**WEB AND REST API**

**DEVELOPMENT**

**USING**

**FLASK/FLASK-LOGIN/FLASK-RESTFUL AND OTHER FLASK EXTENSIONS**

Table of Contents

[1 HTTP 5](#_Toc28160222)

[1.1 Basic aspects of HTTP 5](#_Toc28160223)

[1.2 Methods 5](#_Toc28160224)

[1.3 HTTP Headers 6](#_Toc28160225)

[1.4 De-constructing Requests 7](#_Toc28160226)

[1.5 De-constructing Responses 9](#_Toc28160227)

[1.6 The Server doesn’t know who you are 10](#_Toc28160228)

[1.7 500, 404 and other errors 10](#_Toc28160229)

[2 Cookies 10](#_Toc28160230)

[2.1 Introduction 10](#_Toc28160231)

[2.2 Main uses of Cookies 10](#_Toc28160232)

[2.2.1 Session management 10](#_Toc28160233)

[2.2.2 Personalization 10](#_Toc28160234)

[2.2.3 Tracking 10](#_Toc28160235)

[3 Better alternatives to cookies 11](#_Toc28160236)

[3.1 HTML Web Storage 11](#_Toc28160237)

[3.1.1 LocalStorage 11](#_Toc28160238)

[3.1.2 SessionStorage 11](#_Toc28160239)

[3.2 IndexedDB 11](#_Toc28160240)

[4 What are REST APIs ? 12](#_Toc28160241)

[4.1 Why have REST APIs become so popular over the years 12](#_Toc28160242)

[4.2 REST vs SOA vs Micro Services 12](#_Toc28160243)

[5 The Open API Specification 12](#_Toc28160244)

[5.1 The Open API 12](#_Toc28160245)

[5.2 The Open API Specification Document 12](#_Toc28160246)

[5.2.1 Samples 12](#_Toc28160247)

[5.3 Auto generation of the Open API specification document from code annotations 12](#_Toc28160248)

[6 Swagger UI 13](#_Toc28160249)

[6.1 What is Swagger UI 13](#_Toc28160250)

[6.2 Installing Swagger UI 13](#_Toc28160251)

[6.2.1 Method 1 13](#_Toc28160252)

[6.3 Tweaking Swagger UI 14](#_Toc28160253)

[7 Alternatives to Swagger 15](#_Toc28160254)

[7.1 PostMan 15](#_Toc28160255)

[8 Authentication and Authorization techniques 16](#_Toc28160256)

[9 A sample REST API using Flask-Restful 16](#_Toc28160257)

[9.1 Sample 16](#_Toc28160258)

[9.2 Documenting this REST API 16](#_Toc28160259)

[9.3 Auto generation of Open API spec documentation 16](#_Toc28160260)

[9.4 Viewing the API on Swagger UI 16](#_Toc28160261)

[9.5 Testing this API 16](#_Toc28160262)

[10 Full Fledged REST API development using Flask-Restful 16](#_Toc28160263)

[11 Deployment to production environments 16](#_Toc28160264)

[11.1 The basics 16](#_Toc28160265)

[11.2 Setup.py 16](#_Toc28160266)

[11.2.1 Examples / Samples 17](#_Toc28160267)

[11.3 MANIFEST.in 17](#_Toc28160268)

[11.4 Distributing Resources 17](#_Toc28160269)

[12 APPENDIX A – FLASK BLUEPRINT FOLDER STRUCTURES 19](#_Toc28160270)

[12.1 Divisional Structure 19](#_Toc28160271)

[12.2 Functional Structure 20](#_Toc28160272)

[13 APPENDIX B – ALTERNATIVE PYTHON WEB/REST API FRAMEWORKS 20](#_Toc28160273)

[13.1 Falcon 20](#_Toc28160274)

[13.2 TORNADO 20](#_Toc28160275)

[13.3 Django 20](#_Toc28160276)

[14 APPENDIX C – WSGI SERVERS 20](#_Toc28160277)

[14.1 Gunicorn 20](#_Toc28160278)

[14.2 Waitress 20](#_Toc28160279)

[14.3 CherryPy 20](#_Toc28160280)

# HTTP

## Basic aspects of HTTP

|  |  |
| --- | --- |
| Simple |  |
| Extensible | [HTTP headers](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers) make the protocol easy to extend and experiment with.  New functionality can even be introduced by a simple agreement between a client and a server about a new header's semantics |
| [HTTP is stateless, but not sessionless](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#HTTP_is_stateless_but_not_sessionless) | HTTP is stateless: there is no link between two requests being successively carried out on the same connection. This immediately has the prospect of being problematic for users attempting to interact with certain pages coherently, for example, using e-commerce shopping baskets. But while the core of HTTP itself is stateless, HTTP cookies allow the use of stateful sessions. Using header extensibility, HTTP Cookies are added to the workflow, allowing session creation on each HTTP request to share the same context, or the same state |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## Methods

|  |  |
| --- | --- |
| GET |  |
| POST |  |
| PUT |  |
| HEAD |  |
| DELETE |  |
| PATCH |  |
| OPTIONS |  |

## HTTP Headers

**HTTP headers** let the client and the server pass additional information with an HTTP request or response. An HTTP header consists of its case-insensitive name followed by a colon (:), then by its value. [Whitespace](https://developer.mozilla.org/en-US/docs/Glossary/Whitespace) before the value is ignored.

Custom proprietary headers have historically been used with an X- prefix, but this convention was deprecated in June 2012 because of the inconveniences it caused when nonstandard fields became standard in [RFC 6648](https://tools.ietf.org/html/rfc6648); others are listed in an [IANA registry](http://www.iana.org/assignments/message-headers/perm-headers.html), whose original content was defined in [RFC 4229](http://tools.ietf.org/html/rfc4229). IANA also maintains a [registry of proposed new HTTP headers](http://www.iana.org/assignments/message-headers/prov-headers.html).

Headers can be grouped according to their contexts:

* [General headers](https://developer.mozilla.org/en-US/docs/Glossary/General_header) apply to both requests and responses, but with no relation to the data transmitted in the body.
* [Request headers](https://developer.mozilla.org/en-US/docs/Glossary/Request_header) contain more information about the resource to be fetched, or about the client requesting the resource.
* [Response headers](https://developer.mozilla.org/en-US/docs/Glossary/Response_header) hold additional information about the response, like its location or about the server providing it.
* [Entity headers](https://developer.mozilla.org/en-US/docs/Glossary/Entity_header) contain information about the body of the resource, like its [content length](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Length) or [MIME type](https://developer.mozilla.org/en-US/docs/Web/SVG/Content_type).

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## De-constructing Requests

|  |  |
| --- | --- |
| Sample | GET / HTTP/1.1  Host: developer.mozilla.org  Accept-Language: fr |
| Sample 1 |  |
| Sample 2 |  |
| Sample 3 |  |
| Sample 4 |  |
| Sample 5 |  |
| Format of a GET request |  |
| Format of a POST request | http://127.0.0.1/pass.php  POST /pass.php HTTP/1.1  Host: 127.0.0.1  User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:18.0) Gecko/20100101 Firefox/18.0  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8  Accept-Language: en-US,en;q=0.5  Accept-Encoding: gzip, deflate  DNT: 1  Referer: http://127.0.0.1/pass.php  Cookie: passx=87e8af376bc9d9bfec2c7c0193e6af70; PHPSESSID=l9hk7mfh0ppqecg8gialak6gt5  Connection: keep-alive  Content-Type: application/x-www-form-urlencoded  Content-Length: 30  username=zurfyx&pass=password |
| Tools to inspect Requests |  |

## De-constructing Responses

|  |  |
| --- | --- |
|  |  |
|  | HTTP/1.1 200 OK  Date: Sat, 09 Oct 2010 14:28:02 GMT  Server: Apache  Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT  ETag: "51142bc1-7449-479b075b2891b"  Accept-Ranges: bytes  Content-Length: 29769  Content-Type: text/html  <!DOCTYPE html... (here comes the 29769 bytes of the requested web page) |
|  |  |
|  |  |
|  |  |
|  |  |

## The Server doesn’t know who you are

The **HTTP protocol** is a stateless protocol meaning that the server isn’t required to store session information and each request is independent of the other.-

## 500, 404 and other errors

# Cookies

The overwhelming desire to maintain state across requests.

## Introduction

An HTTP cookie (web cookie, browser cookie) is a small piece of data that a server sends to the user's web browser. The browser may store it and send it back with the next request to the same server. Typically, it's used to tell if two requests came from the same browser — keeping a user logged-in, for example. It remembers stateful information for the [stateless](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#HTTP_is_stateless_but_not_sessionless) HTTP protocol.

## Main uses of Cookies

Cookies are mainly used for three purposes:

### Session management

Logins, shopping carts, game scores, or anything else the server should remember

### Personalization

User preferences, themes, and other settings

### Tracking

Recording and analyzing user behavior

Cookies were once used for general client-side storage. While this was legitimate when they were the only way to store data on the client, it is recommended nowadays to prefer modern storage APIs. Cookies are sent with every request, so they can worsen performance (especially for mobile data connections). Modern APIs for client storage are the [Web storage API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API)(localStorage and sessionStorage) and [IndexedDB](https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API).

# Better alternatives to cookies

## HTML Web Storage

Before HTML5, application data had to be stored in cookies, included in every server request. Web storage is more secure, and large amounts of data can be stored locally, without affecting website performance.

Unlike cookies, the storage limit is far larger (at least 5MB) and information is never transferred to the server.

The **Web Storage API** provides mechanisms by which browsers can store key/value pairs, in a much more intuitive fashion than using cookies.

Links - <https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API>

### LocalStorage

The localStorage object stores the data with no expiration date. The data will not be deleted when the browser is closed, and will be available the next day, week, or year.

### SessionStorage

The sessionStorage object is equal to the localStorage object, **except** that it stores the data for only one session. The data is deleted when the user closes the specific browser tab.

## IndexedDB

IndexedDB is a way for you to persistently store data inside a user's browser. Because it lets you create web applications with rich query abilities regardless of network availability, your applications can work both online and offline.

Links

* <https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Concepts_Behind_IndexedDB>

# What are REST APIs ?

## Why have REST APIs become so popular over the years

## REST vs SOA vs Micro Services

# The Open API Specification

## The Open API

## The Open API Specification Document

### Samples

|  |  |
| --- | --- |
| # | Open weather map [YAML format] |

## Auto generation of the Open API specification document from code annotations

# Swagger UI

## What is Swagger UI

Swagger UI provides a display framework that reads an [OpenAPI specification document](https://github.com/OAI/OpenAPI-Specification) and generates an interactive documentation website.

The UI generates an interactive API console for users to quickly learn about various APIs and actually experiment with requests.

One can get familiar with Swagger UI through the Petstore demo.

First time users are encouraged to explore the [Swagger Petstore example](http://petstore.swagger.io/) to get a better understanding of Swagger UI, let’s.

In the Petstore example, the site is generated using [Swagger UI](https://github.com/swagger-api/swagger-ui)

More information available at - <https://idratherbewriting.com/learnapidoc/pubapis_swagger.html>

## Installing Swagger UI

### Method 1

|  |  |
| --- | --- |
| # | Step |
| 1 | Install Node JS |
| 2 | Install swagger-ui-dist module |
| 3 | Install httpster to serve the swagger-ui-dist module |

## Tweaking Swagger UI

By default, the Swagger UI will launch the petstore demo. This can be changed to launch other documents.

|  |
| --- |
| The index.html file in the swagger-ui-dist folder has been modified to read another .yml file |

# Alternatives to Swagger

## PostMan

# Authentication and Authorization techniques

This cuts across technologies, applications, servers and whatever else one can think of. Authenticating a user and authorizing access to a set of resources is now fundamental to any application.

There are many different ways to do this and, in some way or the other, involve the ability to maintain state across requests.

# A sample REST API using Flask-Restful

## Sample

## Documenting this REST API

## Auto generation of Open API spec documentation

## Viewing the API on Swagger UI

## Testing this API

# Full Fledged REST API development using Flask-Restful

# Deployment to production environments

## The basics

Making your project installable means that you can build a distribution file and install that in another environment,

When you want to deploy your application elsewhere, you build a distribution file. The current standard for Python distribution is the wheel format, with the .whl extension. Make sure the wheel library is installed first:

python setup.py bdist\_wheel

python setup.py sdist bdist\_wheel

## Setup.py

### Examples / Samples

|  |
| --- |
| **import** **setuptools**  **with** open("README.md", "r") **as** fh:  long\_description = fh.read()  setuptools.setup(  name="example-pkg-your-username",  version="0.0.1",  author="Example Author",  author\_email="author@example.com",  description="A small example package",  long\_description=long\_description,  long\_description\_content\_type="text/markdown",  url="https://github.com/pypa/sampleproject",  packages=setuptools.find\_packages(),  classifiers=[  "Programming Language :: Python :: 3",  "License :: OSI Approved :: MIT License",  "Operating System :: OS Independent",  ],  python\_requires='>=3.6',  ) |
|  |
|  |

## MANIFEST.in

## Distributing Resources

If you try to install the package you just created, you will notice that folders like static or templates are not installed for you. The reason for this is that setuptools does not know which files to add for you. What you should do, is to create a MANIFEST.in file next to your setup.py file. This file lists all the files that should be added to your tarball:

recursive-include yourapplication/templates \*

recursive-include yourapplication/static \*

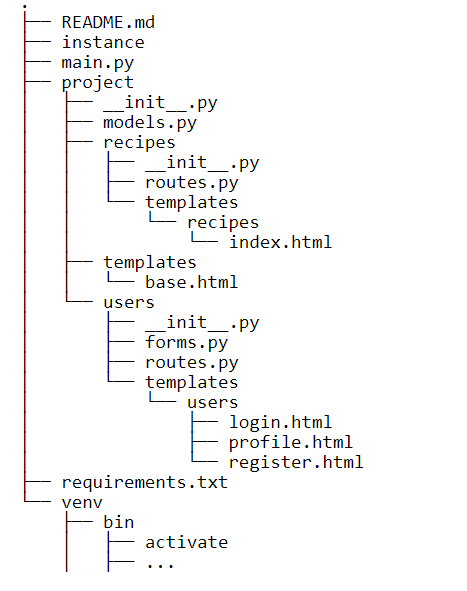
Don’t forget that even if you enlist them in your MANIFEST.in file, they won’t be installed for you unless you set the include\_package\_data parameter of the setup function to True!

# APPENDIX A – FLASK BLUEPRINT FOLDER STRUCTURES

Flask is a compact, lean framework that does not force developers to use standard folder structures.

Proper design is very important

## Divisional Structure



## Functional Structure

## Hybrid Flask Blueprint folder structures

# APPENDIX B – ALTERNATIVE PYTHON WEB/REST API FRAMEWORKS

## Falcon

## TORNADO

## Django

# APPENDIX C – WSGI SERVERS

## Gunicorn

## Waitress

## CherryPy

| **MAIN TOPIC** | **SUB TOPIC** | **DETAILS**  **/**  **(LINKS FOR FURTHER STUDY)**  **/**  **(FEEDBACK)**  **/**  **(SAMPLE PROGRAMS)** | **(CLASSROOM EXERCISES)**  **/**  **(ASSIGNMENTS)** | **ETA** |
| --- | --- | --- | --- | --- |
| **OVERALL CONTEXT** | WHAT ARE YOU EXPECTING ? | <Update after feedback from the students> | **N/A** | DAY 1  (<=15 mins) |
|  | MY EXPECTATIONS FROM THE STUDENTS/YOU | * Be aware of the course content (*Have all of you gone through the course details [separate doc] ?)* * Do the class room exercises * Complete your assignments * Make notes *(I do it and it helps me)* * Don’t just nod your head to what I say. Digest it slowly. Stop me if I am going too fast | **N/A** | DAY 1  (<= 15 mins) |
|  |  |  |  |  |
|  |  | Do all these assignments/exercises in a virtual environment |  |  |
|  |  |  |  |  |
| **FLASK** | THE BASICS | Creating a really simple flask app |  |  |
|  |  | A flask app with a few routes |  |  |
|  |  | ROUTES |  |  |
|  |  | Launching a flask app from the command line |  |  |
|  |  | Using Flask’s in-built server for launching web apps |  |  |
|  |  | Specifying GET, POST (HTTP Methods) |  |  |
|  |  | Using url\_for, redirect and a few other features |  |  |
|  |  | Processing Requests |  |  |
|  |  | Responses |  |  |
|  |  | Stuff on Context Locals, Thread Safety, Proxies and some other important terms |  |  |
|  |  | Session Management | **‘g’ object** |  |
|  |  | Error handling |  |  |
|  |  | Opening a flask shell |  |  |
|  |  | Debug Mode |  |  |
|  |  |  |  |  |
|  |  | Building modular apps using blueprints |  |  |
|  |  | A basic introduction to Blue Prints |  |  |
|  |  | Divisional structures for flask apps |  |  |
|  |  |  |  |  |
|  |  | Functional structures for flask apps |  |  |
|  |  |  |  |  |
|  |  | Hybrid structures for flask app |  |  |
|  |  |  |  |  |
|  |  | Flask Extensions that are commonly used |  |  |
|  |  |  |  |  |
|  | PROJECTS | Creating a plain vanilla flask app with routes that implement all possible combinations/rules  [At least 50 routes] |  |  |
|  |  |  |  |  |
|  |  | A flask app with plain vanilla session management |  |  |
|  |  |  |  |  |
|  |  | A flask app that uses flask-login for session management |  |  |
|  |  | A flask app that authenticates users using SAML |  |  |
|  |  |  |  |  |
|  |  | A flask app that uses token based authentication |  |  |
|  |  |  |  |  |
|  |  | A flask app that uses OAuth 2 and OpenID Connect (OIDC) |  |  |
|  |  |  |  |  |
|  |  | A flask app that connects to a database using ORM |  |  |
|  |  |  |  |  |
|  |  | ORM + Migrations  [Alembic, Flask-Migrate, SQL-Alchemy] |  |  |
|  |  |  |  |  |
|  |  | Passwords supplied by the users are stored in encrypted and salted format |  |  |
|  |  |  |  |  |
|  |  | Using flask-restful to implement around 10 REST APIs with Swagger documentation |  |  |
|  |  |  |  |  |
|  |  | Strengthening a flask app using Flask-Security |  |  |
|  |  |  |  |  |
|  |  | A completely dockerized Flask App  [docker, docker compose, nginx, gunicorn, postgresql, multiple containers, …….] |  |  |
|  |  |  |  |  |
|  |  | Taking a few flask apps and deploying these apps on stand alone WSGI containers |  |  |
|  |  |  |  |  |
|  |  | Debugging a flask web app using a third party editor like PyCharm |  |  |
|  |  |  |  |  |
|  |  | Flask app with full Pytest coverage |  |  |
|  |  |  |  |  |