Module 1: Fundamentals

HTML/Tailwind/DOM, JS Async & Fetch, TypeScript

Adrián Catalán

adriancatalan@galileo.edu

Agenda

1. **Project: Country Explorer**
2. **HTML, Tailwind CSS & DOM**
3. **JavaScript Async & Fetch API**
4. **TypeScript Introduction**
5. **Deep Dive**
6. **Challenge Lab**

Country Explorer App

We are building a Country Explorer that fetches data from a public API.

Core Requirements:

1. **Search**: Filter countries by name in real-time.
2. **Cards**: Display country info (flag, name, population, region).
3. **Modal**: Show detailed information on click.
4. **Performance**: Debounced search input.


center

2. TypeScript Fundamentals

Why TypeScript?

JavaScript is dynamically typed. Errors appear at **runtime**.

The Problem:

```
function greet(name) {
    return "Hello, " + name.toUpperCase();
}
greet(123); // Runtime Error: name.toUpperCase is not a function
```

The Solution (TypeScript):

```
function greet(name: string): string {
    return "Hello, " + name.toUpperCase();
}
greet(123); // Compile Error: Argument of type 'number' is not assignable
```

Type Annotations

TypeScript adds type information to JavaScript.

```
// Primitive Types
let country: string = "Guatemala";
let population: number = 17_000_000;
let isLandlocked: boolean = false;

// Arrays
let languages: string[] = ["Spanish", "Mayan"];

// Objects
let capital: { name: string; population: number } = {
    name: "Guatemala City",
    population: 2_500_000
};
```

Interfaces: Defining Shapes

Interfaces define the structure of objects.

```
// types/country.ts
interface CountryName {
    common: string;
    official: string;
}

interface Country {
    name: CountryName;
    cca3: string;                // 3-letter code (ESP, MEX)
    capital?: string[];          // Optional (some have none)
    population: number;
    region: string;
    flags: { png: string; svg: string };
}
```

Benefits:

Type vs Interface

Both define object shapes. When to use each?

Interface: Extendable, better for objects.

```
interface Animal {
    name: string;
}
interface Dog extends Animal {
    breed: string;
}
```

Type: Unions, intersections, primitives.

```
type Status = "loading" | "success" | "error";
type ID = string | number;
```

Rule of Thumb: Use `interface` for objects, `type` for everything else.

Generics: Reusable Types

Generics allow type parameters.

```
// Without generics - specific to Country
function getFirst(items: Country[]): Country {
    return items[0];
}

// With generics - works with any type
function getFirst<T>(items: T[]): T {
    return items[0];
}

// Usage
const firstCountry = getFirst<Country>(countries);
const firstNumber = getFirst<number>([1, 2, 3]);
```

Optional & Readonly

```
interface Country {
    name: string;            // Required
    capital?: string[];      // Optional (may be undefined)
    readonly code: string;   // Cannot be modified after creation
}

const country: Country = {
    name: "Guatemala",
    code: "GT"
    // capital is optional, can be omitted
};

country.code = "MX"; // Error: Cannot assign to 'code'
```

Null Safety

TypeScript helps prevent null reference errors.

```
// Strict null checks (tsconfig: "strict": true)
let name: string = null; // Error!

// Explicit nullable
let name: string | null = null; // OK

// Optional chaining
const capitalName = country.capital?.[0]?.toUpperCase();

// Nullish coalescing
const population = country.population ?? 0;
```

3. DOM Manipulation

The Document Object Model

The DOM is a tree representation of HTML.

```
document
└── html
    ├── head
    │   └── title
    └── body
        ├── header
        ├── main
        │   ├── input#search
        │   └── div#countries-grid
        └── footer
```

JavaScript/TypeScript can read and modify this tree.

Selecting Elements

```
// By ID (returns single element or null)
const grid = document.getElementById('countries-grid');

// By CSS Selector (returns first match)
const searchInput = document.querySelector<HTMLInputElement>('#search');

// By CSS Selector (returns all matches)
const cards = document.querySelectorAll<HTMLDivElement>('.country-card');

// Type assertion for TypeScript
const input = document.querySelector('#search') as HTMLInputElement;
```

Note: Always use generics or type assertions for proper typing.

Creating Elements

```
// utils/dom.ts
export function createElement<K extends keyof HTMLElementTagNameMap>(
    tag: K,
    ...classes: string[]
): HTMLElementTagNameMap[K] {
    const element = document.createElement(tag);
    if (classes.length > 0) {
        element.classList.add(...classes);
    }
    return element;
}

// Usage
const div = createElement('div', 'p-4', 'bg-slate-800', 'rounded-lg');
```

Building Components

```
// components/CountryCard.ts
export function createCountryCard(country: Country): HTMLElement {
    const card = createElement('article', {
        className: 'bg-white rounded-lg shadow-md overflow-hidden'
    });

    card.innerHTML = `
        <img src="${country.flags.svg}" alt="Flag of ${country.name.common}">
        <div class="p-4">
            <h2 class="font-bold text-lg">${country.name.common}</h2>
            <p>Population: ${formatNumber(country.population)}</p>
            <p>Region: ${country.region}</p>
        </div>
    `;

    return card;
}
```

Event Handling

```
// Adding event listeners
const searchInput = document.querySelector<HTMLInputElement>('#search');

searchInput?.addEventListener('input', (event) => {
    const target = event.target as HTMLInputElement;
    const query = target.value;
    filterCountries(query);
});

// Removing event listeners
const handler = (e: Event) => console.log(e);
element.addEventListener('click', handler);
element.removeEventListener('click', handler);
```

Event Delegation

Instead of adding listeners to each element, delegate to a parent.

```
// Bad: One listener per card (memory intensive)
cards.forEach(card => {
    card.addEventListener('click', handleClick);
});

// Good: Single listener on parent
grid.addEventListener('click', (event) => {
    const target = event.target as HTMLElement;
    const card = target.closest('.country-card');

    if (card) {
        const countryCode = card.dataset.code;
        openModal(countryCode);
    }
});
```

Data Attributes

Store custom data on HTML elements.

```
<article class="country-card" data-code="GT" data-region="Americas">
    ...
</article>
```

```
// Reading data attributes
const card = document.querySelector('.country-card');
const code = card?.dataset.code;      // "GT"
const region = card?.dataset.region;  // "Americas"

// Setting data attributes
card.dataset.favorite = "true";
```

4. Fetch API & REST

REST API Basics

REST (Representational State Transfer) is an architectural style for APIs.

HTTP Methods:

Method	Purpose	Example
GET	Read data	`GET /countries`
POST	Create data	`POST /countries`
PUT	Update (full)	`PUT /countries/GT`
PATCH	Update (partial)	`PATCH /countries/GT`
DELETE	Delete data	`DELETE /countries/GT`

The Fetch API

Modern way to make HTTP requests in the browser.

```
// Basic GET request
const response = await fetch('https://restcountries.com/v3.1/all');
const countries: Country[] = await response.json();

// With error handling
try {
    const response = await fetch(url);
    if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
    }
    const data = await response.json();
} catch (error) {
    console.error('Fetch failed:', error);
}
```

Async/Await

JavaScript uses Promises for asynchronous operations.

```
// Promise chain (old way)
fetch(url)
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error(error));

// Async/Await (modern way)
async function fetchCountries(): Promise<Country[]> {
    const response = await fetch(url);
    const data = await response.json();
    return data;
}
```

Note: `await` can only be used inside `async` functions.

API Service Pattern

Encapsulate API calls in a service module.

```
// services/countryApi.ts
const BASE_URL = 'https://restcountries.com/v3.1';

export async function searchCountries(name: string): Promise<Country[]> {
    const url = `${BASE_URL}/name/${encodeURIComponent(name)}`;
    const response = await fetch(url);

    if (!response.ok) {
        if (response.status === 404) return []; // Not found
        throw new Error(`HTTP error! status: ${response.status}`);
    }
    return response.json();
}

export async function getCountryByCode(code: string): Promise<Country | null> {
    const response = await fetch(`${BASE_URL}/alpha/${code}`);
    if (!response.ok) return null;
    const data = await response.json();
    return Array.isArray(data) ? data[0] : data;
```

Handling Loading States

```
// main.ts
const grid = document.getElementById('countries-grid')!;
const loading = document.getElementById('loading')!;

async function loadCountries(): Promise<void> {
    try {
        loading.classList.remove('hidden');
        grid.innerHTML = '';

        const countries = await getAllCountries();

        countries.forEach(country => {
            const card = createCountryCard(country);
            grid.appendChild(card);
        });
    } catch (error) {
        grid.innerHTML = '<p class="error">Failed to load countries</p>';
    } finally {
        loading.classList.add('hidden');
    }
}
```

Debouncing Input

Prevent excessive API calls while user types.

```
// utils/debounce.ts
export function debounce<T extends (...args: unknown[]) => void>(
    fn: T,
    delay: number
): (...args: Parameters<T>) => void {
    let timeoutId: ReturnType<typeof setTimeout>;

    return (...args: Parameters<T>) => {
        clearTimeout(timeoutId);
        timeoutId = setTimeout(() => fn(...args), delay);
    };
}

// Usage
const debouncedSearch = debounce((query: string) => {
    filterCountries(query);
}, 300);

searchInput.addEventListener('input', (e) => {
    debouncedSearch((e.target as HTMLInputElement).value);
```

5. Deep Dive

1. TypeScript Compilation

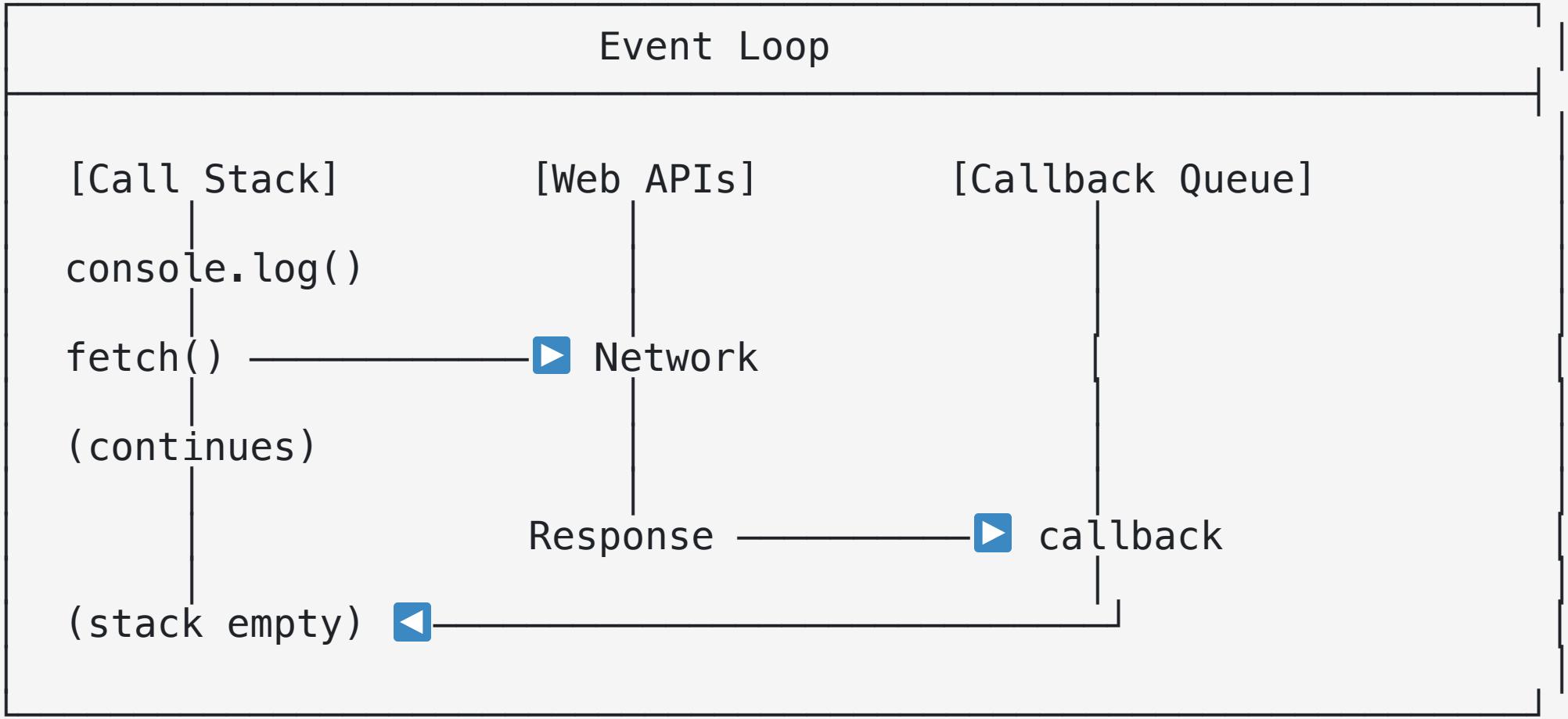
TypeScript doesn't run in the browser. It compiles to JavaScript.

Vite handles this automatically during development:

1. Serves `.ts` files directly (esbuild transforms on-the-fly)
2. Full TypeScript compilation only for production build

2. The Event Loop

JavaScript is single-threaded but handles async with the Event Loop.



3. Module System

ES Modules are the standard for JavaScript.

```
// Named exports
export function formatNumber(n: number): string { ... }
export const API_URL = 'https://...';

// Default export
export default class CountryService { ... }

// Named imports
import { formatNumber, API_URL } from './utils';

// Default import
import CountryService from './CountryService';

// Namespace import
import * as Utils from './utils';
```

4. Tailwind CSS

Utility-first CSS framework. Classes instead of custom CSS.

```
<!-- Traditional CSS -->
<div class="country-card">...</div>
<style>
.country-card {
    background: white;
    border-radius: 0.5rem;
    box-shadow: 0 1px 3px rgba(0,0,0,0.1);
    padding: 1rem;
}
</style>

<!-- Tailwind CSS -->
<div class="bg-white rounded-lg shadow-md p-4">...</div>
```

Benefits: Faster development, consistent spacing, smaller bundle.

5. Vite Build System

Vite is a modern build tool optimized for speed.

Development:

```
Browser ──▶ Vite Dev Server ──▶ On-demand transform ──▶ Source files
         (HMR enabled)          (esbuild for TS)
```

Production:

```
Source files ──▶ Rollup bundler ──▶ Optimized bundle
                (Tree shaking, minification, code splitting)
```

Commands:

- `npm run dev` - Development server with HMR
- `npm run build` - Production build

6. Challenge Lab

Practice & Application

Part 1: Region Filter

Context:

Users want to filter countries not just by name, but also by region (Americas, Europe, Asia, etc.).

Your Task:

Add a dropdown filter that:

- Shows all unique regions from the API
- Filters countries by selected region
- Works in combination with the search input
- Shows "All Regions" option to reset

Files to Modify:

- `index.html` (add dropdown)

Part 1: Definition of Done

Criteria	Description
Dropdown exists	`<select>` element with region options
Regions populated	Options from API data (Africa, Americas, Asia, Europe, Oceania)
Filter works	Selecting region shows only matching countries
Combined filters	Search + Region filter work together
Reset option	"All Regions" shows all countries
State preserved	Filter persists when typing in search
No errors	Console free of errors

Part 2: Favorites System

Context:

Users want to save their favorite countries locally so they can access them quickly.

Your Task:

Implement a favorites feature that:

- Adds a heart icon to each country card
- Toggles favorite status on click
- Stores favorites in localStorage
- Shows a "Favorites Only" toggle
- Persists across page reloads

Files to Create/Modify:

- `utils/storage.ts` (localStorage helper)

Part 2: Definition of Done

Criteria	Description
Heart icon visible	Each card shows outlined/filled heart
Toggle works	Clicking heart adds/removes from favorites
localStorage used	Favorites persist in `localStorage`
Favorites filter	Checkbox/toggle to show only favorites
Visual feedback	Filled heart for favorites, outline for others
Page reload	Favorites survive browser refresh
Clear option	Button to clear all favorites

Resources & Wrap-up

Resources

TypeScript

- TypeScript Handbook
- TypeScript Playground
- Total TypeScript (Free)
- TypeScript Deep Dive (Book)

DOM & Browser APIs

- MDN Web Docs: DOM
- MDN: Fetch API
- JavaScript.info: Document

Tailwind CSS

Recommended Articles

TypeScript

- TypeScript Tips and Tricks - Total TypeScript
- Strict Mode in TypeScript - Marius Schulz
- Type vs Interface - Total TypeScript

Web APIs

- Using the Fetch API - MDN
- Event Delegation Explained - JavaScript.info
- Debouncing and Throttling - CSS-Tricks