<https://www.linkedin.com/learning/learning-flask/>

## Course details

1h 37m Beginner Released: September 22, 2016

Note: Learning Flask was created by Packt Publishing. It was originally released on 7/31/15. We are pleased to host this training in our library.  
  
The need for talented developers is greater than ever before. A basic grounding in a framework as simple, powerful, and easy-to-learn as Flask—a Python micro framework—can help launch your programming career. Get a basic grounding in the fundamental concepts of web development, as well as the hands-on experience required to successfully build web apps with Flask, in this course with Packt trainer Lalith Polepeddi.  
  
It starts with turning your computer into a modern development machine. Then learn everything you need to know about Flask: models, views, controllers, web templates, forms, validation, and API interactions. Lalith shows how to apply these concepts and develop a powerful database-backed Flask app, including a full authentication system. Finally, Lalith shows how to extend the app's functionality by integrating it with third-party APIs. When you finish this course, you will be fully equipped to build your very own custom web apps with Flash.

### Learning Objectives:

* Setting up your development environment
* Creating a homepage
* Building templates
* Routing with Flask
* Deploying to Heroku
* Storing data
* Creating and rendering forms
* Validating form data
* Creating a user authentication system for login and logout
* Implementing location-based app features

Let's trace what happens when a user visits the page in the Flask app. The cycle starts with user opening his browser, typing a URL and pressing enter. So step one, when the user presses enter, the browser issues a request for that URL. Step two, the request hits routes.py, routes.py has code that maps the URL to a Python function. Step three, the Python function finds the correct HTML file living in the templates folder.

Step four, this HTML file fetches CSS, Javascript or images it needs from the static folder. Step five, rendered HTML is sent back to routes.py. Step six, routes.py sends the HTML back to the browser and the page loads and the user This sequence is called the Request-Response Cycle and it's how the files and folders in a Flask app connect to each other.

|  |
| --- |
|  |

[bou@fed27 flask\_projects]$ mkdir learn-flask

[bou@fed27 flask\_projects]$ cd learn-flask/

[bou@fed27 learn-flask]$ mkdir static

[bou@fed27 learn-flask]$ cd static

[bou@fed27 static]$ mkdir css

[bou@fed27 static]$ mkdir js

[bou@fed27 static]$ mkdir img

[bou@fed27 static]$ cd ../

[bou@fed27 learn-flask]$ mkdir templates

[bou@fed27 learn-flask]$ touch routes.py

[bou@fed27 learn-flask]$ touch README.md

[bou@fed27 learn-flask]$ tree

.

├── README.md

├── routes.py

├── static

│   ├── css

│   ├── img

│   └── js

└── templates

### Creating a homepage

[bou@fed27 learn-flask]$ cd templates/

[bou@fed27 templates]$ touch index.html

Suppose I now want to make another page, say an about page. My website should have a similar style and layout across all of its pages, so that means I will need to copy all of this HTML from this index.html file for this new file that I want to create. It seems pretty wasteful to write all of this HTML again when the only part I'd be changing is this small bit of content in the body after the header.

Thinking even further, what happens if I need to add a new logo, or a CSS file to my website? I would need to go into every one of these pages and then add them manually. This is pretty time consuming and error prone. Therefore, instead of repeatedly writing the same HTML in multiple different files, we need a way to write things once. This is what web templates are for.

### Making templates

[bou@fed27 templates]$ touch layout.html

How to use web templates to reuse the same HTML across multiple pages. To understand web templates, let's start by breaking apart index.html. Here's what index.html looks like now. So let's begin by making a new file in templates called: layout.html. So cd templates touch layout.html. In layout.html, we'll write all the html that shouldn't change across pages, like the doc type, the head, and the header of the page. We'll call **layout.html the base template.**

layout.html

<!DOCTYPE html>

<html>

<head>

<link href='http://fonts.googleapis.com/css?family=Open+Sans:300,400,600' rel='stylesheet'>

<link href="{{ url\_for('static', filename='css/main.css') }}" rel="stylesheet">

</head>

<body>

<header>

<div class="container">

<h1 class="title">Learning Flask</h1>

</div>

</header>

{% block content %}

{% endblock %}

</body>

</html>

Now there is this **block content endblock** part. To understand what's going on here see what index.html looks like now.

index.html

{% extends "layout.html" %}

{% block content %}

<main class="hero-section">

<div class="container">

<div class="section-content">

<h2>Discover places near you.</h2>

<a href="#" class="btn-primary">Sign up</a>

<a href="{{ url\_for('about') }}" class="btn-secondary">Learn more</a>

</div>

<div class="section-device">

<img src="static/img/device.png">

</div>

<div class="clearfix"></div>

</div>

</main>

{% endblock %}

Index.html inherits the html from the base template layout.html, and it fills that content block with its own text. In other words, we wrote all the common html in the base template, layout.html, and defined an empty block, named content, to be filled in by child templates, like index.html.

In this way, *the base template defines the common elements of the website, while the child elements customize it with their own content*. So what are web templates? They are html files that contain variables and control close statements. They solve the problem we faced in a previous video. Rather than writing the same html over and over again in multiple webpages, we write the common html once in a base template, and inherit it and customize it from the child templates. In this way, web templates make your app easier to maintain.

We just created a template, indexed at html, and we placed it inside the templates folder. For us to see this page in the browser, we need to map a URL to it, and we do this in **routes.py**.

from flask import Flask, render\_template

app = Flask(\_\_name\_\_)

@app.route("/")

def index():

return render\_template("index.html")

@app.route("/about")

def about():

return render\_template("about.html")

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

create a new usable instance of the Flask class and save it into the variable app.

map the URL "/" to the Python function index. The Python function uses the Flask function render\_template to render index.html. So now when a user types in the URL "/", the function index will run and return the page index.html.

app.run runs the app on a local server, the debug=True flag here is set so that we'll see any error messages along the way.

Go to the top level of the Flask app, and then type python routes.py. So, this will start the local server.

[bou@fed27 learn-flask]$ ls -ltr

total 97

drwxrwxr-x. 1 bou bou 0 Nov 28 13:11 static

-rw-rw-r--. 1 bou bou 0 Nov 28 13:12 README.md

-rw-rw-r--. 1 bou bou 94378 Nov 28 13:26 learn.docx

-rw-rw-r--. 1 bou bou 256 Nov 28 15:19 routes.py

drwxrwxr-x. 1 bou bou 360 Nov 28 15:20 templates

[bou@fed27 learn-flask]$ python routes.py

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

\* Restarting with stat

\* Debugger is active!

\* Debugger PIN: 273-114-815

127.0.0.1 - - [28/Nov/2018 15:22:07] "GET / HTTP/1.1" 200 -

127.0.0.1 - - [28/Nov/2018 15:22:07] "GET /static/css/main.css HTTP/1.1" 200 -

127.0.0.1 - - [28/Nov/2018 15:22:07] "GET /static/img/device.png HTTP/1.1" 404 -

And as per the output here in the terminal, go to localhost:5000 in the browser, and there's the page.

|  |
| --- |
|  |

### Using static files

We created a file called main.css inside the static css folder before.

[bou@fed27 learn-flask]$ tree

.

├── README.md

├── routes.py

├── static

│   ├── css

│   ├── img

│   └── js

└── templates

Copy this into the main.css file.

/\*

\* General

\*/

html, body, h1 {

margin: 0;

padding: 0;

}

body {

background-color: #f4f4f4;

font-family: 'Open Sans', sans-serif;

}

h2 {

font-weight: 300;

letter-spacing: -1.2px;

}

a {

text-decoration: none;

display: inline-block;

}

.container {

width: 1000px;

margin: 0 auto;

}

.pull-left {

float: left;

}

.pull-right {

float: right;

}

.clearfix {

clear: both;

}

.section-content {

color: #3a3a3a;

float: left;

width: 340px;

}

.section-device {

float: right;

position: relative;

top: 65px;

}

.section-device img {

width: 600px;

}

.btn-primary {

background: #4285f4;

border: 1px solid #1266f1;

color: #ffffff;

font-family: 'Open Sans', sans-serif;

font-size: 13px;

font-weight: 600;

letter-spacing: 0.23px;

padding: 12px 32px;

text-align: center;

text-decoration: none;

text-transform: uppercase;

}

.btn-secondary {

background: #ffffff;

border: 1px solid #e6e6e6;

color: #3372df;

font-family: 'Open Sans', sans-serif;

font-size: 13px;

font-weight: 600;

letter-spacing: 0.23px;

padding: 12px 32px;

text-align: center;

text-decoration: none;

text-transform: uppercase;

}

/\*

\* Header

\*/

header {

background-color: #fff;

padding: 25px 0;

}

.title {

float: left;

font-family: 'Helvetica Neue', Arial, Helvetica, sans-serif;

font-size: 17px;

letter-spacing: 2.5px;

text-transform: uppercase;

}

.title a, .title a:visited {

color: #3a3a3a;

}

.main-nav {

float: right;

}

.main-nav ul {

list-style: none;

margin: 0;

padding: 0;

}

.main-nav li {

display: inline-block;

font-family: 'Open Sans', sans-serif;

font-size: 13px;

font-weight: 600;

text-transform: uppercase;

}

.main-nav li a {

color: #3a3a3a;

}

/\*

\* Hero section

\*/

.hero-section .section-content {

position: relative;

top: 100px;

}

.hero-section .section-content h2 {

font-size: 40px;

}

.hero-section .section-content .btn-primary {

margin-right: 10px;

}

/\*

\* Sign up

\*/

.signup-section .section-content h2 {

font-size: 30px;

}

.signup-section .section-content {

position: relative;

top: 53px;

}

.form-group {

margin-bottom: 20px;

}

.form-group label {

color: #3a3a3a;

display: block;

font-family: 'Open Sans', sans-serif;

font-size: 15px;

font-weight: 600;

margin-bottom: 5px;

}

.form-group input {

border: none;

border-bottom: 2px solid #4285f4;

background-color: transparent;

color: #3a3a3a;

font-size: 18px;

padding: 5px 0;

width: 100%;

}

.form-group input:focus {

outline: 0;

}

.form-group .error-message {

margin: 10px 0;

color: #db4437;

}

.form-group .error-message + input {

border-bottom: 2px solid #db4437;

}

.error-message {

color: #db4437;

}

/\*

\* Sign up

\*/

.about-section h2 {

font-size: 30px;

margin-bottom: 0;

}

.about-section p {

font-size: 16px;

}

/\*

\* Home

\*/

.section-tabs {

float: left;

width: 340px;

}

.section-tabs .places {

margin-top: 62px;

padding-right: 20px;

}

.section-tabs .places a,

.section-tabs .places a:visited {

color: #3372df;

}

.section-tabs article {

margin-bottom: 20px;

}

.section-tabs article .name {

font-size: 18px;

}

.section-tabs article .walking-distance {

color: #6e6e6e;

font-size: 13px;

margin: 0;

}

.section-map {

float: right;

width: 660px;

}

.section-map .nav {

padding: 20px 0;

}

.section-map .form-group {

display: inline;

}

.section-map #address {

font-size: 14px;

margin-right: 10px;

width: 87%;

}

.section-map #submit {

font-size: 10px;

padding: 5px 15px;

}

#map {

height: 500px;

}

.leaflet-popup-content h3 {

margin: 0;

}

.leaflet-popup-content p {

margin: 10px 0 ;

}

We want to use the CSS file across all the web pages – so we add the CSS file to the base template layout.html. Open up layout.html, and add this line inside the head element.

So replace the static/css/main.css line

<html>

<head>

<link href='http://fonts.googleapis.com/css?family=Open+Sans:300,400,600' rel='stylesheet'>

<link href="static/css/main.css" rel="stylesheet">

</head>

with

<html>

<head>

<link href='http://fonts.googleapis.com/css?family=Open+Sans:300,400,600' rel='stylesheet'>

<link href="{{ url\_for('static', filename='css/main.css') }}" rel="stylesheet">

</head>

Reason for doing this is rather than hardcovering the filepath to main.css, we're using the Flask function *url\_for* to generate a URL for us.

Here, *url\_for* is telling Flask to go to the static folder and look for the file main.css. We've gone through a full turn of the request-response cycle. Trace what we've done.

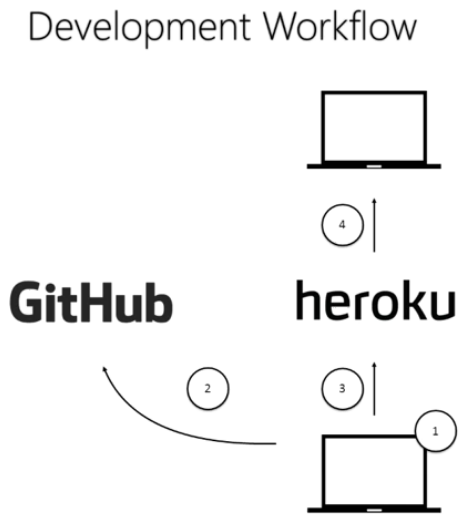
When we type in ***http://localhost:5000/*** into the browser, the browser issues a request for that URL. “/” The request hits routes.py where the URL slash was mapped to the function index(). The function index fetched the web template *index.html* from the templates folder. The web template used a CSS file, so the CSS file was fetched from the static folder. Routes.py then sent all the rendered HTML back to the browser for us to see.

127.0.0.1 - - [28/Nov/2018 15:45:24] "GET / HTTP/1.1" 200 - [Get / Fetches index page]

127.0.0.1 - - [28/Nov/2018 15:45:24] "GET /favicon.ico HTTP/1.1" 404 -

### Saving to GitHub

Let's review the workflow we're following. The workflow starts on your computer. You're using a text editor to write your code, and you're using the command line to run your code. Now, as you write more code you should keep track of it using Git and and store that code on GitHub. When the application is ready to share with others, we should deploy Heroku, so that users can go to that app's URL and view the site.



|  |
| --- |
|  |

So far, we've finished step one, by making a homepage, so let's proceed with step two and push the code up to GitHub.

Create a new Git repository. Type git init inside your application. Let's verify that the new Git repository was created. Type ls -al. This command lets you see all files, even hidden ones. And here we see that the .git folder was created, so a new repository was created successfully.

[bou@fed27 learn-flask]$ pwd

/home/bou/stats-R/flask\_projects/learn-flask

[bou@fed27 learn-flask]$ git init

Initialized empty Git repository in /mnt/win\_data/shared/stats-R/flask\_projects/learn-flask/.git/

[bou@fed27 learn-flask]$ ls -al

total 302

drwxrwxr-x. 1 bou bou 4096 Nov 28 15:49 .

drwxrwxr-x. 1 bou bou 4096 Nov 28 13:24 ..

drwxrwxr-x. 1 bou bou 544 Nov 28 15:49 .git

-rw-rw-r--. 1 bou bou 296985 Nov 28 15:46 learn.docx

-rw-rw-r--. 1 bou bou 75 Nov 28 15:46 .~lock.learn.docx#

-rw-rw-r--. 1 bou bou 0 Nov 28 13:12 README.md

-rw-rw-r--. 1 bou bou 256 Nov 28 15:19 routes.py

drwxrwxr-x. 1 bou bou 0 Nov 28 13:11 static

drwxrwxr-x. 1 bou bou 0 Nov 28 15:20 templates

[bou@fed27 learn-flask]$ ls -al .git

total 7

drwxrwxr-x. 1 bou bou 544 Nov 28 15:49 .

drwxrwxr-x. 1 bou bou 4096 Nov 28 15:49 ..

drwxrwxr-x. 1 bou bou 0 Nov 28 15:49 branches

-rw-rw-r--. 1 bou bou 92 Nov 28 15:49 config

-rw-rw-r--. 1 bou bou 73 Nov 28 15:49 description

-rw-rw-r--. 1 bou bou 23 Nov 28 15:49 HEAD

drwxrwxr-x. 1 bou bou 536 Nov 28 15:49 hooks

drwxrwxr-x. 1 bou bou 144 Nov 28 15:49 info

drwxrwxr-x. 1 bou bou 240 Nov 28 15:49 objects

drwxrwxr-x. 1 bou bou 240 Nov 28 15:49 refs

Next, let's add the files we changed to this repository. To see which files changed, type git status.

The output shows that these were the files that we have modified and this is right, these are the ones that we actually did modify earlier. So let's go ahead and add them: git add routes.py. This command starts tracking changes to the routes.py file. Type git status to double check that the file was added, and there it is in green, the new file routes.py was added, and then go ahead and add the other files that we've modified, so git add README, git add static, git add templates, type git status one more time to confirm that all these files were added, and they were.

Go ahead and commit these new files. Type git commit -m "Initial commit" and new snapshot of the project is created. Now all the code is committed to a Git repository locally so let's get set up with GitHub next. Go to GitHub.com, and create a new repository. Type in a name for your repository, I'm giving mine learning-flask, and create the repository, then scroll down and take these two lines and copy them into your terminal.

The first line connects your local Git repository to the GitHub repo, and the second line actually pushes your code from your local repo to the GitHub repository. The code should be up on GitHub now, so refresh the page and you should see your commits. So, those are the common Git commands that we'll use: git status, git add, git commit, git push, and that's the basic sequence we'll be using them in.

There's much more you can do with Git, and check out this book to learn how you can make full use of Git and understand it more. Now that the code is under version control and pushed to GitHub, let's take a look at how to deploy the app to Heroku in the next video