you're on Windows, the command export doesn't exist, and you should use set instead, like this. So this is the command if you use Windows. Set FLASK_APP= flashcards.py, and on Linux and macOS we use export. So let me fix that again. Now I also need to define a second variable called FLASK_ENV, and I set that to the value development. Again, when you're on Windows, use set instead of export. Now this tells Flask that we are still developing the application, and it enables some development feature like the debugger. Now with both of these variables set, we can run Flask by saying flask run. And if everything goes well, you should see this. We're serving flashcards.py, and the server is running on the address shown here. I can copy/paste this address into a browser, and it shows as a page with the string from my code. So there you have it, our first web page with Flask. I always think it's quite impressive how fast you can get up and running with this framework. Now by the way, one important remark. Although it's really nice and easy to get started this way, you should never, ever run your project like this on production. The Flask development server is simply not meant to be used for production, and, among other things, it's simply not secure. It's perfectly fine for development purposes, but once you want to put your website live on the internet, there are better ways, and we'll talk about that later in this course.

Demo: If the Server Doesn't Run

Now in case running the server doesn't work for you, there are a couple of things you can check. I'm going to start by stopping the current process, and we can do that as Flask shows us here by pressing Ctrl+C. This brings me back at the command line. And first of all, I check that I'm actually in the correct folder. We should be in the folder where my Python file is actually located. Also, I need to make sure that the value of the FLASK_APP variable is exactly the same as the name of my Python file, flashcards.py. Also make sure that the line where you set the variable is exactly correct, so no extra spaces, etc., etc. Remember on Windows you use set instead of export here, and make sure you didn't add spaces around the equals signs. In some cases you get an error about not being able to ______ the module flask. This might be caused by a slight misconfiguration where the Flask executable can be found but not the Python module. And usually you can work around this by typing python -m flask run, instead, and that should work for you. By the way, a similar problem can also occur if you mistakenly called your Python file flask.py. If you call your file flask.py, that will make it impossible for the Flask library to reload it, so make sure not to do that. Never call you script flask.py.

Demo: Flask and the Flow of Control

Now that we have a running application, let's go a little deeper. We'll add some extra pages and get some deeper understanding. We'll learn about the way Flask controls the flow of our application, how Flask maps URLs to functions, and the Flask debugger. Let's take a moment to talk about control flow. Looking at our code, you might have noticed that it doesn't behave like a regular script. It doesn't simply run from top to bottom. Instead, what controls the flow of our code is the Flask server process. Right now, I have a running Flask process, but it doesn't really do anything because what a web server does is to wait for incoming requests. In other words, our program will do nothing until a request comes in. To make a request, I have to visit our site. Whenever I visit a page in my browser like this, the browser sends an HTTP GET request to the server. And you can think of this like a tiny text message asking the server for a specific page. In this case, my URL specifies that the browser should try to contact 127.0 .0 .1 at port 5000. The IP address 127.0 .0 .1 is a special address that always points to the localhost, so on my machine, it'll point to my machine, and on your machine, it will point to your machine. And the process listening for connections on port 5000 is my Flask server. If there's nothing else in the URL, we're visiting the root page for this site, and the name of that page is /. So there's an implicit slash here at the end of the URL that you usually cannot see. That is why the server log now shows that the server received the GET message for the URL /. Flask looks up the view function mapped to the URL, and here we see that I've mapped the welcome method to the URL /. So Flask will call this view method and send the return value back to the browser, which then shows the message. So, the flow of my application is dictated by the HTTP requests that come in. Any view function I define will only be called if a request is made for the corresponding URL, which, for now, I can assume to mean that the URL has been visited in the browser. So this also means that every time I refresh the page, a new request is made, as we can see in the log, and the view function is executed. To show you that the view is executed every single time, let me add a second page to the application. I'm going to copy/paste the existing function, and I'm going to serve this new page at the URL /date. And let's call the function date as well, and the page will return the message, This page was served at, followed by the current time. For this to work, I have to import datetime, so at the top of my file, I'm going to say from_datetime import_datetime. Very well. Now, when the server is in development mode, it will automatically detect changes in our code. Remember that we set the FLASK_ENV variable to development before running Flask? Well, because we did that, the server now shows that it detects changes and restarts the application automatically. So we don't have to restart the server ourselves. We can simply go to our new URL /date, and, pressing enter, we now see the request come in for /date. Every time I press refresh, you now see the time shown in the resulting page changing. I'd now like to add a third

Demo: The Flask Debug View

page to our web app and go over some of the possible problems that you might see. Now, if you're working alone, please take a moment to try to implement this yourself. The exercise is as follows: Please add: A page that shows how many times it has been viewed. If you're going to try to implement this yourselves, please pause the video right now and write the code. I'll wait for a couple of seconds to show the solution. So, here's some code that hopefully looks more or less like what you have tried. First, I create a counter variable that I initialize at 0, and then I create a new view function and assign it a new URL /count_views. Then in the view function, I add 1 to the counter. Because the variable is declared outside the local scope, I have to use the global keyword to be able to do so. Next, I return the text, This page was served, followed by the value of the counter and then the string times. And then when we visit the new URL count_views, we see a page that counts views, as we can check by refreshing the page. So again, this shows you that the view function gets called every time the browser requests the page from the server. Now when you create a new view by copying an old view function, there's one common problem that may occur. So let's say I created this function here by copying the welcome view above, and what happens a lot is you forget to change the name of the function, so the result would look like this where we now have two functions called welcome. And this is perfectly good code. In fact, there's no connection between the URL and the name of the method. So just because the URL is count_views, that doesn't mean that the function should be named anything like that, so we can call it welcome. But now when I refresh the page, I see this, showing me that I have an error in my code. View function mapping is overwriting an existing endpoint function: welcome. So whenever an error occurs in our code, Flask will show an error page like this. Usually it's very helpful to read these messages carefully whenever a debug page like this shows up. By the way, showing this kind of debug information to the outside world is quite dangerous. That's why Flask only shows this when you explicitly tell it to, which we've done by running Flask in development mode. Remember when we set the FLASK_ENV variable to development? So setting this causes Flask to display helpful debug messages. But that is also one of the reasons you never want to run your application like this in production. Good. Let's fix our view by changing the name of our function. Again, this name doesn't have to be related to the URL, but it has to be unique. There cannot be a different function with the same name. So let's set it to something else. Let's say count_demo. By the way, in many cases the debugger can even show you exactly where an error occurs in your code. For example, if I forget to use the global keyword, and let's just ignore the errors in PyCharm right now, refreshing the page, the debugger tells me exactly where my error occurs, which is on line 24. So how does Flask map URLs to

Demo: How Flask Maps URLs to View Functions

view functions exactly? Well, let's stop Flask for a moment, again by pressing Ctrl+C, and now I'm going to start the Python interpreter by running Python. And if you work along, make sure you do this inside our virtual environment. So doing this I can now import my own module by saying import flashcards, and once I do that, I can access the app objects, which, as you may remember, is created at the top of our script. So I'm going to say flashcards.app, and this object has an attribute called url_map. And this shows an entry for each URL we have defined, count_views, date, /, and it also shows to which view function each URL has been mapped. And the middle part, HEAD, GET, OPTIONS, are the HTTP method that each mapping supports. And we'll see more about that later in this course. So when I go to any URL, let me just start Flask again for a moment, let's say I'm going to go to date. What happens is that Flask looks up the URL date in this map, and it then calls the corresponding method. In this case, it's also called date. So when you try to visit a URL for which no mapping exists, let's make a typo, datte instead of date, Flask tells you this URL is not found. So the first thing to do is to check that the URL you have in the browser matches the URL in your Python code. Going back to the URL map, here's one rule we didn't talk about yet. This line is added by default in any Flask application, and it points to a function called static. And this function will serve any files found in a directory called static and will serve those files under the URL /static, followed by the filename. So as an example, I created the static directory, and in there, I put the Flask logo. So in every Flask application, by default, there's already a static view function to serve this file and a URL mapping for that function. So now we can go to the URL static/ flashlogo.png, and the browser then shows me this picture, which is actually the flasklogo.png in the static folder in my project. So Flask provides this static URL mapping to make it easy for us to add things like images, CSS, and JavaScript to our application, and soon we'll see more of that.

Review

Let's go over the main things we've just learned. We started by installing Flask by calling pip install flask. It's best practice to always install things inside a virtual environment, and things are no different with Flask. If you use PyCharm, it will actually create an environment for you automatically when you set up a new project. By the way, for people using Anaconda, in your case, Flask comes installed by default, so you shouldn't actually need to install anything. Once Flask is installed, you can create a new Python file and start coding. At the core of every Flask application, there's a Flask object, and we start by creating that. First we say from flask import Flask, and then we call the Flask constructor and pass the name of our current module. The result is stored in a variable called app. Now we can start adding pages to our application. To do so, we add functions that we assign to URLs. In the code, the URL mapping comes first. We say @ app.route, which is a so-called decorator, and this applies the URL mapping to the function defined immediately below it. In the demo, we've seen that this actually adds the mapping to a special attribute of the app object called the URL map. Now @ app.route is actually a special kind of function call, which can take arguments, and we pass it the URL for this page. In this example, the URL is my_url. Make sure to always start these URLs with a slash. And by the way, there's no connection between the URL and the name of the function, even though in practice they'll often be the same. Now the result of all of this is that when a user now browses through the URL /my_url, the browser will send a request for this URL to our Flask server, which in turn will call our view function. Once we have an app object and a view function, we're ready to run our application. There are three steps to this. First we have to tell Flask where our application is located, and we do so by setting a variable in the terminal called FLASK_APP. On macOS and Linux, you do this by saying export FLASK_APP=, followed by the name of your Python file. On Windows, it works the same, but you use set instead of export. And make sure not to add any extra spaces around the equals sign, or this may not work. Next, we define a second variable FLASK_ENV, and we use this to tell Flask that we're running in a development environment. This enables things like automatically reloading changes in our code and the Flask debugger. With both of these variables set, you can run the application by saying flask run. In some cases where your Python path is not set up completely correctly, you can try to run by saying python -m flask run. Now if you need to stop the server, you can do so by pressing Ctrl+C. Now one thing I cannot repeat enough is that you should never, ever do this in production. The Flask production server is not secure or performant enough for a real-world deployment, so you should never do that. We will cover deployment later in this course, and we'll learn how to handle this correctly. Finally, if you want to work along with this course, here's the URL to the GitHub repository that contains all of the source codes. Or, if you prefer, you can download the code for each module from Pluralsight. And that's it for this first module. I've given a short introduction about Flask and why it's so popular, and then we went right into coding, installing Flask, creating our first pages, and running them. The two major concepts we've needed so far to create web pages are view functions, which are just regular Python functions which we can assign to a URL in our web application by adding the app.route decorator, which creates a Flask URL mapping. Very well. Let's move on to our next module, in which we'll learn about the model-template-view architectural pattern and add actual model and template code to our projects.

Understanding the Model Template View Pattern

Introducing Model-Template-View

Hi. I'm Reindert-Jan Ekker, and in this module, we'll learn about the architectural pattern used by most Flask applications called model-template-view, and how to implement it. Just about every modern web framework, and that includes Python frameworks, follows the same basic pattern called model-template view. You can see this as a set of best practices for how to organize your code. There are three types of components, models, templates, and views, and each has a clear responsibility. In this module, I'll teach you how to implement this pattern in its most simple form with Flask. By the way, you might know the same pattern by the name model-view-controller, and that's the same thing by another name. In the world of Python web frameworks, people are just used to calling it MTV instead of MVC. Now views, or view functions, we've already seen. So, to make our application follow the MTV pattern, we have to add templates, and we'll use a library called Jinja for that. And after that, we'll add a data model. Flask doesn't give us any standard way to do this, and usually you would use an extension to connect with some kind of database like MySQL or Oracle. But in this course, I want to focus on the core of the Flask framework, and that means that I will use a file with some text data in a format called JSON, and I won't connect to an actual database. This will keep our code much simpler, and we still get to follow the MTV pattern. And of course, I will talk a bit about what the responsibilities of each of these components is and how they are supposed to interact.

Demo: Jinja Templates

Let's start by implementing a new kind of component called a template. We'll use a library called Jinja for that. Templates are the components that are responsible for displaying our data to our users. In a web application, displaying data means creating a web page, so we'll use these templates to generate HTML that the browser can display. We'll see how to call these templates from the view and how to pass the data we want to show from the view to the template. To display that data, we will use something called Jinja variables. Before I start, let's remove some of the example views that we wrote in the last module and that we don't actually need anymore. So I'm going to remove date here, as well as the count_views page. After doing that, there's an import at the top that we also don't need anymore, so I'm removing that as well. Now so far, we've written views that return strings, and those strings are the content of our pages. But in a real application, you want HTML pages, and those can quickly become large and complex. And because we don't like to have huge strings with complex HTML code and maybe even JavaScript code inside our Python code, we want to create a separate component called a template that will generate that HTML for us. And that will keep the view function nice and small and clean. Now Flask has a default location where it expects these templates to be, and we need to make this ourselves, so I'm going to create a folder here. It has to be in the top level of our project, and I'm going to call it templates. Now make sure to call this templates. Don't forget the s, or it will not work. And in here, I'm going to create a new HTML file. I'm going to call it welcome.html. Now if your editor doesn't have this option, simply create a text file with the same name, welcome.html. And now we see some code that PyCharm gives to us when we create an empty HTML file. It's a very minimal piece of HTML that has all the basic building blocks for an HTML page. There's a DOCTYPE, there's a head and a body part, and in the head we can set the title for the page. That's what will show up in the tab of the browser. Let me fill this in. And the body will contain the actual content for our page. I'm going to copy/paste some code in there. What we see here is an h1 tag, which will show in the browser as a header saying, Welcome to my page!, followed by a paragraph tag containing come text. This is the demo application for the course Getting Started with Flask, on Pluralsight. There's a link pointing to the Pluralsight homepage, which will be clickable, and I've added an image tag as well. Note how I'm referring to the image in our static folder. This is served by Flask at the URL /static/ flasklogo.png, and that makes it possible for me to use that URL as the source attribute for this image tag here. Now to make this page actually show up in the browser, we have to use this template from a view function. So let's go back to our Python file, and the first thing I have to do is import a function called render_template. And in the view function, instead of just returning the text for the page here, we call render_template. So we delegate generating the content for our page to a separate component, the template. And we pass the name of the template file to render_template, and when this function is called, Flask will look for a file called welcome.html in the templates folder, and it will find the template we just created. At the moment, our templates file is simply a plain HTML file, and this function call will just return the contents of our templates file. This will be a string containing the HTML we wrote. In turn, we return that value from the view function, and Flask will make sure to send it to the browser. Finally, the browser receives the HTML and displays the page that we see here. Now in case you're working along and you get an error, let me go over some pitfalls. First of all, what happens a lot is that you forget to use the return keyword. And this happens to the best of us. You call render_template but forget to return the value. The result is, refreshing the browser, this error message, the view function didn't return a valid response. So if you see this, you probably forgot to return the value from render_template. Another thing that happens frequently is that the templates folder is not named correctly, so make sure it's called templates with an s at the end. Or maybe your template name doesn't exactly match the arguments to render_template. So for example, let me remove something here. And again, going back to the browser, now we see another error, TemplateNotFound. So if you see this error, you may have a problem with the name of your template. Now let me fix that as well. And before we move on, there's a last point I want to make about templates.

Demo: Jinja Variables

So this template file looks like a regular HTML file, but it can actually be a bit more than that. Let me insert something. Now this here, the structure with the double curly braces, is called a Jinja variable, and it's a placeholder that will be filled in by Jinja when the template is rendered. Now let me show the view functions side by side here. We can supply the value for this variable in the view methods. So here I'm passing the variable message with this string as a value. So now every time the browser requests this page, the function will run, it will pass this value to the template, and this string will be placed here in the HTML. Soon we'll see that these values might come from a database. Now, this allows us to have dynamic HTML pages where the content of the page is not always the same but changes with circumstances, depending on who's logged in or a form you filled in or a product you've selected, etc., etc. So we call this file welcome.html a Jinja template because it's not actually an HTML file at all. It's a template for an HTML file. Only after Jinja fills in these placeholders with the values provided from our view function do we have actual HTML to send to the browser. Now a question I get asked a lot is, can I use these Jinja variables everywhere in the HTML? And the answer is yes. You see, as far as Jinja is concerned, this is just a text file with placeholders. Everything that's not a placeholder or some other kind of Jinja structure is left alone by Jinja and sent to the browser as it is. If I want to set a placeholder somewhere in the middle of an HTML tag, let's say here in the link, that's totally fine. Let's also give a value for this new variable x in the view. And now I'm going to refresh the browser. First of all, we see the message here. this data was sent from the view from the template. So this is the output from the Jinja variable called message. We also see that the link to Pluralsight is now broken. Why is that? Well, let's look at the source for this page. Here in the tag for the link is the value 42, which was inserted there by Jinja. And this obviously isn't correct HTML, so that's why the link doesn't work. So this goes to show you that you can use Jinja variables everywhere in a text file. Jinja doesn't really care that this is supposed to be HTML, and it just fills in the placeholders you give it. Although, soon we'll see that it can do a lot more than that. Now let's remove the code that broke the link. So, the files and templates ______ dynamically generate HTML pages based on the data that our view functions send to these templates. And that also explains why we have a separate folder called static. The files in there are always the same and they never change. Those are served by Flask just as they are. We call that static content, and typical examples of that are images, JavaScript, and style sheets.

Review: Jinja Templates

We've just learned about templates, and they are the components from the model-template-view pattern that we use to display our data. The actual technology we use to write our templates is called Jinja, a Python library that can generate text files, and in our case, we use it to create the HTML files that are sent to the browser and displayed to the user. The templates that we write are also text files containing special placeholders for variables where data can be filled in, and that's how the actual HTML is generated. Actually, Jinja can do much more than that. It's a very powerful language that supports if statements, for loops, and many more things. We'll get back to that later. So, here's an example of an Jinja template for HTML. You can think of it as an incomplete HTML document containing placeholders to be filled in. The placeholders are the parts highlighted in orange. They are names of variables in double curly braces. The values for these placeholders are looked up in the template context. This is like a dictionary containing names and values. All the other text outside of the Jinja variable blocks is just copied to the HTML output as it is. So in this way, we can create dynamic HTML pages. The content of this page will vary, depending on the value of the variables in the context. Compare this to the files we put in the static folder, like the flasklogo. That's called static content because it doesn't change. Static files are served as they are, the same every single time. Simply creating the template is not enough. It's the responsibility of the view function to call the correct template for a page. To do so, we call the render_template function, which, of course, we shouldn't forget to import from Flask first. Once you do that, you can call render_template with, as its first argument, the name of the template file you want to render. Any other arguments will be passed to the template context, making them available as variables. The Jinja rendering engine will look for the file, welcome.html in a folder called /templates. Note the s at the end there. Jinja will then fill in all the placeholders in the template with the data from the context and return the HTML page, ready to be sent to the browser. And don't forget to actually return the HTML content from the view function. If you don't, you'll see an error.

Demo: A Model Layer

I've added some code for the third and final part of the MTV pattern, the model. In this course, we're not going to use an actual database. Flask doesn't come with its own database connectivity layer, but we could use an extension for that or even roll our own using something like SQLite. But instead, I want to keep it simple, so I'm going to use this JSON file we see here as our database. In case you're not familiar with JSON, it's a file format that allows us to store our data as text. As you can see, it looks a lot like regular Python data structures. Here we see a list containing dictionaries. To be more precise, our JSON file contains a list with flashcard objects, and each object is a dictionary with a question and an answer. Now I've been trying to learn a little bit of Russian, so there are some simple Russian words in here for me to try and remember. In a way, this file is equivalent to a database table where we would otherwise store this data. Now to access this data, I've written this module called model.py. It contains a function load_db, which reads the data from our file, and below we call that function and store the data in a variable db. So this db variable now holds our list with flashcards. I've also written this note at the top saying, don't do this in a real-world production setting. I say that because a simple JSON file like this will not cope with multiple users, high loads, and things like that. In a real production setting, you'll usually want an actual production-ready database server. But for now, I'm going to focus on Flask. You can consider this db object here a simulation of a database layer for the purpose of showing you how Flask works. This will keep our code much simpler, and we still get to follow the MTV pattern. Now, looking at our flashcards module, I'm saying from model import db. And at the bottom, I've added a new view, and this is the first time we see the complete MTV pattern in action. We have a view function called card_view that first gets some data from the database by saying card = db 0, which gets the first card from the list. And then it calls render_template to create a web page showing this card to the user. Of course, we pass the card object to the template and we have a template called card.html. Let's look at that. And this is a small HTML file containing a head and a body tag and a title Flash Card. And in the body, we see the text

Question, followed by this Jinja expression, double curly braces with card.question. This will access the question part of the card object passed to this template. On the next line, I do the same for the answer. Now let's look at the result of all this code. Going to the /card URL now, here we see the contents of our first card displayed as an HTML page.

Review: Model-Template-View

The model-template-view pattern makes up the core structure of a Flask web application. It consists of three types of components, the view, which defines the behavior for a page, and its responsibility is to determine how to send the appropriate content to the client. The views we've seen so far are Python functions mapped to a URL using a decorator. The second type of component is the template with the responsibility to create a user-friendly presentation, and for this we use Jinja 2 to create HTML pages. And finally, the model, which represents the data that lives in our application. Usually you would store this in a database, but since Flask doesn't have a preferred approach for that, I've decided to go for a simple JSON file instead. By the way, all of this might seem familiar. You might know this pattern already by another name, model-view-controller, as it's known pretty much everywhere outside the Python world. So how do these components work together to serve a web page? Well, as you know, a Flask application does nothing on its own. Everything has to start with a request. So let's say I go to the URL /card in my browser. What happens when I press Enter is that the browser sends an HTTP GET request for the /card URL to the Flask development server. Flask will look up this URL in its URL map and see that this URL is mapped to our view function. The view function is responsible for determining how to form a response, and that means it needs to determine whether to call the template or the model and in what order, etc., etc. So in our case, we first call the database, which returns the data for our cards, and then we send that data to the template, which returns an HTML page with that data filled in, which we then send back to the browser. And the browser shows the page to the user. Now in my opinion, understanding this flow and the responsibilities of each component is one of the main keys to becoming a proficient web developer in any framework. When you really understand the order in which each component does its tasks and how information flows between them, you're already a step ahead of the competition. And that brings us to the end of this module. We've learned about the model-template-view pattern, and because we already knew about view functions, we started by looking at templates. We saw how to create a Jinja template, how to use variables, how to call the template from a view function, and how to send data from the view to the template. Then we added a simple data model based on a small JSON file. But probably the most important thing is that we've looked at how these components interact. That's it. I hope you will join me in the next module in which we'll add logic to our view functions and templates, and we'll see the beginnings of a usable application take form.

Adding Logic to Your Application

Overview: Logic in Views and Templates

Hello, my name is Reindert-Jan Ekker, and in this module, we'll make our application a bit more usable by adding logic to our views and templates. The way our code works at the moment, we always serve the same cards. In the real world, you would want to be able to browse the cards and for each card to have its own page. So this module will be focused on how to add some logical behavior to the templates and the views. To start with our view functions, we'll make it possible for the user to select a card to view by putting a parameter in the URL. We'll need to add some error checking to show a 404 error when the user selects a card that doesn't exist, and I'll take a short side step to show you that we can very easily make our views return cards as a REST API with minimal changes in our code. If you don't know about REST, that's okay. It's not really a major thing in this course, but I do want to take a short moment to talk about it. Moving to templates, we'll add some logic to those as well. To make our cards browsable, we'll put if and for loops in our Jinja code, and we'll see how to create links from a template to a view. So that's a conceptual overview of what we'll do in this module, but I won't actually do it in this order. So let's start with a demo, and you'll see what I mean.

Demo: Parameters in URLs

Let's start writing code. First we'll take a look at the code for displaying cards, and we'll change it so that it can display any cards. To do so, we'll learn how to change the URL mapping so that it takes a parameter to select a specific card to show. This will also make it possible to select a card that doesn't exist, so we need to write error handling code to return a 404 error. Once we have a view that can display any card, I want to make it possible for the user to browse through each card one by one. We'll see how to make a link in the template from one card to another, and we'll add an if statement to check when we reach the last card. Finally, we'll add some code to the welcome page to make it show a list of all cards in the database. We'll do that by adding a for loop to the welcome template. Now before I start with the demo, my Flask is still running, and I usually don't restart it a lot during development. In general, Flask will pick up your changes and reload, so there's no need to restart Flask unless it gets really confused. Now especially on Windows, I've seen people have problems when they start multiple Flask servers. So if it seems like you changed your code but you see no effect in your site, that might be because you're looking at an older Flask process that you forgot to stop. So in general, one tip from me to you is, before you start a Flask process, make sure to check that you have no other running Flasks, and if you have any, like I have here, stop them by pressing Ctrl+C. So if you make sure to kill all your existing Flask processes before you start a new one, you will have exactly one Flask development server running at any time and that you prevent any problems. That being said, I'm going to start a new Flask now, and let's look at our Python code. So here's the card_view function that we wrote in the last module. At the moment, it always gets the exact same card from the database, the one with index 0. I want to change this function so that the user can request different cards she wants to view. To start, let's give the function a parameter index. This will be the index of the card requested by the user. Of course, that means we want to retrieve the card with that index, so let's change the 0 to the variable index. So now we have a function that can take a card index and return an HTML page for that card. But how is the user going to input that index? Well, we can put that information in the URL of the page. I'm going to change the app.route decorator here by adding a parameter called index. I do that by adding a less-than sign, followed by the text int:index and ending with a greater-than sign. This is a special piece of syntax that app.route can understand. What we're saying here is that we expect an integer, so a whole number, after the slash, and the value of that should be assigned to a parameter called index. So now the user can go to the URL /card/2, and that should return the card with index 2. Let's test this. The URL /card/0 gives me the same card as before, and if we change the index to 1 and 2, etc., we see different cards each time. But when we enter an index that doesn't exist, say 30, see get a debug view showing an IndexError. And let's see what happens here. So when the user requests a page /card/1, for example, this matches the URL rule for our view function, and the index parameter gets the value 1, which causes the card with index 1 to be retrieved from the database and shown to the user. That's all well and good. Now, when I use an index that doesn't exist in the database, this line here will throw an index error. And the correct way to handle this is to wrap our code in a try...except block, this like. And this will cause Python to try to execute these two lines, but if those cause an index error, that will break out of the block, and we get to handle that error here in the except block. So if we end up in the except block, we know that the user entered an index that's not present in the database. And because the user is asking for something that cannot be found, the correct response is to return an HTTP 404 not found error code. To do that, we call the function abort with the value 404. And to use this function, I'll have to import it at the top of my code. So now I've imported abort, and I'm using it here in case of an index error, and that means that this URL with a non-existing index now results in a 404 Not Found page. It's still an error, but a more correct one. By the way, if you were to enter something that's not an int at all, this also results in a 404 page, and the same is true if you leave out the index all together. And this is because, if we look at our code, we've specified that the URL for this view is /card/, followed by a number, so none of these other cases match this pattern, and that's why Flask returns a 404, because no URL mapping can be found for those URLs. So now we have a view with

Demo: Building Links in Templates with url_for

a URL mapping that defines a unique URL for each card. And this makes it possible for us to browse through all the cards. But of course, this means you will have to edit URLs, and nobody's going to do that in real life. So let's go to the card template, and here I'm going to add a link for each card that takes you to the next card, and that way, you'll be able to browse through the cards without having to edit URLs. So here I'm adding a paragraph tag, and in there, a button tag, and in there, a link. And this URL here, I use /card/, and that's the URL for our card detail view. And after that, I use a Jinja expression to pass the index of the next card, so the current index + 1. But there's one thing. Our cards don't actually know their own index, so we have to make this information available to the template. So let's add this as an argument to render_template in the view function. Here we have the call to render_template, we already passed a card object itself, and I'm going to add the index of the current card. So this will now also be available in the template, and that means that this link should work. But here's something else to think about. Currently, our view and template are tightly coupled, and that's a bad thing. What do I mean with that? Well, if I decide to change the URL for this view to something else, let's say card_detail, that will break the template because the template has a hardcoded link. And if you have hardcoded links in all pages on your site, you will make it very hard for yourself to maintain your templates when URLs change. But there's a better way. Flask defines a function called url_for, which looks up the URL for a view function. We use it inside the double curly braces, and the syntax looks exactly like calling a Python function. We call this URL building. The call to url_for will determine the correct URL to put in the HTML to create a working link. And the first argument is the name of the view to call, and any other arguments to the view function can also be passed to url_for. So what will happen when we render the template is that the whole block with the curly braces will be replaced by the URL for the next cards. And the double quotes are not part of the Jinja expression; those are part of the HTML code to create a link. And as far as Jinja is concerned, all of that is simply text that gets copied to the output. So, let's test this out. Here's our first card, and at the bottom we now see a Next button. Looking at the HTML code for this page, here we see exactly what happened. All normal HTML code is sent to the browser, but the Jinja expression is replaced by the URL. So this allows us to click through all the cards. But when we reach the last card and we click Next on that, this gives us a 404 Not Found error. Now in a moment, we'll add some code that prevents this error, but before we do that, maybe you can take a moment and add a link to the welcome page to our template. So like I've done before, I'll wait a couple of seconds here so you can pause the video and try to implement this. And here is my solution. What I added is a link, so an a tag, and in the href attribute, again, I use two double curly braces and I say url_for welcome because the name of the view function for the home page is welcome, so that's the name that you want to pass to url_for. Now the welcome view function also doesn't take any arguments, so you don't have to pass anything else than just the name welcome to your url_for. And then as the clickable text, I've added the text Home. Then if I check this in the browser, this is what the page for a flashcard now looks like. So I can click on Next card, or I can click on Home, and this takes me back to my welcome page.

Demo: Jinja If Statements

So we've added our Home link, but this doesn't solve the problem that if we browse past the last card, we get a 404 error. So to solve this, I'm going to replace the link to the next card with a new piece of Jinja template syntax. Here you see curly braces and percent signs, and this is the Jinja syntax for a statement like if or for. In this case, I'm making an if statement. I'm saying if index is smaller than max_index, and when that test is true, I want this link here to be included in my HTML. If it isn't, we have an optional else part here, and in that case, we're including this link here. And the main difference between these two links is that the first link has an argument, index+1, which takes us to the next card, and the second link has index=0, which takes us back to the first card. So all of this works very similarly to a normal Python if statement, but there are some differences. Jinja if statements need to have an endif statement, as you see here, marking the end of the block. By the way, did you notice how we are able to use simple calculations inside the expression for the URL here? In Jinja, usually most simple Python constructions will work, like an addition or a comparison of two values. And there's one more thing. We have to add the new max_index variable into the template from the view. So going back to the view, and I'm adding an argument max_index here, and I'm assigning it the value len(db)-1, which will be the index of my last card. So again, I don't have to reload because Flask will detect my changes. Starting at the first card, I can click Next, and again and again, and this is my last card, and here we see that the link has changed. It now says Start over. And clicking on this link sends me back to the first card. To make our site a little

Demo: Jinja For Loops

more user friendly, it would be nice if there was a way to navigate to the cards from the home page, so let's add some links to the home page. First, I'm going to remove the message from the previous module because we don't need it anymore, and I'm simply going to pass the whole list of cards to the template by saying cards=db. By the way, in a real-world setting, you wouldn't usually pass the entire database to a template. Usually this would be some selection of rows from one or more database tables. But because our database is already just a list of cards, I can simply pass the whole thing. The important thing to understand here is that in both my view functions I now determine a specific piece of data to pass to the template to show. The job of the view function is to determine what data to pass to the template, and the job of the template is to create HTML to display that data. So let's look at the template, and I'll start by removing the image and the message. Now, let me copy/paste some code in here. This isn't actually working code just yet, but please bear with me; I'll fix it in a moment. What I'm adding here is a header, Flash Cards, followed by a ul tag, an unordered list. In there, we see another new Jinja structure, a for loop. So we see the curly braces and percent sign, and it says for card in cards. So it has a similar syntax to an if statement in Jinja, and just like an if statement in Jinja, has an endif. A for loop has an endfor. So we say for card in cards, which loops over the cards variable, which holds all cards from the database, and each of those cards will be assigned to the new variable card, one by one. And every time we go through the loop, the entire block gets evaluated, and the resulting text gets included in our page. So this will generate at many li tags and links as there are cards in our database. The li tag is a list item, so that will be one item in the unordered list. And the link has two parts, the clickable text, which is the question from the card, and the actual URL it links to, which is generated by the url_for function. So we say url_for card_view, and then pass the index for the card. The only thing is, this code doesn't actually work. If I go to the home page in the browser now, we get this error, dict object has no attribute index. And this is because the JSON file we're using as a fake database doesn't actually contain the index for each card. If we were using an actual database, we would have the ID for each card, but I didn't add that to our cards. So let's do this another way. Instead of trying to retrieve the index from the card, I'll use a feature of Jinja for loops. They create a special variable called a loop that we can ask for the amount of times we've been through the loop. So here I say loop.index0, and that gives a 0-based count of the index of the current card. There's also a variable called loop.index, and that one starts counting at 1, not 0. But since our first card has index 0, I need this one. So the first time through the loop, card will hold the first card. So here in the expression with double curly braces we retrieve the question for the first card, and the index will be 0, so we generate a link to the first card as well. The second time through the loop, card will hold the second card, and index will be 1, etc., etc. So, let's try this out. Remember there's no need to restart Flask because it autodetects the changes in our code. And on the home page, we now see a list of our cards. That's nice. Clicking on a card takes me to the page for that card, so that works nicely. So this list we see here was generated by the for loop in the Jinja template, and looking at the source code for this page, this is the part of the HTML that was generated by the for loop. Each li tag with a link inside corresponds to a single card and to a single iteration through the for loop. The text is exactly the same every time except for the index in the URL and the question text inside the link. And here's the for loop again. Note that the ul tag that creates the list is outside the for loop because we only need that once. Inside it, I start my for loop, and inside the loop, I create an li and an a tag for every card. The part of the text that change are created by Jinja expressions with double curly braces, and don't forget to end the for loop with an endfor statement.

Demo: Serving a REST API

We just learned a lot about adding logic and behavior to our views and templates, but now I want to take a moment and, as a sort of side note, show you how we can write some code that's very similar to the functions we already have to create a REST service with Flask in a very simple way. So far, we've written two view functions, one for the welcome page that contains a list of cards, and one called card_view which contains a detailed view for our cards. And in this demo, I want to show you that by writing two very similar functions, we can serve the same data as a REST API in a very simple way. Now if you don't know about REST APIs, let me just put it this way: I'm going to create two more functions, one that serves a list of cards and one that serves one card. But in this case, I'm not going to serve HTML pages; I'm going to return the data in a file format called JSON. Serving JSON instead of HTML from an HTTP server is one of the things that defines a so-called REST API. So let's look at the code I wrote, it's now down here, and let's start by looking at the card detail view. The URL is similar to that of our other card detail view, but it starts with API. And that's only because that's the convention for REST APIs. Apart from that, it's simply function that takes an index as an argument, so it's very similar to our card view function. The other main difference is that this view directly returns the card object from the database itself. Flask will automatically make this into a JSON object and return a RESTful response. Let's see how this works. If I don't go to the URL for a card, but I add api in front, this is now the URL for my new REST function. And when I press Enter here, this is what my browser shows. So this is the actual JSON data that was sent back by my view instead of HTML. This would be really easy to use with a front end that you wrote in, for example, React or Angular, where you have a JavaScript front end that consumes this REST API and then uses JavaScript code to display a page for that. Now going to our other REST view, we get an error here. The view function didn't return a valid response. And the reason for that is that for security reasons which are quite complicated to explain so I'm not going to explain that here, we're not allowed to serialize a list like this directly into a JSON response. That's disallowed, so I cannot just say return db here because db is a list, and returning lists in REST APIs in Flask directly is disallowed. What I can use is a function called jsonify, and of course, I have to import that, so let's import that at the top of the file. And now using jsonify in our function, refreshing this page, this returns all my cards as a JSON list. Again, let's look at the raw data, and this is what the actual JSON looks like. So this shows one of the strong points of Flask. If you need to very quickly set up a simple site, whether it's a traditional website that serves HTML or a REST API, all you have to do is write some very straightforward functions that return the data you need, and Flask will do the rest. Showing both functions, I directly return my data, and Flask automatically turns it into a valid RESTful response.

Review: Logic in Views and Templates

We've covered quite some ground in our demos, so let's review the key points you should take away from this module. First of all, we saw that in your call to app.route, it's possible to add parameters to your URLs. An example of this is shown in the code example at the top. The part in blue says int:x between angle brackets, and this expression will match a number in the URL and assign that number to the view parameter called x. In other words, if the user browses to the page /example/5, then my view function will be called with the argument x, having a value 5. So this is a way to use parts of URLs as inputs for our view function. By the way, if a request comes in that doesn't match this, like /example/hello, this doesn't match the URL mapping, and you will receive a 404 error. By the way, you can also leave out the int part of the parameter altogether, as shown here, and in that case, any text in the URL will be assigned to x as a string. So this allows you to also take other kinds of input from the URL. There are also some more advanced things you can do with these parameters, but those are beyond the scope of this course. So we can use the URL to select the card to show, and this also means that it has become possible to enter a URL with an index that doesn't exist, so we need to add error handling. HTTP defines error code for that, and we're all familiar with the most common one, the 404 not found. To return an error code, we start with importing the abort function from Flask import abort. Then in our view function, we have to somehow detect the fact that a non-existing index has been entered. In our demo, I've used a try...except block around our view logic, and when we tried to get a card with a non-existing index, this throws an index error, which we then can catch in the except block. Then I call abort with the appropriate error code 404. By the way, it's also possible to define custom error pages in Flask, so if you want a beautifully styled 404 not found page, you can do that, but I'm not going to cover that in this course. Moving from the view to the template, we've seen that we can use the url_for function inside double curly braces to get the URL that is associated with a view. The argument to url_for is the name of the view function, and the entire expression is replaced with the URL for that view. In the example here, I'm using this expression in the href attribute for a link tag, creating a link to my welcome page. The reason you want to do this instead of simply typing the URL itself is that this allows you to change the URL mapping for that view later without also having to update all the links in all your templates. When you use url_for and you change the URL for a view, the HTML generated by the template automatically changes as well. As a second example, here's a similar link, but in this case, it links to the card_view, which is a function that takes an argument. And url_for lets you pass such an argument as well. But you have to pass it as a keyword argument, so you can say index=3, but simply passing 3 as a second argument would not actually work. In both cases, it's important to note that we have double quotes surrounding the whole Jinja expression, and these double quotes are part of the HTML tag. The Jinja expression that gets replaced by the URL starts and ends with the double curly braces. Jinja has many features, and although I cannot show all of them here, I've shown some of the more important and useful ones. In this slide, you see a for loop being used to build an unordered list, a ul tag. And inside the ul is the for statement, and for that we use curly braces and percent signs. The for statement itself looks like a Python for statement, for card in cards, and it works similar as well. A new local variable card is created, and this will take the value of each item in cards one by one. But one main difference with Python is that in Jinja a for loop has an endfor as well to mark the end of the block. Everything inside the block is repeated every time we go through the loop. So in this case, there will be an li tag, a list item, for each card, with a link in it to the page for that card, and the clickable text for the link will be the question for that card. Jinja also supports if statements, which, again, are very similar to Python if statements, but they use curly braces and percent signs, and they end with an endif. In the example, we see an else block as well, which is optional, just like with a Python if/else statement. If the condition in the if statement is true, the first part will be included in the HTML page, and if it's not true, the second part will end up in the output. And that brings us to the end of this module. What have we seen? Well, we started by looking at our card detail page, and we added a parameter in the URL which made it possible to select which card to show. We added error handling to return a 404 not found when the selected doesn't exist, and we've seen how to create links in the template using the url_for function. We also learned about the Jinja if statement. Then we moved on to the welcome page, to which we've added a list of cards, and to create that, we had to learn about the for loop in Jinja templates. And finally, I made a small sidestep and showed you how we can use very similar code to what we already had to create a REST API using Flask. And let's move on to the next module in which we'll learn how to add user interaction by adding forms to our Flask application.

Adding User Interaction

Intro: User Interaction with Web Forms

Welcome. I'm Reindert-Jan Ekker, and in this module we'll see how to add forms to our application to enable user input. I would like to enable our users to add and remove cards themselves, and we can do so by adding forms to our application. On the template side of things, we do so by implementing HTML forms, and in the corresponding view functions, we'll also have to add some code. Specifically, we will have to change our views to allow a new kind of HTTP request called a POST request. And once we do that, we can process user input to actually add or remove a card according to the user's wishes. And once that's done, we'll learn how to redirect the user to another page to show that the operation was successful. Finally, will talk briefly about forms and security.

Demo: Jinja Form Templates

Let's start by adding a page for adding new cards to our application. I will start by serving a Jinja form template, and then we will add view functionality to process the user input and save the new cards. Let's start by creating a new view function called add_card. It's mapped to the URL add_card, and all it does is render a template called add_card to the HTML. The next step is of course to add the corresponding template add_card. So here we have a new HTML document, and I'll set the title to Add Card, and in the body there's a header Add a new card and a form tag. The form tag by itself does nothing. One way to think of it is that it creates a grouping of tags. The tags inside it will together create a form. So let's put some tags in here. Here's a paragraph that says question, followed by an input tag. It says input type is text, which means that this will be displayed by the browser as a text field. We also gave it a name, question. So this will allow the user to enter a question for the new card. Let's add another input for the corresponding answer. So this is very similar. It's a paragraph with the text answer and a text input field. The main difference is that the name of this input is answer. The reason we give these inputs names is because later when we process the user input we want to know which part of the input came from which fields. We'll get back to that soon. Now we're missing one important thing. You see, the form we have now doesn't do anything yet. It will allow the user to fill in these fields but not to submit the form and send it to the server. To do that, we have to add a submit button. When the user clicks this button, this will cause the entire form to be submitted. The browser will send everything the user typed in to the server. By the way, make sure to put all form elements, so both input tags and the button, inside the form tag, or otherwise things won't work. Let's see this in action. So here's our form, let's fill it in, and let's submit this. We didn't write any code to handle the form on the server yet, but let's just look at what happens here. The form gets sent to the same URL that we were already looking at, but as you can see, the browser adds a question mark, and then it added the user input in the form of key-value pairs. We see the name of each form, question, answer, and then an equals sign and the data that the user filled in. So here you see that the names of our forms are used when sending the data to the server so that the server so that the server then can see which piece of data came from which input field. But actually this is not the usual way you send form data to the server. In most cases, we prefer to send the data back using a different kind of HTTP message called a POST. We can add this as an argument to the form tag. And now, first let's go to the page again to load the new form, and let's fill the form again. And submitting, now we get an error. The HTTP POST method is apparently not supported by our view function. Before we fix this, please note that when we do a POST from a form, the input data is not included in the URL anymore. The user input is still being sent to the server but as a part of the body of the request. And usually this is the preferred way to do this. Among other things, it prevents some privacy and security issues, and it doesn't pollute the URL with all the user input data. To actually be able to process the user input, we need to take another look at the view function. What you need to know is

Demo: Handling POST Requests in the View

that Flask view functions by default only handle GET requests, which are the kinds of requests that are sent when you type a URL in the address bar or click on a link. To make our function support POST requests as well, we have to add an option to app.route, methods, and we pass it a list with all supported HTTP methods. When handling forms, this usually will be GET and POST, like this. Inside the view method, I can now add an if statement to test for the HTTP method of the current requests, if request.method == POST. We're using an object called request here, and to use that, I have to import it, so let's do that. So this is a global object defined by Flask, but even though it's a global, Flask makes sure that inside a view method it represents the current request from the browser that we're processing. So it will contain things like the request method, the URL we requested, cookies and headers, and also the data filled in to the form. When a moment later another request comes in for another URL that gets handled by another method, there the request object will refer to that new request. Very well. So here I'm checking what the request method is. Now if the request method is POST, this means that the user has clicked submit, and we can process the user data. But before doing so, let's add an else clause. If we end up in the else clause, this means the request method has to be GET, and in that case, the user has not submitted the form. In other words, a GET request means that the user simply wants to retrieve the form so that they can see it and fill it in. So in that case, we simply render the template. So what do we do when the method is POST? Let's start by retrieving the data that the user typed in. There's an attribute of the request called request.form, which behaves kind of like a dictionary, so we can get the data for the input named question, like this. And of course, I can do the same with the answer input field, and let me put those together to create a dictionary which I will call card. And once we have this card, I can add it to our list by saying db.append (card). Would we have a real data, then saving this would be slightly more involved. But our database is just a list, so calling append works find.

Demo: Redirecting after Handling Form Submit

Now the final question is, what do we show to the user after creating the new card? We can call render_template again, but that's not actually the most elegant solution. In most cases after processing a form, you want to send the user to a different page that shows a confirmation that the form was successfully submitted and handled. To do that, I'll first have to import two new functions, redirect and url_for. So I'm going to call redirect here and return its value. So just like with render_template, you shouldn't forget to call return with redirect. Now the redirect function takes a URL as its argument, so for example, if I wanted to redirect to example.com, what would happen in this case is that Flask would send a response to the browser with a special status code telling it to go to example.com. The browser would then send a new request to example.com, and the user would end up looking at that site. But of course I want to redirect to one of my own views. To generate the URL for our own view, there is the url_for function, which we have seen before in our templates. And the page I would like to show is the card_view for our newly created card, so I'd like to ask you to pause this video and try if you can fill in this function call so that it redirects to the card_view for the card I just added to the database. I'll wait for a few seconds before showing the solution. So, did you try this? I hope so. Here's the solution to this little exercise. So we get the URL for our card_view function, and as the index we passed length of the database minus 1, which is the index of our new card because it will be the last card in the list. Did you get this as well? If you didn't, don't feel bad. The most important thing is that you understand the general idea. You will start understanding the details in due time. Now, let me show you how this works. Now first of all, I'm going to start the developer tools in my browser so we can see exactly what goes over the network when I pass this form. So here we are on the Network tab of my developer tools, and first I'm going to reload the page to make sure we have the latest version of our template loaded. Let's fill in the form, and I'm going to submit again. And what you see here is that when I click Submit the browser sends a POST message to the server, and this contains the actual user input. And the status code we received from the server was a 302, which is a redirect. So basically what the Flask server is telling the browser here is, please go to a different URL. So then the browser did a new request for /card/4, which is the new page for our new card. By the way, there's also a 404 here, and that's because the browser is always looking for a so-called favicon, a little icon you can put in the address bar, and we didn't provide that in our application, so that's why we get a 404. But that's just a very innocent error, and I'm going to ignore that. Anyway, back to our code. So what happens is the first time the user visits the form, we receive a GET request, and we return the form as HTML. Then the user fills in the form and sends a submit, which causes a POST request to come in, and we handle that in this part of the if statement here. We process the form data and save a card to the database and send a redirect. By the way, let's actually save the data. So going to model.py, here I've added the method save_db to save our new data to the JSON file. It just overwrites everything, so nothing sophisticated is going on here, and again, it's not saved in any real multiuser environment. Now in flashcards.py, I can import this method and call it in our view function to make sure the new data actually gets written to disk.

Exercise: Deleting Cards

So let's implement another form, and this time it will be for deleting flashcards. So this is basically a repetition of what we just did. We'll create the same components, a form template and a view to handle the input, and for those of you who like to work along, I'm going to make this into a kind of exercise where we go through the steps to implement the whole MTV patten together. I already created some code. Here's a template called Remove Card. You can either copy it or download it from Pluralsight or from the GitHub repository shown here. This template is very similar to the card template, but the title is Remove Card, and after showing the card itself, it shows a form that asks the user to confirm that he wants to delete the card. We'll get back to the form in a moment. Note that this template does need a card object to display. Now, let's look at the view. As the first step, we need to write a view function that only shows the remove_card template, and I want to give this as an exercise to you. There are several steps to this, so when you try this, test your solution to see if it works. I'll give you a couple seconds to pause the video and try it. So let's create a solution for this. I'm going to start by copy/pasting some code in here. Here we have an app.route for a new URL, I call it remove_card, and we have a function definition that renders the remove_card template. Now don't forget to use the return keyword here. Also, because the template displays the actual card, we have to retrieve the card from the database and pass it to the template. This means we do need to have the index as a parameter for this function. So I'm going to add an int:index to the URL and a parameter index to the view function. Looking at the template, we see that the form has the method post, so if I click on the submit button, a POST request will be sent. The No button is not actually a button but link that takes us back to the welcome page, so that will be handled by the welcome view. In the view function for remove_card, we need to add code to handle a click on the submit button. So here's my second question to you. Please add code to handle a POST request and remove the card with the given index. This means you will have to add something to the app.route line, as well as to the function body. This is a bit more complicated than our other exercises so far, so if you can't do it all, don't feel bad. Just see how far you can get. I'll wait for a moment. Now let's implement this. To start, we have to declare in app.route that we actually allow POST requests. We do this by adding an argument methods, like this. Next, I will add an if then else to the method body, testing for the request.method. This is a very common structure for Flask view functions. The next question is, what goes in the if part? The else part is straightforward. It renders the template. But what about the if? Well, here's what I do. If you have a slightly different solution, that's okay. There are multiple ways to solve this, and mine isn't the only one that works. What I do is, I remove the card with the given index from the database then save the resulting list of cards to file, and then I redirect the user to the welcome page where they will see the list of cards after removing this one. Good. There's one more thing I want to add, and that's error checking because, of course, our index might not exist. So I'm going to wrap everything in a try...except block. Very well. As a final exercise, let's go to the card template, so this is the template we used to browse through our cards, and here at the bottom of the file, I would like to add a link to the remove_card page. So, that's another question for you. Can you add a link here that will point the user to the remove card page? Again, I'm going to wait for a moment. And here's how you do this. It's a link, and in the href attribute, we use url_for to generate the URL for our remove_card view, and we pass the index of the current card. Now, let me show you that this actually works. Let's go to the last card in our lists. And here's our new link to remove this card. If I click it, we go to the Remove Card form, and when I click on the submit button here, we go back to the home page, and we see that the last card has now been deleted.

A Note About Forms and Security

So maybe you wonder what happens when we have a sneaky user that tries to input stuff that is not just text but code? How will our code handle that? Well, let's try this out. Instead of simple text, I'm going to add some HTML and JavaScript code in the form here. So what happens when I submit this? Well, this is the result. We see our flashcard, it's actually been created, but the HTML here is displayed as text, so the HTML texts that I put in the form are not actually included in the page as HTML code. Let's look at the source of this page to see what happens, and this is what we see. The lesser and greater-than signs that made up my tags have been replaced by these HTML codes. So these little code snippets here will be replaced by the browser by lesser-than and greater-than signs. And this won't be interpreted as an HTML tag. And this is actually something that Jinja does. Jinja, by default, escapes all the text that's displayed on the page as a Jinja variable. So looking at the card template, these parts here where we display the card.question and card.answer are Jinja variables, and the text that's in there will be escaped by Jinja. So this is a safety precaution that makes sure that even if you have a malicious user that slips some bad text into your database, by the way, we call this kind of attack cross-site scripting, if you're just outputting that text on the page as a Jinja variable, Jinja will make sure to escape the special characters that would have made it dangerous to put this data on your page. So Jinja here replaces the greater-than and lesser-than signs with HTML escape codes. Looking at our JSON data, one thing we see is that in the process of reading and saving our database, my beautiful unicode Russian text has been replaced by octal character codes, but that's okay, it's still the same text, it's just not as readable. But here we see our new card. And as you can see here, in the data the malicious code is still there. It's actually a valid paragraph tag here. So what this shows you is that if you have a form, you probably want to make sure to clean up your user input before you actually store it in the database. But that's not actually something Flask can help you with because Flask doesn't know about your model, so it can also not provide you with steps to do this. Now there's also a completely different set of possible tags when you use forms, and that's called CSRF, cross-site request forgery, and that comes into play when you use forms in combination with cookies and sessions. Now we don't have either of those, so right now this doesn't concern us. But let me just say this. If you want more complete form-processing functionality, including input validation and CSRF protection, use a Flask extension like Flask WTF. So even though I'm not going to cover this library, here is the home page of Flask WTF, which is a forms library, which makes it very easy to create beautiful web forms for Flask, validate your user input, and it also gives you CSRF protection.

Review

Let's review what we've learned in this module. First of all, we've seen that we can create HTML forms. I don't want to take too much time to talk about this because this course is about Flask and not about HTML, but anyway, the important parts here are the form tag that groups all the form elements together, and this tag takes an attribute method. And for most forms, you will want to set this post. This will cause the browser to send the input data to the server as an HTTP POST request. Inside the form tag, you can put one or more input tags. There are many kinds of different input fields, but in our demo, I just used two text fields. It's important to give each field a name, and we will use these names in the view function to retrieve the user input for each field. Then there's the submit button. You need to add one to your form. Clicking this is what causes the browser to send the data to the server. Let's see how to handle the form submission in our Flask view code. To handle form submission, first we have to allow our view function to actually receive POST requests, and we do so by adding an argument, methods, to the app.route decorator, and we pass that list with the allowed HTTP methods. Usually these will just be GET and POST. Inside the view method, we can test for the actual method of the current requests. If it's POST, we know that the form was submitted, and we can write code to process user input. If not, we know the form wasn't submitted, and it's a GET request, and that means that the user is visiting the page and just wants to view the form so he can then fill it in. So what we do is, we call render_template and return the HTML form. So here's the usual pattern for processing the user input. First of all, you use the request object to retrieve the input, like you see here. You call request.form, and then you use the name of the input fields you want to get the input for. So this is where you use the names of the inputs from the HTML. By the way, here's a few notes about the request object. First of all, don't forget to import it. And then, the request object represents the HTTP request that Flask has received from the browser, and that means that it contains just about every useful piece of information about that request. For example, you can use it to get info about cookies, headers, form inputs, etc., etc. You should also realize that although the request object is defined globally in Flask, within your view function, it's bound to the current request you're handling, and you cannot use it to get info about other requests from the past. Anyway, once you've retrieved all the data from the form and then processed it, let's say that you check that it's valid and correct input, and in our case we group it together into a new card, usually you would pass the new or updated data to your model. In our case, we save it to a file. And finally, you want to show the user a new page that shows them the form submission was successful. The best practice here is to send a redirect to a different page by calling the redirect methods. To these methods, you can pass a URL, and from our templates, we know that it's best not to hardcode URLs. So we use url_for to generate the URL for one of our views, and this will redirect the browser to that other page. And that's it; we've added functionality to add and remove cards, and in both cases we created a template holding an HTML form with HTML input tags, and we've written view codes to handle the user input. These views have to accept POST requests, retrieve and process the user input, and once that's done, we redirect to another page. We also took a short moment to talk about forms and security. Very well. Let's move on to the next module and add styling to our application.

Apply Styling and Jinja Inheritance

Intro: Template Inheritance and Applying CSS

Hi, I'm Reindert-Jan Ekker, and in this module, we'll learn how to add styling to our site using Jinja inheritance. This is going to be a very short module. What I want to do is to apply some styling to our site to make it look a bit nicer. Now, I'm going to admit that I'm not very good at making things pretty, but I can at least explain the technology that you would use to do that. The most important technology for that is a language called CSS, and we'll not just write some CSS styles; we'll see how to create a simple file with some styles and apply that to the whole site at once using a technique called inheritance in Jinja. One side effect of this will be that it also makes our HTML a bit nicer and cleaner.

Demo: Adding a Stylesheet and Applying It to an HTML Page

So, let's get coding. I'll start by creating a small CSS style sheet and apply that to a single HTML file. Next, we'll use Jinja inheritance to apply the styles to the whole site, and we'll restructure our HTML code to be a bit cleaner in the process. So let's create a CSS style sheet. What I'm going to do is here in static, I'm going to create a text file called style.css. For those of you not familiar with

CSS, this is how it works. We have a selector, in this case it's body, and a list of CSS rules applied to that selector. What this means is that for everything in the body tag of our HTML page, which is just about everything on the page, we apply these rules. First of all, we want to use a sans serif font, we want to make the text color cornflower blue, and we set a background color as well. So, let's see how we can actually apply this CSS to our site. Let's go to the card view for a moment, and what I can do to apply our CSS styles to this page is to add a link element to the head, like this. Adding this will cause the style sheet to be loaded and the CSS styles to be applied to the page. When we now view a card in the browser, let me just reload this page, we see indeed that the color rules have now been applied. The color hasn't been applied to the links, but that's a CSS issue, and this course is not about CSS but about Flask. By adding more rules to your CSS, you could make this page look just wonderful, but that's not what we're here for. I just want to show you that we can actually do this. By the way, let's make this code a little bit better. At the moment, I've hardcoded the link to our style sheet, but we know that that's a bad idea. It's better to generate this URL with the url_for function. So here I'm using url_for with the name of a view that we didn't write ourselves, static. The static view comes with Flask, as we've discussed at the start of this course. It actually takes an argument filename, which is the name of the file you want to link to. In this case, I'm linking to the style sheet, so I'm saying style.css. The reason to do this is that it's very well possible, when our application grows, that at some point we want to host all our static content in a different way with, for example, a dedicated static file server or on a completely different path. If we use url_for, we won't have to change our template; we just configure Flask so that url_for will generate the correct URLs for where our static content will be hosted in that case. So it's a bit more typing right now, but it does save us a headache in the future. Now if I go to another

Demo: Jinja Template Inheritance

page, we see that the CSS is not actually applied here because, if I look in the template for this page, there's no link to the style sheet here. So the obvious solution would be to copy/paste the code that loads the CSS from the card template into every other template. But there's a better way. I'm going to use template inheritance. First, let me create a new template, which I'm going to call base, and I'll copy some code in here. This is our base template, and it defines the base structure for our other templates. All the code that is repeated in every template over and over I can now just put in here. So you see there's a DOCTYPE and an html tag, a head, and a body, all these elements that come back in every page. There's also a new Jinja structure called a block, of which, I've defined two, a block called title and a block called content. You can give these blocks any name you like, so I could just have well called them Bob or Dave. I just think that title and content is a bit more descriptive. These blocks mark the parts of the HTML that can be filled in by the templates that inherit from this template, the children. Let's see how that looks. I'm going to start with the welcome template, and I'm going to remove the whole header, including the body tag, and replace it with this statement, extends base.html. And this tells Jinja that we want base.html to be the parent template of this template. Everything that used to be inside the body tag we will now surround with a block content. And I replace the closing tags at the end of the page with an endblock. So now our code says we extend from base.html, and here's the code that goes inside the content block. So looking at base.html again, the resulting HTML will start with the HTML header, start the body tag, and then this block decoration will be filled in with the content from the welcome template. After that, we close the body and the HTML tag. And here we have a second block called title. Let's fill that in as well. So this will make sure that the browser tab for this page will say Welcome. Good. Now we can repeat this for all other pages, but I've already done that, so let me show you. Here's the Add Card page, and again, you see the same structure, an extends base.html statement, a title block, and a content block. In my opinion, you can really say that the HTML gets cleaner and simpler by using inheritance. All the ugly decorations at the top of the HTML have moved to the base template, and we're only left with the parts that are specific for this page. Now I think you'll believe me when I say I applied the same thing to our other templates, and I'm going to leave that as an exercise to you. It's basically the same thing two more times. Now, if we revisit our site and refresh, we see that although the content of each page has stayed the same, we now have the styling applied everywhere. So, looking at the base template once more, because we used inheritance we can apply styling to the entire site with this one tag here.

Review

That's it. This was a short module, but we've still learned some things that are worthy of review. We've seen that you can do template inheritance with Jinja by starting your template with an extends statement. In the example, we extend from a template called base.html, so that will be the parent template. All HTML in that template will be included in this template, but we can use blocks to fill in parts of the HTML from the parent with content specific to this page. These blocks have to be defined in the parent template. So, if the parent template defines a block title, I can use block title in this page and replace that part of the parent template with my content. We've also seen how to link to a style sheet from HTML with the link tag. Now my style sheet is served as a static file, so as the href attribute, I could use a static URL like static/ style.css. But the preferred way is to use the familiar url_for function, with the first Argument, static, which refers to the Flask built-in function, static, and a second argument, filename, which is style.css. Using url_for, we can build static URLs in such a way that when my setup changes and I want to host my static files in another way than simply from the static folder, we won't have to change our templates. So, we've learned about styling in CSS and how to apply that styling to our HTML as a static file with url_for. We also saw that to apply our styles to the whole site at once, we can use Jinja inheritance. A nice side effect of this is that it also makes our HTML cleaner. Good. That's our module about styling and inheritance. Let's go on with our final module about deploying our application.

Deploying a Flask Application

Intro: Deploying a Flask Application

Hi, I'm Reindert-Jan Ekker, and in this final module, we'll see how to deploy our Flask application. Deploying our application means we're going to run it as a long-running process on the server so it can accept traffic from the internet. This is a very different scenario from running the development server, mostly because the internet is not a safe or clean environment. We don't know who will be sending requests or how many requests will come in or what will be in those requests. So I'm going to show you some best practices to handle these things. We're going to run on a different server than we're used to. There are multiple options to choose from, but for this module I've decided to use Gunicorn, a Python HTTP server that's fairly popular. So that means we're not going to use the built-in Flask server because that's simply not meant to run in production. Gunicorn is meant to be a production-ready server, but it does have some limitations, so it's best to run it behind a so-called reverse proxy, and for that, we'll use nginx. If you wonder what this means and why it's necessary, all of that will become clear during the demo. After doing all of this, we'll take a short look at some of the other available options. There are several other servers we might have chosen, as well as some hosted options. Those are websites that can run your server process for you and take away the hassle of managing your own server. When I say I'm going to deploy a Flask app on Gunicorn with Linux, there are still hundreds of ways you might do that, so let me tell you a little bit more. As my Linux flavor, I've chosen Ubuntu, which is based on Debian and uses the apt package manager. If you use a different Linux, that should work fine; just make sure to install the same packages with the corresponding package manager. Before I start the demo, I've already taken care of setting up some stuff. Python 3 is installed on Ubuntu by default, and I've made sure to install Python3-pip so that we can install our dependencies. I've also installed Git and nginx, the reverse proxy.

Demo: Deploying Flask on Linux with Gunicorn

So, let's give this a go. We're going to deploy our Flask app on Ubuntu with Gunicorn, and we'll run that behind a reverse proxy called nginx. I've set up a Ubuntu server somewhere in the cloud, and I'm going to log in there with ssh. Now I've mapped this hostname to my server's IP address, but it's not a real DNS mapping, so unfortunately, you won't be able to access this same server at all. And the first thing I want to do is to get my project on here. This is a simple question of doing a git clone. Now, you might argue over whether this is something you want to do in production. Maybe you would prefer building a package and installing that on this machine. But let's keep it simple for now and just clone the project. Good. The source code should be here now. Let's see. Seems like it's all there. Now, to be able to run our code, we need to install the dependencies for our project. I can do that with pip. Note that I'm explicitly calling pip3, not pip, because I want this to run installations for Python 3, and on Ubuntu, to do this, I call pip3. So, this installs Flask and all its dependencies. But you're probably thinking, shouldn't you install things inside a virtual environment? And of course, yes, you're right. That's the best practice. But here's the thing. I also want to install Gunicorn, the server we will use to run our application, and I prefer to install that using the Ubuntu package manager, apt. You see, when you do basic system maintenance, one of the things you do is regular updates of your system, and when you install Gunicorn with apt, you will get security updates with your usual updates. To me, that's a big thing, and that's the reason I prefer to install Gunicorn with apt, not with pip. Getting this apt-installed Gunicorn to play nice with a virtual environment is a bit finicky though, and I'm only running one Python application and one instance of Gunicorn on this server, so there's no real need to use a virtual environment anyway. So you could choose to set up a virtual environment, install everything in there, and install Gunicorn with pip inside the same environment. That would work. But then you would have to manually check for updates for Gunicorn and install them, so it's more work, and it will cause even more work in the future. Although if you are planning to run multiple projects on multiple Gunicorn instances, doing everything inside virtual environments might be what you want to do. That choice is up to you. Anyway, the Gunicorn package for Ubuntu is actually a Python 2 package, and for the Python 3 version, I have to add a 3 here. Okay. Now I can run my server by saying gunicorn3 flashcards:app. Here, the part before the colon is my module name, flashcards.py, but without the .py part. And after the colon comes the name of the Flask object inside the module, which is app. Running this, we see that it starts a server process, but it's listening on port 8000. And to actually run a real website, we need this to listen to port 80. The thing is, normal users don't have permission to do that. We might run the whole thing as a superuser to make it listen to port 80, but that's a bad idea. I don't want to run my code as a superuser. What if there's a bad bug or a security hole? No, there's a better way. We use nginx, which is actually a web server that does listen to port 80, and it can accept connections and then forward them to Gunicorn on port 8000. Nginx will also prevent some denial-of-service attacks that Gunicorn is vulnerable to. But first, let's stop Gunicorn by pressing Ctrl+C. And I'm going to run Gunicorn again with the -D switch with a capital D, which means it will run in the background as a daemon, and this also means the program will now keep running even after I log out. So right now Gunicorn 3 is running. We just don't see it. Now, I'll move to the nginx configuration. It's in /etc/nginx, and we want to go into the sites-available folder. So there's a default site here, which is just the nginx welcome page, and I don't want to see that, so let's remove it, and I want to create a new site here. For that, I'm just going to follow the deployment instructions on the Gunicorn website. Here I'm going to copy the example configuration for nginx. Back on my terminal, I start the nano editor to create a new file, which I also call default, and I paste the example code in here. Note that in your case, you probably want to replace the hostname example.org here and the name of the log file with the actual name of your site. I'm just going to leave that as it is right now. Now I use Ctrl+O to save the file and Ctrl+X to exit nano. And now I need to restart nginx, which I do by saying sudo service nginx restart. And that's it. We have an nginx which accepts HTTP connections, it gives us better performance and security, and it forwards requests to Gunicorn, which runs our app. Let's test that this works. And here we see our site happily running on Gunicorn. So we've now successfully deployed our application to the internet.

Review: Deploying Flask

So, when deploying our code with Gunicorn, there's a choice to make, use virtual environments or not. What I chose to do is to install Gunicorn with apt because that means we'll get security updates with our usual system updates. When you do that, it's easiest to not use a virtual environment. But usually that's fine if you only run a single web application on your server. You can also choose to install Gunicorn with pip. This will probably give you a newer version of Gunicorn, but you will have to manage updates yourself. The upside is that this makes it easy to run everything from within a virtual environment, which is what you might want to do if you serve multiple web apps with multiple Gunicorns on a single server. So, to run Gunicorn, you just say gunicorn3, followed by the name of your Python module without adding .py, and then you add a colon, and then the name of the app object inside your module. If you want to run Gunicorn as a daemon in the background, you can add a -D switch with a capital D, as shown here. Actually, there's a lot more to say about the different ways in which you can run Gunicorn. There's a link here to the documentation, and let's check that out for a moment. So here we are on the documentation page about deploying Gunicorn. First of all, here's a much more elaborate example of an nginx configuration. Actually, setting up nginx is a skill on its own, and I'm not going to spend any more time on that here. Just be aware that if your application gets popular and you receive more traffic, you might want to look into nginx and what it has to offer. There's also an item here called Monitoring, and it talks about different ways to run your Gunicorn application. Most importantly, there are items here about Upstart and Systemd, which will allow your application to run as a Linux system service. You probably want that. Among other things, it will allow your website to start automatically after the system reboots. Here's another page I want to show you. It's the Flask documentation page about deployment options. It starts with listing hosted options, which are ways to host your application without having to deal with system maintenance yourself. These services give you a completely installed and ready-to-go system. Usually you just upload your code, and that's it. So you might want to check these out. Under the self-hosted options, we see Gunicorn, as well as some alternatives. To give a short overview, uWSGI is very configurable and flexible but a bit more complex to set up than Gunicorn. Gevent and Twisted are more focused on performance in very specific scenarios. So for now, since this is a getting-started course, I think you'll do fine with Gunicorn. So that brings us to the end of this module and the end of the whole course. What did we see? Well, I showed you how to deploy your application on Linux with Gunicorn. Now because you don't want to run Gunicorn as a superuser and because you want some protection against DoS attacks, we like to run it behind a reverse proxy. In our case, we used nginx. After running our app this way, we saw that there are some alternative options for running your app, as well as some hosted options, which take away the hassle of maintaining your own system. And there we are. Thank you for watching. I hope you enjoyed it, and I wish you happy coding with Flask. This was Reindert-Jan Ekker for Pluralsight.

Course author

Reindert-Jan Ekker

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

Course info

Level	Beginner
Rating	★★★★★ (37)
My rating	★★★★★
Duration	2h 4m
Released	9 Nov 2019

Share course