Web Development for Data Scientists

Why Learn Web Development as a Data Scientist?

Most data science work involves exploring datasets, building models, and analyzing results. However, in many real-world situations, the value of your work increases significantly when others can interact with it. Web development provides a way to make your models and insights accessible through simple user interfaces or APIs.

Even a basic understanding of web technologies allows you to:

- Present your results beyond notebooks
- Create simple forms to collect user input
- Serve your machine learning models via a web application
- Share interactive visualizations or summaries

How Much Web Development is Enough?

As a data scientist, you do not need to become a full-stack web developer. Instead, it’s more practical to focus on a few key skills that complement your existing workflow:

- **HTML** to create and structure web pages
- **CSS** to control the appearance and layout
- **Flask (Python Framework)** to build lightweight web applications and APIs

These skills are sufficient for creating interfaces where users can interact with your models or view results dynamically.

A Practical Approach

This course introduces just enough web development to support your work as a data scientist. The goal is to help you understand how to connect the outputs of your models with simple, user-friendly web interfaces using tools you’re already comfortable with—primarily Python.

We will begin with foundational concepts like HTML and CSS, and then move to using Flask for building basic applications.

HTML for Data Scientists

What is HTML?

HTML (HyperText Markup Language) is the standard language used to define the structure of web pages. It tells the browser how to display content like text, forms, buttons, and tables.

As a data scientist, you can use HTML to:

- Create forms that take input from users
- Display model results on a page
- Structure information clearly

Basic HTML Elements

Page Structure

```html
<!DOCTYPE html>
<html>
<head>
  <title>My Web App</title>
</head>
<body>
  <!-- Content goes here -->
</body>
</html>
```

Headings and Paragraphs

```html
<h1>Model Results</h1>
<p>This page shows the output of a machine learning model.</p>
```

Forms and Inputs

```html
<form action="/predict" method="post">
  <label for="feature">Enter a value:</label>
  <input type="text" name="feature" id="feature">
  <button type="submit">Predict</button>
</form>
```

Displaying Tables

```html
<table>
  <tr>
    <th>Feature</th>
    <th>Value</th>
  </tr>
  <tr>
    <td>Age</td>
    <td>29</td>
  </tr>
</table>
```

Summary

You only need to learn a small set of HTML tags to create basic interfaces:

- Structural: `<html>`, `<head>`, `<body>`
- Text: `<h1>` to `<h6>`, `<p>`, `<span>`
- Forms: `<form>`, `<input>`, `<button>`
- Tables: `<table>`, `<tr>`, `<td>`, `<th>`

This is enough to build simple pages that collect input and display results. You will connect these pages with Python using Flask in later lessons.

CSS for Data Scientists

What is CSS?

CSS (Cascading Style Sheets) is used to control the appearance of HTML elements. It allows you to change fonts, colors, spacing, and layout.

For data scientists, CSS helps make basic web interfaces more readable and user-friendly. Even a few lines of CSS can make your pages easier to use.

Ways to Use CSS

Inline CSS (directly in the HTML tag)

```
<p style="color: green;">Prediction Successful</p>
```

Internal CSS (inside a `<style>` block)

```
<head>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 40px;
    }
    input {
      padding: 5px;
      margin-bottom: 10px;
    }
    button {
      background-color: #007BFF;
      color: white;
      border: none;
```

```
        padding: 8px 12px;
      }
    </style>
  </head>
```

Common CSS Properties

Property	What it does
`color`	Text color
`background`	Background color
`margin`	Space outside the element
`padding`	Space inside the element
`font-size`	Size of the text
`border`	Adds a border around elements
`text-align`	Aligns text (left, center, right)

Example

```
<body>
  <h1>Prediction Result</h1>
  <form>
    <input type="text" placeholder="Enter value">
    <button type="submit">Submit</button>
  </form>
</body>
```

With the internal CSS above, this form will have padding and better spacing.

Summary

You don’t need to master CSS as a data scientist. Knowing how to:

- Set padding and margins
- Style buttons and inputs
- Control font and color

is often enough to make your tools more usable. We will now move on to connecting everything using Python and Flask.

Introduction to Flask Framework**

Why Flask?

- Lightweight Python web framework
- Ideal for serving models and exposing APIs

Installation

```
pip install flask
```

Demo App

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return 'Hello, Data Scientist!'

if __name__ == '__main__':
    app.run(debug=True)
```

Running the Server

```
python app.py
```

Then go to `http://127.0.0.1:5000` in your browser.

Query Parameters in Flask

What Are Query Parameters?

Query parameters are key-value pairs passed in the URL after a `?`. For example:

```
/predict?x=10\&y=20
```

In this case: - `x` has the value `10` - `y` has the value `20`

They are often used to: - Pass values to your route - Trigger filtering or model predictions without using a form

Accessing Query Parameters in Flask

Use `request.args` to access query parameters:

```python
from flask import Flask, request

app = Flask(__name__)

@app.route('/predict')
def predict():
    x = request.args.get('x')
    y = request.args.get('y')
    return f"x = {x}, y = {y}"
```

Example

Visiting this URL:

```
http://localhost:5000/predict?x=12&y=7
```

Will return:

```
x = 12, y = 7
```

Handling Missing Values

You can provide a default value:

```
x = request.args.get('x', default=0)
```

Or use an `if` statement to check if the parameter exists:

```
if 'x' in request.args:
    x = request.args['x']
```

Use Case Example

You might want to use query parameters to trigger a model prediction:

```
@app.route('/predict')
def predict():
    feature = request.args.get('value')
    if not feature:
        return "No value provided"
    # prediction = model.predict([feature])  # pseudocode
    return f"Prediction result for input {feature}"
```

Summary

- Query parameters are passed using `?key=value` in the URL

- Use `request.args.get('key')` to retrieve them
- Useful for triggering actions or passing values without forms

Serving Static Files in Flask

What Are Static Files?

Static files are files that don't change during execution. These typically include:

- CSS files
- JavaScript files
- Images and other media

Flask serves static files by default from a folder named `static`, and they are accessible via the `/static/` URL path.

Default Project Structure

```
my_app/
├── app.py
├── static/
│   └── style.css
├── templates/
│   └── index.html
```

Referencing Static Files in HTML

In your HTML templates (inside the `templates/` folder), use Flask's `url_for()` function to correctly link to static assets:

```
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
```

This will generate `/static/style.css` by default.

Changing the Static Folder and URL Path

If you want Flask to serve static files from a different folder or under a different URL, use these options when creating your Flask app:

```
from flask import Flask

app = Flask(
    __name__,
    static_folder='assets',           # Physical folder on disk
    static_url_path='/files'          # URL path to access those files
)
```

In this setup:

- Flask will **look for files in the `assets/` folder**
- Users will **access files via `/files/...`** in the browser

Example:

```
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
```

Generates:

```
/files/style.css
```

Summary

- `static_folder` : changes the folder Flask looks into for static files (default is `static/`)
- `static_url_path` : changes the URL path used to serve those static files (default is `/static`)
- Use `url_for('static', filename='...')` to refer to static files regardless of the folder or path

Handling Forms in Flask

Why Use Forms?

Forms allow users to input values that can be processed by your Flask app. This is useful for: - Passing feature values to a machine learning model - Uploading data - Triggering analysis

Creating a Simple HTML Form

Here is a basic HTML form that accepts a single input:

```html
<form action="/predict" method="post">
  <input type="text" name="feature1" placeholder="Enter a value">
  <button type="submit">Submit</button>
</form>
```

- `action="/predict"` : the form will send data to the `/predict` route
- `method="post"` : the form uses POST request to send data

Handling Form Data in Flask

In your `app.py` , set up a route to receive and process the form data:

```python
from flask import Flask, request, render_template

app = Flask(__name__)

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

```
@app.route('/predict', methods=['POST'])
def predict():
    value = request.form['feature1']
    return f"Received input: {value}"
```

Explanation:

- `request.form` is used to access submitted form data
- The key `'feature1'` corresponds to the `name` attribute in the HTML input

Example File Structure

```
my_app/
├── app.py
├── templates/
│   └── form.html
```

form.html Template

```
<!DOCTYPE html>
<html>
<head>
  <title>Prediction Input</title>
</head>
<body>
  <h2>Enter Input</h2>
  <form action="/predict" method="post">
    <input type="text" name="feature1">
    <button type="submit">Submit</button>
  </form>
</body>
</html>
```

Summary

- Use `<form>` in your HTML to collect user input
- Use `request.form['input_name']` in your Flask route to read submitted data
- Forms enable dynamic interaction with your web app, such as sending input to a model or triggering analysis

Jinja2 Templating in Flask

What is Jinja2?

Jinja2 is the templating engine used by Flask. It allows you to embed Python-like logic inside your HTML templates. This is useful when you want to:

- Display variables (like model predictions or user input)
- Loop through lists (like rows in a DataFrame)
- Use conditions (like showing a message only when needed)

Passing Variables from Flask to Templates

In your Flask route, use `render_template()` and pass variables like this:

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    result = "Prediction: 42"
    return render_template('index.html', prediction=result)
```

Here, `prediction` is a variable you can use inside `index.html`.

Using Variables in HTML

Inside your HTML template (`index.html`), use double curly braces `{{ }}` to display variables:

```html
<!DOCTYPE html>
<html>
<head>
  <title>Result</title>
</head>
<body>
  <h1>Model Output</h1>
  <p>{{ prediction }}</p>
</body>
</html>
```

Control Structures

If Statements

```
{% if prediction %}
  <p>Result: {{ prediction }}</p>
{% else %}
  <p>No result available.</p>
{% endif %}
```

For Loops

You can loop through a list or dictionary:

```
<ul>
  {% for item in items %}
    <li>{{ item }}</li>
  {% endfor %}
</ul>
```

In your Flask route:

```
@app.route('/list')
def show_list():
    return render_template('list.html', items=['Apple', 'Banana', 'Cherry'])
```

Escaping Output

Jinja2 automatically escapes variables to prevent HTML injection. If you trust the variable and want to allow raw HTML, use `|safe`:

```
<p>{{ some_html | safe }}</p>
```

Summary

- Use `render_template()` to pass variables from Flask to HTML
- Use `{{ variable }}` to display data
- Use `{% ... %}` for control structures like loops and conditionals
- Jinja2 makes it easy to combine logic and layout for dynamic pages

Template Inheritance in Flask (Jinja2)

What is Template Inheritance?

When building web apps, many pages share a common layout — for example, a header, footer, or navigation bar. Instead of repeating this code in every file, Jinja2 allows you to define a **base template** that other templates can extend.

This keeps your code cleaner and easier to maintain.

Step 1: Create a Base Template

Create a file called `base.html` inside your `templates/` folder.

```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}My App{% endblock %}</title>
</head>
<body>
  <header>
    <h1>My Flask App</h1>
  </header>

  <main>
    {% block content %}{% endblock %}
  </main>
</body>
</html>
```

- `{% block title %}` and `{% block content %}` are placeholders that child templates can fill.

Step 2: Extend the Base Template

Create another HTML file, e.g. `index.html`, that extends the base template:

```
{% extends "base.html" %}

{% block title %}Home Page{% endblock %}

{% block content %}
  <p>Welcome to the homepage.</p>
{% endblock %}
```

Flask Route Example

```
from flask import Flask, render_template

app = Flask(__name__)

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

Folder Structure

```
my_app/
├── app.py
├── templates/
│   ├── base.html
│   └── index.html
```

Summary

- Create a `base.html` with `{% block %}` sections

- Use `{% extends "base.html" %}` in child templates
- This avoids repetition and keeps your HTML organized

Template inheritance is useful as your app grows and you start creating multiple pages that share the same layout.

Message Flashing in Flask

What Is Flashing?

Flashing is a way to send temporary messages from the backend (Flask) to the frontend (HTML). These messages are usually used for:

- Status updates (e.g., “Prediction complete”)
- Error messages (e.g., “Invalid input”)
- Notifications (e.g., “File uploaded successfully”)

Flashed messages are stored in the session and automatically cleared after being displayed.

Step 1: Set a Secret Key

Flashing uses Flask’s session, so you must set a secret key:

```
from flask import Flask

app = Flask(__name__)
app.secret_key = 'your_secret_key'
```

Step 2: Flash a Message in Your Route

Use `flash()` to send a message:

```
from flask import flash, redirect, render_template, request

@app.route('/predict', methods=['POST'])
def predict():
    feature = request.form.get('feature1')
    if not feature:
        flash('Please enter a value')
        return redirect('/')

    # process prediction here
    flash('Prediction complete')
    return redirect('/')
```

- `flash('message')` stores the message
- `redirect('/')` sends the user back to a route where the message will be displayed

Step 3: Display Flashed Messages in the Template

In your base or main template (e.g. `index.html` or `base.html`):

```
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul>
      {% for msg in messages %}
        <li>{{ msg }}</li>
      {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

This block retrieves all flashed messages and displays them in a list. You can style them with CSS as needed.

Summary

- Use `flash('your message')` to send a message
- Use `get_flashed_messages()` in your template to retrieve and display them
- Useful for alerts, validation feedback, and user notifications

Creating an API Using `jsonify` in Flask

What Is an API?

An API (Application Programming Interface) allows other programs (like a frontend, script, or another server) to communicate with your Flask app using structured data—typically JSON.

This is useful when: - You want to expose your ML model as a service - You want other tools to send input and receive predictions

What Is `jsonify`?

Flask's `jsonify()` function converts Python dictionaries or lists into properly formatted JSON responses.

Example: Basic API Endpoint

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/api/predict', methods=['GET'])
def predict():
    value = request.args.get('value')

    if value is None:
        return jsonify({'error': 'Missing input'}), 400

    # Simulated prediction (replace with actual model call)
    result = int(value) * 2
    return jsonify({
        'input': value,
        'prediction': result
    })
```

How to Call It

Visit this in your browser or use a tool like Postman or `curl`:

```
http://localhost:5000/api/predict?value=7
```

Response:

```
{
  "input": "7",
  "prediction": 14
}
```

Returning Error Codes

You can return custom status codes with `jsonify()`:

```
return jsonify({'error': 'Invalid input'}), 400
```

This helps the client understand what went wrong.

Summary

- Use `jsonify()` to return JSON from your Flask routes
- Ideal for serving ML model outputs or exposing data
- Combine with `request.args` (GET) or `request.json` (POST) for flexible input handling