JavaScript – Yet Another Language, for the Browser

Databases

Pierre Senellart

PSL UNIVERSITÉ PARIS

10 June 2026

Outline

Where we are

We can already, in Python:

- query a database with SQL or pandas
- serve HTML pages with Flask, generated from the data
- handle forms, redirects, sessions

Where we are

We can already, in Python:

- query a database with SQL or pandas
- serve HTML pages with Flask, generated from the data
- handle forms, redirects, sessions

Two limitations of the pages we have built so far:

- every interaction costs a full HTTP round-trip: each click reloads a page
- no behavior inside the page (no chart that auto-refreshes, no autocomplete, no live validation)

The third Web language

Recall from the HTML lecture:

HTML	structure and content
CSS	style
JavaScript	behavior, on the client side

The third Web language

Recall from the HTML lecture:

HTML	structure and content
CSS	style
JavaScript	behavior, on the client side

Today: a short tour of JavaScript, contrasted with Python, and the two browser APIs you need most as a data person:

- the DOM (to read and modify the page in place)
- `fetch` (to talk back to your server without reloading)

A meta-objective

You already program in Python, C, and C++ (and mark up pages with HTML and CSS). JavaScript is just one more. The goal is not to make you a JavaScript expert; it is to remind you that picking up a new language is:

- much faster than learning to program in the first place (concepts transfer; syntax is mostly skin-deep)
- the normal state of affairs for a software professional
- easier today than ever: the documentation (MDN Web Docs) is one search away, and your editor already knows the language

A meta-objective

You already program in Python, C, and C++ (and mark up pages with HTML and CSS). JavaScript is just one more. The goal is not to make you a JavaScript expert; it is to remind you that picking up a new language is:

- much faster than learning to program in the first place (concepts transfer; syntax is mostly skin-deep)
- the normal state of affairs for a software professional
- easier today than ever: the documentation (MDN Web Docs) is one search away, and your editor already knows the language

Tomorrow it will be Rust, SQL dialect n, Julia, or whatever pays the bills. Do not be afraid.

Outline

A brief history

- Created by Brendan Eich at Netscape in 1995, in ten days, to add behavior to Web pages
- Standardised as ECMAScript (ECMA-262) since 1997
- Despite the name, has nothing to do with the Java language: a 1990s marketing decision
- Has grown enormously, mostly compatibly, with major modernisations in ES2015 (“ES6”) and yearly releases since
- Today runs in every browser, on servers (Node.js, Deno, Bun), in databases (PostgreSQL plv8...), even in spreadsheets

Where the JavaScript runs

in the browser (today's focus) the user's machine runs the code; the server only ships it. Use cases: interactive pages, animations, forms, fetching data.

on the server (Node.js) for situations where one team wants the same language across the stack

Where the JavaScript runs

in the browser (today's focus) the user's machine runs the code; the server only ships it. Use cases: interactive pages, animations, forms, fetching data.

on the server (Node.js) for situations where one team wants the same language across the stack

In our hotel app, JavaScript runs after the Flask response has arrived in the browser. The server's job is unchanged.

Recap: the four languages, side by side

The axes from Programming 1, with a column for JavaScript:

	C	C++	Python	JavaScript
Execution	compiled	compiled	bytecode	JIT-compiled
Typing	static, strong	static, strong	dynamic, weak	dynamic, weak
Conversion	numeric	numeric	minimal	pervasive
Memory	manual	manual	automatic (GC)	automatic (GC)
Parameters	value, address	+ reference	object reference	object reference
Paradigms	imperative	imp., OOP, gen.	imp., func., OOP	imp., func., OOP, event
Objects	none	classes	classes	prototypes
Level	low	low + high	high	high
Home turf	systems, OS	performance	data science	the browser

What is new (1/2): how the code runs

The four highlighted cells of the table have no real equivalent in Python, C, or C++. First, how the code runs:

JIT compilation no separate build step, yet not a plain interpreter: the engine compiles to machine code on the fly and re-optimises the hot paths as the page runs (unlike Python's bytecode interpretation or C's ahead-of-time build). In practice, close to native speed.

Event-driven a script is not a run-to-completion program: once loaded, it sits idle and reacts to events – a click, a key, a network reply, a timer. A single-threaded event loop dispatches them one at a time (more when we reach `fetch`).

What is new (2/2): objects and values

Then, how it treats objects and values:

Prototype-based objects an object inherits from another object (its prototype), not from a class. There are no classes underneath; the `class` keyword (ES2015) is syntactic sugar over prototypes.

Implicit conversion between types – no explicit *cast* needed, unlike C: `1 + "1"` is `"11"`, `"3" * 2` is 6. Handy, but the root of the == trap, a few slides on. Prefer ===.

What is new (2/2): objects and values

Then, how it treats objects and values:

Prototype-based objects an object inherits from another object (its prototype), not from a class. There are no classes underneath; the `class` keyword (ES2015) is syntactic sugar over prototypes.

Implicit conversion between types – no explicit *cast* needed, unlike C: `1 + "1"` is `"11"`, `"3" * 2` is 6. Handy, but the root of the == trap, a few slides on. Prefer ===.

We will not lean on these four traits today, but you will meet them as soon as you read real JavaScript code.

Outline

Hello, browser

Put this in `hello.html` and open it in any browser:

HTML

```
<!DOCTYPE html>
<html>
  <body>
    <h1 id="greet">Loading...</h1>
    <script>
      const name = "world";
      document.getElementById("greet").textContent = `Hello, ${name}!`;
    </script>
  </body>
</html>
```

Hello, browser

Put this in `hello.html` and open it in any browser:

HTML

```
<!DOCTYPE html>
<html>
  <body>
    <h1 id="greet">Loading...</h1>
    <script>
      const name = "world";
      document.getElementById("greet").textContent = `Hello, ${name}!`;
    </script>
  </body>
</html>
```

Three things to notice:

- the JavaScript lives inside a `<script>` tag
- it runs as the page loads, top to bottom
- it modifies the page through `document`

Where you actually run JS

Browser console: the developer tools (F12) have a Console tab. Type any expression and see its value:

JavaScript

```
> 2 + 2
4
> "hello".toUpperCase()
"HELLO"
```

This is to JavaScript what the interactive interpreter (the REPL, read–eval–print loop) is to Python; use it early and often.

Where you actually run JS

Browser console: the developer tools (F12) have a Console tab. Type any expression and see its value:

JavaScript

```
> 2 + 2
4
> "hello".toUpperCase()
"HELLO"
```

This is to JavaScript what the interactive interpreter (the REPL, read–eval–print loop) is to Python; use it early and often.

Inside a script:

JavaScript

```
console.log("hello");      // appears in the Console tab
console.table(rows);       // pretty-printed for arrays of objects
```

Outline

Declaring variables

Three keywords, only two of them in modern code:

JavaScript

```
const PI = 3.1416;        // never reassigned
let counter = 0;          // reassignable
counter = counter + 1;

var legacy = "avoid";     // pre-2015, weird scoping, do not use
```

Declaring variables

Three keywords, only two of them in modern code:

JavaScript

```
const PI = 3.1416;        // never reassigned
let counter = 0;          // reassignable
counter = counter + 1;

var legacy = "avoid";     // pre-2015, weird scoping, do not use
```

`const` default choice; the variable cannot be *reassigned* (the object it points to still can)

`let` only when you really need to reassign

`var` legacy, with error-prone scoping rules; never write this in new code

Style: declare close to first use, prefer `const`.

Primitive types

Unlike Python or C, integers and reals share a single `number` type:

`number`	one type, IEEE-754 64-bit float (also used for integers up to 2^{53})
`bigint`	arbitrary-precision integer: `12345678901234567890n`
`string`	`"hi"`, `'hi'`, or backtick template literals with `` `${expr}` ``
`boolean`	`true`, `false`
`null`	explicit "no value"
`undefined`	implicit "no value" (unassigned, missing property, missing argument)
`symbol`	unique identifiers (rare)

Primitive types

Unlike Python or C, integers and reals share a single `number` type:

`number`	one type, IEEE-754 64-bit float (also used for integers up to 2^{53})
`bigint`	arbitrary-precision integer: `12345678901234567890n`
`string`	`"hi"`, `'hi'`, or backtick template literals with `` `${expr}` ``
`boolean`	`true`, `false`
`null`	explicit "no value"
`undefined`	implicit "no value" (unassigned, missing property, missing argument)
`symbol`	unique identifiers (rare)

Everything else is an object (including arrays, functions, dates, regexes...).

The equality trap: == vs ===

Use === and !==, always.

JavaScript

```
1 == "1"          // true   (converts types)
1 === "1"         // false  (strict)
null == undefined  // true
null === undefined // false
NaN === NaN       // false  (NaN is not equal to itself)
```

The equality trap: == vs ===

Use === and !==, always.

JavaScript

```
1 == "1"          // true   (converts types)
1 === "1"         // false  (strict)
null == undefined  // true
null === undefined // false
NaN === NaN       // false  (NaN is not equal to itself)
```

== was meant to ease comparing the strings the web hands you (form fields, URL parameters) against numbers; its conversion rules turned out incoherent, so the safe default is strict equality.

Truthy and falsy

Like Python's `if value:`, JS lets non-boolean values steer conditionals. The falsy values are:

`false`, `0`, `0n`, `""`, `null`, `undefined`, `NaN`

Truthy and falsy

Like Python's `if value:`, JS lets non-boolean values steer conditionals. The falsy values are:

`false`, `0`, `0n`, `""`, `null`, `undefined`, `NaN`

Everything else is truthy, including the empty array and the empty object (different from Python!):

JavaScript

```
if ([])     console.log("yes");   // prints
if ({})     console.log("yes");   // prints
if (" ")    console.log("yes");   // prints (non-empty string)
```

Outline

Functions

Two equivalent forms:

JavaScript

```
function add(a, b) {
  return a + b;
}

const add = (a, b) => a + b;            // arrow, single expression

const greet = name => `Hi, ${name}`;    // one param, no parens
```

Functions

Two equivalent forms:

JavaScript

```
function add(a, b) {
  return a + b;
}

const add = (a, b) => a + b;            // arrow, single expression

const greet = name => `Hi, ${name}`;    // one param, no parens
```

- Arrow functions are JavaScript's lambdas, like Python's `lambda` or C++ lambdas: concise, and with a less surprising `this` (we will not need `this` today)
- Default values: `function f(x = 0) {...}`
- All functions are first-class values: stored in a variable, passed as arguments, returned from other functions

Control flow

Mostly C-like, mostly familiar:

JavaScript

```
if (x > 0) { ... } else if (x === 0) { ... } else { ... }

for (let i = 0; i < n; i++) { ... }
for (const item of rooms)  { ... }   // rooms: an array;
                                     // iterates values
for (const key in obj)     { ... }   // obj: an object;
                                     // iterates keys (rare)

while (cond) { ... }
do { ... } while (cond);
```

Control flow

Mostly C-like, mostly familiar:

JavaScript

```
if (x > 0) { ... } else if (x === 0) { ... } else { ... }

for (let i = 0; i < n; i++) { ... }
for (const item of rooms)  { ... }   // rooms: an array;
                                     // iterates values
for (const key in obj)     { ... }   // obj: an object;
                                     // iterates keys (rare)

while (cond) { ... }
do { ... } while (cond);
```

Rule of thumb: `for (... of ...)` for arrays, `Object.entries(obj)` when you really need both keys and values.

Control flow

Mostly C-like, mostly familiar:

JavaScript

```
if (x > 0) { ... } else if (x === 0) { ... } else { ... }

for (let i = 0; i < n; i++) { ... }
for (const item of rooms)  { ... }   // rooms: an array;
                                     // iterates values
for (const key in obj)     { ... }   // obj: an object;
                                     // iterates keys (rare)

while (cond) { ... }
do { ... } while (cond);
```

Rule of thumb: `for (... of ...)` for arrays, `Object.entries(obj)` when you really need both keys and values.

Why `const`? `item` takes a new value each pass, yet `const` is right: each iteration introduces a fresh `item`, never reassigned within a single pass.

Outline

Arrays

JavaScript

```
const rooms = [504, 107, 302];
rooms.length;              // 3
rooms[0];                  // 504
rooms.push(401);           // append
rooms.includes(107);       // true

const big = rooms.filter(r => r > 200);
const ids = rooms.map(r => `Room ${r}`);
const sum = rooms.reduce((acc, r) => acc + r, 0);
```

Arrays

JavaScript

```
const rooms = [504, 107, 302];
rooms.length;              // 3
rooms[0];                  // 504
rooms.push(401);           // append
rooms.includes(107);       // true

const big = rooms.filter(r => r > 200);
const ids = rooms.map(r => `Room ${r}`);
const sum = rooms.reduce((acc, r) => acc + r, 0);
```

The same higher-order functions you saw in functional programming (Python's `map`/`filter`/`reduce`, C++'s `std::transform`/`accumulate`); here the callbacks are just arrow functions.

Objects

JavaScript

```
const room = {
  id: 504,
  capacity: 2,
  price: 120.0,
};

room.id;            // 504  -- dot access
room["price"];      // 120  -- bracket access (for dynamic keys)
room.view = "sea";  // adds a property

const { id, capacity } = room;       // destructuring
const room2 = { ...room, id: 505 };  // spread (shallow copy)
```

Objects

JavaScript

```
const room = {
  id: 504,
  capacity: 2,
  price: 120.0,
};

room.id;            // 504  -- dot access
room["price"];      // 120  -- bracket access (for dynamic keys)
room.view = "sea";  // adds a property

const { id, capacity } = room;       // destructuring
const room2 = { ...room, id: 505 };  // spread (shallow copy)
```

Keys are always strings (or symbols); a value can be anything: a number, a string, an array, a function, or another object. The closest Python analogue is the `dict`, but with dot access and a different literal syntax.

JSON: the wire format of the Web

JSON (JavaScript Object Notation) is JavaScript-object syntax, restricted to data:

JavaScript

```
const text = '{"id": 504, "capacity": 2}';
const obj  = JSON.parse(text);       // string -> object
const back = JSON.stringify(obj);    // object -> string
```

- No comments, no trailing commas, keys must be double-quoted
- Only six value kinds: object, array, string, number, boolean, `null`
- Python: `json.loads` / `json.dumps` are the same idea

JSON: the wire format of the Web

JSON (JavaScript Object Notation) is JavaScript-object syntax, restricted to data:

JavaScript

```
const text = '{"id": 504, "capacity": 2}';
const obj  = JSON.parse(text);       // string -> object
const back = JSON.stringify(obj);    // object -> string
```

- No comments, no trailing commas, keys must be double-quoted
- Only six value kinds: object, array, string, number, boolean, `null`
- Python: `json.loads` / `json.dumps` are the same idea

Though born from JavaScript, JSON is now the universal data-exchange format: Web APIs, configuration files, logs, and programs written in any language. We will use it with `fetch` in a few slides.

Outline

Three ways to include JS

1. External file (recommended):

HTML

```
<script src="/static/app.js"></script>
```

Three ways to include JS

1. External file (recommended):

HTML

```
<script src="/static/app.js"></script>
```

2. Inline, inside the page:

HTML

```
<script>
  console.log("hello");
</script>
```

Three ways to include JS

1. External file (recommended):

HTML

```
<script src="/static/app.js"></script>
```

2. Inline, inside the page:

HTML

```
<script>
  console.log("hello");
</script>
```

3. In an attribute (avoid):

HTML

```
<button onclick="alert('hi')">Click</button>
```

When does the script run?

By default, the browser pauses parsing the HTML to download and run the script. That is rarely what you want:

`<script defer src="...">` download in parallel, run after the HTML is fully parsed, in source order. Recommended.

`<script async src="...">` download in parallel, run as soon as ready, possibly out of order. For independent third-party scripts (analytics...).

When does the script run?

By default, the browser pauses parsing the HTML to download and run the script. That is rarely what you want:

`<script defer src="...">` download in parallel, run after the HTML is fully parsed, in source order. Recommended.

`<script async src="...">` download in parallel, run as soon as ready, possibly out of order. For independent third-party scripts (analytics...).

Put scripts in `<head>` with `defer`, not at the end of `<body>` as the old advice used to say.

The browser, asynchronously

JavaScript in a browser is single-threaded and event-driven: the engine processes a queue of tasks, one at a time, never blocking on I/O.

The browser, asynchronously

JavaScript in a browser is single-threaded and event-driven: the engine processes a queue of tasks, one at a time, never blocking on I/O.

Consequence: long-running computations freeze the page. Network and timer code is built on callbacks, promises and async/await (next section). There is nothing like Python's `time.sleep` that blocks the rest of the program; use `setTimeout` instead.

Outline

Selecting elements

`document` is the entry point to the page tree (the DOM, lecture 10):

JavaScript

```
const title = document.querySelector("h1");
const items = document.querySelectorAll("ul.menu > li");
const byId  = document.getElementById("submit");
```

Selecting elements

`document` is the entry point to the page tree (the DOM, lecture 10):

JavaScript

```
const title = document.querySelector("h1");
const items = document.querySelectorAll("ul.menu > li");
const byId  = document.getElementById("submit");
```

- `querySelector` takes any CSS selector you already know: type, class, id, combinators, attributes
- Returns the first match, or `null`
- `querySelectorAll` returns a NodeList (iterable; use `for (... of ...)`)

Reading and changing content

JavaScript

```
const greet = document.querySelector("#greet");

greet.textContent = "Hello!";         // safe: plain text
greet.innerHTML   = "<em>Hi</em>";    // PARSES HTML -- careful

greet.classList.add("highlighted");
greet.classList.toggle("active");

greet.setAttribute("data-room", "504");
```

Reading and changing content

JavaScript

```
const greet = document.querySelector("#greet");

greet.textContent = "Hello!";         // safe: plain text
greet.innerHTML   = "<em>Hi</em>";    // PARSES HTML -- careful

greet.classList.add("highlighted");
greet.classList.toggle("active");

greet.setAttribute("data-room", "504");
```

Security: `innerHTML` parses the assigned string as HTML. Combined with user-supplied data, this is the JavaScript twin of `|safe` in Jinja: a textbook XSS hole. Prefer `textContent`, and build elements with the DOM API when you need structure.

Creating elements

JavaScript

```
const ul = document.querySelector("#rooms");

for (const room of rooms) {
  const li = document.createElement("li");
  li.textContent = `Room ${room.id} -- ${room.capacity} guests`;
  ul.appendChild(li);
}
```

Creating elements

JavaScript

```
const ul = document.querySelector("#rooms");

for (const room of rooms) {
  const li = document.createElement("li");
  li.textContent = `Room ${room.id} -- ${room.capacity} guests`;
  ul.appendChild(li);
}
```

This pattern, fetched data → created DOM nodes, is the core of every interactive table or list on the Web.

Listening to events

JavaScript

```
const button = document.querySelector("#book");

button.addEventListener("click", () => {
  console.log("Book button was clicked!");
});

document.querySelector("form").addEventListener("submit", e => {
  e.preventDefault();           // do NOT submit and reload
  // ... handle the form here, e.g. with fetch ...
});
```

Listening to events

JavaScript

```
const button = document.querySelector("#book");

button.addEventListener("click", () => {
  console.log("Book button was clicked!");
});

document.querySelector("form").addEventListener("submit", e => {
  e.preventDefault();            // do NOT submit and reload
  // ... handle the form here, e.g. with fetch ...
});
```

- Pass a function; the browser calls it when the event fires
- The argument (`e`) is the event object: target, coordinates, key pressed…
- `e.preventDefault()` cancels the browser's default behavior (form submit, link navigation)

Outline

The problem: I/O is asynchronous

A network request takes tens or hundreds of milliseconds. We cannot freeze the page while we wait.

The problem: I/O is asynchronous

A network request takes tens or hundreds of milliseconds. We cannot freeze the page while we wait.

JavaScript represents "a value that will be available later" as a promise:

JavaScript

```
const p = fetch("/api/rooms");   // returns a promise immediately

p.then(response => {
  console.log("got", response.status);
});
```

async/await: the readable form

Chained `.then` calls quickly become unreadable. Modern code uses `async` / `await`:

JavaScript

```
async function loadRooms() {
  const response = await fetch("/api/rooms");
  if (!response.ok) {
    throw new Error(`HTTP ${response.status}`);
  }
  const rooms = await response.json();
  return rooms;
}
```

async/await: the readable form

Chained `.then` calls quickly become unreadable. Modern code uses `async` / `await`:

JavaScript

```
async function loadRooms() {
  const response = await fetch("/api/rooms");
  if (!response.ok) {
    throw new Error(`HTTP ${response.status}`);
  }
  const rooms = await response.json();
  return rooms;
}
```

- Any function declared `async` returns a promise
- `await` pauses the function (not the page) until the promise resolves

Outline

GET

JavaScript

```
async function getRooms() {
  const r = await fetch("/api/rooms");
  if (!r.ok) throw new Error(`HTTP ${r.status}`);
  return await r.json();          // parses the JSON body
}
```

GET

JavaScript

```
async function getRooms() {
  const r = await fetch("/api/rooms");
  if (!r.ok) throw new Error(`HTTP ${r.status}`);
  return await r.json();          // parses the JSON body
}
```

- Returns a `Response` object whose body can be read as JSON, text, a blob…
- `r.ok` is true for status codes 200–299
- Network failures (DNS, offline, CORS) make the `await` throw an error; HTTP errors (404, 500) do not, so you must check `r.ok` yourself

POST with a JSON body

JavaScript

```
async function book(roomId, guest, arrival, nights) {
  const r = await fetch("/api/book", {
    method:  "POST",
    headers: { "Content-Type": "application/json" },
    body:    JSON.stringify({ roomId, guest, arrival, nights }),
  });
  if (!r.ok) throw new Error(await r.text());
  return await r.json();
}
```

POST with a JSON body

JavaScript

```
async function book(roomId, guest, arrival, nights) {
  const r = await fetch("/api/book", {
    method:  "POST",
    headers: { "Content-Type": "application/json" },
    body:    JSON.stringify({ roomId, guest, arrival, nights }),
  });
  if (!r.ok) throw new Error(await r.text());
  return await r.json();
}
```

Note `{ roomId, guest, arrival, nights }`: the shorthand when the property name equals the variable name. Equivalent to `{ roomId: roomId, ... }`.

The Flask side, in two lines

A JSON endpoint, with the database knowledge of lecture 13:

Python

```python
from datetime import date
from flask import jsonify, request

@app.route("/api/rooms")
def api_rooms():
    rooms = db.session.scalars(db.select(Room)).all()
    return jsonify([{"id": r.id, "capacity": r.capacity}
                    for r in rooms])

@app.route("/api/book", methods=["POST"])
def api_book():
    data = request.get_json()
    r = Reservation(room_id=data["roomId"],
                    guest=Guest(name=data["guest"]),
                    arrival=date.fromisoformat(data["arrival"]),
                    nights=data["nights"])
```

The Flask side, in two lines

A JSON endpoint, with the database knowledge of lecture 13:

Python

```python
from datetime import date
from flask import jsonify, request

@app.route("/api/rooms")
def api_rooms():
    rooms = db.session.scalars(db.select(Room)).all()
    return jsonify([{"id": r.id, "capacity": r.capacity}
                    for r in rooms])

@app.route("/api/book", methods=["POST"])
def api_book():
    data = request.get_json()
    r = Reservation(room_id=data["roomId"],
                    guest=Guest(name=data["guest"]),
                    arrival=date.fromisoformat(data["arrival"]),
                    nights=data["nights"])
```

Same-origin and CORS

Browsers refuse, by default, to let a page on `a.com` call `fetch` on `b.com`. This is the Same-Origin Policy, the first line of defense against malicious sites reading your session.

Same-origin and CORS

Browsers refuse, by default, to let a page on `a.com` call `fetch` on `b.com`. This is the Same-Origin Policy, the first line of defense against malicious sites reading your session.

- Same origin: same scheme, host *and* port; everything else is cross-origin
- A server can opt in to cross-origin access by replying with the `Access-Control-Allow-Origin` header (CORS)
- In Flask, the `flask-cors` extension does this
- For our hotel app, server and browser share the origin, so CORS is not an issue

Outline

The HTML shell

templates/live.html:

HTML

```
{% extends "base.html" %}
{% block content %}
  <h2>Live reservations</h2>

  <form id="search">
    <input name="q" placeholder="Guest name..." autocomplete="off">
  </form>

  <table>
    <thead><tr><th>Guest</th><th>Room</th><th>Arrival</th></tr></thead>
    <tbody id="rows"><!-- filled by JS --></tbody>
  </table>

  <script defer src="{{ url_for('static',
                                filename='live.js') }}"></script>
{% endblock %}
```

The JavaScript

JavaScript

```
const form = document.querySelector("#search");
const tbody = document.querySelector("#rows");
async function refresh(q = "") {
  const r = await fetch(`/api/reservations?q=${encodeURIComponent(q)}`);
  const rows = await r.json();
  tbody.replaceChildren();                    // clear existing rows
  for (const row of rows) {
    const tr = document.createElement("tr");
    for (const cell of [row.guest, row.room, row.arrival]) {
      const td = document.createElement("td");
      td.textContent = cell;                  // safe (no XSS)
      tr.appendChild(td);
    }
    tbody.appendChild(tr);
  }
}
form.addEventListener("input", e => refresh(e.target.value));
refresh();                                    // initial load
```

The Flask endpoint

Python

```python
@app.route("/api/reservations")
def api_reservations():
    q = request.args.get("q", "").strip()
    query = db.select(Reservation).join(Guest)
    if q:
        query = query.where(Guest.name.icontains(q))
    rows = db.session.scalars(query.order_by(Reservation.arrival)).all()
    return jsonify([
        {"guest": r.guest.name, "room": r.room_id,
         "arrival": r.arrival.isoformat()}
        for r in rows
    ])
```

The Flask endpoint

Python

```python
@app.route("/api/reservations")
def api_reservations():
    q = request.args.get("q", "").strip()
    query = db.select(Reservation).join(Guest)
    if q:
        query = query.where(Guest.name.icontains(q))
    rows = db.session.scalars(query.order_by(Reservation.arrival)).all()
    return jsonify([
        {"guest": r.guest.name, "room": r.room_id,
         "arrival": r.arrival.isoformat()}
        for r in rows
    ])
```

As the user types in the search box, the page issues a fresh request per keystroke; the server returns matching reservations as JSON; JavaScript redraws the table. No page reload.

What we just built

- A small single-page interaction layered on the existing Flask app
- HTML for the shell, CSS for the look, JavaScript for the behavior, SQL behind the scenes
- A clear separation: the server exposes JSON (declarative data), the client decides how to render it (declarative HTML + JS that diffs the DOM)
- The same pattern, scaled up, is what every modern framework (React, Vue, Svelte…) gives you with more machinery

Outline

The minimum useful JavaScript

- `const`, `let`, `===`, template literals: the everyday syntax you will reach for constantly
- Array methods: `map`, `filter`, `reduce`, `forEach` (the same shapes you use in Python and SQL)
- The DOM API: `querySelector`, `textContent`, `createElement`, `appendChild`, `addEventListener`
- `fetch` + `async`/`await`: to read and write data without leaving the page
- JSON: the lingua franca between your Flask views and your JavaScript

Picking up a new language: practical tips

- Concepts you already know (variables, types, functions, I/O, modules) almost always transfer; budget more time for the idiomatic parts (event loop, promises, build tools)
- Use a reliable reference (MDN Web Docs), not random blog posts from 2014
- Make the browser console your interactive interpreter; type expressions, inspect objects with `console.log`
- Read other people's small programs end-to-end before writing your own
- Be ready to throw away your first version; the second is always shorter

To explore on your own

- MDN Web Docs, especially:
  - `https://developer.mozilla.org/en-US/docs/Web/JavaScript`
  - `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`
  - `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API`
- javascript.info: a thorough, free tutorial
- In your browser, open the Network tab while clicking around any modern site; you will see exactly which JSON requests it makes and what comes back
- Try rewriting one page of our Flask app as a JSON endpoint + a JavaScript renderer

License