First Steps in Web Development with Python

Miguel Grinberg

@miguelgrinberg

europython

July 8-14 2019

BASEL

About Me

- My blog: https://blog.miguelgrinberg.com
- My books / courses:
  - Flask Web Development (O'Reilly)
  - The New and Improved Flask Mega-Tutorial
  - MicroPython and the Internet of Things
- My open source: https://github.com/miguelgrinberg
  - Python port of the Socket.IO server and client
  - Flask extensions: Flask-SocketIO, Flask-Migrate, Flask-HTTPAuth, Flask-Moment, etc.
  - Flask examples: Lots of them, check my GitHub page and my blog!
- I gladly answer Python or web development questions on social media :)
- I take tutoring, consulting and/or contract work (I'm also on Patreon!)
- Portland, OR, USA 🌹 is home; but currently living in Ireland ☘️

Agenda

Part I - Theory

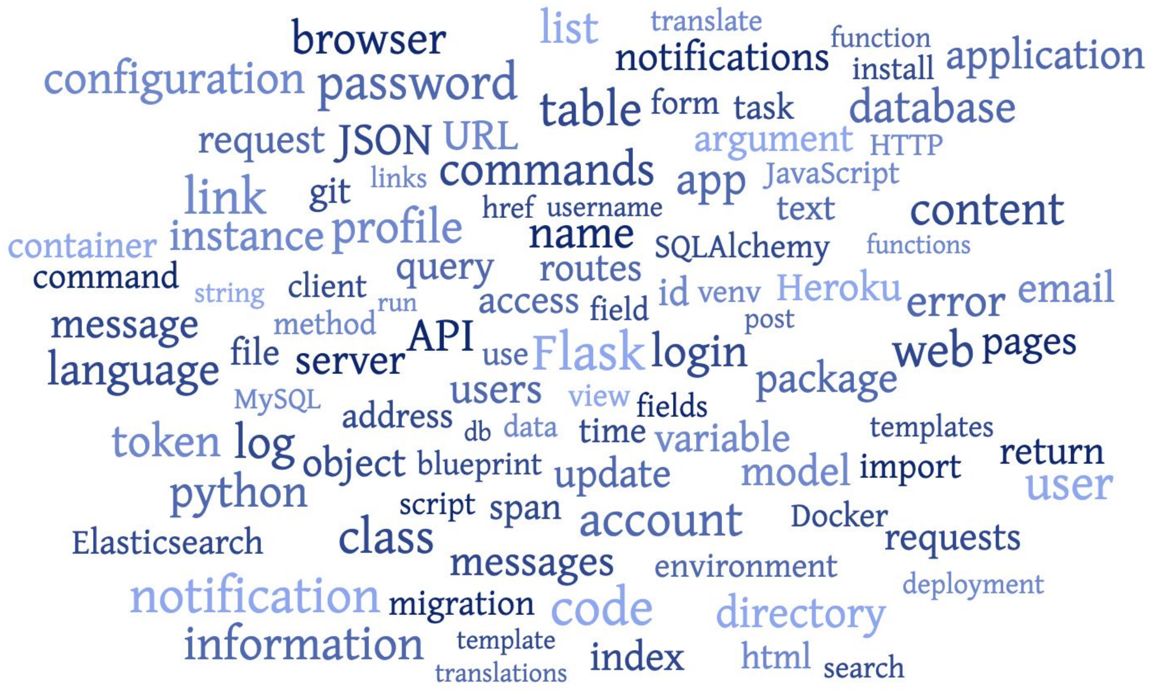
- Introduction to Web Development
- How Web Browsers Work
- URLs
- Requests and Responses
- How Web Servers Work

Part II - Practice

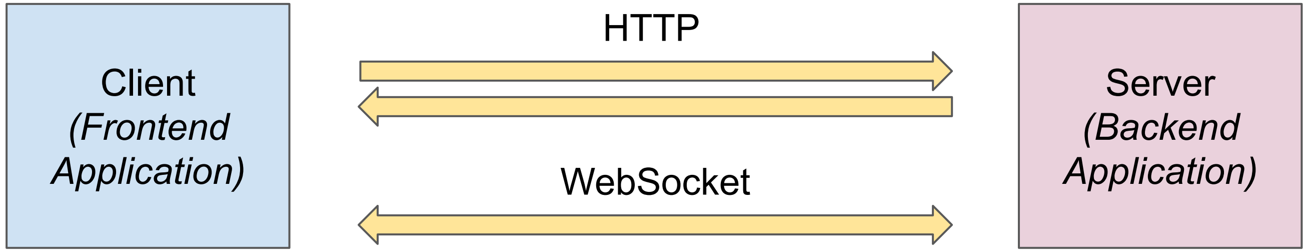
- Let’s write a web application!

Introduction to Web Development

Where Do I Start?

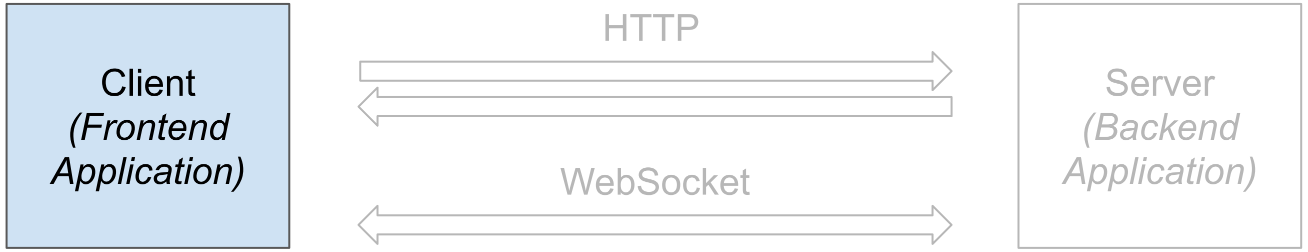


Web Development in One Slide!



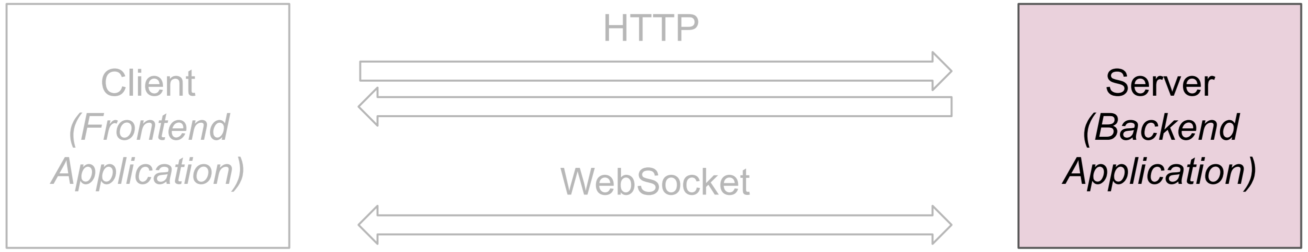
Client (Frontend Application)

- Runs on the user’s hardware (usually a web browser)
- Shows content to the user and accepts user’s input
- Varying degrees of application logic
    - Thin client: depends on most application logic provided by the server
    - Rich client: implements its own application logic
- Client languages: HTML, CSS, JavaScript



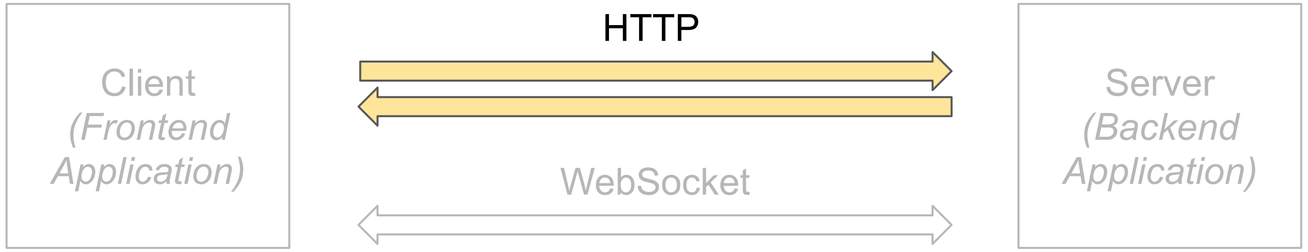
Server (Backend Application)

- Runs on the developer’s hardware, usually in a data center
- Provides supporting functions to client applications
- Varying degrees of application logic
    - Fat server: Implements most of the application logic
    - API server: only implements some aspects of the application logic, authentication and storage being the most common
- Server languages: any (but we prefer Python!)



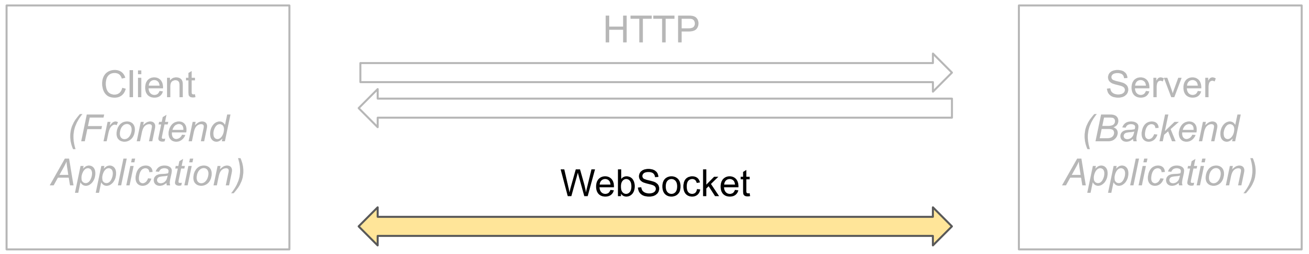
HTTP (Hypertext Transfer Protocol)

- Most common form of communication between clients and servers
- Client sends a *Request*, server replies with a *Response*
- The server cannot initiate an exchange, it only responds to client requests
- Each request/response cycle uses a separate connection
- Nicely fits the web browser model
- Vast majority of web applications use HTTP



WebSocket

- Newer protocol that attempts to address some of the limitations of HTTP
- Client or server can send data to the other side at any time
- Permanent connection between each client and the server
- Server needs to maintain a large number of long lived connections
- Nicely fits the async model
- Commonly used in highly interactive web sites (chat rooms, gaming, etc.)



Why Is Web Development Hard Then?

- Developing one robust application of any kind is hard. Two is harder!!!
- Server applications usually have a lot of moving parts
  - Database (MySQL, Postgres, MongoDB, etc.)
  - Caching layer (Redis, Memcached, etc.)
  - Horizontal Scaling
  - Proxy servers and/or load balancers (nginx, Apache, etc.)
  - Background and/or scheduled jobs
  - Cloud services (work queues, object storage, etc.)
- The browser is a terrible development platform for client applications
  - Can’t pick a language, must use the HTML, CSS and JavaScript triad
  - Lots of runtime platforms (Chrome, Firefox, Safari, IE, Edge, Opera, etc.)
  - Immature ecosystem without any great framework choices

How Web Browsers Work

GET Requests

- User types a URL → browser sends a GET request to the server for that URL
- The server response contains the web page for the URL, typically as HTML
- Browser clears the previous page and shows the new web page to the user
- If the web page references other resources then it sends an additional GET request for each
    - For images, the browser displays the image data in the response within the page
    - For CSS stylesheets, the browser uses the data to render the page appropriately
    - For JavaScript code, the browser executes the code
- User clicks on a link → browser sends a GET request for that link and the process repeats

POST Requests

- User fills out some fields and submits a form → browser sends a POST request with the data entered by the user
- The server response is handled in the same way as for GET requests

Redirects

- A server can optionally respond with a “redirect” response, which includes a redirect URL
- The browser sends a GET request to the redirect URL as soon as it receives the response

Background or Asynchronous Requests (Ajax)

- Custom JavaScript code running within a web page can also issue requests
- Background requests do not replace the current web page
- The server response for a background request must be handled by a JavaScript callback function

URLs

Scheme

`https://example.com:8041/api/users?online=1&role=mods#form`

- Specifies the protocol used
    - https:// is for HTTP protocol, with encryption
    - http:// is for HTTP protocol, without encryption
- There are other protocols besides HTTP

Host

```
https://example.com:8041/api/users?online=1&role=mods#form
```

- The name or IP address of the server
- Authentication information can be included as part of the host with the format username:password@example.com

Port

```
https://example.com:8041/api/users?online=1&role=mods#form
```

- The network port number on which the server is listening for connections
- Defaults to 443 for https:// and 80 for http:// if omitted
- Port numbers below 1024 can only be used from admin/root accounts

Path

`https://example.com:8041/api/users?online=1&role=mods#form`

- Address of the requested resource
- Can be a reference a static file or to an application defined resource

Query String

```
https://example.com:8041/api/users?online=1&role=mods#form
```

- Optional arguments included with the request
- The ? separates the path from the query string
- The & separates multiple arguments
- The = separates the argument name from the value

Fragment

```
https://example.com:8041/api/users?online=1&role=mods#form
```

- Usually indicates a bookmark location within the resource
- Fragments are handled entirely by the web browser
- The server does not receive the fragment part of a URL

URL Encoding

- Some characters are reserved and need to be escaped:

:	%3B	#	%23
/	%2F	@	%40
?	%3F	Space	%20 or +
&	%24	+	%2B
=	%3D	%	%25

- Percent encoding can be used for any other characters as well

URL Mapping

- Web applications map URLs to server resources
- URLs can map to static files
  - Example: https://example.com/static/{file} maps to /home/miguel/website/files/{file}
  - A request to https://example.com/static/images/hello.jpg returns the contents of file /home/miguel/website/files/images/hello.jpg
- Other URLs may map directly to pieces of application logic
  - Example: https://example.com/users/{id} maps to function `get_user({id})` in the application
  - A request to https://example.com/users/1234 triggers `get_user(1234)` to be invoked by the server

Requests and Responses

The HTTP Request

- Method: GET, POST, PUT, DELETE, and others
- URL
- Headers: name/value pairs that provide additional information
  - Authentication
  - Client capabilities
  - Cookies
  - Format and length of request body
- Body: optional data submitted by the client

The HTTP Response

- Status code: numeric code that indicates results
  - 2xx codes: success
  - 3xx codes: redirect
  - 4xx codes: client error
  - 5xx codes: server error
- Headers: name/value pairs that provide additional information to the client
  - Caching instructions
  - New cookies
  - New URL for a redirect
  - Format and length of the response body
- Body: optional data returned by the server

How Web Servers Work

Basic Structure of a Web Server

- Wait for incoming HTTP connections from clients
- If a GET request for a static file arrives, the contents of the file are returned as the response body
- If a request for an application defined URL arrives, a “handler” function in the application is invoked to generate the response
- Web frameworks such as Flask or Django help with web server tasks:
  - URL routing to functions
  - URL routing to static files
  - High-level representations of HTTP requests and responses
  - Authenticating users
  - etc.

Fat Servers (with Thin Clients)

- Most or all of the application logic is in the server
- HTML pages rendered by the server are returned as responses
- CSS stylesheets and images directly referenced in generated HTML are served as static files
- Interaction between client and server is through foreground GET and POST requests
- You can write an entire web application in Python + HTML + CSS (no or minimal JavaScript!)

API Servers (with Rich Clients)

- Server returns the bootstrapping web page with embedded or referenced JavaScript in initial request(s)
- Client application is controlled by JavaScript from then on
- All requests issued by JavaScript are background requests
- Server accepts requests from JavaScript code in the client to retrieve and store information, authenticate, etc.
- JavaScript APIs in the browser are used to generate the page content
- Applications are more complex, but can offer a better UX
- Client-side frameworks such as React or Angular simplify the task of writing browser applications (but not by much!!!)

Hands-On Exercise #1

HTML and CSS

Code: bit.ly/firststepswebdev

A Simple HTML File

A Simple HTML File: index.html (step1)

```
<!doctype html>
<html>
   <head>
       <title>My First Web Application</title>
   </head>
   <body>
       <h1>Hello, user!</h1>
   </body>
</html>
```

Styling with CSS

Styling with CSS: styles.css (step2)

```css
body {
   max-width: 50em;
   margin: 0 auto;
}
```

Styling with CSS: index.html (step3)

```
<!doctype html>
<html>
   <head>
       <title>My First Web Application</title>
       <link rel="stylesheet" href="styles.css">
   </head>
   <body>
       <h1>Hello, user!</h1>
   </body>
</html>
```

Hands-On Exercise #2

Set Up a Python Virtual Environment

Code: bit.ly/firststepswebdev

Creating a Python Virtual Environment

Creating a Python Virtual Environment

Mac OS X and Linux

```
$ mkdir webapp
$ cd webapp
webapp $ python3 -m venv venv
webapp $ source venv/bin/activate
(venv) webapp $ pip install flask
```

Windows

```
$ mkdir webapp
$ cd webapp
webapp $ python3 -m venv venv
webapp $ venv\Scripts\activate
(venv) webapp $ pip install flask
```

Hands-On Exercise #3

Let’s Write a Fat Server with Flask!

Code: bit.ly/firststepswebdev

The Simplest Web Application

The Simplest Web Application: app.py (step4)

```
from flask import Flask


app = Flask(__name__)


@app.route('/')
def index():
    return '<h1>Hello, user!</h1>'
```

Running the Application

Running the Application

```
(venv) $ flask run
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Debug Mode - Mac OS X and Linux

```
(venv) webapp $ export FLASK_DEBUG=1
(venv) webapp $ flask run
```

Debug Mode - Windows

```
(venv) webapp $ set FLASK_DEBUG=1
(venv) webapp $ flask run
```

Returning a Complete Web Page

Returning a Complete Web Page

Mac OS X and Linux

```
(venv) webapp $ mkdir static
(venv) webapp $ mkdir templates
(venv) webapp $ mv ../styles.css static
(venv) webapp $ mv ../index.html templates
```

Windows

```
(venv) webapp $ mkdir static
(venv) webapp $ mkdir templates
(venv) webapp $ move ../styles.css static
(venv) webapp $ move ../index.html templates
```

Returning a Complete Web Page: app.py (step5)

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')
```

Returning a Complete Web Page: index.html (step6)

```
<!doctype html>
<html>
   <head>
       <title>My First Web Application</title>
       <link rel="stylesheet" href= "/static/styles.css" >
   </head>
   <body>
       <h1>Hello, user!</h1>
   </body>
</html>
```

Hands-On Exercise #4

Templates and Forms

Code: bit.ly/firststepswebdev

Templates

Templates: index.html (step7)

```
<!doctype html>
<html>
   <head> … </head>
   <body>
       {% if name %}
       <h1>Hello, {{ name }}!</h1>
       {% else %}
       <h1>Hello, user!</h1>
       {% endif %}
   </body>
</html>
```

Templates: app.py (step8)

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
   return render_template('index.html' , name='Miguel')
```

Web Forms

Web Forms: index.html (step9)

```
<!doctype html>
<html>
   <head> … </head>
   <body>
       {% if name %} … {% endif %}
       <form method="POST" action="">
           <p>Your name: <input type="text" name="name"></p>
           <p><input type="submit"></p>
       </form>
   </body>
</html>
```

Web Forms: app.py (step10)

```
from flask import Flask, render_template, request


app = Flask(__name__)


@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        return render_template('index.html', name=request.form['name'])
    return render_template('index.html')
```

Congrats!

You wrote your first web application!

Next Steps

- Take a more complete Python web development tutorial
  - Watch my video tutorials on web development with Flask
    (I suggest you start with the “Flask Workshop” tutorial I gave at PyCon US in 2015)
  - Flask Mega-Tutorial on my blog
  - Feel like trying something different? Find YouTube tutorials for your favorite framework!
- Learn a Database
- Learn JavaScript
- If you are on Windows, learn Unix
- Get a Raspberry Pi and set up a home web server for your personal projects
- And the most important: keep writing code!

Thanks!

Find these slides at

speakerdeck.com/miguelgrinberg

Q&A