Introduction to the Flask Microframework

by Reindert-Jan Ekker

Course Overview

Introduction

Hi my name is Reindert-Jan Ekker and welcome to this course about web development with the Flask Microframework. In this module I'll tell you a bit about what Flask it, why you may choose to use it, and what you need to know to be able to follow this course. I'll also tell you what you can expect from this course. Flask is a framework for writing web applications. It's a micro framework which, as the name suggests, means it's small, clean, and simple. Flask comes with a minimal amount of features and that mean it consists of a minimal amount of code as well. Currently the Flask code base is around 12, 000 lines of code not counting the unit tests. Now although this may sound like a very limited kind of project, Flask, as a web framework, is actually surprisingly flexible. It gives you a lot of room for making your own design decisions and generally doesn't tend to get in your way like some other feature rich frameworks do. Where they make you follow certain patterns that aren't always as convenient as you might like. Another nice thing about the Flask code base is that it's very clean and readable. Any time you wonder what's going on, it's real easy to just dig into the source code and figure out what's going on. Not having so many features also means that the features that are there are 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 all of your questions on sites like Stack Overflow in no

time. And not just Flask itself is small, especially when you're writing a smaller web application you'll find that your own codebase will be small and simple as well. 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. And of course, sometimes you'll need features that Flask doesn't provide, but fortunately extending Flask is easy and there are many good extensions around. So basically, building a more complex website with Flask feels like playing with a Lego. You choose the extensions you like and leave out the rest and you'll end up with a project that's designed the way you like it.

Flask Features

So what features does Flask provide? Well 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. Now if you're familiar with Jinja templates you'll also find that they're really similar to that so there's not a huge learning curve. 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 thus the routing that maps URLs to Python functions. Now if you're familiar with the model view controller pattern this means that our views and controllers are basically covered with these two components. We have something to render HTML, which you might call the view layer and there's functions to prepare the data into logic, which we may call the controllers. I'll explain a bit more about this pattern in the next module and we'll see that with Flask like with Jinja, we call this pattern by a different name and that is model template view. But for now I want to focus on the third kind of component which is shown here in parenthesis and that's called the model. Flask does not provide anything for this, and that's actually a nice thing. Because with Flask you can choose any kind of precisions technology you like. So that might be SQL light or a no SQL database or more traditional approaches like MySQL or Postgres. In this course we'll use a Python library called SQLAlchemy, but you could do this in any way you like. So if you prefer to save your data in a text file, no one's going to stop you. Then when your application grows larger and you need a little more structure than you just one or two files, Flask provides something called blueprints. These let you organize your code in nice reusable packages. There's also a built in development server and a debugger and Flask comes with built in unit testing support. And of course don't forget Flask is very extensible and you'll see throughout the course that there's little it cannot do.

What You Should Already Know

So now you know why you might want to use Flask, but before we start here's a short overview of things that you should already know about 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 introduction 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 the basics of web development and that means you should at least know how to write a simple HTML page. It may be helpful if you also know a bit about the CSS styling and JavaScript, although you should be able to follow everything even if you don't. We'll also touch on the HTTP protocol a bit, which is the protocol that enables a browser to ask a server for a webpage. But if don't know about HTTP, don't worry it's not hard at all and I'll explain everything you need to know. Finally, it's also important that you know the basics of databases. We will use a database to store some of our applications data so we'll need to create, view, and modify database tables. So you should know the basics of SQL and database design.

Course Overview

Finally let me give you a short overview of what to expect from this course. The course revolves around a project which is a social bookmarking site where people can store and share their bookmarks. We'll start simple and add features to the website as we learn more about Flask. The first module after this introduction is called getting started. And it shows you how to obtain and install Flask. We'll build a simple hello world page and we'll see how HTTP routing works and what the model template view pattern means. After that we'll start our real social bookmarking application. We'll start with some simple views and templates, writing our first HTML pages, and adding styling and JavaScripts. From there we'll add our first HTML forms, so the user can actually enter a bookmark and we'll see how to handle that data. For the forms we'll use the Flask WTF extension and we'll see some more Jinja 2 features. Next is persistence, which we'll tackle with the Flask SQLAlchemy extension. This means we'll be able to actually store the user inputs. Then we're going to add user profiles, login and signup and learn about the Flask login extension. We'll add some more features like editing and deleting bookmarks and add text. In this module we'll see some of the more advanced features of Jinja and Flask and then in the last module I'll show you what to do when an application grows larger and more complex. Among other things, we'll see how to reorganize our code with blueprints and how to do unit testing. So, now you know what to expect, let's not wait any longer and let's get started.

Getting Started

Module Overview

Hi, my name is Reindert-Jan Ekker and welcome to this module in which we'll get started with the first simple Flask project. In this module we'll get everything up and running for our first simple Flask project. Of course we'll need Python and then we'll need to obtain and install Flask and several other tools and libraries as well. For that we're going to use a tool call pip. With that done, we'll set up our development environment with another tool called virtualenv. Then we'll be ready to create our first simple application. And I'll give you a nice short overview of what Flask has to offer.

Installation Steps

Now the steps you have to take for installation and setup are similar on all platforms, but there are some important differences. So this is what we'll do. I'll show you the steps for each platform separately. So we'll start with installing Python and pip. So once we have those two, we can move on to installing virtualenv and virtualenvwrapper. And once we have that, we can create our projects development environment. Now as I said, these first three steps are slightly different on different platforms. So in a moment I'll start with showing you how to perform these three steps on Windows. Followed by a demo on Linux, and finally the same steps on a Mac. After showing you all of that, the final step is to install Flask and start working. And fortunately all of that is identical on all platforms. Before we start, let's talk about installing the tools we need. And of course the first one should be Python. The Flask documentation recommends the latest Python 2 version installed. Currently that's Python 2. 7. For now you can try to run Flask on Python 3, but it's not recommended and if you want to know why, here's a link where you can read up on that. But for the near future it's best to stick with Python 2. So let's go over the installation process before showing you a little demo. On Windows you can go to python. org and download and run the MSI installer. That's basically all you need to do. On a Mac you get a Python version pre-installed but it's a good idea to update to an newer version. For that I recommend using Homebrew, which is a very nice package manager for Mac OS. Now for those of you who are on Linux just about every Linux distribution comes with a version of Python, but like with Mac OS, it's probably a good idea to upgrade and you can use your package manager to do that. We're also going to install Pip, a package manager for Python. Now starting with Python 3. 4, pip is preinstalled. So you won't need to install anything, but you're probably working with Python 2,

which means we'll have to install pip ourselves. Fortunately it's simple, we just go to pipinstaller. org and download and run a script called get_pip. py. Now you should know that on Linus or Mac OS, you probably need to use the sudo command to install pip. Now before I show you how to do this in a demo, two other short remarks about installing pip. First of all, if you installed Python with Homebrew, like I suggested above, pip will be installed too and you won't need to do anything for that. Secondly, Windows users need to be aware that pip will be installed in the scripts directory of their Python installation. And you should add that directly to your path.

Demo: Installation on Windows

Let's see how to perform the first three steps of the installation on Windows. Here we are on the Phython. org download page and I'm going to download and run the MSI installer for Python 2. 7. I'm going to select the default options here and the default path and this point here is important I'm adding the location of the Python installation to the path. Okay let's let this finish and now I can immediately go onto downloading pip at pipinstaller. org. I'm going to click install pip here and then right-click on the getpip. py scripts. Saving that in my downloads folder, opening the folder, and double clicking the script will run it and install pip for me. Now before we can go to the command line and run pip we first have to change our path. So let's go to the properties of this computer choose advanced system settings, and here we click environment variables. Down here among the system variables, we find the path. I can edit it and at the front here is the path to my new Python installation. This has just been added by the Python installer. I'm going to add the scripts directory of the Python installation to the path because that's where pip and virtualenv will be. So let's copy and paste this and add scripts. Then I'll press OK and OK again and now we can start the command line. You can now use pip to install other packages and I'm starting with virtualenv. Now virtualenv by itself isn't very easy to use, so I'll also install some nice extra tools with a package called virtualenvwrapper-win. Okay, that's all the installation steps we need. I installed Python, pip, virtualenv, and virtualenvwrapper and now we're ready to start working. Let's say I want to start a simple project called hello world. When I type make virtualenv hello world it tells me it's put a new Python executable in hello world/scripts. And it has installed setup tools and pip as well. Our project now has its own dedicated Python on pip inside the virtual environment. You should also notice that the prompts now starts with hello world in parenthesis. That means that the hello world virtual environment is active, so anything we install now will be specific to this project. Finally I'm going to change into my dif directory and make a directory for my actual projects called hello world as well. So this is where I will actually put my Python code. I can bind this directory to the virtual environment with the setprojectdir command like this. Now it

tells me that dif hello world is now the project directory for my virtual environment. And that means that the next time I activate the environment I will automatically move into this directory. Now if you're not familiar with virtual environments you should check out the course titled the Python developer's toolkit, which has a module dedicated to this very important tool.

Demo: Installation on Linux

In this demo we'll go over the first three installation steps on a Linux machine. So here we are, I'm logged into a virtual machine running _____ to Linux. I'm going to start by using this systems package manager to update Python. Now in the case of ubunto we will use app get, very well. So I've already used my browser to download the get pip script from pip installer. org and it's in the downloads directory. I can run it with Python by typing Python followed by the path to the get pip scripts. Running that installs pip for me, but actually I'm getting an error here saying permission denied. And that's because I'm trying to install pip system write. So I should run the script with sudo instead. Let's change that and try again. Okay, now that pips installed let's install virtualenv. We can do that with the command pip install virtualenv. So pip is actually a package that we use to install other packages. Now that we've installed virtualenv, let's install another package called virtualenvwrapper, which is a user friendly wrapper around virtualenv. Okay, and again the command is pip install virtualenvwrapper. So with that done now we need to do a little set up for virtualenvwrapper. I'm going to add some text to a file called. profile in my home directory. Let's use nano to edit it. Now this file will typically have some default configuration in it for you distribution. You can safely leave everything in here that's already in here, and add the virtualenvwrapper configuration to the end of the file. Now this does three things. First it tells virtualenvwrapper where your virtual environments are, namely in a directory called envs in your home directory. It also sets the location of my Python projects, which in my case is under the dif directory in my home directory. Then it reads the virtualenvwrapper initialization scripts. So, let's save this file and now to load the new configuration I can start a new shell or I can tell my current shell to read the profile file like this, a dot followed by the name of the file. And now I can use the virtualenvwrapper commands. I start by creating a new project called hello world with the make project command. It says it put a new Python executable in hello world and it has installed setup tools and pip as well. Our project now has its own dedicated Python in pip inside the virtual environment. It also created a new project directory in development/helloworld. Notice that the prompt starts with hello world in parenthesis. That means the hello world virtual environment is active, so anything we install now will be specific to this project. Finally looking at the prompt we can also see that we're now in the dev/helloworld directory so we're ready to start creating our

first Flask project. Now again, if you're not familiar with virtual environments, you should check out the course titled the Python developer's toolkit, which has a module dedicated to this very important tool.

Demo: Installation on Mac OS

In the next short demo I'm going to show you how to install pip and Python on Mac OS. And we start on the Homebrew home page. Homebrew calls itself the missing package manager for OS10. If we scroll down here's a single line of Ruby code we can copy and paste onto the command line, let's do so. Now running this will install Homebrew. First it asks me for confirmation, so let's press Enter and then I have to enter my passwords and then we only have to wait for a few moments. Once Homebrew finishes installing, it tells you to run the command brew doctor. So let's do that for a moment. And what you see here is just a couple of warnings about updates that don't really apply to the Python installation, so I can safely ignore those for now. Of course it's entirely possible that brew doctor gives you different warnings and it's always important to read the instructions it gives you very well and then decide whether you have to run the commands it gives you. So with Homebrew installed we are now ready to update Python. And we can do that with the command brew install python. We let that run for a moment and when it's finished I'm going to run Python to see whether we have actually updated. And as you can see even though the Homebrew installer told us that we were installing Python 2. 7. 8, I still see Python 2. 7. 5 here. And this is because the path variable still points to the original Python installation. So I'm going to have to change that so that it finds the Homebrew installed Python first. And to do that I'm going to create a file called. profile. Let's use the nano editor for that. So I'm going to type nano. profile that opens my file, and let me copy/paste a line in here. Now this file will be executed every time you start a terminal and what it does is it puts the path in which Homebrew installs software, which is /user/local/bin, in front of the path and that will make sure that the Homebrew installed Python will always be found first. Very well, I'm going to save and exit the editor and now if course we have to make sure that the profile is actually executed. I can do that by restarting the shell or I can say. profile, like this, and that will run. profile as well. Now let's do the Python check again, I'm going to run Python, and now we see that we have an updated Python because the version now says 2. 7. 8, very well. Now as I told you before, Homebrew installs pip for us, so we don't have to do that ourselves and that means we can go straight to the next step and that's installing virtualenv. So I can say, pip install virtualenv, press Enter, and we can see it installing. Now virtualenv also has a more user friendly frontend called virtualenvwrapper and now I need to do a little bit of setup for virtualenvwrapper. Again I'm going to open. profile with the nano editor

and let's copy some lines in here. Now these are the same lines that I used on the Linux install. Again, the first line tells virtualenvwrapper where my virtual environments are, which is in my home directory/. envs. The second line tells virtualenvwrapper where I actually develop, so where my projects are, which is in my home directory/dev. And finally I have a line that reads the virtualenvwrapper initializer script and makes all its commands available to us. Again I can save and exit the editor and of course I shouldn't forget to reload the profile again with the. command, very well. Now I just told virtualenvwrapper that it should look in the dev folder for my projects but I don't have a dev folder yet, so let's make that and then my final step is to make my first project with the make project command. And the name of the project is hello world. Now the make project command tells us that it installs our own Python interpreter and our own pip specifically for this project in the virtual environment for the project. Also you can see at the start of the prompt now it says hello world in parenthesis and that means that we're inside an active environment. That means everything I install with pip right now will be specifically for this project. And any other projects won't have access to those installed libraries. And another thing the made project command did, we're now in a directory called hello world and actually this is the directory inside dev that it just made and where we can put our Python files. So right now we're ready to start working on our first Flask project.

Review Slides: Pip and Virtualenv

So we've seen how to download and install Python and pip, from there we saw how to use pip to install other packages, like virtualenv. And the command for that is simply pip install virtualenv. We also worked another tool that makes virtualenv a little easier to use and it's called virtualenvwrapper. But that one doesn't work on Windows so Windows users have to install virtualenvwrapper-win. Now if you want to know more about these tools check out another course called the Python developers toolkit in which pip virtualenv and virtualenvwrapper are explained in more detail. So the pip install command will let us install packages, now the thing is pip will install your packages in a system wide manner. And when you're working on multiple Python projects you usually want to be able to install libraries and tools for a specific project, instead of for the whole system. And that's exactly what virtualenv does. It enables us to install the libraries that our project depends on in an isolated environment so that we don't get dependency conflicts with other projects. Virtualenvwrapper simply provides some extra nice commands for virtualenv. Once virtualenv and the wrapper are installed, we can setup a so called virtual environment. Now it's a good habit to do this for every Python project you work on. Now we can use the make project commands to create our first projects. For the demo we've created

a project called hello world, on Windows this is a bit more complicated because the virtualenvwrapper variance for Windows, called virtualenvwrapper-win doesn't contain a make project command. So we use a combination of make virtualenv and set projectdir. We've also seen that after setting up a project environment we can issue the command work on hello world to activate the environment. This is reflected by the command prompts showing the name of the active environment in parenthesis. It'll also move you into the project directory automatically. Now we can then type deactivate at any time to leave the project environment if you need that.

Creating Our First Flask Project

And this brings us to our final step, installing Flask and getting to work on our first projects. So regardless of your operating system, you should by now have created a virtual environment called hello world. You may have exited the command line in the meantime and in that case we can reactive the virtual environment with the workon command like this. Now again notice how the prompt starts with the name of the active environment in parenthesis. You can also see that our working directory is now hello world. So now we can install Flask again with pip and of course this will now be installed inside the active environment and so other projects outside of the environment won't have access to this particular installed instance of Flask. So we see that apart from Flask several other libraries have been installed on which Flask depends. Most noticeably here is Jinja 2, which is the template library we'll learn a lot about in this course. Now I already put a file of my own in our project directory and it's called helloworld. py as you can see. Its contents look like this. We start by importing Flask with the first line from Flask import Flask. The second line here calls the Flask constructor which will create a global Flask application objects. The argument to the constructor is the name of the main module or package of the application. And in most simple cases you should simply use the variable __name__ here. Then I define a so called view function called index, which returns the string hello world. This function is decorated with the route decorator, which is actually an attribute of the app objects. So that's why we say app. route. And tell the application that this view function should respond to requests for the /index URL. Finally there's the main function that simply calls app. run. This will start the Flask application. So, let's see this in action. Back at the command line I say python helloworld. py and press Enter, of course, and Flask answers that it's running on 127. 0. 0. 1, which is a local machine at port 5000. Starting in browser and going to local host port 5000 I get a 404 not found error. Well actually that's what I expected, because looking at the code again I remember that I told Flask to map my index function to the /index URL. So, let's go to /index instead and now we get to see the helloworld string that my function returns. So, let me give you a short explanation of

how this works. So we simply install Flask by saying pip install flask. After that we only need to create a single file called helloworld. py with a minimal amount of code to create our first Flask application. This simple hello world application has two things we need in every Flask application. First of all there's the application object created by calling the Flask constructor with the name of our current module. We'll use the app object globally in our application. We also defined something called a view function. It looks like this and the first line is a decorator, which actually is an attribute of the app object called app. route. It tells Flask that any incoming requests for a page at /index will be handled by this view function. I'll explain how this works in a moment. Now the function itself is also called index but that's not connected to the fact that the route is index2. We could have called the function anything, it simply returns the string hello world and that is what shows up in the browser.

Flask Routing

So how does this work exactly? Well suppose I type the address for the index page into the address bar of my browser. In our case it's local host port 5000 to /index. The browser will ask the server running on local host port 5000 for the page at that address. And of course in this case, it's Flask that listens to port 5000 on my local host. So Flask will try to find a view function that is mapped to the /index address. Now the app. route decorator does just that, it maps the route given as its argument to the function it decorates. So that means that to answer the request for /index, Flask will execute this function, which returns the string hello world. And as a result the browser will display the text to the user. So how does Flask look up a function for a route? Well it's simple. Every Flask app has an attribute called URL map and we can inspect that. So if I start a Python interpreter and import our app, I can ask that app for its URL map. And this is a variable that contains the mapping rules. Now this rule here is mapping the URL index to the function called index. It maps several kinds of so called HTTP methods like get, head, and options to this function. Now as you see there's also always a default route for a function called static. This function will serve any files found in a directory called static to the server. Under the URL/static followed by the file name. As an example I created the static directory and put the Flask logo in there. Now if go to static/flasklogo. png the browser shows me this picture, which is actually the flasklogo. png file in the static folder in my project.

The Model-Template-View Pattern in Flask

Now flask gives us a lot of freedom, but most Flask applications will follow the model template view pattern. So let me shortly describe what that is. Basically these three words, model, template, and view are names for components that are a part of our application. We have models, which are software components that hold our persistent data. And most of the times these objects will represent routes of data in table in a database. Flask doesn't include any specific library for writing these components and leaves all of that to you. Two popular choices for models in a Flask application would be SQLite which comes standard with Python or SQLAlchemy, which we will be using in this course. Then there are templates. If you're familiar with the MVC or model view controller pattern in that case you would be calling them views. Actually model template view or model view controller are exactly the same pattern with two different names. So it doesn't really matter which ones you choose. For this course we're going to stick with the model template view naming. Now templates are the components we will use to actually generate the HTML output for our webpages. Flask includes a very powerful template library called Jinja 2. And that's what we will be exploring in the next module. Finally there's views. Again if you're used to the model view controller convention you would call these controllers. They are the software components that control how to generate the response to a specific web request. So they will be reacting to requests for specific URLs. And they may retrieve data from a model and use a template to generate HTML. And possibly a lot of other things as well. When we're working with Flask a view is nothing more than a Python function that generates a HTTP response for a given HTTP request. So let's see how this model template view pattern applies to the request response cycle we've seen with our hello world application. Like we've seen before we go to localhostport500/index and the browser fires a request to that address, which gets picked up by Flask, which uses the URL map to determine whether any view was mapped to that URL. And that causes us our index function to be called. So this is what we actually call the view in the model template view pattern. Now if this were not just a simple hello world application but something more complicated like let's say a new site, we would like to retrieve some data now. Like for example, the latest articles to display. So the view function will retrieve data from the model. Of course it may also store data in the database using the model. Now the view function may need to apply some logic on the data and at some point then we'll want to generate a nice HTML page that contains that data. Now instead of writing ugly Python code inside the function to generate HTML, we can call a template component that will enable us to generate beautiful HTML pages in a friendly way. Finally with the HTML generated the view function will return a response object which will be sent to the browser. We'll see more about templates in the next module and we'll learn about models two modules after that.

Resources and Summary

Now finally here are some resources for you if you want to look up more information. First of all here's a link to the course called the Python developers toolkit that goes deeper into pip and virtualenv. Then here's the link to the pip-installer. org page where you can download the get pip scripts and to read more about pip itself. And here's a link to the page for virtualenv. And of course then there's the page for virtualenvwrapper and its Windows counterpart virtualenvwrapper-win. And that brings us to the end of this module. What have we seen? Well first of all we've seen how to obtain and install pip. And once you've done that you can start installing virtualenv and virtualenvwrapper, which enables you to setup a project environment. From there we went onto create a simple example application by installing Flask and adding one very simple file. And then I gave you a very short overview of Flask by telling you about the model template view pattern and by giving you a little insight into how routing works for Flask pages. So with all of that out of the way we can go onto the next module, in which we'll explore how to create basic templates and views.

Basic Templates and Views

Introduction

Hi I'm Reindert-Jan Ekker and welcome to this module in which we'll start a new Flask project and to implement some basic templates and views. What can you expect from this module? Well we're going to be starting a new Flask project called thermos and that's going to be a simple social bookmarking site like Delicious. It's what we'll be working on for the rest of this course. To this project we'll be adding some simple pages starting with an index page. The HTML for that will be generated using Jinga 2, the template engine that comes included with Flask. We'll also use CSS to add some styling and we'll see how to serve CSS files using the static content feature of Flask. We're also going to add a second page containing a form where users will be able to add their bookmarks. For now we'll just create the page so we're not going to add any logic to process user input. In other words, in this module we're going to focus on generating HTML and we'll see how to use template inheritance to make sure both pages will be styled uniformly without having to duplicate any code. And how to create maintainable links in our pages with a Flask function called URL for. Then finally we'll add some custom error pages to the project.

Demo: Starting a Project

So we're creating a new project called thermos and for that we can use the make project command. As you can see it creates a new project directory and activates the new environment as well. Now remember that if you're using Windows you cannot use the make project command and you should use make virtualenv and set project directory for that. Now of course the first thing we're going to do now is install Flask with the pip install Flask command. Very well with that done let me show you the simple directory structure I set up for our project. Now this project will be a little more elaborate than just a single file. Let me make this clear because it might be a little confusing. What you're seeing right now is the thermos directory, which is my project directory. In there I put a readme file for people new to the project as well as another directory that's also called thermos. The second thermos directory will hold our actual projects and it will contain a Python package called thermos, because it's a package it needs to have a file called __init__. py. Which is completely empty, but it needs to be there for this to be a valid package. Now inside the package there's a static _____ which will hold static files like CSS, JavaScript, etc. Then there's the templates directory where the templates for generating HTML pages go. And finally, there's a file called thermos. py which contains our actual Flask projects. Now I don't like to write a complete HTML page from scratch and that's why I'll be going to this site called initalizr. com. Which provides me with some boiler plates for starting up the projects. And we can of course choose Bootstrap layout, but I find that that complicates the HTML and CSS a bit and to keep it simple for this course I'll chose the simple responsive boiler plate. But of course if you need Bootstrap you can choose that here as well. Now the only option I'm going to enable down here is the 404 error page and of course you can enable other options if you want to, but I like to keep this site simple for now. Now if you open the file we just downloaded, we find these files. I'm going to move the HTML into the templates folder because these files will become the bases on which we build our projects. And the rest of the files will not be used to generate anything, but they'll just be served as they are from the static directory. So let's move them into the static folder. And that's basically the basis of our projects. Let's open the Python file and see what's in there. Now as you can see my thermos. py file is right now almost exactly the same as the hello world example we saw in the previous module. But of course we'll need to make some changes. To start I'm importing two more globals rendered template and URL for. I've also added an extra route for the index view so that it gets hosted both at the root URL/ and at the URL/index. Another change we have to make is that we want a view function to generate a real HTML page and not just a string like hello world. Generating HTML inside a Python function is very ugly and messy and we really don't want to do that. So instead let's make use of the HTML file we put in

the templates directory. To do so we can use the rendered template function I imported above. Now rendered template will look inside the templates directory by default and there it will find the index. html file we just downloaded from initializr. com. Now of course we still need to start the program to see whether this works. So let's go to the command line again, of course I should move into the new thermos package first and then say Python thermos. py. And now we can see that Flask is running. So now if I go to local host port 5000 again we see an error, but this time it's an internal error and not a 404, like in the previous course. And that means that although Flask has a view function mapped to this URL, there's an error in the Python code it tries to execute. So how do we find out what's wrong here? Well we can start with putting Flask in debug mode. And we do that by passing debug is true as an argument to the run function. Then of course we have to restart our application and for that I go to the command line and press Ctrl + C. This stops the program. And then I run the Python thermos. py command again to restart. So now let's see what causes our error. When I refresh the browser now I see that it says view function doesn't return a response. So let's check the code again. And I simply forgot the return statement that actually returns the response object from the render template method. Now after fixing that and refreshing now we see the actual index page, but there's something else wrong because we don't see the styling yet. And the reason for this is because in our index. html file from initializr, the CSS files are referenced like they're in a CSS folder. But they actually aren't because I put them inside the static folder. So the file you're looking at right now is how the original index page from initializr looked and in the meantime I have fixed the references. And let me show you what the fixed HTML looks like. So we can fix the problem by changing all these references and prefix them with a relative path to the static folder. Because that's where they actually are. Now I've also added my own title, three descriptions for links that don't actually do anything yet, and a short welcome text. And by the way, don't forget to fix this year for the JavaScript at the bottom, because that's in the static folder as well. So now if we reload we can see what the application will more or less look like. You can see the title here as well as the three links and some of the article text. Now the orange color here is the default from the initializr download, but I like to customize the look and feel a bit so let's take a look at the CSS. So in the CSS that was provided by initializr I only changed some colors. Here where it says orange theme I replaced these three color values to a shade of blue instead of orange. Then going down a bit the background color for the nav tec for mobile, again I change into a shade of blue. Then when we go a little bit further down we find a comment that says author's custom styles. And that's where I put some extra CSS rules and again I'm only changing some colors from orange to blue. Then if we reload the page we now see a slightly customized version of the default initializr layout. And that concludes our first steps in setting up this application.

Review: Debug Mode and render_template

So we've seen that the Flask application needs an application object that's created using the Flask constructor. We will use this application instance globally in our code. Now to run the application we call app. run. And we can pass it an arguments debug, which turns debugging on when true. What we also just saw is how we can use templates from a view method to generate HTML. For this Flask provides the rendered template method. Its first argument is the templates file name and after that we can pass data through the template using something called the template context. We'll see how this works in the next demo. For now there's two things you should realize about templates. By default templates are expected to be in a directory called templates and that's where Flask will look up the file you give as its first arguments. Also when we call rendered template it generates an HTML page and simply returns that as a string. Now of course the view function should in its turn return that string so that it can be sent to the browser. Now if you forget that, you're going to get an internal server error.

Demo: HTML Templates With Jinja2

So in the previous module I explained that we're going to use something called templates for generating HTML. And we've already seen that we have HTML files in our project. Furthermore, we've just seen a slide that mentioned passing data from our view function to the template context. Now let's see how that works. Flask comes with a template engine called Jinja 2, this engine can generate any kind of text spaced file format, not just HTML. But in a Flask project, of course, its main use will be to generate HTML. Now our HTML files are actually not really HTML files, but they're Jinja 2 templates which generate HTML. I will now show you some basic Jinja 2 syntax for generating some HTML from data that we pass from the view function to the template. I'm going to take this section here and make it so that its contents are generated from data passed in from the view. Now the syntax I want to show you is two curly braces followed by the name of a variable in this case called title and text and then two closing curly braces. Jinja 2 will take the value of these variables and replace the whole expression with the value of the variable. Now let's check this page in the browser to see how it looks and we see that there's nothing in the variables yet. And apparently Jinja doesn't give an error in that case, it simply replaces the expressions with the curly braces by the value of the variables, which at the moment is nothing. Giving the variables a value is actually very easy, we just pass them to the rendered template function like this. Of course the name of the variables, in this case text and title, should be the same as the names you use in this template. So when I refresh the page we see this text actually showing up. Now while that's very nice, but let's take it a step further. Can I pass things other than

strings to a template? Yes I can. For example I can pass a list like this. And as you see the template will contain a _____ presentation of the list and that's normally not what you want. Fortunately, I can use an index in the expression in the template. So I can just say I want the second item in the list, which of course has index1. And refreshing again we now see the second string from the list. But I can also pass objects to the template. As an example let's add a user class. A user has a first name and a last name attribute and let's create an instance with my name and pass it to the template with the name user instead of text. Now in the template I can use the user variable, but when I look at the page I see an ugly default representation for the user objects. One way to improve this is by adding an __stre method so that the user instance can be converted to a string like Python. But let's leave that as an exercise to you. I'm going to show you something else. We can use normal attribute access in Jinja. So I can ask for the users first name like this, user. firstname and user. lastname for the last name. Now if I look in the browser I see that the two parts are outputted together, so I'm going to add a space here between the expression to make it a bit prettier. Very well. Now the last thing I want to show you is that we can call methods and objects too, by user class as initials method and I can access that like a normal method from the Jinja template. Of course, I shouldn't forget to add parenthesis and again we can check in the browser that it works. Okay, let's go over what we just saw in a couple of slides.

Review: Jinja2 Basics

So we've seen how to use the Jinja 2 template language to render some of our data to an HTML page. To start with, we saw that a variable name surrounded by double curly braces will look up that variable in the template context and render its value to the HTML page. You can add variables to the template context by passing them to the rendered template function as keyword arguments. Like you can see here in the example with a var called var and a value of hello. Now we also saw that we can look up attributes on an object with regular Python dot notation, but actually the dots in Jinja 2 does even more. It checks several things in order. So suppose I say var. x inside the double curly braces? Jinja 2 will first try to find an attribute called x of the object var. Now in case var doesn't have such an attribute, Jinja 2 tries to find an item in var, which is nice if var, for example, is a dict. In which case there can be an item inside dict that we can look up. Now if both of these fail, Jinja will output the empty string. This is, by the way, also what happens when the variable var doesn't exist in the template context at all. In that case, it also simply outputs an empty string. And finally we also saw that we can call functions objects. Again the syntax for this is identical to the normal Python syntax and actually you can also pass arguments in the normal

way. And we're going to see that later in this course, but of course I encourage you to try this out for yourself and see how that works.

Demo: Using url_for to Generate Links

We're going to be creating a social bookmarking site and of course one of the things that you'd include is a page for adding a new bookmark. I've added a new template called add. html and for now it has no styling at all. It looks like this, it's a very simple HTML page with a simple form. It contains an input field for entering the URL for the new bookmark and a submit pattern. Now with this new page we also need a new view. So I added a function called add. Right now it doesn't do anything with the user input it simply shows the page. We'll see how to add logic for storing the new bookmark in the next module. So for now the function body simply calls render template with the name of the template file. Remember that render template will look inside the templates folder by default so that's where it will look for our add. html file. So, of course, add. html should again be in the templates folder. So in the browser we type add into the address bar and now we see the form. As I said there's no styling yet and no logic, but we're going to add that soon. For now I want to focus on something different. We actually want the add URL link in the home page to point to this page. But right now if you look at the status bar you'll see that this link doesn't point anywhere at the moment. We can fix that in the index. html template by calling a special Jinja 2 function that's globally available in every template and it's called a URL for. It takes as its argument the name of a view function, in this case, of course, it's going to be add. And it returns the URL to which that function is mapped. Now notice how the Jinja expression with the double curly braces is used inside the href attribute for the HTML tag. So the quotes for the HTML attribute surround the Jinja expression. Now because the entire Jinja expression will be replaced with a URL, this will end up being a normal HTML link. So with this added we can go back to the page and now we see that the link now points to my add page. And if we click it, we end up at the form.

Demo: Template Inheritance

Now let me show you how we can apply the same styling to the add bookmark form as to the index page without duplicating any code. We'll use a technique called template inheritance. To start I created a new template called base. html, you can think of it as the parent template of all other templates. I basically copied the entire index page into the base template apart from the actual content that's specific to the index page. So right at the top in the title tag we see some

new Jinja syntax. This new syntax is an expression surrounded by curly braces and percent signs. Basically we use the double curly braces we saw before only when we want to output the value from a variable or a function to the HTML page. All other cases like for loops or if and else statements go inside this syntax, the curly braces and percent sign. Now the construction we see here is a block, blocks have a name, like in this case title. They start with the block statement and their name and they end with the end block statement. I also made the top title tag a link to the index page so that we can always return to the home page by clicking that on every page. For this I use the now familiar URL for function again. And scrolling down a bit we see some more blocks, one called content and one called sidebar. And the block called sidebar actually already has some content in it. Now why do I do this? Well let's look at the new index template. As you can see it's now much cleaner and simpler. It starts with an extends statement, again with curly braces and percent signs, telling Jinja 2 that we want this template to extend from the parent template base. html. And that means that we are inheriting everything from that parent except those parts that are marked as blocks. Because we can override these with our own content. And so you see that I define my own title block here with my own title and that will replace the title block from the parent. In the same way I override the content block with my own content. Also note that I'm not providing my own sidebar block in this template. You can probably imagine what the result will be when the index template is now rendered. Jinja will take the base template and replace the contents of the title tag with the title block from the index template. Below it will add the content from the index template, but since that template does not have its own sidebar block, we keep the default sidebar as it's defined here. So, let's look at this in the browser. The result looks the same, but this time almost everything we see comes from base. html. And the content and the title are the only things provided by the index. html templates, but now we get to the really good parts. When I click the add URL link it now has the same styling and I accomplish that by having this add. html template, let's open it. Extend from the same base template as index. html. It also has its own title block and its own content block which holds the form. Now I don't want to see the sidebar on this page so I'm going to add a sidebar block that's actually empty. This will override the default sidebar. So now when I refresh, I can see the form with no sidebar.

Review: Maintainable Links With url_for

So we can call the URL for function with the name of the view and it will return the URL for that view. Now of course, behind the scenes Flask will use the URL map variable to resolve the view name. In the next module we'll see how to pass arguments to a view with URL for. Now Flask

makes sure that URL for is available automatically in the template context. Now you might ask, isn't it simpler to just write a simple HTML link to, let's say /index? Well that may be true, but the advantage of using URL for is that it's more maintainable. Look at it like this. Suppose I have some view and I want to change the URL it's hosted at. Now if I use URL for all I have to do is change the app. route call. So a single line in my Python. And all the links in my app to that page will automatically be changed to the new URL. Whereas, if I hard code HTML links, I have to go through every HTML page in the application and change every single link that links there. Now a last thing we have to fix is that I want to use URL for for any static content. The reason why I want this is that when I deploy my application to a server I might want to serve the static contents separately without using Flask. Now if I use URL for I can configure it to generate URLs that point to the externally served contents, so that might be, for example, a completely separate file server that serves my static content. So, to fix this in base. html let's change the relative path to the static directory into a call to URL for that will generate that same path. Basically, like I just explained, this makes my application more maintainable in the long term. As you can see, we are now passing an extra argument to the URL for function. The first argument is a so called endpoint and after that comes any keyword argument who want to pass the view function. Let me show you how this works. I'm going to start a Python interpreter and import the application object from the thermos module. Now let's inspect what's in the URL map. The last rule here is the static rule. So normally what would happen is that browser asks for a file, for example a CSS file, and that will be hosted at the URL that matches this rule. So that would be mapped to the static endpoint which will serve the file. Now what URL for does is it takes the reverse of such a rule. So in our HTML I'm doing exactly the reverse from what a browser would do, basically I already know the endpoint, which is static and I want to know the URL. So URL for looks up the static endpoint in the URL map and finds that the URL attached to return looks like this. But a part of this URL is this file name variable and we need to fill that with the name of a file. And that's why, in our template, we need to pass the file name argument to URL for. So that URL for can generate and complete URL. So that's why the call looks like this. And of course, we shouldn't forget to fix this for the JavaScript at the bottom of the file as well.

Review: Template Inheritance

So we've just learned about template inheritance. The basic principle is simple, we define a parent template that contains one or more blocks that can be overridden by its children. Now the children define their parent template by using the extends expression. Note how we use a curly brace and the percent sign here instead of double curly braces. The double curly braces are for

rendering values from variables and functions. And all other expressions go inside this construct. Extends is followed by the name of the template in quotes. It's important to know that the extends expression should be the very first expression in the template. Now the block that's you override in the children are defined with the block expression. You start with block followed by the name of the block, so in this example it's a block called content. After that comes any default content. Then you end the block with the end block expression. Now overriding this in a child template uses exactly the same syntax. So to override the content block from the parent we say block content between curly braces and percent signs, follow that up with any content we want to see and end it with an end block. The end user will then see the content from the child block instead of that from the parents.

Demo: Custom Error Pages

The last thing I want to show you in this module is how to use the 404 error page that we downloaded from initializr. It's quite simple. We define a function, just like a view function, but this time we decorate it with app. error handler instead of app. route. I defined two such functions, page not found, which will handle a 404 error and server error, which will handle a 500 server error. Both functions call render template with the respective templates for their errors. Note that we don't just return the rendered template, but we actually return a _____ that includes the status code for the page. This is important because otherwise Flask will not set the HTTP status code correctly. So let's start by testing this with a non-existing URL like hello. And that works. By the way this is simply the default for a for error page from initializr and I didn't change anything about that. Now triggering a 500 error is a little harder. We can try by causing an error in a view function, for example, by mistyping the name of the rendered template function in the index view. Going to the index view then we don't actually get a 500 error because debug mode is on and it catches the error for us. So just for now I'm going to turn off debugging for a moment so that we can test the 500 page. And of course I have to stop and start the application now remember that you can press Ctrl + C to stop Flask and then if we refresh, we see the 500 error page. By the way this error page is simply made by copying the 404 page and changing the body text, like you can see here. So, that tells you how to add custom error pages to your application and that brings us to the end of this module.

Resources and Summary

In case you want some more in-depth information here are some resources for you. This is a link to the documentation for the Jinja 2 template engine. I really encourage you to go there and read about it because Jinja 2 is very powerful and this course only just scratches the surface of all it can do. Here's a link to the initializr site I used to start up the project. It's a great way to get up and running with any website project. Then there's a link to the Flask documentation about URL building. And finally, for those of you who prefer to use Bootstrap in their projects, here's a link to the Flask Bootstrap extension, which integrates Flask with Bootstrap. And that concludes this module. We've learned how to run Flask and turn on debug mode. We've written our first templates and got introduced to Jinja2. We've learned about rendered templates and we've seen some Jinja syntax, like the double curly braces for rendering values. And the block and extends expressions for template inheritance. We've seen how to include static content in a page, like CSS and JavaScript. And finally I explained how to generate maintainable links with the URL for function. Now let's go onto the next module where we'll see how to handle the logic for an HTML form.

Forms and View Logic

Introduction

Welcome, my name is Reindert-Jan Ekker and in this module we'll add some logic to our views and forms. So in this module we're going to expand on forms. We'll see how to make our life a lot easier with the Flask-WTF extension. It will help us greatly with generating the HTML for forms and validating and processing the user inputs. We'll also see how to handle forms from a view function. We'll see how to get the input when it's sent through the HTTP post method and how to redirect to another page after a form has been submitted successfully. There's also a Flask feature called message flashing, which we'll use to notify users about the results of their actions.

Demo: Views and Forms

So this is the add URL page as we created it in the previous module and this is the HTML form that generates it. So let's see if we can write some code in a view function that actually stores what we enter into this form. Before we look at the view function though, please note that the method attribute on the form tag here is set to post, which means that when a user submits the form the data will be sent to the server in an HTTP post request instead of a regular HTTP get.

Also note that the main attribute of the input tag is URL, which means that the text from this input will be sent in the request as a request attribute called URL. We'll see why we need to know these things in a moment. So here we are in thermos. py and I already made some changes for handling the data that a user enters into the form. I'm now importing a global Flask object called request, which inside a view function will always be bound to the current HTTP requests. So now in our view function, I can say if request. method = post and as we saw a moment ago we get a post request when the user has submitted the form. So we can start processing the input data. Now if the request is not a post, we don't do any of that and simply return the form as an HTML page, just like we did before. But when the user has submitted the form we get a post request, so we can process the data. Now remember that we gave the input the name URL. We can now ask for the text the user typed into that input by asking the request for any form data with the name URL. By the way, I'm not doing any validation here. So if the user didn't type in a valid URL or maybe even nothing at all, we simply accept that for now. The next thing I do is store the bookmark. Because we don't have a database collection yet, I've created a global list to which we append the new bookmark to simulate storing the data in a database. Of course you should never do this in a real application because with multiple connections using a global to store data in, is asking for trouble. But I'll teach you about persistence in a database in the next module, for right now let's just go with the global list instead. Then after storing the bookmark I log a message and for that I use the logger instance for our Flask app. I'm logging at the debug level so that's why I added a line at the start of the file to set the log level to debug. So this way we will see a confirmation of every stored bookmark on the command line. Now let's go and add a bookmark. I can enter a URL and submit, but unfortunately this gives me an error, method not allowed. This is because of something I forgot to do in a view function. You see, when we map a view function to a URL with the app. route decorator, normally it will respond to a normal get request. Which is what the browser sends when you request a page, but when we submit the form with the post method, it will not respond. So we have to change the call to the decorator to make sure it responds to a post as well. Now we do this by adding a parameter called methods with a list of the methods we want mapped. In this case, get and post. So now let's try to add a bookmark again and now we're not giving the user any feedback yet about what happened, but we don't see an error, so it probably went well. Let's check the command line and here in the log we see that our bookmark is stored. So our view function works and we can process input data from the form. Now normally when a user submits a form and there's no errors, you want to send them to another page instead of just showing the form again. We can do this by having the view return an HTTP response code that signals to the browser that it should load another page. And we call that a redirect. Flask has a function for this conveniently called redirect and because we don't

want to hardcode URLs, we'll use URL for as well. So now instead of calling rendered template, we'll call redirect and give it as an argument the URL of the index page. This will cause the browser to load the index page when the form has been submitted correctly. So now when I type in some data in the form and submit, we end up on the index page. Now before I go onto really make this form user friendly, let's go over the things we just learned.

Review: Views, Forms, Post-Redirect-Get

Let's start with going over the logical flow of form handling. In web development there's a pattern called post redirect get and it works as follows. When the user clicks the link to view our form for the first time, the browser will send something called a get request. And as response it receives the HTML for the page and displays it to the user. The user then fills in the form and clicks submit. Now in most cases you'll want this to use a post request like in the demo. On the server side you can then check whether the form was filled in correctly. For example, are all required fields filled in, is the URL valid, things like that. In case everything works and the form gets handled correctly, you return a redirect to another page. This might, for example, be a success page where you tell the user that everything went well. The browser will then send a NuGet request for the other page you redirect to. On the other hand, if there are errors normally you'll want to render the form again and show some messages that tell the user which fields are not filled in correctly. So in this last case where we rendered the form again with rendered template, the browser doesn't have to do a new request, because the HTML it receives is simply in the response to the post request that contains the data that the user just typed in. Now in the demo I only showed you the case where everything went well and we didn't do any validation yet. We'll be adding that to the project in a moment, but we'll be using an extension called WT forms to help us with that. Now although we don't have a complete post redirect get cycle yet, we do already have a kind of simple blueprint of what a view function that handles a form might look like. Let's go over the new things in our code. First of all, if you want to be able to handle a form that uses the post method, make sure to declare that in the app. route decorator, because otherwise Flask will reject the request and the form data will not be processed. Inside the view method you can use the global request object to inspect the HTTP request. Here I check whether the request method is post and when it is, I know the format has been submitted by the user. If not, the user is simply viewing the form and they will not have filled in anything yet. There's also an attribute called form which is a dict that you can ask for the values of the various inputs in the form. In this case I'm retrieving the data from the input called URL. Then after storing the data, I redirect the user to the index page. In a moment we'll make sure that a confirmation message will be shown on that

page to tell the user that everything went well, but in any case we're causing the browser to leave the form page and load another page. Note how I'm using URL for here, just like we've seen in templates. So if I ever want to serve the index page on another URL, we can do that without having to change this call. Now in case we didn't get a post request but a get request instead, we know that the user is just viewing the form, so in that case we simply rendered the HTML page and returned the empty form for them to fill in. Now a short word on the request object. It's kind of a magic Flask feature, it's globally available, but it doesn't globally represent the same thing. Flask makes sure that when an HTTP request is active, the request object is bound to that current request. So if your application is handling a thousand concurrent requests, in multiple threads, this variable will be bound to a separate instance that represents that specific request. Now that also means that normally you shouldn't use a request objects when there is no active HTTP request. Basically that means you can only use it in a valid way inside a view function. Now there are many useful attributes on the request objects and here's a table showing some of the more important ones. We've just seen how the form attribute gives you access to the input data from an HTML form. There's also the args attribute that holds data that is sent as part of a URL. Where you put key value pairs after a question mark in the URL. Now this is mostly useful when you're submitting a form with get instead of post. Other things that you can access through the request are any cookies that are set in the browser the HTTP headers from the request, any files that have been uploaded through the form, and last but not least, the HTTP method for the current request. And we've just seen a use case for that. Now you should know that this list is not complete and there are many more things a request object provides. But it would go too far to explain all of them right here.

The Session and Message Flashing

So before we go into form validation and error handling, I want to take a moment to focus on something else. Besides the requests, there's another global object in Flask and it's called the session object. You can use it to remember data between requests. It works by setting a cookie in the browser and that means it's coupled to a users HTTP session in that browser. Now just like the request object, it's a Flask global object that depends on the context in which a request is active. So normally you should only use it inside a view function. So to use the session, Flask needs to set a cookie and because of security considerations, Flask sets a secure cookie using a secret key for cryptography. And that means we will need to configure a secret key for our application so we can use a session. Now in case you want to use the session directly and store some data in it, you can use it like you would a normal Python dict to store key value pairs. But in

this course we won't need that because we'll be using it indirectly to show messages to the user with the flash function. This is what we're going to use to tell the user that's storing a bookmark has succeeded. The flash function will store a message in the session so that we can show it to the user with a future request. Now when that next request comes in and we may want to show that message to the user, we can use the get flask messages function in a template to retrieve and display the message. Now because we want to use the flash function, of course, we start by importing the flash function from Flask. Now I can replace the local by calling that function. I simply pass it the string I want to show. Of course we can now also remove the lines that import and configure the logger. Now when we try to add a bookmark again we get an error, because no secret key was set. As was to be expected, because we know we need a secret key to use the session. So where do we get a secret key? Well one way is by generating it with Python. We simply start Python in import os and then call os to urandom. In this example I'll create a string of 24 random bytes, this is going to be my secret key and I can copy/paste it straight into my code. So in thermos. py I'll add a line that uses the config attribute of the app object. This config object is a dict that holds configuration values and as a key we'll use secret key and as a value we use the string we just generated. And that's all Flask needs to be able to generate a cryptographically secure session cookie. So now when I fill in the form and submit, the view function stores this flash message in the session and then I get redirected to the index page, which will cause a new request from the browser. And then I'll see the flash message at the top here. So how do I retrieve the flash message from the session to show it here? Well let's examine the template. I've put the code to show flash messages in the base template so that any flash messages will always be shown regardless of which page you are looking at. First of all, I'm calling a Flask function get flashed messages that retrieves a list of messages from the session. I assign the results to a variable called message with the width expression you see here. It defines a new template variable called messages for the duration of the width block only. Now inside the width block, first I check if there's any message at all. If not, I'm not rendering any output to the template, but in case there are, we output a ul tag and then we use a Jinja for loop, which works almost exactly like a normal Python for loop. So we say, for message in messages and every message in the list gets assigned to the message variable. And for every message, we output an li tag with a Jinja expression that renders the value of the message variable. Most of this reads more or less exactly like Python code, there are two things you have to be careful of. First of all, make sure you always end for, if, and with statements with an end tag, like endfor and/or endwith. If you forget that, Jinja will give an error. Also make sure to surround the if, for, and with expressions with curly braces and percent signs. Whereas the expression that simply outputs a variable value uses two curly braces. So now you know how to render a list of messages to HTML. Let me show you the

CSS I added for this to make it a bit prettier. I'm just adding some colors and margins here and removing the list bullets so nothing really fancy going on here.

If and For

So now you've seen how to use the flash function which is a convenient way to show messages to a user. But we also saw some new Jinja syntax. We saw that Jinja has an if expression which works pretty much like a standard Python if statement. We didn't use an else clause in the demo but if you want you can and it works just like you'd expect. But don't forget to end the block with an endif, because otherwise Jinja will give an error. Now a nice thing is that Jinja also supports elif, just like the Python if statement. Another Jinja control structure is for which also works just like the Python equivalent, but you have to end it with an endfor. Now sometimes inside the loop you need to know the number of the current iteration, or other things about the loop. Jinja defines a loop variable inside the loop for that. In the resources I will provide a link to the documentation about this so you can read more on how to use this if you need it. Finally, we saw the width statement. This designs a new variable for the duration of the block and sets it to the value of the expression. Again, don't forget to end this with an endwith expression. So let's follow this up with another application of the if and for statements. At the moment we're showing the most recently added bookmark only in a flash message and that only occurs once directly after adding it, but of course I'd rather see a number of the most recently added bookmarks in a list. So I've added some code to do exactly that. I added a function called new bookmarks that returns the last bookmarks sorted by date. Then in the index view I call this function to retrieve the five newest bookmarks and pass that to the template context as a variable called new bookmarks. Moving to the template, I added some code there that's very similar to the code we saw for showing flash messages, although in this case I'm not using a width statement. First I check if there's anything in new bookmarks at all, and if not I show a message, no bookmarks yet. When there are bookmarks, I generate a ul tag and a closing ul tag. And inside that I use a for loop to iterate over all items in new bookmarks. For each of those I generate a list item, which contains a link to the bookmarks URL. The text for the link at the moment is simply the URL itself. So let's restart the application and test our new code. I'll add two links, one to Pluralsight and one to example. com and now these links show up in a nice list on the index page.

Demo: Using WTForms

So we have a basic form that stores bookmark and shows a message to the user. One of the things we're missing though, at the moment, is validation. And that means we want to check the user input and handle any errors. Now of course we can retrieve the data for every input in the form and validate that with standard Python string functions, but there's a nicer and more efficient way and that's to use a Flask extension called Flask WTF. It provides integration with a very nice library called WTForms. Now this library allows us to define our HTML forms as Python classes and from there it will render the forms HTML and validate the input. Now generally speaking it will make our view functions cleaner and simpler and our forms more maintainable. So let's see how to use Flask WTF. Before we can use the Flask WTF extension, we should, of course first install it. And like always, we do this with pip so we type pip install flask-wtf. Now let me show you a first form class that represents the HTML form for adding a bookmark. I call it bookmark form and it has to inherit from the Flask WTF form class, which of course I have to import as well. Notice that I created a new file called forms. py that will contain my form classes. This is not necessary, you can just as easily put all your Python code in thermos. py but to keep things clean and maintainable, I like to break up the code into smaller files. And so I decided to create forms. py. Now for every input in the form I add a class attribute one called URL and one called description. To these attributes I add instances of WTForms field classes. These are Python classes that represent HTML input fields. So for example, there's a string field which represents a simple HTML text input field and a URL field, which represents an HTML5 URL input. Which is basically a text input field for which the browser can check whether the user has entered a valid URL. Now WTForms works as follows. When we create a new bookmark for an object by calling the constructor for this class, it will generate attributes on the new instance with the same names, URL and description. These instance attributes will represent the actual fields for that instance of the class. So you should remember that this code here only defines a couple of class attributes and that WTForms will use these attributes to generate field instances on any bookmark form objects. Then we can use those instance fields to generate the HTML for each field in a template, but also to validate the input after the form is submitted and to retrieve the data from the form. We'll start with that in a moment, but first, of course, we shouldn't forget to import the string field and URL field classes in order to be able to use them. I'm also passing some arguments to each field and the first argument is the description label, which can be used in the HTML to show a label. Right now I'm not really using that functionality so there's only two short placeholder strings there. And you can also pass lots of configuration options and we'll see more about those later. So let's see how to use this class from a view function. Switching to thermos supply of course I need to import my new bookmark from class first. And scrolling down this is what our view function now looks like. Now the first difference is that we start by creating an instance of

our form class. The Flask WTF extension contains code that checks whether the current request contains any user input from the form and adds that to the fields. So if this is a get request where nothing is filled in yet, after this line the form variable represents an empty form. But if the user has just submitted the input, the form variable will contain the data the user typed into the form. Now on the next line I replaced the check for the request method by another test, which is form. validate on submits. And this does two things. It checks the HTTP method and it validates the form. And that means that if either the method is get or the form contains errors, we skip the code inside the if block and go straight to rendering the form. So if the user submits a form with errors, we don't do anything but re-rendering the form, which as we see in a moment, will also show the errors to the user. Now if the form has been submitted and validates correctly, we can go on to store the data. And here we see that the form instance has two attributes, URL and description. Like I explained earlier, these are instance attributes that have been alter generated based on the class attributes of the bookmark form class. These attributes represent the form fields and they both have a data attribute that contains the data that the user typed in. From here it's basically the same, I can store the bookmark and show a flash message, etc., etc. Then on the last line of the function we shouldn't forget to pass the form to the template so that we can use it to generate HTML. Now let's look at the template to see how that works. Inside the form tag we simply use a Jinja variable expression to render the HTML for each field. So we render form. url and form. description by simply putting them inside double curly braces. This will result in two HTML input tags, one with type text and the other with type URL. Now any arguments we passed to these fields will become HTML attributes. So in this case both HTML input tags will get a size is 50 attribute. Now there's also something else here and that is the form. hidden tag function I'm calling at the start of the template. This will render a hidden field with a special token that we need to protect our application against a form of a tag called CSRF or cross side request forgery. I don't have the time to explain that here, because it's quite a complex topic, but let me just say that you should always include this hidden tag call in your forms when you use Flask WTF. If you don't, you will not be able to submit any data because Flask WTF will reject it. Now with all that done let's go to the add URL page again and we now see the fields are clearly wider, so the size attribute from the template worked. We also see the new description field I added, and let's add a URL and a description and of course we shouldn't forget to add HTTP:// in front because otherwise it wouldn't be a valid URL and we don't have any error handling yet. Now as you see the URL has now been stored and going over the code we added quickly again here's the bookmark form class that represents the form with a field instance for every input in the form. Now these field instances are assigned to class attributes on the bookmark form class. Now in the view we create an instance of the form and ask it whether it has been submitted with valid data. If

so, we retrieve the data and store it. If not, we render the form, which can mean either an empty form because nothing was submitted or a field in form with errors. Now in the template we use the double curly braces syntax to render the form fields and also you should never forget the call to hidden tag.

Demo: Showing Form Errors

So let's see how we can handle errors. Let's start by taking another look at the bookmark form class and in particular the URL fields. One of the options we can pass to the field is to specify validators so that WTForms can check the input data for errors. I pass these validators as a keyword argument with a list of validator objects. In the case of the URL I pass it data require which means, of course, that this field has to be filled. And the URL validator, which will check whether the data is a valid URL. Now in the view function when we call validate on submit on the form, this will trigger these validators so that the data that the user has typed into the form gets checked against them. So if the user doesn't fill in the fields, the data required validator will generate an error message. But if he does fill it in, but it's an invalid URL, the URL validator will generate an error. Now in both cases validate on submit will return false, so suppose the user forgets to fill in the URL then validate on submit will return false and we render the template again. So now the only thing we need is to actually show the error on the page. And looking at the template, we see that this is quite simple. I simply check if there's anything in form. url. errors which will be the case if the URL field has any errors, of course. And if there are, I add a CSS class of error to the URL field so that we can show the user which field caused the error. Now below the field I'm also adding a list of errors in a for loop looping over that same form. url. errors attribute. I've also added two blocks of CSS code for styling the errors and two other blocks for making the form layout a little bit prettier. So now we can go and test. So for example, if I type in a URL without HTTP in front, this triggers the URL validator into returning and invalid URL error. With the new changes in our template this is shown with a nice red color and the field that caused the error has a red outline as well. So now let's fix the URL to show that everything still works. And it does.

Review: Using WTForms

So let's do a quick review of what we saw in the demo. First of all we saw how to create Python classes that can render and validate HTML forms. These are normal Python classes which should inherit from the flask_wtf. form class. WTForms provides field classes for many different kinds of

possible HTML inputs. In the demo we only saw a URL field and a string field but we'll see some other ones as well in this course. Of course I can't cover them all, but I'll give you a link to the documentation where you can read all about them. By the way, some HTML5 inputs are not included in the WTForms by default. The URL field is one of those, but fortunately Flask WTF includes an extension which does have a URL field, so that's why I could use it anyway. So when you start reading the WTForms documentation but you cannot find anything about the URL field, that's correct because it's not one of the standard WTForms field classes. Now we also saw that to use the power of WTForms for validation you can pass a list of validators to the field. These will be applied to the input data when you call validate on submit in the view. In the view we started by creating a form instance simply by calling the bookmark form constructor. This will create a new object representing the instance of the HTML form we are generating for the current request. Now if the form was just submitted, this form object will contain the data from the request. And if not, the form will be empty. Then we call form. validate on submit which will only return true if the form was submitted by the user and the data does not contain any errors. So only if this returns true do we want to start processing the data and eventually redirect to another page. If it returns false, we simply want to render the form again, possibly with errors. Now most of the time we'll need access to the data in the view function and for that each field has a data attribute. So for example, to retrieve the URL the user types in we use form. url. data. Now you should realize that using this is normally only meaningful after the form has been validated successfully. And of course, after all of this you shouldn't forget to pass the form to the rendered template function call to make it available in the template. So how do you use the bookmark form class in the template? Well first of all there's something you should never forget and that's to add the form. hidden tag call for protection against cross side request forgery attacks. If you forget this, WTForms will not let you submit the form. Now to generate a cryptographically secure CRFS token, just like with the session cookie, you have to make sure to configure the Flask secret key. Now to render the HTML for the inputs we can simply use double curly braces. Doing this with the URL field for our form, for example, will replace a Jinja expression with an HTML input tag with type URL. But as you may remember, in the demo we used parenthesis to call the URL as a function. This does basically the same thing, but any keyword arguments passed to that function will become HTML attributes on the input tag. So the example here would add a class is fancy attribute to the HTML. Finally we saw that we can access any errors on the form by asking for the errors attribute of a field, which gives a list of all the errors on that field.

Demo: Custom Validation

Now for the URL we are using a relatively new HTML feature which is the input of type URL. And at the moment some browsers support this and others don't. For example, when I open the page in Firefox and I type in a URL without the HTTP part, it tells me this is not a valid URL. And in my case this message is in Dutch because my Firefox is using Dutch as the default language. Now this check is done by the browser and the form isn't submitted in this case, so the data doesn't even reach my Flask code. So in this case the browser prevents the user from entering a URL without the HTTP part in front of it. It will allow me to enter a URL without a top level domain though, like this. And Safari, the browser I use for most of my demos, doesn't support the new URL input at all. So Safari will let me enter anything and that's one of the reasons why I always have to check for valid data on the server side, so in my Flask code. Now of course I want things to be a bit more convenient for our users and that's why I'm going to customize the validation a bit. Let's take another look at forms. py and see my changes. Here I have overridden the validated method of the form class. This method will be called when we call validate on submit and we can change its behavior a bit if we need. So I'm going to take away the need to specify a protocol by checking whether the URL starts with HTTP or HTTPS. And if not we'll add that here automatically. So now even if the user types in something that doesn't start with HTTP it can still validate but it will have to end in a top level domain because that's one of the things that the URL validator still checks for. After we do this check we call the original validate method on the parent class. This will check all other validators against the data and if it detects an error we return false like it does. Then finally I check whether the user has filled in a description. If not, I use the URL the user typed in as the description. This makes sure the description field is never empty. Then at the end I return true and this is very important because the validate function has to return a Boolean value and in this case we want a return value to mean there is nothing wrong. Now let me show you that this works. I'm going to load the page in Safari and enter a URL without HTTP. We pass validation, so far so good, and when I hover over the link you can see that HTTP was added in front automatically. And of course I can click the link and this will bring us to the Pluralsight home page.

Demo: A Jinja Macro for Rendering Forms

Now as a final step I want to show you an advanced feature of the Jinja templating engine. I'm defining a macro here called render field and such a macro behaves more or less like a regular Python function. In this case it takes as an argument a form field and then renders it. And that means we won't have to write all the HTML and Jinja code to render each field and its errors every time for each field in every form, but instead we only have to call render field for each field

and that will take care of everything. Now since my forms will be inside table structures with a table row for every field and another row for the errors for field, that's structure this macro will generate. It has the same logic my template used to have checking for errors on the field and looping over the errors will show them as well rendering the fields. Now in the template for the add bookmark form, I hard coded the text for the label for each field but let's make things more maintainable and use the label attribute of the fields. So if this macro will be called with the URL field of our bookmark form, this label here would probably say something like, please enter a URL. This will be taken from the form definition and we'll see that in a moment. Now in the next tag we render the fields, so this will result in the HTML input tag. Now there's two new things here. Remember that you can add arguments to a call to a field and that those arguments will become HTML attributes on the input tag. Well here we pass any remaining keyword arguments to this field. So I can pass, for example, the size is 50 argument to the macro render fields and that will again become a size is 50 attribute on the HTML tag, because that's where I'm applying these keyword arguments. The second new thing is that we use a Jinja filter called safe here. This pipe symbol means we take the output from the field, which is a string with an HTML input tag and in this case use the safe filter, which means that Jinja will not try to escape any special characters in the output. We'll see more of Jinja filters later in this course. Now looking at the template for our add bookmark form, I have replaced all HTML for rendering the form with two calls to our render field macro. Now in both cases I passed the field as the first argument followed by any keyword arguments that will become HTML attributes on the input tag. Now to use the macro I have to import it first and for that I use the code on the second line of the file where we import render field from my new form macros file. Looking at the forms. py file I now pass a meaningful text as the first argument. As you may recall, this will become the label for each field in the HTML form. So now if we load the page in a browser this is the form as rendered with our new macro. You can see the new labels but a part from that all HTML is basically the same as it was before, but of course the benefit of using a macro, in this case, is not in generating different HTML, but I'm using a macro because it will make writing templates for forms much easier from now on.

Resources and Summary

So we've gone over a lot of information in this module and let me point you to some resources where you can find out even more if you want to. Here's a link into a page of the Jinja documentation that gives you a list of all control structures. And in there you will find more detail about among other things, with, for, and if, the structures we learned about in this module. Then here's documentation page for Flask WTF, the extension we're using to integrate WTForms with

Flask. And this is the page for WTForms itself. Finally here's some more information about post redirect get, the pattern I've been explaining, that basically defines the flow of logic when you handle a form. If you want to know more about it here's a very nice article about it on Wikipedia. So like I said, we saw a lot of new stuff and let me give you a very short summary of all the things that you saw. First of all, we learned how to use forms with Flask WTF and to do that there's several aspects we need to consider. One side of that is how to use Flask WTF in a view. And for that you need some understanding of the post/redirect/get pattern. We also saw how to use message flashing, so to show a user a message after handling the form. And for that we also need to configure a secret key. On the other hand, we also need to change our templates and we saw three new structures if, for, and with. We saw how to render form fields and errors. And we saw how to use CSRF protection with a hidden tag function and finally we even saw how to use macros in templates to make rendering forms a little less tedious. And then at the very end of the module we even saw a little bit about custom validation with a Flask WTF form class. And that brings us to the end of this module, let's go onto the next module where we'll learn how to persist our data in a database.

Persistence

Introduction

Hi, I'm Reindert-Jan Ekker, and in this module we'll see how to persist the data for our Flask application. What can we expect from this module? Well the main topic is persistence, and that means we will be storing bookmarks and user profiles in a database. In the model template view pattern, the classes we used to represent this data are called models. Now Flask doesn't provide any standard component for writing models, but I'm going to use a library called SQLAlchemy, which is probably the most popular tool for mapping classes to database tables. We call such a tool an object relational mapper or ORM. We'll only have to write Python classes and methods, and SQLAlchemy will generate the corresponding SQL queries. In this module we'll only see some very simple relations and queries, and I'll show you some more advanced use cases later. We'll also use another Flask extension called Flask-script that will help us with database administration tasks. Now one thing we're going to see along the way is how to make our app, which is slowly becoming larger and harder to maintain, into separate Python files.

Demo: Our First Model Classes

So let's jump right in and look at some code. Of course, every time when I haven't been working on my project for awhile and I want to start again, I should remember to activate the virtual environment. Now I want to install the Flask extension that integrates Flask with SQLAlchemy, which is the library that does our object relational mapping. Like always we use pip for this, and after this has finished we see that it installed the Flask extension and SQLAlchemy itself too. Now I've added a file to our project called models. py, and of course I called it that because it contains two model classes, Bookmark and User. You can think of these classes as representing a database table, and they work pretty similar to the form classes we saw in the previous module. In this case, each class doesn't represent an HTML form, but a database table, and for each column in that table we have a class attribute. These attributes are instances of the Column class, and conveniently this Column class is a property of the db objects. You can think of this db object as representing the database connection. So the Bookmark table has four columns, and each column gets as its first argument the data type. For the id this is an Integer, for the url Text fields, the date is a DateTime, and the description is a String. Now each of these classes are attributes of the db objects just like Column. Now the nice thing about this is that we only have to import the db object from our main thermos module, and we don't have to import anything else. Now I'll show you where the db object is defined in a moment. We can also add extra options to the columns like that the id is a primary_key, and that the url should be filled, and that the default value for the date is the value of the utcnow function. Note that we don't use parentheses here because if we did that the default would be set to the value of utcnow at the moment that the module is loaded. Instead, we pass the utcnow function itself as an argument, and that will be called every time a bookmark is created. I've also added an __repr method to enable clear printing and logging of values. The User class is very similar to the Bookmark class. It has an id column, a username, and an email field, and both of the String fields have a unique restraint. Based on these model classes SQLAlchemy can generate our database tables, and it can create SQL queries and insert, update, and delete rows in our database. But of course we need to do some set up first, so let's see what our main file looks like. I've added a few simple lines to set up an SQLite database connection, and the nice thing about SQLite is that it comes bundled with Python, so we don't need to install anything, and we also don't have to configure or set up a database server. So that's very easy, and for our simple application SQLite is powerful enough. SQLAlchemy also supports many other databases, and chances are it will support your favorite database as well. Now we start with importing SQLAlchemy from flask_sqlalchemy. Then we need to set up the database collection URI as a config parameter for our app. Now the value of this URI starts with sqlite:// and then

another slash, which is part of the file system path to our database file, and then the directory form which our application runs followed by thermos. db, which will be the file name in which SQLite stores our data. Now to use the os. path. join function here, of course I need to import the os module too, and the location of the basedir directory I have to set in this line here where I determine the path to the current Python file. So all of this makes sure that we have a connection URI for SQLite that points to a file called thermos. db in the same directory as thermos. py. On the next line I actual initialize SQLAlchemy by calling the SQLAlchemy constructor and passing our app objects. This constructor returns new objects, and I assign that to my db variable. From here on the db variable represents my database connection and provides access to all the functionality of flask_sqlalchemy. So going back for a moment to the models module, in this file I import the db object form thermos so that I get access to all the classes that it provides so we can set up our model classes. Now, the next step is to let SQLAlchemy initialize our database and to see how we can insert data and query the database. But before we do that, let's do a short review.

Review: Models With Flask-SQLAlchemy

So we only need a few simple steps to get up and running with SQLAlchemy and Flask. First we install the flask-sqlalchemy module with pip. Now, of course, you don't need to do all of this because Flask really gives you complete and total freedom for choosing a persistence layer, so that means you can also choose to use SQLite directly or use any other kind of persistence technology. Now after installing we need to write a little code to import and configure SQLAlchemy. This only takes three lines. First we import the SQLAlchemy object from flask_sqlalchemy, then we use the string SQLALCHEMY_DATABASE_URI as a configuration key for our app. config dictionary, and we give that as a value, an sqlite URI with the absolute path to our database. Now in the demo I showed you how to automatically construct this path based on the location of our Python count. Now after configuring the database connection like this, we call the SQLAlchemy constructor with our app object and assign the results to the db variable. This db variable from now on represents our database connection, and this is what we will use to create tables and queries. Now apart from SQLite, SQLAlchemy supports many databases, so if you prefer another engine than SQLite that should be more problem at all. Like with our WD Forms classes for generating HTML forms, SQLAlchemy lets us create classes that represent our database tables, so each instance of such a class will represent a single row a data in that database table. Now these classes are fairly simple, and basically the only real requirement is that they should inherit from the db. Model class. Now Flask-SQLAlchemy is a sort of wrapper around the SQLAlchemy library, and as such it provides some convenient extras like putting the model

and field classes on the db objects to make importing unnecessary. Now another thing is that you won't have to explicitly set the SQL tablename for your models. In plain SQLAlchemy you do need to do this, but Flask-SQLAlchemy has that automated, and you won't have to do it. Now it also adds a query attribute to the model classes, which is not present on vanilla SQLAlchemy models. Now in the next demo I'll show you how to use this. So inside the body of our module classes we define the columns as class attributes. For each column the name of the attribute, in this case id, will be the name of the database column when SQLAlchemy generates that database. Each of the columns is an instance of the db. Column class, and as its first argument the db. Column constructor takes the data type. This is followed by any options, which will mostly be SQL constraints like that the field is nullable or unique or a primary_key, etc, etc. So, let's do another little demo to see how SQLAlchemy generates our database tables from our Python count.

Demo: Storing and Retrieving Data

So now we're back at our models module, and let's open my terminal to show the low-level workings of SQLAlchemy. I'm only going to show the basics because this course is about Flask and not about all the intricacies of SQLAlchemy. I'm in my top-level project directory, and my environment is active so I can start Python. And now I'll import the db object from the thermos module inside the thermos package, as well as the User and Bookmarks models. And now to create the database I call the create_all function on the db objects. This will create a new SQLite file and run the SQL for creating a new bookmark table and a new user table. Note that I need to import the model classes for that. If I just import db from thermos. py, this stipend will create a completely empty database without any tables. So if I want to create the tables for my models, I have to import my models as well. Now, after running this statement there's a new file in here called thermos. db, and let's take a closer look. So SQLAlchemy has created a bookmark and a user table for us. Let's compare this table with the Python code, and looking at the bookmark table we see it has columns id, url, date, and description with the data types as specified in our code. The same is true for the user table. So SQLAlchemy inspects our Python classes and generates the right SQL code to create these tables. Now I'm going to back to the Python interpreter, and I'll add some data to these tables. Let's create a User object by calling the constructor with a name and email address. Note that I didn't write a constructor myself. It's provided by SQLAlchemy based on our class attributes that tell it which fields to accept. Now SQLAlchemy follows a pattern that's called unit of work. That basically means that it has a place where it stores all changes to the database until we tell it to enact these changes at once. For that it uses an object called db. session. My new user instance right now does not have any association

with the database yet, and to add it to the database first we add it to this session object. Now SQLAlchemy knows that there's a user object that's ready to be stored in the database. To execute the necessary SQL and really store this data we say db. session. commit, and immediately we see that my table now has one row. Looking at the data we see that my information has indeed been stored correctly. Note how there's also an id of 1, which I didn't set myself. This is the primary key, and it will be generated for us. We never set it ourself. Let's repeat this exercise and add a bookmark. It's very similar, but first I need to import datetime because we have to add a timestamp. We can now add a bookmark, and note how I call the datetime. utcnow function to set the timestamp to the current moment. In this line I'm immediately adding the bookmark to the session. And I'll commit again, and again we see that there's now a row containing our bookmark in the bookmark table. Of course I could have added both objects to a single session in one go and committed them both at the same time. There's no real need to make a new session for every object. Finally, let me show you that we can do queries. I'll only show you two very simple examples right now. Each model class has a query attribute that contains an sqlalchemy. BaseQuery instance. This is the class that we use to build SQL queries. Probably the simplest example is to retrieve an object by primary key with the get function. So to retire the data we just stored we might say user. query. get(1), and this returns our user objects. If we call it with a nonexistent id like 2, it returns none. We can also use filter_by, which results in a query with a select clause. So here I select any user rows where the username is reindert. Now as you see this returns a BaseQuery object, so you can think of filter_by as a function that modifies the query to add a where clause, in this case where username is reindert, but it doesn't actually execute that query. For that I can call the all method on the query, and that will return all matching rows. And now we receive a list with a single user object. And that's a minimal introduction to SQLAlchemy. We will see more intricate queries in due time.

Review: Storing Data and Simple Queries

So the steps for adding new data to the database are quite simple. First you call the constructor to create your new objects. This constructor is provided by SQLAlchemy, so you shouldn't write your own. It takes keyword arguments for each field, and you never have to specify a value for the id field here because that will be added automatically. Next you can add the new object to the session with db. session add. Now this doesn't execute any SQL yet. It simply registers the object as one of the things that have changed and that should be stored. Then you should, of course, not forget to commit. This will run the actual SQL statements that results in a new row in the database table. SQLAlchemy also lets us query the database. For this we use the query attribute

on a model class. This is actually something that the flask-sqlalchemy extension adds for convenience. Normal SQLAlchemy classes don't have these attributes. The query class lets you build any query you like. It offers lots of functionality and enables you to build the most complex queries you can think of, but for now let's stick with the simple ones. We saw that the get function retrieves a row by primary_key and that the all function retrieves all rows matching a query. In this case we've not added any filtering, so using all on the query object directly will yield all rows in the table. You can add a where clause to select rows that match a criterion with the filter_by function. It takes a keyword argument that will become the where part. Remember that the filter_by method just modifies the query. It doesn't execute it. Filter_by returns a new query object, and we can call another function on that to have it executed and receive the results. In this example I'm using the first function, which obviously returns the first matching row. You can also use all, which will return a list of all matching rows.

Demo: Storing and Retrieving New Bookmarks

So I've introduced you to SQLAlchemy, and now let's see how to use it in our application. To start with I'm importing the Bookmark class from the models module, and I've removed our old function that stored the bookmarks in a global list. Now in the view function I've replaced the call to that stored Bookmark function with two lines, one that creates a new bookmark and one that adds is to the session. Let's start our application and see what happens. First I'm going to have to cd into the thermos package and start the projects. But now we get an error, and this is actually a circular import problem. While the interpreter is reading the thermos module it encounters the line that imports the models module, but then when reading the models module that one tries to import the db object from thermos. So that's a recursive input call, but soon we'll see how to solve this in a very eloquent way that also provides for a nicer way to start our application. But for now I'll fix this by changing the line like this. And this of course means that now I have to say models. Bookmark instead of just Bookmark because now we don't use any names from the models module at the time of loading that module. That's why we can get away with a circular import like this. But like I said, we're going to see a better solution soon. Now, let's run again, and this seems to work. Let's try and add a URL, and that seems to work. It says Stored 'test'. Let's check the database, but this still seems to be empty, and you probably saw this coming. This is because I forgot to commit, so let's add another line to the view method and try again. And now we see our bookmark has indeed been stored. But of course it would be better if we gave the user some real feedback. So I'm going to remove the old method for showing the newest bookmarks, and instead I'm going to add a method to our Bookmark model. Now you see the

model class represents our data, so in my opinion this kind of logic should be a part of the model class because it operates on that data. As you see it's a static method because we don't need a bookmark object to be able to call it, and we also see some new features of SQLAlchemy. I'm using order_by instead of filter_by, and I can pass order_by a field of our class, so in this case I'm passing it Bookmark. date, and SQLAlchemy will know how to generate the SQL to order by the date column. But I want it in descending order, and for that SQLAlchemy has the desc function, which I import and call on the date fields. Now all of this, just like filter_by, just modifies the query, but doesn't execute it. This time instead of calling all I'm calling limit on the query so that it returns only a limited number of rows. We give that the parameter num, and of course we need to fix the view as well. The list of new_bookmarks we pass to the index template now comes from our static function on the Bookmark class. Now refreshing our index page we see the list of new bookmarks, which at the moment only contains the single bookmark I just added.

Demo: A Manager Script With Flask-Script

Now before we go on and add a foreign key relation between the user and bookmark tables, I want to make our life a little bit easier. I've added a script called manage. py, which will automatically create or drop the database with a single command. I'm using an extension called flask-script for that, so let's first install that with pip. Now I import an object called manager from this extension, and I define two methods, initdb and dropdb. By decorating these methods with manager. command these functions will become available on the command line. The first method calls db. create_all, and the second one db. drop_all. Now of course there's also a main method, which calls manager. run. Now to use these scripts I type python manage. py, and that gives me an overview of the commands it reports. Let's start by calling dropdb to remove the database, and we now see that the database has no tables at all anymore. Running initdb creates the tables again. And that's really nice, especially when we change the model and we want to recreate the database. Something else that flask-script provides by default is a command called runserver. And as you can see this runs our application, and soon we'll going to make this our standard way of starting our application. But now let's first add a database relation.

Demo: A One-To-Many Relation

Let's go back to our models. I added a new field user_id to the Bookmark class, and as you can see it's an Integer field that's set to be a ForeignKey linked to the id field of the user table. So this is a basic foreign key relation where each bookmark has a reference to a user. I also set nullable

as False, so a bookmark always has to be linked to a user. Now the User class, on the other hand, has a bookmarks list. We tell SQLAlchemy to use the relation between the User and the Bookmark class so it will use the foreign key we set up on the Bookmark class. We also give it a backref argument, which means that on the other side of the relation, which is on the Bookmark side, there will be a property called user, which will hold the user objects, so we'll not have to work with the user. id field on the Bookmark class. Instead we can work with real Python objects since there will be a list of bookmark objects on every user, and there will be a user object on every bookmark. That way we hide the mechanics of setting up foreign keys on the database. Finally, there's an argument lazy, which tells SQLAlchemy how to load the Bookmark objects. I'll explain that in a moment in one of the slides. Now to try this out we have to regenerate our database tables because their layout has changed, so I'm going to start with dropping and creating the database. For now I added two lines to our initdb function to add two default users so we have something to work with. And let's start an interpreter and import the classes we need. And I'm retrieving our first user, and now I'm going to add some bookmarks in a single for loop. So now our bookmark table has three rows, and now we can ask for these bookmarks on the user objects. But as you can see this is a query object, so we have to execute it first, and for that we can use the all function. And here we see our new bookmarks. Each of these bookmarks has a user attribute as well, and that points back to our user. And that's how we can work with database relations just like they're Python references. So we are not using SQL, we are not even aware of there being an id and a foreign key, we just use normal Python attributes and objects. Now how do we apply this in our application? Well first of all when we create a bookmark we link it to a user. Normally that would be the currently logged in user, but we don't have a log in page yet, so let's fake that for now. I added a logged_in_user method which simply always returns the user reindert, so this basically mocks the login process. We pass that to the constructor of the bookmark class, and that will make sure the foreign key will be set correctly. By the way, in the next module we'll see how to actually authenticate at login, but for now we'll stick with this mockup for login function. Then I also change the template a bit so that we're now showing the user who edited the bookmark in the bookmark list. So let's start our application again, and this time we'll use the manage scripts, and we now see the bookmark list including the username. I didn't add descriptions to the bookmarks, so that's why they all say none. Let's add a new one to test whether our view still works. And as you can see now our new bookmark with a new foreign key relation to the user, is stored. As a last note let me tell you that I removed the main method from thermos. py because I will never run our thermos. py as a standalone script again. From now on I will start our application with the manage scripts.

Review: One-To-Many Relations and Lazy Loading

So adding a one-to-many relation is not that hard either. You have to make sure that on the many end of the relation you add a ForeignKey. In our case that's the Bookmark class, and it will hold a ForeignKey to the user table. Then you add a db. relationship attribute to the one side of the relation, so that's the User class. This defines the one-to-many relation. The first argument of the call refers to the class that is on the many side of the relation, and the backref defines a property that will be set on the many side that will refer back to the user objects. So with a call to db. relationship SQLAlchemy will create both a bookmarks attribute on the user side and a user attribute on the bookmark side. Finally there's the lazy argument, and I set that to dynamic. It defines how the related data should be loaded. Let's explain that a little bit better. Now when we add a relation like this it might mean that when you retrieve a user object the bookmarks attribute contains many hundreds of items. You may not want to load all these rows of bookmarks every time when you retrieve a user, so we can have SQLAlchemy lazy load that data instead, and that's where the lazy argument comes in. There are several options, but the most important ones are the following: Lazy is select means that the bookmarks will only be loaded when you access the bookmark attribute on the user. They will all be loaded at that moment with a select statement. Lazy is joined will load the bookmarks with the user in a join, so no lazy loading at all. And subquery will do the same, but in an SQL subquery. And finally there's dynamic loading, which is what we are using. This is what you probably want when there may be lots of items. In this case the bookmarks list on the user object is not a normal list, but actually a lazy query object. You can call methods like all or filter_by on that object, so this gives you some more control over performance because it lets you limit the data to specific routes before you actually load all of them.

Demo: Breaking the Project Up into Smaller Files

Now I find that the thermos. py file is getting a bit over crowded. We also run into some problems importing files with a circular import problem, and I want to make things a bit cleaner, so I'm going to break up the application in multiple files. First of all I'm moving all application setup into the __init. py, and I moved all views into a new module called views. py. Now here, at the end of the init file I import the models and the views. So now our application is a real Python package, and we can use it like that. Now the manage. py script I have moved out the regular project directory since it's not really a part of the site itself, but rather a script that manipulates it. Note how I'm now importing from thermos. models instead of from models directly because, of course, models is inside the package. In views. py I can now say from models import User and Bookmark

because our circular import problem has been solved, so this module now contains all my view functions and nothing else. So now I have a very cleanly separated set of files with forms. py holding all my HTML form objects, models containing all the models for my database layer, views. py holding all my views, and init. py is basically the thing that binds the package together, holds the configuration, and imports the views and models. So all of this has several nice consequences. First of all we now have a real functioning Python package that we can use from the top level of our project through the manage scripts. I can run the project by calling manage. py run server. We also have separated views and configuration, which is a bit cleaner, and we also don't have any import problems anymore.

Demo: Adding a User Page

And now for the final step in this module. I've added a new template called user. html, and it shows the bookmarks for a single user as you can see in the title here. It also contains a for loop that shows all bookmarks for that user by looping over the bookmarks attribute. Although the bookmarks attribute is an SQLAlchemy query object, looping over it like this will cause the SQL to be executed, and it will yield each bookmark in turn. And I'm also showing the timestamp for the bookmark, and I've added and else clause that shows a message when a user has no bookmarks at all, so this is basically a user page which shows all bookmarks for a specific user. Now of course we also need a view function, and this function is called user, and we see a new feature of the flask routing decorator. If we put in name and angle brackets, it will become a parameter for the function, so this route will match any URL starting with user/ and then any username. This part of the URL will be passed to the view function as the username argument. Now inside the view function we can of course write code that gets a user from the database with a query that matches the username and then check if that returns any user by that name and return a 404 not found error if the user doesn't exist. But fortunately flask-SQLAlchemy provides a nice convenience function that does all of this in a single call. We use filter_by to get a user with the give username, and then we call first_or_404. And that does exactly what it just said. It checks whether the query returns anything, and if not it returns a 404 error. And if there are any matching items in the database, it returns the first one. Finally, of course, we need to be able to go to a user page, so let's add a link. And here in the bookmarks list on the index page I'm adding a url_for call that links to the user view. As we already saw with static content we can pass data to that view by simply adding it as a keyword argument. So here I tell url_for to pass an argument username to the user view, and that username is of course the username of the user that is related the current bookmark through our foreign key relation. So let's test this. Here's my

bookmark list, and my username is bow clickable. And when I click it I go to /user/reindert. Now this page says bookmarks for reindert and shows all my bookmarks. Now let's try to go to the page for other user, arjen, and it says here we have no bookmarks at all, which is correct. Now and if we try any other username, let's say test, we get a 404 error because there's no such user. And so you see how using the foreign key relation we can very easily use the bookmark to link to a user page, which then shows all the bookmarks for that user simply by using Python references. And we don't have to use any kind of low-level SQL statement or make use of the foreign key relation. We can simply use Python objects, and all the SQL is generated for us.

Final Review

So there are a few final things I want to review. First of all we saw how the use of a name in brackets in a URL will cause flask to take that part of the URL and pass it to the view. In this example we're taking the username part of the view, which is everything after the slash, and pass it to the view as an argument. Given that username we can construct a query to look up the user with that name. On that query we can use the first_or_404 function, which will either return the first matching records or a 404 not found error. This is one of the very nice features of flask-SQLAlchemy. Now there's also a get_or_404 method, which is nice if the URL contains a number, which is the database ID for a row. Get_or_404 will retrieve a row by primary_key or return a 404 error. During this course we also saw how to do simple database administration tasks. First of all we saw that the create_all method on the database object will create a new database with a table for each model, and similarly the drop_all method will drop the database. And I showed you that we can use flask-scripts to create a nice admin interface. It also gives us the run server method to start our application.

Resources And Summary

So if you want to learn more, like always, here are some resources. First, here's a link to the Flask-SQLAlchemy extensions homepage. This link points to the documentation for SQLAlchemy, which is the ORM library that is used by Flask-SQLAlchemy. And this is the homepage for flask-script, which I used for the manage script. And that's it for this module. I've introduced you to Flask-Sqlalchemy, and I've shown you how to write model classes, which represent database tables. We've seen how to automatically create and drop the database, how to insert data and do simple queries, and how to define a one-to-many relation and do lazy loading on the data from that relation. We've also seen a convenience function called first_or_404. Then along the way we saw

how to use flask-script to set up nice scripts to run database administration and to run our application. And we saw how to break up our application into multiple files. Now, let's move on to the next module in which we'll see how to add user authentication and log in.

Users and Authentication

Introduction

Welcome. I'm Reindert-Jan Ekker, and in this module we'll add user sign up and authentication to our projects. In the previous module we added a fake function that always returns the same logged in user. In this module we'll fix that by implementing real login authentication. We'll start with installing and configuring another Flask extension called Flask-Login. We also need to do a small adaptation to the User class to make it work with that extension. The next thing is to decorate all views for which we want users to be logged in with a decorator provided by Flask-Login, and that decorator is called login_required. Of course, that means we'll have to add a mechanism for the user to actually log in, so we'll add a login view with a form and a template, and of course we shouldn't forget a logout counterpart to that. Finally we have to enable users to sign up for the site, and that of course also entails adding a view, a form, and a template.

Demo: Flask-Login Setup

So by now you're probably getting used to it. We start by installing an extension, and this time it's called flask-login, and so we call pip install flask-login. And when that finishes we open init. py and add four lines. Now we start by importing the LoginManager class from flask-login, and then the first line here calls the constructor for the login_manager object and assigns that to a LoginManager variable. Then we can set several options on the LoginManager, and right now I just set the session_protection to strong for some extra security, and then I add the LoginManager's functionality to our Flask app by calling login_manager. init_app. And of course, I pass that our Flask app as an argument. And that's basically all the set up we need. With this done we can declare that some views are only accessible for users that are logged in, so in the views module let's start by decorating our view for adding a new bookmark with login_required. Of course we have to import login_required from flask_login. And now when I open the site in my browser and I go to the page for adding a bookmark I get an unauthorized error. So this tells me I

need to log in to look at this page, and that means the next step, of course, is to actually enable the user to log in.

Demo: Preparing the User Model

So let's add a login page, and there are quite a few steps we need to do, so we're going to cover a lot of ground in the next couple of demos. Just try to follow me, and don't worry. We will do a review afterwards. Now to start flask_login requires that our User class has a couple of extra methods, so let's open models. py. Now fortunately we don't have to write any methods ourselves. We can simply have the user class inherit from the UserMixin class, which you can import form flask_login. Now let's take a short look at the UserMixin class to see what it adds. There are four methods here: is_active, is_authenticated, is_anonymous, and get_id. All of these are required for flask_login, and let me explain how this works. Remember we decorated our app bookmark view with the login_required decorator. That decorator will check if there is a user object in the session for which is_anonymous returns false, among other things. Now when a user has not successfully logged in yet, let's go down here, the object in the session will be an instance of this class, AnonymousUserMixin. That means that is_anonymous is true, is_active is false, and is_authenticated is false as well. Of course, that means that login_required will make sure that you cannot view the add bookmark page if you try to go there. So now suppose the user goes to the login page, fills in the form correctly, and the login view, which we will be writing in a moment, will retrieve the user data from the database which will be represented as our model class, which is an instance of UserMixin. Now flask_login stores the ID of these model objects in the HTTP sessions so that it can be retrieved with every request, which means that for every view that is decorated with login_required it can retrieve this UserMixin instance, which represents the user. Now for the UserMixin class is_authenticated is true just like is_active. And is_anonymous is false, so that means that now if you go to a view that is decorated with login required it will let you through, and you can view that page. So I hope that clears up why our user class needs to inherit from UserMixin. Actually, you can also choose not to do this and implement the four functions yourself. For example, you may want to be able to suspend a user, and one way to do that is to store a suspend flag with the user in the database. You could then overwrite the is_active function from UserMixin with the value of that flag from the database. Very well. Let's move on to actually making this work and add a login page.

Demo: Adding the Login Page

So let's start with adding a LoginForm class in forms. py. So here I import a couple of new form field classes, namely PasswordField, BooleanField, and SubmitField. The new LoginForm class is a normal WTForms class, which inherits from the BaseForm class, and we have a username field, which takes a string input; the PasswordField, which will become an HTML input with type password, so the text the user types will not show up; and a remember_me field of type BooleanField. This will become a checkbox on the HTML page, which the user can activate if he wants to be remembered. This is a standard functionality from flask-login. It will set a cookie so that the user doesn't have to log in again the next time he visits our site. Of course, both the username and the password have to be filled in, so both get a DataRequired validator. I also added the SubmitField, which will become the submit button for the form. Personally I like to keep these classes minimal, so normally I just hard code the submit button in the template and leave it out of the form class, but I put it in here just so you can see the WTForms can also generate the submit button. Now of course there's a template too. Let's look at it. And basically is the same HTML as the add bookmark page. The main difference is that I changed the title, and there are a different set of fields. I set the size for the username and password, but not for the remember me checkbox, nor for the submit button. So the final component, as always, is the view, so let's go to views. py and check it out. I have to start with importing login_user, which, as you can guess, is the function to call when the user correctly fills in the form. Also, don't forget to import the new LoginForm class. Scrolling down we find my new login view. It works just like most functions that handle a form. We map it to the URL /login, and it handles both GET and POST requests. We start by creating a form instance, and remember that this will either be an empty form in the case of a GET request, or it will be filled with the data the user filled in in case of a POST request. Then we validate the form, and in case all required fields are filled in we are ready to check the password. Now I didn't implement password checking yet, so at the moment I'll just look up the user and assume the password is correct. If the user exists, we call the login_user method, and remember that flask_login will store the user's ID in the HTTP session so that it can retrieve the user at every subsequent request. I also passed the value of the remember_me checkbox, and that will be handled by flask_login without any more work from our part. We also flash a friendly message to the user and redirect to another page. Now this line is a bit peculiar. It redirects to either one thing or another, and I'm going to come back to this in a moment. For now the first part will be false, and we redirect to the index page. Finally, of course, if the user doesn't exist, we flash a message about that and rerender the form. Okay, so let's test all this. Now I didn't fix the Sign in link here just yet, so I'll have to mainly type in the URL and go to /login. And let's fill in my credentials. But now I get an error. NoneType is not callable. Now this is not really a clear error, but with some reasoning we can deduct what's going wrong. See how

the URL has changed from login to index. Well, that means that the view seems to have functioned correctly and that we were redirected to the index page. So what goes wrong here? Well, flask_login only has the logged in user's ID in the HTTP session, and it needs to get the actual user object from the database. But of course flask-login doesn't know about our database setup and model classes, so we need to do a bit ore configuration so it knows how to retrieve the user. To fix this let's go back to views. py. At the top here I added a new method called load_user that takes an ID and calls user. query. get, which will retrieve a user by primary key. Now decorating this method with login_manager. user_loader tells the login manager that this is the way to retrieve a user object based on the ID. By the way, of course I also removed our old fake logged in user function because we don't need that anymore. So now going back to the browser we find that we can log in, and reentering my credentials we find that we can log in, and we're successfully redirected to the index page. And I can visit the add bookmark page as well. Now we can't actually add a bookmark at the moment because I just broke that piece of functionality, but we're going to fix that again in a moment.

Demo: Redirect After Login

Now I know you already got a lot of information seeing all these demos, but bear with me because there's still one last step before the review. To show you I have to log out, but we don't have a log out button, so I'm just going to open the developer tools and remove the cookie, and that's the same as logging out. Now clicking Add URL we still get our ugly error message, and I'd prefer to automatically redirect the user to the login page in such a case. And flask_login does do that, but we need to tell it the name of our login view so that it can use url_for with that name and redirect to the right URL. So here we are in __init__. py, and all I have to do is add one line to the configuration where I set the name of our login_view function. So now when I'm not logged in the login_required decorator on the add view will know what the name of our login view is, and it can redirect us there. Going back to our login view you should know that when flask_login redirects a user to the login page it can also pass the login view and argument called next in the request, which holds the page that the user was trying to visit. Here in the login view, if there is a next argument in the request, we redirect there. So a user might be trying to visit the add bookmark page, gets redirected to the login page, but under water the next argument here makes sure that after logging in we get redirected back to the add bookmark page after all. So let's test that. Let's reload this, and as you can see in the URL, here we are now at the login page, and there's a next argument that says we're trying to visit the add page. And then when I enter

my credentials this redirects me back to the add bookmark page. Very well. That was a lot of information. Let's review.

Review: Flask-Login Overview

So in the demo, as always, we started with installing flask-login with pip. With that out of the way, the next thing is to set everything up so we can use it in our project. That of course involves writing some code to integrate flask-login with our application. First we import the login_manager class, and then we instantiate it so that we can do some configuration. For our app I only need two lines of configuration, and that is to set the session_protection to strong for some extra security against session hijacking, and then I tell the login_manager what the name of our login_view function is, and I have appropriately called that function login. Remember that we need to do this so that flask-login can redirect users to that view when they try to access a page that is restricted to logged in users only. Finally we call init_app and pass in our flask app objects. This integrates the login function _____ with our application so that we can use in the login_required decorator and other features. But that's not all. Flask-login needs to access user instances, and for that we have to declare a user_loader. This gives the extension or mechanism to retrieve users by id. When a user is logged in, the user id is stored in the HTTP session, and with every request the user object gets loaded. Configuring this is easy. We simply define a function that returns a user with the given id. In this case, of course, I implemented a using SQLAlchemy, but if you chose a different persistence method you would use that here. We decorate this function with login_manager. user_loader, and the final step is to make a small change on the user class. Flask-login needs the user class to implement a couple of methods like is_authenticated, but the UserMixin provides default implementations for all of these. So in most simple use cases all we need to do is to make sure that the user class inherits from UserMixin, and we're all done. And that means we're now ready to start using flask-login. Some of the things in this slide we've already seen in the demo, and others we'll see in action in a moment. To start, any views that should only be accessible to logged in users we mark with login_required. A user that's not logged in trying to access such a view will be redirected to the login page we configured previously. After a successful login, they will be redirected back to the page they were trying to visit. Inside these view functions, a variable called current_user always points to the currently logged in user. We'll use this soon to set a relation between a new bookmark and the current user. Now, to make it possible for there to be a logged in user, we need to create a login page, and in the view for that we call the login¬_user function. This takes the user you want logged in as its first argument, and optionally as a second argument you can pass it a Boolean for the

remember_me functionality. If this is set to true, the login will be remembered in a cookie, and the user won't have to log in again the next time he visits the site. Now the last two things on this slide we haven't seen in practice yet. There's a logout_user function that logs user out, and flask-login sets the current user variable in our templates as well so we can check whether the current user is authenticated. Now let's see some more demos where I actually use current_user and logout_user.

Demo: Current_user and Logout

So flask-login provides a variable called current_user. As always, we need to import it first, and note that I'm importing logout_user too. We're going to need that. So now when adding a bookmark we can link the bookmark to that user so that it will be added to their profile page. Another use for the current_user variable is in templates. So let's open the base template, and here I changed the sign in link in such a way that the contents of the page will depend on the status of the user. I call the is_authenticated function on current_user, and if it's false we show a login link, but when the user is logged in we show a logout link. Of course that means we need to have a logout view as well. So let's go back to the views again, and down here is a simple view that does nothing else but log the user out with logout_user and then redirect them to the homepage. We don't even need a template for this view. So let's open a browser, and I'm going to log in as another user than myself this time, and immediately you can see how the links at the top change. I can add a URL, and we can redirect it to the index page, but we can see that the URL now shows the name of the user that created this bookmark. Finally I can log out, and now when I click Add URL I get a message that I should first log in.

Demo: Password Hashing

There's still one important thing missing in our login mechanism, and that is checking the credentials. We need secure password storage and a means to check these passwords as well. Now of course I'm not going to build everything from scratch. Werkzeug, that's the library that Flask uses for routing, among other things, gives us two nice functions, which I import here: check_password_hash and generate_password_hash. As the names suggest these can create and verify password hashes. Now let's scroll down a bit so we can see the entire user model, and to the user model I'm adding a new Column password_hash that contains a String. This will not contain the password itself, but a secure hash. We use Python properties to make this field a write only field so we can set the value as a string, but only read it as a hash. So I'm creating a property

password, and this will not be represented in the database table. If we try to read it directly, it raises an AttributeError. The setter for this attribute generates a hash and stores it in the password_hash field. Then I add a method, check_password, that checks a string against that hash. So when a user tries to log in and types in a password, that is passed as a string to this function, and that will hash it and compare the hashes. Finally I added a staticmethod called get_by_username. We can pass this a username, and it returns to corresponding user. I like this approach better than doing the filter_by in a view because this kind of operation, in my opinion, should be part of a model class and shouldn't be in the view code. Now all of this gives us the tools to finish up our login view. Let's go there. And this is the final version of my login view. We call get_by_username, and then we check whether that username exists, and if so we check the passwords. If one of these fails, we say incorrect username or password and rerender the form. If they succeed, we call login_user and redirect. We now redirect either to the page from the next argument, or if there is no next argument we don't redirect to the index page anymore, but we can now redirect straight to the user's homepage. Finally, let's add some default passwords, and purely as a convenience I'm adding that to manage. py. I'm just setting both users' passwords to test for now. Of course, now I have to drop and initialize the database because I changed the model. So let's stop our application with Ctrl+C, and let's say python manage. py dropdb followed by python manage. py initdb. Of course I'll have to restart my application as well, and testing this I can try to log in with an incorrect username, and we get an error message. Giving a real username and password takes me right to my own bookmark page.

Demo: Adding a Signup Page

Let's finish up this module by adding a page where new visitors can sign up for an account. Let's start with an HTML form for this, and the SignupForm class has four fields: a username, two password fields with second one for validation of course, and an email field. As you can see I added some more validation rules than with our other form classes so far. The StringField for the username has a Length validator that sets the allowed length for a username between three and 80 characters. I also added a regular expression validator that only matches letters, numbers, and underscores. The PasswordField gets an EqualTo validator that checks that the contents of the two password fields are equal. Of course, we need to import all these validators as well. Finally I wrote two custom validation routines. Now looking back at the bookmark form class, what I did here is overwrite the validate method, which is for validating the whole form at once, but in our new class I'm adding two methods, which validate specific fields. The nice thing here is that these validator functions will generate errors on these exact fields, so that error will show up in our

HTML page next to the input that the error is for. The validate_email function checks that the email address is not already in the database, and validate_username does the same thing for the username. In both cases we raise a ValidationError with an error message if anything is wrong. Then of course there's a template. The signup template won't surprise you. It's basically the same as all our other forms, but of course we need to call render_field for our specific form fields, and in this case it's form. username, email, password, and password2. And of course we have a view as well. It's called signup and the URL is signup as well with methods GET and POST as with every form so far. As always we create the SignupForm instance, do a validate_on_submit, and if everything is filled in correctly we create a new user instance, add it to the database session, and commit. Then we flash a Welcome message and ask the user to login. We also redirect them to the login page. Now the last step is to make this new page reachable, so let's go to the base template again, and I added two more lines to our if statement here. Now when you're not logged in you see a Sign up link to make a new account and a Sign in link for when you already have an account. After logging in there's a link to your own bookmark page and the logout link as well. So now we can go to the sign up page, and let's try to make a user with a very short strange name. And now we see two error messages, one from the regular expression validator and one form the length validator. Of course when we fill in everything we get the login page that shows that the user has correctly been created.

Resources and Summary

That concludes this module, and this time I only have one link for you and that will take you to the flask-login documentation. So this module was all about users and login. We have installed and configured Flask-Login, and among other things we needed to implement a user_loader and have the user model inherit from UserMixin, and with that done we can decorate views with login_required to make them accessible only to members. After login, the current user variable holds the user instance so we can access their data. We can also use this in templates, for example, to change the page based on the user's login status. There's also login and logout_user, which do exactly as their name suggests. Now, apart from Flask-Login, we also saw how to implement password hashing on the user model, and we added the signup page to enable new members to create an account. So let's move on to the next module. We're going to see some advanced features like template filters and includes, and we'll add tagging to our bookmarks with a many-to-many relation.

Managing Bookmarks

Introduction

Hi, my name is Reindert-Jan Ekker and in this module we'll add some features to our site for editing and deleting bookmarks. Along the way we'll learn some slightly more advanced techniques. We have now pretty much seen the basics of setting up a site with Flask, let's move on to some more advanced techniques. We'll implement four things. First we're going to fix the layout of the bookmark list, then we're going to add pages for deleting and editing bookmarks. First we'll add the edit page, and the delete page will be at the very end of the module. We'll be adding text to bookmarks, and while implementing these things we'll come across some new and advanced things. These can be roughly separated into two categories. First we'll have things that have to do with templates. So these will have to do with the Jinja2 template engine, we'll see how to use Jinja filters to do some new Operations on our data. We'll create a partial template that lays out a single bookmark and then include that in our bookmark list. We'll learn about something called a context processor that lets us inject things into the template context, and we'll get to use some JavaScript in a template. The other category concerns the database. To add text to bookmarks we need to implement a many-to-many relation. And that means the database model changes, and we can use a flask extension to make our life easier and do automated database migration. So now you know what to expect, let's get started by fixing the layout of our bookmark_list.

Demo: Fixing the Bookmark Layout

Let's start with fixing the way our bookmarks look. I've added this new template called bookmark_list. I will never be using this page on its own, we'll only be including it on pages that show list of bookmarks, like the index page, or a user's own bookmark_list. So this is not a complete HTML page, instead this template only returns the HTML for ul tag containing a list of bookmarks, and we'll include that in other pages. Now we assume there will be a variable in the template context called bookmarks that contains the bookmarks to display in the list. We loop over that and create a list item for everyone. There's an anchor tag here that links to the bookmark URL and it contains the bookmarks description as and h3 tag, and the bookmark's URL itself gets displayed on the page as well. Now because the description and the URL can be very long, I'm using a Jinja2 filter called truncate here. Such a filter is basically a function that is present in the template context and that you can apply to a variable to transform it. Here I use the

standard filter truncate with an argument of 50, which means that the description will be truncated after 50 characters. The killwords option here means that truncate will not leave words intact. So it will truncate the description in the middle of a word. I do the same with URL because that can be quite long too. There's also another div inside the list item and that contains some extra info about the bookmark. It shows the text Added by with the user that added the bookmark, and a link to their bookmark list. And then there's something new here, a function called moment, and this is actually a way to call moment. js, which is a nice JavaScript library that formats dates and times in a friendly way. Because remember we stored the date for bookmark as a utctime, which is not really meaningful for most people. Now using moment we can make this a bit nicer. To use it I have to install another flask extension called flask-moment, which pulls in the JavaScript library and makes it available in our templates. And let's do so by typing pip install flask-moment. And of course we need to do some set up as well. So let's open init. py, and here we import moment from the extension and call the constructor with our app as an argument. And that's all we need to do to make the moment. js library available inside our templates So looking at our base template now, at the bottom here I remove the lines that included jQuery because flask-moment provides a nicer way to do this. I also include the moment library here. Because we do this in base. html, moment and jQuery will now be available in every page. Now finally of course, we need to actually use our new bookmark list. On the index page I'm not looping over bookmarks and generating a list anymore, instead I'm now using the include directive from Jinja to include the bookmark list template. Remember this will loop over a variable called bookmarks, but index. html gets a variable from the view called new bookmarks, which contains only the newest bookmarks. Fortunately Jinja provides a directive called with, which we can use to give the variable a new name within the scope of the with statement. So locally the new bookmark list is now called bookmarks and we can safely include the template now because it can now retrieve a variable with the name bookmarks. Now on the user page we basically do the same thing. We need to retrieve the bookmarks as an attribute form the user, so again we assign it the name bookmarks first, with a with statement. Also notice that I'm calling count on the bookmarks here. That doesn't look very Pythonic, you would expect this to simply read if user. bookmarks, right? Well, yes. But the thing is that books is actually an SQLAlchemy query object from the user model, and that will evaluate to true in an if statement. So if I remove the count, like this. The if statement would always be true and we would never see the text in the else statement, not even when there are no bookmarks at all in the database. So that's why I'm using count here. It wouldn't be necessary on a normal list, but on SQL query object it is necessary. Let me show you how this looks now. So the list of new bookmarks here on the index page is now generated by the bookmark list template. Both the description and the URL are clickable and they will send me to

the bookmarks page. Finally, notice how our call to moment sows the date in a very human-friendly way. Now finally let me show you how I made this layout. I added some simple CSS rules for that in main. css. And first of all I decided to change the colorful links so that they are blue regardless of whether they've been visited or not. And scrolling down a bit I added these simple CSS rules for the bookmark list, the header part for every bookmark, and the info part of every bookmark. I also added a rule for a span with the class editlinks, and this span will contain an edit and a delete link for every bookmark, we're going to add that in a moment. So with all of this out of the way let's go ahead and add the edit_bookmark page.

Demo: Editing Bookmarks

So now I want to add a page that lets the user edit a bookmark. Now we already have a form for adding a bookmark. So what I've done, I've moved the template for adding a bookmark, which was called add. html, to a new file called bookmark_form. html. And we'll be using this form both when adding a new bookmark and when editing it. Now the title of this page is now a variable, and that variable I used both for the browser title, as for the h1 tag. And this way we can set the title to add a new bookmark or edit bookmark depending on the use case. And that's really all I changed here in this template, so apart form that it's exactly the same as the old add. html template. So of course, now we need to add a view function. Looking in views. py this is our new function called edit_bookmark. Its URL is /edit followed by the id of the bookmark, which of course has to be an int. The user also has to be logged into be able to visit this view, hence login_required. The id of the bookmark gets passed to the function and we use the get_or_404 function to retrieve the bookmark with that id. If none exists it will cause a 404 error. And next I check whether this bookmark is actually owned by this user. And if not they're not allowed to edit this bookmark and I want to show 403 forbidden error. Flask has a function for that and it's called abort, which we can call with an HTTP status code. This will make Flask stop processing the request and return that status to the client. And now we use the same bookmark form class we also use for the add view, but we use it in a slightly different way. We now pass it the bookmark as an argument named obj, or objects. This will fill the form with the data we just retrieved from the database, so the user will not see an empty form as with the add bookmark page, but a pre-filled form. But note that the form class only takes data from this argument is there's no data in the form from the request. That is special flask WTF feature. It means that if the user has already edited something in the browser, that data will be in the request, and it takes precedence over the data we got from the database. So here flask WTF does exactly what we want. If there is user input we take that, but if there is no user input yet we pre-fill the form with the data from the

database. Now after validating the form we need to transfer the data from the user to the database. And of course we can copy each field one by one, but flask WTF has a better way and that is the populate object method. Which does the same thing, it loops over all attributes and copies them onto the object we pass it. So it will copy the form data onto the bookmark. Now bookmark is an object we got form the database with an SQLAlchemy query five lines earlier. And that means the bookmark object is already part of our database session. I don't have to call session. add or anything like it, I can simply commit the session and any changes will be written to the database. Finally, we flash a message and redirect back to the user page. And of course, I shouldn't forget to import the abort function like this, and of course I added a custom error handler to show the message in a nice way. Like the other error pages, the 403 error page is a copy of the error page I downloaded form initializer. It only shows a different message saying you don't have permissions to view this page. Now of course, we need a way for the user to access the added page, so in the bookmark_list template I added the span with class added links, which contains a link to the edit_bookmark view. Using url_for we pass it the id of this bookmark. Note that I use an if statement to only show this link for bookmarks owned by the current user. Also remember that in the previous clip I showed you the CSS where I gave this editlinks span a float right property. And that's all I need for my edit page, let's go ahead and try it in a browser. And of course I have to sign in first, and as you can see once I log in as a user I see editlinks for my bookmarks. Clicking on one of them takes me to the familiar form where I can change the URL and the description. Doing so we return to our own page and see that the bookmark has changed. Now of course I also checked that I cannot edit another user's bookmark even if I wanted to. As an exercise you might want to try and see if you can think of a way to trigger the 403 error page. I leave that as an exercise to you.

Review: Jinja Filters and Populating Objects

So you've seen a couple of new things. First of all we've seen Jinja2 filters, which basically are functions that are available in the template context, and that you can apply in a variable expression. And you can use these to transform the content of your variables when you output them in your template. Now one of the nice things about filters is that they can be chained using the pipe symbol. But first let me show you a simple example, we've seen this in the demo as well, here we want to output the variable title but when it gets longer than 10 characters we want to truncate it. And Jinja has a default filter for that called truncate. And as you can see it looks very much like we're simply calling a function, but we're applying that to the title using the pipe symbol. So this means truncate takes title as its input, truncates it at 10 characters, and then

returns the truncated title as its output. Now as I said you can chain filters as well. So this example here will do multiple filters from left to right. So here first we take title, then we call the trim filter, which strips all white space from the beginning and the end of the title, then we transform it to upper case with the upper filter, and then again we truncate it. So each time we use a pipe symbol, that will cause a filter to take whatever it is at its left-hand side, as input and pass its output the right-hand side. Now Jinja provides a large number of built-in filters, so many common operations you would want to do on your data before you display it in your HTML can be done with filters. And you can even add custom filters, and there's a special flask decorator called app. template_filter. Now apart from all the Jinja built-in filters, there's also one special filter added by Flask, and that called tojson. And that will convert data you pass into it to JavaScript JSON adaptation. And we will see one instance of using this to generate JavaScript with our Jinja templates. We also saw two new directives, include and with. The include directive will take another template, render it, and the output will be included in our current template. So in this case we have a template that includes other. html, and other. html will be rendered, and whatever HTML that returns will be included in our current template. This is very useful when you have templates that generate parts of HTML pages, so HTML fragments. A nice thing is that the template you include also has access to context variables from the current context. But as we saw in the demo, the names that in include template expects to be in the template context might not match the name of the variables in the current templates. And one possibility to deal with this is the width directive, this will create a new scope with a variable in it. So in this example we create a new scope, with a new variable called x with the value 42. But in the demo showed, that you can of course also use this directive, to copy the value from a variable into a variable with another name. So this can be very nice if you combine it with an include directive with a template that expects certain variables with certain names to be present in the template context. We also saw two new ways to populate form and model instances. First we saw a new way to call the constructor for one of our WD form objects. Here we use the obj arguments to populate a form with data from another object. And that object could, for example, come from the database so it could be a model instance. This is very nice to pre-populate the form, like an edit for, with data that's already in the database. And a nice feature of WD Forms is that if there is an request data present, that will take precedence. So that will be used instead of the obj arguments. And that means that if the user has filled in the form hat data will be used to fill the form object. But if the user has not filled in anything yet we pre-populate the form with the given objects. So this fills WD Forms form object from a model instance, but what if we want to do it the other way around? Well, we've seen that as well. WD Forms form objects have a method called populate_object, and we can pass that another objects, again that could for example be a model

instance. And what this method does is it simple copies all attributes from the form to that object. So you can use this as shorthand to simply copy everything in the form into your model. One of the nice things, of course, about this is that when you're model changes, or your form changes, you won't have to update all your views and make sure you copy every field because this method will make sure all the fields are copied, and you only have to use one line of code.

Demo: Database Migrations With Flask-Migrate

So we've added an edit page to our project, and now I want to be able to add a text to my bookmarks. I'm going to have to add a new model class and a many-to-may relation, which means we'll be adding two new database tables. Now when you're working on project and your database model changes a lot, this can get really annoying when you're working with version control and you want to be able to switch between versions and branches of your code. Every time you need to drop and rebuild the database. But there's a good solution for this and it's called automated database migration. For this we use flask extension called flask-migrate. Let me show you how this works. Now because I don't want to overcomplicate things I'm going to start with removing our current database file so we can start from scratch. Next I'm going to install the flask-migrate extension. And of course the command for that is pip install flask-migrate. And as always we will need to do some set up for this extension, but this time we don't need any integration with our app. Instead I'm adding some code to our manage-scripts. I'm importing Migrate and MigrateCommand, and I call the migrate constructor with our app and db. Next we tell the flask script manger to add the migrate command under the name db. This gives us access to the database access commands with a prefix of db. Now you should also notice that I'm importing both bookmark classes, user and bookmark, at the top here even though I don' use bookmark in the code here, but these classes have to be available, they have to be loaded so that flask-migrate can detect them, and generate the code to generate the database tables for them. So again I added the migrate command class to our manager with a prefix db. What does that mean? Going back to the command line, now when I say Python manage. pydb I get a list of supported operations. And the first step here is something you only do once for your project, and that is to initialize the migrations with the init command, so Python manay. py db init. And this tells you it created a migrations directory with some files in it. This is all we have to do for setup, now we can add our first migration. Remember I removed our database file so currently we don't have any database at all. Now our first migration will be the migration from no database at all to an initial database with our current bookmark and user class. And I can add that migration with the migrate command like this, manage. py db migrate, and then I can add the name of this

migration with the -m switch. You don't have to do this, but I think it's a good practice. So now I'm setting up our first migration with the name initial. Flask-migrate now detects our model classes, and it generates a migration script that will enable us to automatically regenerate this version of the database structure. We can of course open this file, and this is a Python file containing two functions, upgrade and downgrade. Upgrade will take you to this version of the database from the previous version, and downgrade works the other way around. So this works more or less like a change set in version control, it contains the changes we did to our database. Since we just started over with an empty database, this script contains code to generate our user and bookmark tables. Now our database at the moment is still empty and so we can ask flask-migrate to upgrade it to the latest version by saying Python manage. py db upgrade. And this will make it run the actual SQL code to generate our tables. So basically, now it runs the python scripts that it just generated in the previous step. Opening the database we see that our bookmark and user tables are back, but there's also now a table called alembic_version, which holds the version number of this version of the database so that the migration engine that's used by flask-migrate, which is called alembic, can identify it. So if we look at the data inside this table we see this number which corresponds with the name of our migration script. So now you know a little bit about database migrations, let's go ahead and add some new model classes.

Demo: A Many to Many Relation

So to be able to add text, my bookmarks of course, I added a new model class called Tag, and it's quite simple. It has an id and a name, the id of course is the primary_key, and names cannot be null, they have to be unique, and there's an index on the name. Now a bookmark can have any number of tags, and a tag can be associated with any number of bookmarks, so we need a many-to-many relation. to do this we setup a junction table containing foreign keys to both models. I do this here at the top of the file. I use the db. table class instead of creating a model class for this because I really don't need a model class for this table. You see my models are bookmarks and tags, and those are the things I want to be reasoning about in my code. So this table we define as a low-level table and we'll never be using it directly. So as you can see in the db. table call I add two columns, tag_id and bookmark_id, and both of them are integers, and their foreign keys either to tag. id or bookmark. id. In the bookmark class I add an attribute called tags, and that's the relationship with the tag class. I need to use the class name tag here as a string because the tag class is defined below, at the end of the file so at this point the name of the tag class is not known to the interpreter yet. The second argument to the relationship call is the argument's secondary. And that tells the relationship to use out junction table, called tags, which is the table I

defined above. We also define a backref called bookmarks, which will add an attribute called bookmarks to the other side of the relation. So each tag will get a bookmarks attribute containing a list of the associated bookmarks. We define this to be loaded dynamically because there may be a large number of bookmarks associated with each tag, and we don't want to retrieve all of those every time we look at a tag. This side of the relationship on the bookmark class is not lazy though. So when we retrieve a bookmark we also immediately retrieve all associated tag objects, because normally we'll want to see those too, and in that case we don't want to generate SQL queries to load them later. Now you may have noticed that I prefixed the name tags here with an underscore, and that is because I don't want to access this list directly from other classes. We'll see why that is soon. For now let's go ahead and create my two new database tables, the junction table and the tag model table, and let's see how we can do that using flask-migrate. First I have to take another look at mamange. py, and as you can see I'm now importing the tag class above here as well because, as I said previously, these have to be loaded here for flask-migrate to detect them. Something else I did is I renamed the first function here from initdb to insert_data because it doesn't create any tables anymore, we now have flask-migrate for that. So from now on all I use this for is as a utility function to insert some test data into the database. So it has some code here to add a new user and I wrote a little internal function that adds new bookmarks, and then I call that a number of time sot add number of bookmarks. I also have a for loop that add a list of tags, and if I scroll right a bit you can see that it's quite a list and that I also add all these tags to each bookmark. But that's just a utility for myself, the important thing here is that I imported the tag class at the top of the file, and then going back to the command line I can tell flask-migrate to generate a new migration with the command Python manage. py db migrate, and I'll add a name as well, with -m tags. So my new migration is called tags. And as you can see two new tables are detected, as well as the index on the tag table. The next step of course is to run this migration, and again I can do that with the db upgrade command. So looking at the database again, now we see our two new tables. This is a junction table that contains two foreign keys, and here we have the tag table with an id and a name. Now if I need to go back to an earlier version of my code, I can do so by saying Python manage. py db downgrade, which will take us back to the previous version of the database, or I can add the name of a tag with a --tag initial switch. I'm going to make this a little bit wider here, you can see I use two minus signs here. So when I pass this a name flask-migrate will run all downgrade functions one-by-one until we're back at the migration called initial. So after doing this you see now our tag table and our junction table are gone again. And if I want to get back to the current version, again, I can say Python manage. py db upgrade and it will take us back to our latest version. And again I can also add the name of the migration I want to upgrade to with --tag tags. And again, now our bookmark_tag and tag tables are back.

So this is how flask-migrate very easily let's you switch between different version of your database. And this is just the tip of the iceberg of all you can do with it, so let's do a short review before we go on and add tagging to our user interface.

Review: Flask-Migrate

So we've seen we can use flask-migrate for database migrations and, as always, we start with installing it by using pip install flask-migrate. Then we have to set it up in manage. py and this is different from most other extensions in that we don't have to integrate is with our app, all we do is add the migrate command class to the manager in our manage. py file. So the first line here imports the Migrate class and the MigrateCommand class. Then on the second line we call the migrate constructor and we pass it our app and our database, and on the third line we enable the command line interface by passing the MigrateCommand class to the add_command function. And the first argument there is a string, in this case I used db which will be the prefix for all these commands. So we say dbinit, dbupgrade, etc, etc. So very a short overview of the commands that this offers you first there's dbinit, which initializes you migrations, you only have to run that once for your project and it will set up everything in the migrations directory. Then there's a dbmigrate command, which adds a migration, so every time you change something in your models you can run this to generate a Python script that will update your database tables. And with the -m switch you can give your migration a name. Then there's a dbupgrade command, which will upgrade your database to the latest version. This will actually run the migration script that was generated by the migrate command. So the migrate command only generates a Python script, and this will run it. Now of course you have to realize that flask-migrate is a very complete solution, and it does many many more things than just this.

Demo: Forms and Views for Tagging

So let's make it possible to actually store our tags in the database. Let's begin by adding a field to our form for editing bookmarks. So here in forms. py I added the string field called tags to the bookmark form class. We're expecting a comma separated list of tags, and to check that that is what we get I'm using a regular expression here that accepts letters, numbers, commas, and spaces. So when the user submits the form this input field will contain some combination of letters, numbers, commas, and spaces. And down here in the validate method, then I do some clean up. I remove empty and duplicate tags from the input without showing any errors. So first I split the input data at every comma, and then from each part after that split I strip all white space.

Then I remove all empty strings form the list and I put everything that remains in a set, which removes all duplicates. Then I use the join function to put the list of non-empty, non-duplicate tags into a comma-separate string again. So basically, now we have a validate function that takes the input from the text field, and cleans it up, and makes sure there will be no duplicates or empty tags left. And that means we now have a form that allows us to input and edit lists of tags. Let's add the HTML for that. So here, in the bookmark form template, we simply render this field like all other fields, with the render_field macro. Now of course, after editing a bookmark and adding text we also want them to show up, so in the bookmark list template I now add a for loop that loops over all the text for a bookmark and renders them. Actually it renders a link for each tag, but these links don't do anything yet. And then finally there's some simple things we need to change in views. py as well. Of course we need to import tag class at the top, and then in the add bookmark view we need to take the list of tag and pass them to the bookmark, like I did here. Now we don't have to the same here in the edit bookmark view because that one uses populate objects and it will copy the text automatically. So what were' passing to the model here is just a string containing a comma separated list of words because that's what the HTML input receives. So how do we convert that into a list of tag model objects? Well, let's look at the models. So let me explain the reason why I prefixed the tags field of the bookmark class with an underscore. You see in the view in the form we handle the list of tags as a comma separated string. So it's convenient to make a tags property that provides a list of strings as well. So here is the property tags and the getter takes the contents of the _tags list, which holds actual tag model objects. Then it takes the name from each and joins that into a string. So when we ask for the value of the tags property on a bookmark we get a string with a list of tag names. So what about the setter? Well, when we pass a string with the list of tags to be set to this property, we need to find out for each of those tags whether it already exists in the database. If it doesn't we need to insert a new tag, in the tag table, and then add the new model object to the tag list for this bookmark. If it does exist we can simply retrieve it, and again put it in the lists. So what I did is I created a function called get_or_create that takes the name of a tag and returns a tag model instance by either creating or retrieving a tag with that name. We can now to a for loop over all the words in the string we received and then we call get_or_create on each of these words. The resulting list is a list of tag model objects, and we can assign that to the _tags attribute. Assigning a list of tag objects is all we have to do, and SQLAlchemy will take it from there and create all the relevant rows in the database. Then, down below in the tag model, here's the get_or_create method and this is what it looks like. It's a static method and it uses a try statement to retrieve a tag with a given name. And in case that throws an exception, which means that such a tag does not exist, we create a new tag and return that. So this way we can convert strings to tags and back. So

when we test this in the browser we now see that I already added some bookmarks and tags, and you see the list of tags for each bookmark, and we can click on them, but nothing happens yet. But then I can click the edit link here, and now we can edit the list of tags for this bookmark as a comma-separated string. So for example, I can add the tag test here, and now the test tag has been added to this bookmark.

Demo: Integrating JavaScript

But of course this isn't really user friendly. I'd prefer to have a list of suggestions that I can click and we can do that with the help of some JavaScript. I'm going to use a library called select2, which you can download from GitHub. Now inside this zip file I just downloaded there's a lot of files, but I'm only going to copy a couple of them into my projects. Mainly there' select2. min. js, which I'm copying into the js vendor directory. And into the static CSS directory I'm going to copy select2's spinner. gif, selct2. css, select2. png, and select2x2. png. And those are all the files we need to make our tagging more user friendly. Now select2 is a simple JavaScript library with no flask integration at all. But that doesn't really matter, I can use it inside my templates anyway. Let's start with looking at base. html. I only changed two things here. I added a block called styles that will contain the references to my style sheets. And below, I also added block scripts providing a place for other pages to add JavaScripts. Now is I look at my bookmark form, here at the top I overwrite the styles block. Inside the block I start with calling a function called super, this will take the contents from the styles block in the parent template and include them here. So I'm including the same style sheets here as the base template, but I'm also adding another style sheet and that is the CSS file form the select2 JavaScript library. Next to the tag field here I added the id attribute tags, and that's so that we can look up this input by id with jQuery. And then at the bottom of the file I'm overriding the scripts block with my own page specific scripts. I'm starting with including the select2. min. js JavaScript file to load the select2 library. Then in a second script tag I add some JavaScript. Now in case you're not familiar with JavaScript or jQuery don't worry, this is pretty standard stuff and if you don't understand that's okay. I create a jQuery document. ready function, which contains some code that will be run after the page has loaded. Then we have a Jinja expression here. You should realize that the Jinja template is executed on the server. So the part here between curly braces gets executed first by Jinja, and it gets replaced by a string. The resulting page gets sent to the browser and there the JavaScript gets executed. So the JavaScript interpreter never sees this expression in curly braces, but only its results. Now this expression works as follows, it calls the all_tags method, which we will see in a moment, which returns a list of all tags in the database. Then we apply a filter called map, which takes the name

attribute of each, and returns a generator expression that we transform to a list with the list filter. Then we pass that list to the tojson filter so that it gets transformed into a valid JavaScript list expression. Finally we have to call the safe filter because Jinja will escape all special characters in variable expressions by default, and that would result in a non-valid JavaScript string. So we call safe to tell Jinja that we trust this expression and it doesn't have to escape anything here. So this shows you how to apply multiple filters at once. Transforming a list of tags into a list of names, and from there to a JSON expression, and finally using the safe filter. This results in a string that is a valid JavaScript list of names, and we assign that to a JavaScript variable called tags. In the next line I look up our tags input by id with this jQuery expression, and call the select2 method on that. This select2 method is what the select2 library provides to make our tagging input more user friendly. We pass it a dictionary for configuration, and this dictionary contains two values. The list of tag names we just created, and a list of valid separators between tags, which we set to spaces and commas. This means that when using this input the user can use spaces or commas between tags. Select2 will convert this to comma separated list automatically. So let's see how this works. This time when I click the edit link we see that the tags field now contains a button for every tag. These buttons are created by select2, and when I try to add something new, select2 shows me a list of all available tags. So this list of tags is what we pass to select2 in the JavaScript code. So if we check out our HTML source for a moment, here you see the actual JSON list that was created by our Jinja2 expression with all these filters. So as I said, on the client-side in the HTML you're never going to see the Jinja2 expression you're only going to see the JSON string that is the result from that. So this means we can add and remove tags with a simple click, and select2 will prevent us from accidentally adding the same tag twice. So this doesn't just look better, it works better as well.

Demo: A Context Processor

Now there's one thing I didn't explain yet and that's where this all_tags function comes from. I'm not passing it in form the view like other variables in the template context, but instead I'm making this available as a template context global so that it's always present in each template. I added this in views. py. Now to make something globally accessible in the template context we use the app. context_processor decorator. And you can use this to decorate a function that does only one thing and that is to return a dict. To this dict I add a reference to the static method, all from the tag class, and now because we annotated this function with app. context_processor this methods will be executed every time before a template is rendered. And the content of the dict that we return is then added to the template context. So this will give every template access to

the old tags method, and also everything else that we might add to this dicts. Now I could have added parens here, like this, but that would mean we actually run the database query before every template, even if we don't show any text in that template. So instead I chose to add a reference to the function, and in the template we cn call the function if necessary. So now we have this, I can show a list of all these tags, and make them clickable to show all bookmarks for a tag. So here I added a view called tag, which takes the name of a tag as an argument. We try to retrieve a tag with that name with tag. query. filter_by and then call first_or_404 so that we show a 404 message if such a tag doesn't exist. Next we call render template to render tag. html, and pass it the tag model object we just got from a database. So let's look at tag. html and this is tag. html, it's basically a copy of the user. html template, but it shows the bookmarks for a tag instead of for a user. So it says Bookmarks tagged followed by the tag instead of bookmarks for user. And we say tag. bookmarks instead of users. bookmarks, but structurally it's exactly the same as the user template. Now of course we want to be able to visit the page for a tag so let's add some links, and here in base. html I'm now changing the contents of the sidebar so that it will show a list of all available tags. It's quite simple, we call all tags and loop over it, and for each of them insert a link to the page for that tag. And in the bookmark list it's exactly the same, we were already showing the tags, but I'm adding a real link to the tag page now. So this is what the sidebar looks like now. We can click on any tag and see the bookmarks for that tag. This works in the sidebar, but also on the bookmark itself. Now to make sure the tags in the sidebar are visible I have to give them another color than links on the rest of the page. So let me show you the CSS for this. And here are some simple CSS rules for giving links in the sidebar a lighter color.

Demo: Context Processors and Super()

So we just saw two things about Jinja templates we haven't seen before. First of all we saw how we can inject variables into the template context without having to pass them from a view. What you do is you define a function that returns dicts, and that function will then be run before each template is loaded. The content of the dict you return are then added to the template context so that basically means they will be globally available to every template. And to do this you annotate your function with app. context_processor and that will make sure flask picks it up and runs it before each template. Then another small feature we saw is that you cn call a function called super inside a block directive. So when you're overriding a block inside your Jinja template, and this will insert the contents of the parent block. We saw that this can be very handy, for example, when you have a styles block where you define CSS and you want to overwrite the styles, but you

still want to use the CSS from the parent SOL, and in this case you'll first call super to include all the CSS that is defined in the parent and then you add your own.

Demo: A Delete Page

Now the last thing I want to add is the possibility for the user to delete a bookmark. So what I did is I added view called delete, which takes the database id of a bookmark. Of course, again, it does get_or_404, and checks that the current user is the owner of the bookmark and if all of that checks out we delete the bookmark with a call to db. session. delete. And of course we have to commit that as well. And as always, we flash a confirmation to the user and redirect them to their own page. Now please note that if the recurse method is not post here we first render a template confirm_delete to confirm deletion. So that's a very simple page that asks the user to confirm deletion of the bookmark before actually doing a post to this view and really doing the delete on the database. So then, if we look at confirm delete, that's a very simple page as you can see here, and it contains a simple form with a submit button that will do a post. And we also have another button that's basically a link back to our user's page. Now as you can see I also include a new template bookmark. html and that simply contains the HTML for a single bookmark, so we can show the single bookmark that we're deleting here inside the delete page. So basically what I did here in bookmark_list is that I took everything inside a list item for single bookmark and I moved that into another template. So here inside bookmark_list, for every bookmark I'm listing I'm including bookmark. html, and bookmark. html is a new template that contains div with information for one single bookmark. That way we can include this from the delete confirmation page as well and show only the HTML for a single bookmark there. In main. css I'd had to do a very simple change here and change the selector for this role here from a selector that used to select a list item to a selector that selects anything that has the class. bookmark. And that's because on the delete page a bookmark doesn't show up on a list, but on its own in its own div. Finally here in bookmark. html I added one simple line, and that is the link to the delete page. And as you can see that's right next to the link to the edit page. Very well, if we look at this in the browser, now we see both an edit and a delete link on every bookmark and if we click delete we go to a very simple page that shows the bookmark in question with the question, are you sure you want to delete this bookmark? And if I click yes, delete we get a confirmation that the bookmark is deleted and it's not in the database anymore. And that means that right now we have a more or less complete functionality where we can add, edit, and delete our bookmarks. So let's do a final short review.

Resources and Summary

So let me show you some resources where you can read more about all the things we learned this module. First of all here's the link to the flask-migration homepage, which is a flask extension that uses the alembic database migration tool. Then we also saw two JavaScript libraries. One is moment. js, which we included using the flask-moment extension. And then there's select2, which we used to add a nice user interface for the text. And there's no extension for that so this link goes straight to the page for the JavaScript library itself. Finally, here's a link to the Jinja documentation where you can read up on the list of all built-in Jinja filters. And that's it for this module. What have we seen? Well we started by fixing the layout of our bookmark list. In doing this we learned a little more about Jinja templates, mainly about filters and how to include templates that generate only parts of HTML into other templates. Then we added some functionality to edit bookmarks. And along the way we saw how to populate model objects from forms, and forms from models. From there we went on to add tagging to our projects. And to do that we had to add a many-to-many relation between the tag and bookmark classes. This gave us a good reason to investigate flask-migrate to do database migrations. Finally, we had to add a nice user interface for the tags, and I showed you how to do this with a JavaScript library and how to use a context processor to make a function globally available in all templates. And the last step in the module was adding a delete page.

When Your App Grows

Introduction

Hi, my name is Reindert-Jan Ekker and in this last module I'll discuss various topics that become more and more important as your app grows larger. So with the last couple of modules we've seen our tiny app slowly becoming larger and larger. And as it grows things like debugging become more complicated and we'll want to add unit testing for better quality control. We'll start with adding a nice tool to make debugging easier, but to implement unit testing in a convenient way we need to do some more work. First, we'll have to split our application into modules called blueprints. That adds some nice structure to the project, and it also paves the way for two other structural changes, namely creating our app object with a factory function, which as we'll see is necessary to dynamically configure our project according to different scenarios. And this means we'll be able to, for example, use a separate database file for unit testing or to turn debug mode

on and off depending on the context. So let's get started with the first small step, which is adding a debug toolbar.

Demo: The Flask Debug Toolbar

So let's start with simple task and that is installing an extension which will let us conveniently debug our application. I'm going to install it by saying pip install flask-debugtoolbar. And of course here in init. py I'm going to add some code to enable it. We import the debugtoolbar extension class, and call the constructor, and pass it our app. Now also note that I'm setting the debug configuration option to true here. If we don't do that the toolbar will not be shown, but since I'm putting flask in debug mode here, when we start our app now we see something new on the right side here. And this is the flask-debugtoolbar and it will give us access to a wealth of information about our app. For example you can see how much time it costs to deliver this page. And you can see all the HTTP headers that were sent in the request, you can also see all the data that was sent with the request, like the cookies and the session variables that are stored on the server side. Furthermore you can see the complete configuration of your flask application. So for example, the debug setting is here, but you can also see the secrete key that we use to securely generate a session cookie, for example, and another example here is the URI for the connection with our database. Now something that I find comes in handy a lot is the templates tab here, which shows you all the context variables that are present inside the template. So when this template, which is the index template, was rendered these are the things that were present inside the template context. Now you can probably understand why this makes debugging your templates a lot easier. Then finally let's focus on the SQLAlchemy tab, and that shows all the SQL queries that were done to present us this page. Now strangely enough I see that we used nine queries to make this page, but it's a very simple page and I didn't really expect it to do that. And looking at the list here you can see that line 31 in models. py causes five SQL queries. so let's check out why that is, and apparently this is where it happens. So when we ask a bookmark for its tags list it has to get the name for every tag and that results in separate SQL queries for all these different tags. Now one way to fix this is to change our relationship. So here on the tags relationship I can change the behavior of the bookmark side of the relationship, and I want to do that in such a way that when I retrieve a bookmark all the tag models that are associated with that are joined in the query immediately. To do that I say lazy is joined here. And now, when I load the page again, now we see to deliver the index page we only need four queries. And you can see that now in views. py, when I retrieve the bookmark we get this huge query, which retries all the tags associated with the bookmark at once. Now I think this is pretty reasonable because

normally when we load a bookmark we will want to see all the tags as well. So here you see how the debug toolbar really makes life easier for us. Now there's also two tabs that I'm not going to use right now, one of them is logging, which is a subject I'm not going to cover in this course, and the other is a profiler. So the debug toolbar actually comes with a built-in profiler, but you'll have to enable it to actually use it. So that's the debug toolbar for you, let's move on.

Demo: A Blueprint

Now our thermos application has grown a lot since we started it and I really find that, for example, our views. py file is getting quite large and unwieldy, so I'd like to break this up into multiple components. Well, actually flask has a mechanism for that and it's called a blueprint. If you're familiar with Jengo, blueprints are quite similar to Jengo apps Now to illustrate the use of blueprints for you I moved all authentication-related views into their own blueprint. To do so I added a new Python package called auth inside my thermos app. Now because it's a Python package it has an init. py file. I also added a views. py file, which holds the authentication-related views, login, logout, and signup. And with the signup view comes the signup form, so there's a forms. py inside the auth blueprint too. Now let's start by login at the init. py file. And basically all this does is it creates an actual blueprint object and we give it a name, auth, and we pass it the name of the current Python module. We simply import the blueprint class from flask and then we import the views we want to be a part of this blueprint. Now if we look at views. py inside this blueprint, there's a couple of interesting things here. First of all, of course, this only imports what we need so the list of imports here is a lot shorter than the list of imports in our original views file. Then you should notice that I'm not importing app anywhere. I am importing the d b object for the parent package, which is thermos, but I'm not importing the app object here. Instead I'm saying from. import auth. So form the current package I'm importing our auth object, which his the blueprint object. And inside the blueprint that kind of takes the place of the app object. so the main difference here is that where I register the views with the route decorator. I don't say @app. route, I now say @auth. route. So what I do is I register these views on the blueprint, not on the app yet. And of course, notice here, that I'm importing the user model form the parent package, and I'm importing the LoginForm and the SignupForm from the. forms module. So that's the forms module inside the blueprint. Now if you look at the init file for the app itself, here you see that I'm now importing the blueprint form the auth package, and then I'm registering with app. register_blueprint. And then first I pass it the blueprint object, and then I say URL prefix is /auth. And now what happens is that all the view inside the blueprint will be registered on the app, but their URLs will be prefixed with /auth. And to demo that for you, now when I click sign in you can

see here in the URL bar that the URL for the login form is now /auth/login. Now all of this also has an effect on how we build our URLs. Remember thte se links here at the top are actually generated with a call to url_for. So if we open the template, for example, here in base. html you can now see that where I call url_for for an auth-related view, I have to prefix the name of the view with the name of the blueprint. If I don't do that, for example, I would still say url_for logout, then this template would give an error. And it will tell me that there's no such view called logout. So basically views inside blueprints are namespaced, and you have to prefix their names with the name of the blueprint. And of course I have to change this for every occurrence of url_for that references a view that's now inside a blueprint. So in this case I didn't only have to change base. html, I also had to fix some links in index. html. Finally a last look at the init file, I also had to change the configuration of the login_manager because I tell it what the login_view is, and of course, the name of the login_view now isn't just login it's auth. login.

Review: Blueprints

So we just saw how I separated out the authentication functionality of our site, and moved that into a blueprint called auth. Now in the flask documentation such a blueprint is described as follows. A blueprint object works similarly to a flask application object, but it's not actually an application. Rather it's a blueprint of how to construct or extend an application. Now a blueprint may provide view functions, or static files, or its own templates, or a combination of all of those, etc, etc. So you can use this to break up a growing application into modules with clearly defined chunks of functionality. This makes your code more clear and maintainable. Or you can write reusable blueprints that you can reuse between apps. You might even want to create a blueprint as part of an extension. Now the blueprint we just saw is in its own package. Actually this is not necessary because you could create an entire blueprint inside a single Python file, but it's really a best practice to use packages and there's no really good reason not to. So my advice to you would be if you create a blueprint do it in its own package. And don't forget, a Python package always has to have an init file. So how do we create a blueprint? Well, of course, a blueprint is in a sense just an object. So we import the blueprint class, and then we create an instance. We pass it a name, in this case auth, and the name of the current Python module. Then, if necessary, we import any modules that are part of the blueprints package. So in our case that's simply the views module inside the auth package. Note that I'm not importing the forms here because the views file imports the forms by itself. Now then we also need to register the blueprint as part of our application. So in the application's init file I import the blueprint object we created form the auth package. Now to prevent namespace collisions I rename it to auth_blueprint. Next, I call app.

register_blueprint and pass it that sane blueprint instance. The second argument of this call specifies the URL prefix that will be inserted before all the URLs for the views inside the blueprint. So for example, the login view will now be housed at the /auth/login. One important difference when writing views within blueprint is so that we use the route decorator from the blueprint, not from the app object. So we say @auth. route instead of @app. route. You should also try to remember that when you reference a view from a blueprint, which you do when you call url_for you need to prefix the name of the view with the name of the blueprint. So instead of calling url_for with the name login, we now all it with auth. login. Now if you're referencing a local view from within the blueprint itself you can leave out the blueprint name, but you do have to start with a dot. So in that case you'll say. login.

Demo: More Blueprints

So I went along and broke up the app even further so that now all views are inside of blueprints and this is necessary for our next step, and we'll see that soon. But first let's take a short look at the blueprints. There's a bookmarks blueprint now, if we look at the init file this looks pretty much exactly like the one for the auth bookmark. And the views it contains are the add bookmark view, the edit view, the delete view, the user's bookmark view, and the bookmark list for a tag. Apart from this it's pretty much the same as the auth blueprint. You should note that this time I register all the views with @bookmarks. route because bookmarks is now the name of the current blueprint. Another thing to note is that, for example, in the delete view I now use. user in the call to url_for when I redirect. This is because this view is inside the same bookmarks blueprint and so we don't have to prefix is with the bookmarks name, we just can use a dot because it's a local view. Similarly I now have a main blueprint, and again, the init file looks the same. And here we don't have a lot of views. Basically I use this for the index page and the error handlers. Now normally you register an error handler on the app directly, and you can't actually register an error handler on a blueprint, so that's why the blueprint class has a function called app_errorhandler. And once you register the blueprint this error handler will be registered on the app. And same goes when we scroll a bit down for the context processor. Again, you can't register context_processor on the blueprint, but you can register it on the app. So right now we tell the main blueprint that we have a context_processor that we want to register on the app once this blueprint is registered. Now of course when breaking up our application like this I had to fix all the calls to url_for in all the views, and all the templates. Of course I could go over that with you, but I think the idea is clear, every reference to a view that's now in the main blueprint get s a prefix of main, and a view that's now I the bookmarks blueprint gets a prefix of bookmarks, etc, etc. Now a

very important effect of all of this is that we have no Python modules anymore that depend on the existence of an app object. And remember that before we needed to use the app. route decorator to create view functions. And for that it was necessary that the app object existed as soon as the views module was loaded. But with the blueprints we don't have that dependency anymore, and that means I can now delay the creation of my app objects. And let's see a little bit more about that.

Demo: An App Factory and Dynamic Configuration

So here's the thermos/__init__. py file, and as you can see I've completely changed the structure of this file. I now instantiate all extensions before I even create my app instance. And that means I count past the app instance to these constructors anymore So now I create instances of the login_manager and the db object that don't even have access to the app yet. All flask extensions actually support this way of creating them. And then, when I scroll down, you can now see I created the app itself in an application factory function called create_app. It starts by creating the app, and then calls init_app on all extension objects. Now all extensions are in the same state as they will be if I had passed the app as an argument to the constructor. Next I register all my blueprints. Note how import the blueprints inside the create_app function as well. Well, this is because blueprints may depend on things like the db object, or the login_manager when loading. So I want to make sure that I don't load these modules before I have set up the entire application, including the extensions. Now you may wonder why I would go through all this trouble when everything was working correctly without an application factory. Well that's because I want to be able to unit test my code. And to do that I want to be able to start the application with a testing configuration, which is slightly different from the normal development configuration. And this app factory function makes exactly that a possibility. So as you see here, the create_app function takes an argument config_name. Now app. config is pretty much a normal dictionary with some added functions like the ability to load configuration from a file or, as I'm doing here, from an object with the app. config from_object function. Config by name is a dictionary from a new module config. py, and when I open that you see I set up a little class hierarchy with config being the base class that contains my secret key. And sets the value debug to false by default. The development config class inherits from that and sets debug to true, and set the path to my database. The testing config sets testing to true, points SQLAlchemy to a different database file, and sets the WTF_CSRF_ENABLED option to false. Now that's a WD forms option that we need to disable to allow unit testing forms. Finally I have a little dict called config_by_name that maps each of the names, def, test, and prod, to the appropriate configuration. Now back in init. py you

should now understand that passing the create_app function, one of the names dev, test, or prod, will load the correct configuration. Because basically we look up the object with that name from the dictionary config_by_name. Now finally, in manage. py, I've cleaned this file up a bit and now it looks very simple. I start with calling create_app, to determine which configuration to load I check for an environment variable called THERMOS_ENV. If it isn't set we use the development settings and otherwise we use which ever value is in the environment variable. Then we create the manager and, as always, add the migration commands. So now on the command line it can set the THERMOS_ENV variable, let's set it to dev first, and now when I run the server you can see the toolbar is loaded, and that means that the debug flag is set. But when I stop the server and change the value of THERMOS_ENV to, let's say, production and of course restart the server, and now we see the same page without the debug toolbar because in a production setting the debug flag isn't set. So let's go on an view a little example of a unit test.

Unit Testing

So now we've set up dynamic configuration and an app factory, and we've split up the entire application into blueprints, we're ready to run some unit tests. And for this I added a test package, it contains in it an empty init. py file, and here are the unit tests. Now as you cn see I'm getting an error here because I'm importing something, and that's because I first have to install a new extension. Now you can write unit tests for flask with a vanilla Python unit test package, but there's a very nice extension that takes care of all the setup you have to do, and that's called flask testing. And as you can see at the top here, I'm importing from that extension, but it gives an error because I didn't install the extension yet. So let's start by doing that. The extension is called flask-testing so when I do pip install flask-testing, and I run it, now the error has disappeared. Now because I don't like the standard behavior of the unit test package when discovering unit tests, I like to use another package called nose, so let's install that as well. Very well. Now let's go over the unit test. My ThermosTestCase class inherits from TestCase from the flask-testing extension. And one of the things that provides is a create_app function that we should override. And of course what I do there is I call the create_app function from my on module, and I pass it test as the name for the configuration. Then there's the setup function and that gets run before the unit tests. Basically what I do here is I create a database and insert a user in a bookmark, but I'm also retrieving an instance of something called a test client. And this will let us make requests to our application, and it keeps track of everything like cookies, etc, so we can do things like logging in. And that's exactly what I'm doing at the end of the set up function. I'm logging myself in as the user I just created. Now this is important because I'm going to test that added bookmark view

and we have to be logged in to get access to that view. Now there's also a tear down function which is run at the end of testing, and it simple clears out the database. Now remember that I set up a different database path in testing. So this isn't going to affect the database I use in development or production. So let's take a look at my actual test case. I called it test_delete_all_tags and it calls the self. client. post function. Of course this does a post request to my application. Its first argument is going to be the URL, and I'm going to use url_for to ask for the bookmarks form with a bookmark id of 1 because there's only one bookmark in my database so the id is going to be 1. Then I pass some form data. For that I use the data argument and I pass it a dictionary with a URL and an empty tags list. So this should remove all the tags from my bookmark. Finally I tell the post function to follow redirects. That's because after editing I get sent to another page. And I'm going to check whether I'm going to get a 200 okay after that redirect. So here, first, I'm doing an assert that tests whether the status code, after all of this has been handled, is a 200 okay. Next I retrieve my bookmark form the database, and I'm going to assert that there's no tags left on that bookmark. So let's run this test. Now the nice thing about nose is that it discovers all my tests automatically. So it can run all of them by saying nosetests. And as you can see my assertion fails, apparently there's still some tags left on the bookmark and fortunately I know what causes this. Let me show you. In models. py here we have the setter for the tags property that tests whether there's anything in the string I passed there. And of course, what I just did is set that string to empty, so the test is going to fail and nothing will be executed, so we can simply fix this, like this, where I say that when I get an empty string for the tags property the underlying list of tag objects should be an empty list. And this will cause SQLAlchemy to remove all foreign key references to the tags table for this bookmark from the junction table. In other words it will remove all tags from this bookmark. So after fixing this I can rerun my tests and now the tests pass.

Resources and Summary

And that brings us to the end of this course. So let's go over our final list of resources. Let's start with the tools we saw in this module. Of course, there's the flask debug toolbar, which is a really nice project you should check out. Then there's flask-testing, which makes unit testing with flask a lot easier. And then there's nose, which has nothing to do specifically to do with flask, but it's just a really good test runner for Python. Then let's see some of the new flask development techniques. First there's blueprints, and here's the link to the documentation about them. There's also a page about app factories, and finally here's a link that talks about best practices for configuring your flask application. And this is final summary of this course. What have we seen in

this module? Well, we started by installing the flask debug toolbar, which is a very powerful tool that gives you a lot of introspection into the inner workings of your application. Then we split up our entire application into blueprints, and after that we didn't have a single module that had a load time dependency on the app object. And that means we can delay the creation of our app, and use an app factory function. And that in turn gives us the possibility to do dynamic configuration, and reads the configuration from a file or objects. Once you've done that it's very easy to start unit testing. And that brings us to the end of this course. I hope you enjoyed watching and, of course, I hope you learned a lot, and you should now be ready to build your own flask application. I'm Reindert-Jan Ekker Pluralsight, thank you for watching.

Course author

Reindert-Jan Ekker

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

Course info

Level	Intermediate
Rating	★★★★½ (202)
My rating	★★★★★
Duration	3h 57m
Released	26 Dec 2014

Share course