Spring Boot Initials

Oct 6, 2025 | Farhan Tausif

Setup Github

Just google
Add, git config --global credential.helper store

Create a Demo Spring boot project

Learn Spring Boot Series | Baeldung

Security with Spring Series | Baeldung

Initial Tasks - Kube Kit

Node.js (How it literally works)

- Internals
- API calls
- node

SVELT

- Dynamic Components
- Reactivity
- Component
- Kit
- Common ?
- Others (touch)

HAPIjs

- Handlers / Controllers
- Routes

Structure/Grouping

- **FRONTEND** (SVELT/SVELTKit)
- **BACKEND** (HapiJs)

Kubernetes basics

Summary –

What is Kubernetes? | Kubernetes Explained

Piyush Garg

Theme and Objectives: The video serves as a comprehensive guide to understanding Kubernetes, including its history, architecture, and benefits. It aims to illustrate Kubernetes as a solution to the complexities of deploying and managing applications in a cloud environment.

Central Idea: The central idea revolves around Kubernetes being a powerful tool that simplifies container orchestration and deployment across various platforms, thus making it suitable for developers, tech enthusiasts, and business owners looking to improve their deployment processes.

Reader Benefits:

- **Who is it suitable for?** Developers, tech students, startup founders, and those interested in cloud computing and containerization.
- **What can the reader learn?** Gain in-depth knowledge of Kubernetes, its operational mechanics, and advantages over traditional deployment methods while learning how to navigate its interface effectively.

Timeline Summary

🕑 00:00 - 02:45: Introduction to Kubernetes

- Welcome and introduction to Kubernetes, outlining the core themes of the discussion.
- Emphasis on understanding the problems Kubernetes solves and the importance of learning it.

🕑 02:46 - 06:14: Traditional Development Challenges

- Discussion about traditional deployment methods, highlighting challenges such as server purchases, maintaining uptime, and scaling issues.
- Explanation of the costly and skill-intensive nature of traditional deployments.

🕑 06:15 - 10:38: Emergence of AWS and Cloud Computing

- Introduction to Amazon Web Services (AWS) and how it revolutionized accessibility to cloud resources.
- Explanation of the dawn of cloud-native technologies and their impact on application deployment.

🕒 10:39 - 13:36: Containerization

- Insights into containerization as a lightweight solution that emerged to address the complexities of traditional deployments.
- Introductory discussion on how containers simplify application management.

🕒 13:37 - 17:09: Need for Container Orchestration

- Explanation of the necessity for container orchestration services due to increasing complexities in managing multiple container instances.
- Introduction to the concept of automating container management.

🕒 17:10 - 21:31: Birth of Kubernetes

- Historical overview discussing Kubernetes' development at Google and its release as an open-source project under the Cloud Native Computing Foundation (CNCF).
- Introduction to Kubernetes terminology and its Greek origin meaning helmsman.

🕒 21:32 - 28:03: Kubernetes Architecture Overview

- Breakdown of Kubernetes architecture, including control planes and worker nodes.
- Explanation of Kubernetes components such as API servers, controllers, and ETCD.

🕒 28:04 - 34:26: Containment Operations

- Description of how Kubernetes manages containers, including scaling operations and resource allocation.
- Explanation of desired states versus current states and how Kubernetes maintains synchronization.

🕒 34:27 - 42:40: Kubernetes Control Mechanisms

- Details on managing workloads in Kubernetes, including scheduling and deploying containers.
- Insights into the use of Kubernetes components like kubelet and kube-proxy in managing workloads effectively.

🕑 42:41 - 48:13: Final Thoughts and Learning Resources

- Recap of Kubernetes functionalities and benefits for developers looking to streamline their workflow.
- Promotion of additional resources for learning Kubernetes and Docker, encouraging viewers to explore further.

Key Points

- 🚀 **Kubernetes Overview:** Simplifies application deployment and management in cloud environments.
- 🛠️ **Traditional Challenges:** Traditional methods require high skills for maintenance, making deployments difficult.
- ☁️ **Cloud Computing Impact:** AWS made cloud resources accessible to individuals and small startups.
- 🔍 **Containerization Benefits:** Improved scalability and easy sharing of applications through lightweight containers.
- 📈 **Automation Requirement:** Container orchestration automates management, crucial for efficiently handling workloads.

Key Insights

- **Thought-Provoking Idea:** Kubernetes is the ‘helmsman’ of container management, steering applications on cloud seas.
- **Highlight (6:15):** The transition from physical servers to cloud configurations drastically reduced deployment complexities.
- **Recommended Methodology:** Learn to leverage infrastructure as code to enhance deployment strategies across various platforms.

Frequently Asked Questions (FAQs)

- **Q1: What is Kubernetes?**
  A1: Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications.

- **Q2: Why should I learn Kubernetes?**
  A2: Learning Kubernetes can enhance your ability to deploy, manage, and scale applications efficiently in a cloud-based environment.
- **Q3: What are the benefits of cloud-native technologies?**
  A3: Cloud-native technologies, like Kubernetes, offer scalability, flexibility, and efficient resource utilization, reducing maintenance overhead.
- **Q4: How can Kubernetes help with scaling applications?**
  A4: Kubernetes automatically manages scaling based on demand, adjusting the number of running container instances.
- **Q5: Is Kubernetes suitable for small businesses?**
  A5: Yes, Kubernetes provides powerful orchestration capabilities that can be tailored for businesses of all sizes, enhancing application management and deployment.

Conclusion

The video serves as a valuable introduction to Kubernetes, outlining its critical role in simplifying application management through container orchestration. By understanding its principles and architectural foundations, viewers gain insights into deploying applications more efficiently. For action steps, it is advisable for viewers to explore resources on both Kubernetes and Docker to improve their deployment strategies and stay updated with cloud-native computing trends. Consider starting with practical hands-on projects to gain real-world experience.

Summary – Kubernetes Crash Course for Absolute Beginners […

~ Tech with nana

Summary

The video provides a comprehensive **Kubernetes crash course** aimed at beginners looking to understand and utilize Kubernetes effectively. The course covers the **definition of Kubernetes**, its architecture, key components, and practical deployment through hands-on demonstrations. It serves as an essential guide for those interested in entering the field of DevOps or cloud engineering, offering

insights into the workings of Kubernetes for robust application management.
Benefits to the readers include:

- Understanding the purpose and features of Kubernetes.
- Gaining hands-on experience with practical projects.
- Identifying avenues for further learning and career advancement, especially in positions like Kubernetes Administrator.

Timeline Summary

Overview of the Course

- **00:00 - 01:08**: Introduction to the course by Nana, emphasizing the intent to teach Kubernetes fundamentals in a short time while mentioning resources for further learning.

What is Kubernetes?

- **01:09 - 02:50**: Defining Kubernetes as an open-source container orchestration framework developed by Google, explaining its role in managing applications across different environments, including cloud and on-premises.

Challenges Addressed by Kubernetes

- **02:51 - 04:30**: Discussing the problems Kubernetes solves, such as high availability, scalability, and disaster recovery, in the context of managing complex applications built with microservices.

Kubernetes Architecture

- **04:31 - 07:40**: Overview of Kubernetes architecture, describing the roles of master nodes, worker nodes, and essential components like the API server and etcd for cluster state management.

Main Components of Kubernetes

- **07:41 - 09:59**: Introduction to Kubernetes components, focusing on pods, services, and their interactions, laying the groundwork for deploying applications.

Hands-On Demonstration

- **10:00 - 24:50**: Step-by-step demonstration of setting up a simple Kubernetes project with a web application and a database, outlining the function of pods, services, and configuration management with config maps and secrets.

Persisting Data in Kubernetes

- **24:51 - 26:50**: Explanation of how data persistence works in Kubernetes using volumes, ensuring that database data remains intact even after pod restarts.

Scaling with Deployments and StatefulSets

- **26:51 - 30:35**: Highlighting how to replicate application pods using deployments and discussing the specific use of StatefulSets for databases, emphasizing the importance of maintaining data integrity.

Setting Up and Interacting with Minikube

- **30:36 - 39:05**: Overview of setting up Minikube, a local single-node Kubernetes environment to test deployments, and how to use the command-line tool `kubectl`.

Configuration and Creation of Kubernetes Components

- **39:06 - 48:50**: Detailed walkthrough of creating configuration files for deployments, services, config maps, and secrets, explaining the YAML structure used in Kubernetes.

Deploying Applications

- **48:51 - 54:40**: Finalizing the configurations and deploying the application with a database in Kubernetes while ensuring it's accessible from a browser through an external service.

Conclusion

- **54:41 - 57:40**: Recap of the key points covered, recommendations for continued learning, and the importance of hands-on experience to become adept at using Kubernetes.

Key Points

- 🔍 **Introduction**: Kubernetes simplifies the management of complex containerized applications.
- 🏛️ **Architecture**: Understanding master and worker nodes, along with key components, is crucial for deploying and managing applications.
- 🛠️ **Core Components**: Familiarity with pods, services, and their networking aspects is foundational for successful operations in Kubernetes.
- 🗄️ **Data Management**: Critical insights into data persistence highlight the importance of using volumes to store stateful data.
- ⚙️ **Configuration Management**: Using config maps and secrets helps streamline application deployment by avoiding hardcoded values.

Key Insights

- **💡 Kubernetes abstracts away the complexities of managing containers.**
- **🚀 Deploying applications with Kubernetes enhances flexibility, scalability, and reliability.**
- **📚 Using the Kubernetes API and `kubectl` is essential for effective interaction with the cluster.**
- **🔒 Configuration best practices involve using secrets and config maps to handle sensitive data and application settings securely.**

Frequently Asked Questions (FAQs)

- **❓ What is the purpose of Kubernetes?**
  - **🌟 Kubernetes is designed to automate the deployment, scaling, and management of containerized applications.**
- **❓ How can I get hands-on experience with Kubernetes?**
  - **📊 You can use Minikube to create a local development environment for practicing Kubernetes deployments.**
- **❓ What are the main components of a Kubernetes cluster?**
  - **⚙️ Key components include master nodes, worker nodes, pods, services, and the etcd key-value store.**
- **❓ How does Kubernetes ensure high availability?**
  - **🛡️ Kubernetes achieves high availability by distributing application instances across multiple nodes and using self-healing mechanisms that allow pods to restart automatically.**
- **❓ Is experience with Kubernetes beneficial for my career?**
  - **🚀 Yes, Kubernetes skills are in high demand in DevOps and cloud computing roles, making it a valuable expertise to have.**

Conclusion

This Kubernetes crash course offers a holistic introduction to the essentials of Kubernetes and its components, equipping viewers with foundational knowledge to start deploying applications efficiently. By emphasizing hands-on practical experience and best practices, viewers are encouraged to delve deeper into Kubernetes and consider further training, especially in specialized programs like the **Kubernetes Administrator course**. The final takeaway is to practice regularly, leverage available resources, and experiment within local environments to master Kubernetes functionality effectively.

NodeJs

#2 - Node.js কি & কিভাবে কাজ করে? - Node.js Tutorial Bangla | Bangla Node js Tutorial

1. Node.js Fundamentals & Modules

- **What is Node.js?**
  - It's a **JavaScript Runtime Environment** built on Google's **V8 Engine** (the same one Chrome uses) that allows JS to run on the server/outside the browser.
- **npm (Node Package Manager)**: Package manager to install and manage third-party packages. We can create a `package.json` file, and use commands like `npm install`.
- **Node.js Modules:**
  - **CommonJS (`require`/`module.exports`)** vs. **ES Modules (`import`/`export`)**. Understand the difference and when to use each.
  - **Core Modules:** Learn to use essential built-in modules like:
    - `fs` (File System): For reading and writing files.
    - `path`: For handling and normalizing file paths.
    - `http`: For creating basic web servers (The **API calls** foundation).

Feature	CommonJS (CJS)	ES Modules (ESM)	When to Use
Syntax	- **require()** to import. - **module.exports** or **exports** to export.	- **import** to import. - **export** to export.	Older Node.js projects, synchronous loading, when compatibility is a concern.
Loading	**Synchronous** (blocks execution until loaded).	**Asynchronous** (non-blocking).	All modern JavaScript projects (Node & Browser),
Usage	Default for older Node.js versions.	Default for modern browsers and increasingly for Node.js (requires .mjs or "type": "module" in package.json).	for features like **tree-shaking** (optimization).

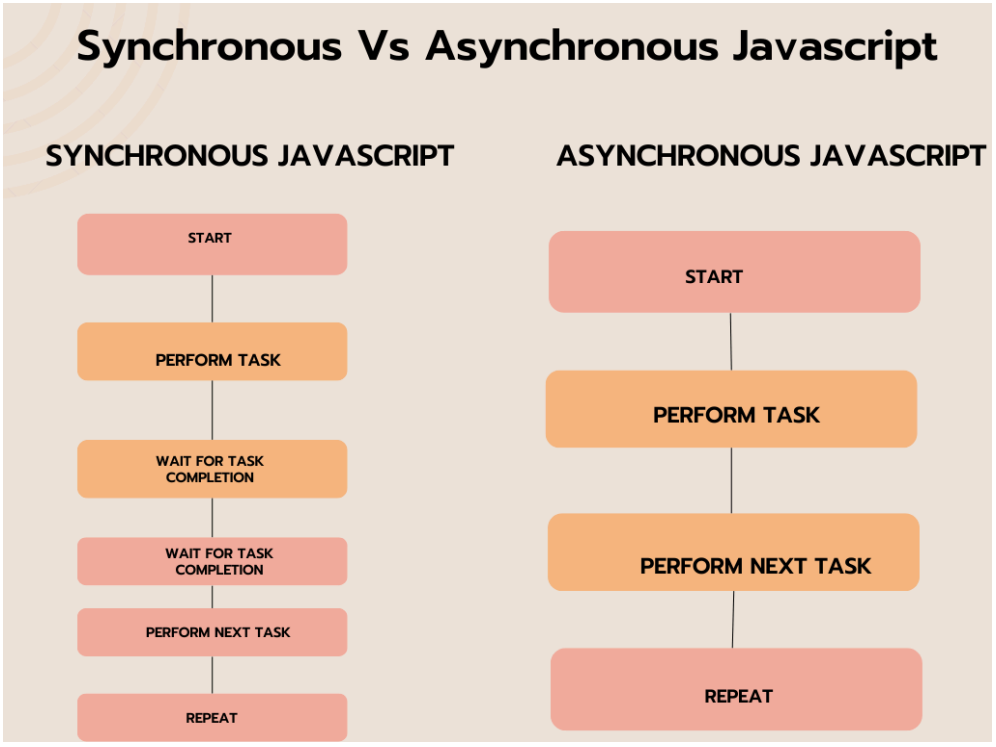
2. Core Architecture: How it Literally Works (Internals)

This is the most crucial part for understanding performance and the **non-blocking nature** of Node.js.

1. Non-Blocking I/O: The Node.js Philosophy

I/O (Input/Output) refers to any operation that involves waiting for an external resource, such as reading a file from disk, querying a database, or making a network request.

- **Blocking (Synchronous) Operation:**
  - The main thread **stops** and waits for the I/O operation to complete before moving to the next line of code.
  - In a server context, this means the server cannot handle *any* other user requests while one request is waiting for a database query to finish. This is **inefficient.**



- **Non-Blocking (Asynchronous) Operation:**
  - The main thread tells the operating system (OS) or a background utility to perform the I/O operation.
  - It immediately moves to the next line of JavaScript code.
  - When the I/O operation finishes, its designated function (a **callback**) is put in a queue to be run later.

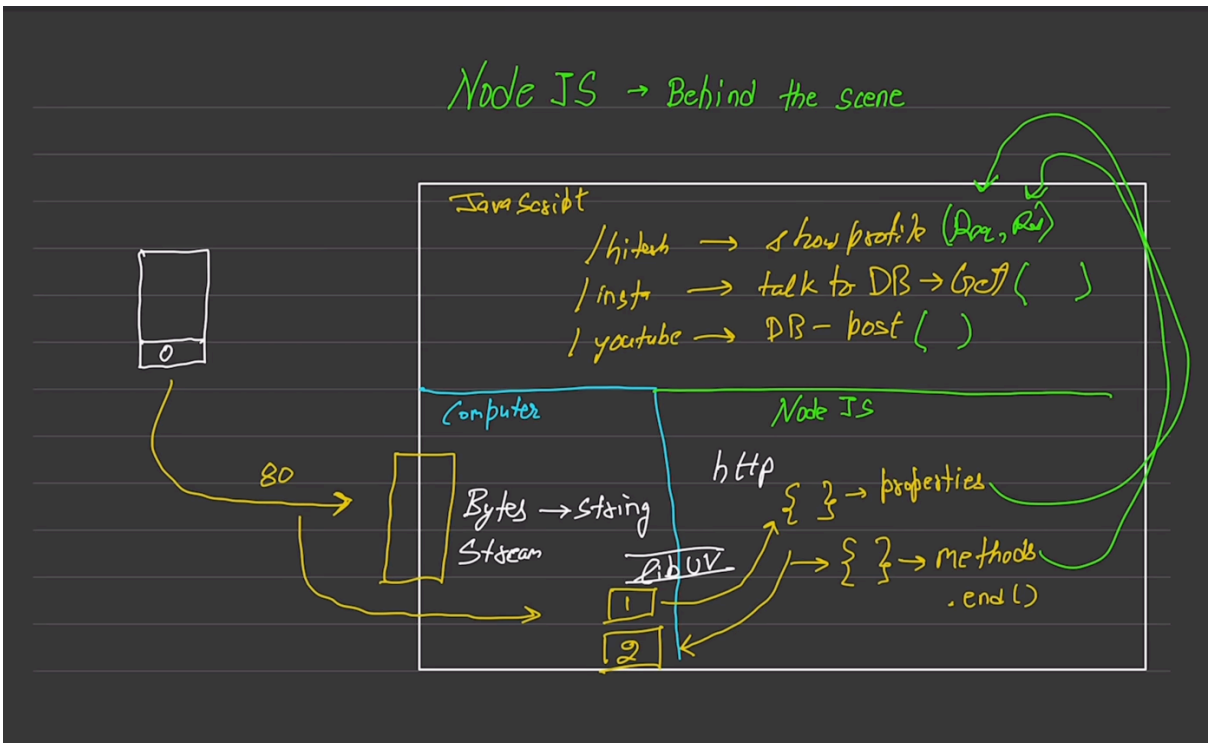


Fig: NodeJS → Behind the scene

2. The Core Components

Node.js is a **runtime** that is built from **three major parts** working together:

A. The V8 Engine (The JavaScript Executor)

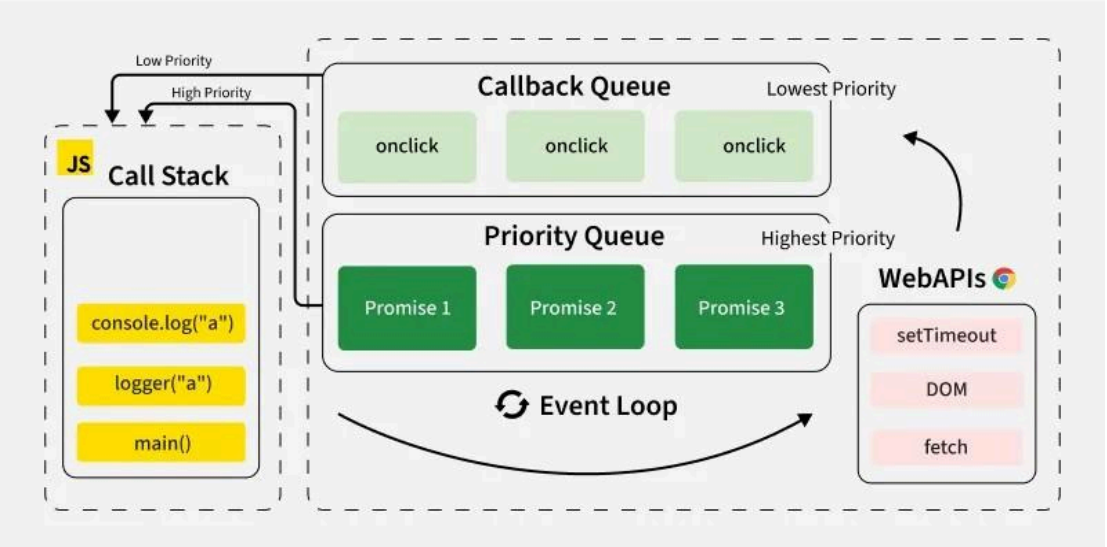
- **Role:** V8 is Google's open-source JavaScript and WebAssembly engine, written in C++. It powers Chrome and Node.js.
- **Function:** It takes your human-readable JavaScript code and compiles it directly into highly optimized **machine code**.
- **Core Feature:** It provides the **Call Stack**, which executes your JavaScript code in a **single thread**. This is why your *JavaScript logic* itself is single-threaded.

B. `libuv` (The I/O Utility Belt)

- **Role:** `libuv` is a C library that handles all the heavy-lifting of I/O operations for Node.js in a non-blocking way.
- **Functionality:**
    1. **I/O Abstraction:** It abstracts the differences between operating systems (Windows, Linux, macOS) regarding network, file, and timer operations.
    2. **Worker Thread Pool:** For computationally expensive or disk-intensive I/O (like file system operations or DNS lookups), `libuv` maintains a pool of background threads (**Worker Threads**). It offloads these tasks to the pool so the main JavaScript thread remains free.

3. The Event Loop (The Orchestrator)

The **Event Loop** is the mechanism that allows Node.js to perform non-blocking I/O operations despite JavaScript being single-threaded. It is a continuous loop that checks if the **Call Stack** is empty and then moves completed asynchronous operations from a queue into the Call Stack for execution.



Phases of the Event Loop (In Cycle Order)

The Event Loop continuously cycles through these six phases in order:

1. **Timers Phase ⏳:** Executes callbacks scheduled by `setTimeout()` and `setInterval()` whose expiration time has been reached.
2. **Pending Callbacks Phase:** Executes I/O callbacks deferred to the next loop iteration (e.g., some system error callbacks).
3. **Idle, Prepare Phase:** Used internally by Node.js.
4. **Poll Phase 🔄 (Most Important):**
    - Retrieves new I/O events (like completed network requests or file reads) and executes their callbacks.
    - If there are no I/O events, it may block/wait here until a new I/O event arrives or a timer is due.
5. **Check Phase ✅:** Executes callbacks scheduled with `setImmediate()`.
6. **Close Callbacks Phase 🚪:** Executes close event callbacks, like `socket.on('close', ...)`.

```javascript
console.log("Start");

setTimeout(() => {
    console.log("setTimeout Callback");
}, 0);

Promise.resolve().then(() => {
    console.log("Promise Resolved");
});

console.log("End");
```

Output

```
Start
End
Promise Resolved
setTimeout Callback
```

In this example,

a. console.log("Start") executes first.
   setTimeout schedules its callback but does not execute it immediately.
b. Promise.resolve().then() is placed in the microtask queue and executes before the callback queue.
c. Promise Resolved appears before setTimeout Callback due to microtask priority.
d. JavaScript executes code synchronously in a single thread. However, it can handle asynchronous operations such as fetching data from an API, handling user events, or setting timeouts without pausing execution. This is made possible by the event loop.

The event loop continuously checks whether the call stack is empty and whether there are pending tasks in the callback queue or microtask queue.

1. **Call Stack:** JavaScript has a call stack where function execution is managed in a Last-In, First-Out (LIFO) order.

2. **Web APIs (or Background Tasks):** These include setTimeout, setInterval, fetch, DOM events, and other non-blocking operations.

3. **Callback Queue (Task Queue):** When an asynchronous operation is completed, its callback is pushed into the task queue.

4. **Microtask Queue:** Promises and other microtasks go into the microtask queue, which is processed before the task queue.

5. **Event Loop:** It continuously checks the call stack and, if empty, moves tasks from the queue to the stack for execution.

Phases of the Event Loop

The event loop operates in multiple phases.

Timers Phase: Executes callbacks from setTimeout and setInterval.
I/O Callbacks Phase: Handles I/O operations like file reading, network requests, etc.
Prepare Phase: Internal phase used by Node.js.
Poll Phase: Retrieves new I/O events and executes callbacks.
Check Phase: Executes callbacks from setImmediate.
Close Callbacks Phase: Executes close event callbacks, e.g., socket.on('close').
Microtasks Execution: After each phase, the event loop processes the microtask queue before moving to the next phase.

1. Callbacks (The Original Pattern)

A **callback** is simply a function passed as an argument to another function, which is then executed once the original function has finished its work. This is the oldest way to handle asynchronous operations in JavaScript.

- **How it Works:** When you start an async task (like reading a file), you provide a function (the callback) to be executed when the task completes.

JavaScript

```
fs.readFile('data.txt', (err, data) => {
  if (err) throw err;
  console.log(data);
});
// Node.js immediately continues executing code here, it doesn't wait.
```

The Problem: "Callback Hell" (or the Pyramid of Doom) When you have a series of dependent asynchronous operations, you end up nesting callbacks deeply within each other, leading to code that is difficult to read, debug, and maintain.

JavaScript

```
// Example of Callback Hell
asyncTask1((result1) => {
  asyncTask2(result1, (result2) => {
    asyncTask3(result2, (result3) => {
      // ... very deep nesting
    });
  });
});
```

2. Promises (The Solution to Callback Hell)

A **Promise** is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value. They provide a much cleaner, linear way to structure asynchronous code.

- **States:** A Promise exists in one of three states:
  - **Pending:** Initial state, neither fulfilled nor rejected.
  - **Fulfilled (Resolved):** The operation completed successfully.
  - **Rejected:** The operation failed (an error occurred).
- **Key Methods (Chaining):** Promises allow you to chain operations using the following methods:
  - **`.then(onFulfilled)`:** Registers a callback to be invoked when the Promise is **fulfilled** (successful). You can chain multiple `.then()` calls together.
  - **`.catch(onRejected)`:** Registers a callback to be invoked when the Promise is **rejected** (error). This handles errors from any part of the chain above it.
  - **`.finally()`:** Registers a callback to be executed regardless of whether the Promise was fulfilled or rejected (used for cleanup tasks).

```
fetchData()
  .then(data => processData(data)) // Successfully process the data
  .then(finalResult => console.log(finalResult))
  .catch(error => console.error('An error occurred:', error)) // Catch any error
  .finally(() => console.log('Operation complete.')); // Always runs
```

4. Building APIs (The Hapi.js Pre-requisite)

Before jumping into Hapi.js, you need the general server-side concepts that are universally applicable.

- **HTTP Protocol Basics:** Understand HTTP methods (`GET`, `POST`, `PUT`, `DELETE`), status codes (e.g., 200, 404, 500), and headers.
- **Requests and Responses:** Know how to handle incoming **Request** data (`req`) and send back a **Response** (`res`), including setting headers and status codes.
- **Streams and Buffers:**
  - **Buffers:** Used to handle raw binary data (essential for files/network).
  - **Streams:** A way to efficiently process data chunk by chunk (e.g., streaming a large video file or processing a large file upload) without loading it all into memory.

Suggested Learning Path

1. **Node.js Installation & "Hello World":** Set up Node and run your first script.
2. **Modules and npm:** Learn to import/export code and install packages.
3. **The Event Loop & Async/Await:** Focus heavily on this to grasp the core "Internals."
4. **Core Modules:** Use `fs` and `http` to build a simple static file server.
5. **Introduction to Hapi.js:** Once you have the fundamentals, you can easily transition to using **Hapi.js** to handle the Routing and Controller logic for your **API calls**.

Svelte + SvelteKit

Svelte Documentation: Svelte Docs

SvelteKit Documentation: Introduction • SvelteKit Docs

Basic Web Dev

HTML Basics

Basic HTML-w3Schools: HTML Tutorial
Basic HTML-MDN: HTML: HyperText Markup Language - Web

Document Structure, Metadata, and Sectioning

Element	Summary
<html>	Represents the **root** (top-level element) of an HTML document.
<head>	Contains **machine-readable information (metadata)** about the document (title, scripts, styles).
<title>	Defines the document's **title** that is shown in a browser's title bar or page tab.
<base>	Specifies the **base URL** to use for all relative URLs in a document.
<link>	Specifies relationships between the current document and an **external resource** (most commonly CSS).
<meta>	Represents **metadata** that cannot be represented by other HTML meta-related elements.
<style>	Contains **style information (CSS)** for a document or part of a document.
<body>	Represents the **visible content** of an HTML document.
<main>	Represents the **dominant content** of the body of a document.
<section>	Represents a **generic standalone section** of a document.

`<article>`	Represents a **self-contained composition** intended to be independently distributable (like a blog post).
`<aside>`	Represents a portion of a document whose content is **only indirectly related** to the main content (often a sidebar).
`<nav>`	Represents a section of a page whose purpose is to provide **navigation links**.
`<header>`	Represents **introductory content**, typically a group of introductory or navigational aids.
`<footer>`	Represents a **footer** for its nearest ancestor, typically containing author or copyright information.
`<h1>` to `<h6>`	Represent **six levels of section headings**, with `<h1>` being the highest.
`<hgroup>`	Represents a **heading grouped with any secondary content**, such as a subheading.
`<address>`	Indicates that the enclosed HTML provides **contact information** for a person or organization.

Text Content and Semantics

Element	Summary
<p>	Represents a **paragraph** of text.
	The **generic container** for flow content, having no effect until styled with CSS.
<blockquote>	Indicates that the enclosed text is an **extended quotation**.
<pre>	Represents **preformatted text** which is to be presented exactly as written (retains whitespace).
	Represents an **unordered list** of items (typically bulleted).
	Represents an **ordered list** of items (typically numbered).
	Represents an **item in a list** (, , or <menu>).
<dl>	Represents a **description list**, enclosing terms and descriptions.
<dt>	Specifies a **term** in a description or definition list.
<dd>	Provides the **description, definition, or value** for the preceding term (<dt>).
<figure>	Represents **self-contained content**, potentially with an optional caption.

<figcaption>	Represents a **caption or legend** for its parent <figure> element.
<hr>	Represents a **thematic break** between paragraph-level elements.
<a>	Creates a **hyperlink** to a web page, file, or location on the current page.
	Indicates that its contents are of **strong importance** or seriousness.
	Marks text that has **stress emphasis**.
<b>	Used to draw the reader's **attention** to the contents, without granting special importance (historically bold).
<i>	Represents a range of text that is **set off** from the normal text, such as idiomatic text or technical terms (historically italics).
<code>	Displays its contents styled to indicate a **short fragment of computer code**.
<abbr>	Represents an **abbreviation** or acronym.
 	Produces a **line break** in text (carriage-return).
<q>	Indicates that the enclosed text is a **short inline quotation**.
<mark>	Represents text which is **marked or highlighted** for reference or notation purposes.
<data>	Links a given piece of content with a **machine-readable translation**.

<time>	Represents a specific **period in time**.
<dfn>	Used to indicate the **term being defined**.
<kbd>	Represents inline text denoting textual **user input** from a device.

Forms, Tables, and Interactive Elements

Element	Summary
<form>	Represents a document section containing **interactive controls for submitting information**.
<input>	Used to create a wide variety of **interactive controls** (text fields, checkboxes, buttons, etc.) for forms.
<button>	An **interactive element** activated by a user to perform an action.
<label>	Represents a **caption for an item** in a user interface, associating text with a form control.
<textarea>	Represents a **multi-line plain-text editing control** for free-form text input.
<select>	Represents a **control that provides a menu of options**.

<option>	Used to define an **item contained in a <select>** or <datalist> element.
<fieldset>	Used to **group several controls** and labels within a web form.
<legend>	Represents a **caption for the content of its parent <fieldset>**.
<progress>	Displays an indicator showing the **completion progress of a task**.
<output>	Container element into which a site or app can **inject the results of a calculation**.
<details>	Creates a **disclosure widget** where information is visible only when toggled open.
<summary>	Specifies a **summary, caption, or legend** for a <details> element.
<dialog>	Represents a **dialog box** or other interactive component, such as a dismissible alert.
<tr>	Defines a **row of cells** in a table.
<th>	Defines a **header cell** in a table.
<td>	Defines a **data cell** in a table.

Semantic HTML tags are elements that clearly define the meaning and purpose of the content they enclose, both to the browser and to developers. Unlike non-semantic elements like `<div>` and `<span>`, which are generic containers, semantic tags convey the role of the content within the webpage structure.

Key Semantic HTML Tags and their Purpose:

- `<header>`: Defines the introductory content or a set of navigational links for a document or a section.
- `<nav>`: Represents a section of the page intended for navigation links.
- `<main>`: Specifies the dominant content of the `<body>` of a document. There should only be one `<main>` element per document.
- `<article>`: Represents self-contained content that is independently distributable or reusable, such as a blog post, a news article, or a forum comment.
- `<section>`: Defines a standalone section within a document, often with a heading. It's used to group related content.
- `<aside>`: Represents content that is tangentially related to the content around it, often presented as a sidebar or pull-quote.
- `<footer>`: Defines the footer for a document or a section, typically containing information like copyright, author, or contact details.
- `<figure>` and `<figcaption>`: Used for embedding self-contained content (like images, diagrams, code snippets) with an optional caption.
- `<time>`: Represents a specific point in time or a time range.

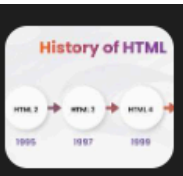
Benefits of using Semantic HTML:

- **Improved SEO:** Search engines can better understand the structure and content of a webpage, potentially leading to higher rankings.
- **Enhanced Accessibility:** Assistive technologies like screen readers can more accurately interpret and convey the page's structure to users with disabilities.
- **Better Code Readability and Maintainability:** Semantic tags make the HTML structure more intuitive and easier for developers to understand and maintain.
- **Cross-Device Compatibility:** A well-structured semantic document ensures better compatibility across various devices and browsers.

HTML to Web Frameworks

History of HTML

HTML was created by Tim Berners-Lee in 1991 while at CERN to share documents online, with the first official version released in 1993. It has since evolved through several versions, including HTML 4.01, which was widely used in the 2000s, and XHTML, a stricter, XML-based version. The latest standard, HTML5, was published in 2012 and added new features for multimedia and a better semantic structure.



Key milestones in HTML's history

- **1991:** Tim Berners-Lee created the first version, HTML 1.0, though it wasn't officially released until 1993.
- **1995:** HTML 2.0 was released.
- **1997:** HTML 3.2 was released, replacing the more complex HTML 3.0.
- **1999:** HTML 4.01 became an official standard, becoming the most widely used version in the early 2000s.
- **2000:** XHTML 1.0 was completed, which was HTML 4 structured as an XML language.
- **2012:** HTML5 was officially published, incorporating new technologies like multimedia tags ($<audio>, <video>) and semantic tags (<article>, <section>$) without needing plugins.

- **HTML 1.0 (1993):** The first public version, consisting of 18 tags, established the basic structure of web documents with hyperlinks, headings, and paragraphs.
- **HTML 2.0 (1995):** Introduced forms and tables, enabling more interactive web pages.
- **HTML 3.2 (1997):** Enhanced design capabilities with support for sophisticated tables and introduced the `<img>` tag for images.
- **HTML 4.01 (1999):** Became a stable and widely used standard, introducing support for Cascading Style Sheets (CSS), scripting, and better multimedia support.
- **XHTML (2000):** A reformulation of HTML as an XML-based language with stricter rules for better interoperability.
- **HTML5 (2014):** A significant milestone that introduced semantic elements, better multimedia support (like `<audio>` and `<video>`), offline storage, and features for mobile devices.
- **HTML5.1 and later (2016-present):** Continues to evolve with new features like the `<picture>` element, while future updates are expected to focus on core application foundations.

Why Pure HTML Was Not Enough and we needed frameworks

বা En

Pure HTML, while foundational to the web, proved insufficient for building modern, dynamic, and scalable web applications, leading to the development and widespread adoption of frameworks. Here's why:

- **Static Nature and Lack of Interactivity:** HTML alone is static. It defines the structure and content of a webpage but lacks the ability to handle user interactions, dynamic data updates, or complex animations. Frameworks, often built on JavaScript, provide the tools to create interactive elements, manage state, and respond to user input, transforming static pages into dynamic applications.
- **Complexity and Maintainability:** As web applications grew in complexity, managing large codebases with pure HTML, CSS, and JavaScript became challenging. Frameworks introduce structured approaches like component-based architectures (e.g., React, Vue) or Model-View-Controller (MVC) patterns (e.g., Angular), which promote code organization, reusability, and easier maintenance, especially in team environments.
- **Developer Efficiency and Productivity:** Frameworks offer pre-built functionalities, libraries, and conventions that streamline development. Features like routing, state management, data fetching, and templating are often integrated, reducing the need to write boilerplate code and allowing developers to focus on unique application logic. This significantly boosts productivity and speeds up development cycles.
- **Scalability and Performance:** Building scalable applications with pure HTML and client-side JavaScript can be difficult. Frameworks often incorporate optimizations like virtual DOM (React, Vue) or server-side rendering (Next.js, Nuxt.js) to enhance performance and improve user experience, particularly for large-scale applications with frequent updates.
- **Community Support and Ecosystem:** Popular frameworks benefit from vast communities, extensive documentation, and a rich ecosystem of third-party libraries and tools. This provides developers with readily available resources, solutions to common problems, and a supportive environment for learning and collaboration.

The evolution of frameworks like React, Vue, and Angular was a **direct response to the shortcomings of using pure HTML, CSS, and JavaScript** for complex applications.[1]

🛑 Why Pure HTML Was Not Enough

HTML (HyperText Markup Language) is a **markup language**, not a programming language.[2] It was designed to **structure documents** with headings, paragraphs, and links, which is why the original web was a collection of static, interconnected pages.[3]

As the web evolved from static pages to dynamic **web applications** (like Gmail, Twitter, or Google Maps), three major problems emerged that pure HTML, CSS, and vanilla JavaScript couldn't handle efficiently:

1. **Lack of Modularity (Component Reusability):**
    - In a complex app, you need the same button, navbar, or date picker on dozens of pages. With pure HTML, you have to **copy and paste** the code (HTML structure, CSS styles, and JavaScript logic) everywhere.[4]
    - If you needed to change the button, you'd have to manually update *every single copy*. This made code maintenance a nightmare.
2. **Inefficient DOM Manipulation:**
    - The **DOM (Document Object Model)** is the tree-like structure of a web page. To update the screen, JavaScript directly manipulates this tree.
    - In a large application with real-time updates (like a social media feed), every small data change could require a large, slow, and manual update to the DOM. This led to **performance issues** and slow, janky user interfaces.
3. **State Management (Keeping UI and Data in Sync):**
    - Web applications are constantly dealing with "state"—data that changes (e.g., a user's logged-in status, the contents of a shopping cart, or a new chat message).[5]
    - Manually writing JavaScript to ensure that every part of the screen displaying that data is updated correctly became incredibly complex, leading to bugs and hard-to-maintain "spaghetti code."

📈 The Evolution of Web Frameworks

The modern frameworks solve the problems above by introducing **Component-Based Architecture** and **Efficient Data Handling**.[6]

1. The Pre-Framework Era (1995–2010)

- **HTML, CSS, JavaScript (1990s):** The foundation.[7]
- **jQuery (2006):** This library's primary goal was to simplify **DOM manipulation** and solve **cross-browser compatibility** issues, making it easier to select elements and

handle events.[8] It helped a lot but didn't solve the structure/maintenance problem for large apps.

2. Full-Fledged Frameworks: AngularJS and Angular (2010)[9]

Framework	Year	Key Innovation	Status
AngularJS	2010	**Two-Way Data Binding:** Changes in the data model automatically update the view (HTML), and changes in the view (like typing in a form) automatically update the data model. This eliminated a ton of manual JavaScript.	Deprecated (Replaced by Angular)
Angular	2016	**Full Framework:** A complete rewrite (not compatible with AngularJS). It enforced a rigid structure, embraced **TypeScript** (for better scalability), and became a massive, full-featured framework ideal for large **Enterprise Applications**.	Active

3. The Library Revolution: React (2013)[10]

Framework	Year	Key Innovation	Status
React	2013	**Virtual DOM (VDOM):** React created an in-memory representation of the real DOM. When data changes, React calculates the difference (the "diff") between the current VDOM and the new VDOM, and only applies the *minimum necessary changes* to the real, slower DOM.	Active (The most popular)
		Component-Based: Focused purely on the "View" layer (the V in MVC), emphasizing building UIs from small, reusable components with clear inputs (**props**) and internal state.	

4. The Progressive Approach: Vue.js (2014)[11]

Framework	Year	Key Innovation	Status
Vue.js	2014	**Progressive Adoption:** Designed to be easy to adopt incrementally—you can drop a small Vue app into a corner of an existing page.	Active (Fastest-growing)
		Simplicity & Best of Both Worlds: It combines the **Two-Way Data Binding** of Angular with the **Virtual DOM and Component Model** of React, often with a gentler learning curve.	

In short, these frameworks didn't replace HTML; they built a **smarter, standardized layer *on top of JavaScript*** to make developing modern, complex, interactive web applications **maintainable and performant**.

Basic CSS

Basic CSS: CSS Tutorial

CSS Flexbox: Flexbox Froggy

CSS Grid: Grid Garden

The term **"cascading"** in CSS (Cascading Style Sheets) refers to the algorithm by which the browser determines which style rules to apply when multiple rules target the same element and potentially conflict.

It's a core principle that dictates the hierarchy and order of precedence for applying styles. Essentially, when an element has multiple CSS declarations attempting to style the same property (e.g., `color`, `font-size`), the cascade determines which rule "wins" and is ultimately applied. This process involves several factors:

Origin: Where the style comes from (e.g., browser's default styles, user's custom styles, author's stylesheet). Author styles generally take precedence over browser defaults.

Specificity: A measure of how specific a CSS selector is. More specific selectors (e.g., ID selectors, class selectors) have higher specificity than less specific ones (e.g., element selectors). A rule with higher specificity will override a rule with lower specificity, even if the latter appears later in the stylesheet.

Order of Appearance: If two rules have the same origin and specificity, the rule that appears later in the stylesheet (or in the document's CSS sources) will take precedence. This is often referred to as "the last one wins."

Importance: The !important declaration can be used to explicitly give a rule higher precedence, overriding even more specific rules or those appearing later. However, its use is generally discouraged as it can make stylesheets harder to maintain.

Cascade Layers: A more recent addition to CSS, cascade layers provide a way to explicitly define the order of precedence for groups of styles, offering more control over the cascade. In essence, the "cascading" aspect ensures a predictable and consistent way for browsers to resolve styling conflicts and apply the correct visual presentation to web elements.

Summary: CSS Summary

Evolution of CSS

Evolution of CSS:

- **CSS1 (1996):** Introduced foundational styling capabilities, separating presentation from structure. This included basic font, color, and spacing controls.
- **CSS2 (1998):** Expanded on CSS1 with features like media types for different devices and advanced positioning tools, allowing for more complex layouts.
- **CSS3 (2001-2021):** A modular evolution, CSS3 introduced a vast array of new features in distinct modules. Key additions include:
  - **Animations and Transitions:** Enabling dynamic visual effects.
  - **Flexbox and CSS Grid:** Powerful layout modules for creating flexible and responsive designs.
  - **Media Queries:** Crucial for responsive web design, allowing styles to adapt based on screen size and other device characteristics.

Evolution of CSS Frameworks and Libraries:

- **Early Frameworks (Mid-2000s):** The need for efficient and consistent styling led to the emergence of frameworks like Blueprint, 960, YUI Grids, and YAML. These introduced grid systems to simplify content layout and offered cross-browser compatibility, addressing a major pain point for developers.
- **Rise of Responsive Design:** With the proliferation of mobile devices, frameworks began to heavily incorporate responsive design principles, using techniques like media queries to ensure websites looked good on various screen sizes.
- **Component-Based Frameworks (e.g., Bootstrap):** Frameworks like Bootstrap gained immense popularity by providing pre-styled, reusable components (buttons, forms, navigation bars) and a robust grid system, significantly accelerating development.
- **Utility-First Frameworks (e.g., Tailwind CSS):** A more recent shift, utility-first frameworks like Tailwind CSS offer low-level utility classes that directly correspond to single CSS properties. This approach prioritizes customization and allows developers to build unique designs directly within their HTML, minimizing the need for custom CSS.
- **CSS-in-JS Libraries (e.g., styled-components):** These libraries allow developers to write CSS directly within their JavaScript code, often within the context of component-based architectures like React. This offers benefits like scoped styles and dynamic styling based on component state.

In essence, the evolution of CSS and its frameworks has moved from basic styling to powerful layout tools, responsive design, and more specialized approaches like utility-first and CSS-in-JS, all aimed at improving developer efficiency and enabling richer, more adaptable user experiences.

CSS frameworks and libraries address several common challenges in web development, offering benefits that streamline the styling process and improve project outcomes.

Key reasons for using CSS frameworks and libraries:

- **Faster Development Time:** They provide pre-built components (buttons, forms, navigation bars, etc.) and established styling conventions, eliminating the need to write CSS from scratch for common elements. This accelerates development and allows developers to focus on unique aspects of a project.
- **Consistent Design and Branding:** Frameworks promote a unified visual style across an entire website or application. By using standardized components and styling rules, they help maintain design consistency, ensuring a cohesive user experience and adherence to branding guidelines.
- **Built-in Responsiveness:** Most modern CSS frameworks include robust grid systems and responsive utilities, making it easier to create layouts that adapt seamlessly to different screen sizes and devices (mobile-first design).
- **Improved Collaboration and Maintainability:** Frameworks often come with clear documentation and established conventions, making it easier for multiple developers to collaborate on a project and understand each other's code. This also simplifies long-term maintenance and updates.
- **Cross-Browser Compatibility:** Frameworks are typically built with cross-browser compatibility in mind, addressing potential rendering inconsistencies across different web browsers, saving developers time and effort in debugging.
- **Access to Community Support and Resources:** Popular CSS frameworks have large communities and extensive documentation, providing readily available support and resources for troubleshooting and learning.
- **Foundation for Prototyping:** They offer a quick way to build functional prototypes of websites or applications, allowing for rapid iteration and testing of design ideas before investing in custom styling.

CSS Libraries

Bootstrap Basics

Get started with Bootstrap

Tailwind CSS Basics

Tailwind CSS Docs

Difference

Key Differences Summarized:

Aspect	Bootstrap	Tailwind CSS
Philosophy	Component-based, ready-made UI	Utility-first, granular styling control
Customization	Limited, primarily through overriding styles	Extensive, build designs from scratch
Development Speed	Faster initial setup with pre-built components	Slower initially, but offers more precise control
Learning Curve	Easier for beginners	Steeper, requires mastering utility classes
Performance	Potentially larger file size	Optimized, smaller file size with JIT compiler

Basic website layout

CSS Website Layout

Basic website Layout - CSS

⚓ Tasks-Basic Frontend

Complete Roadmaps > Roadmaps.sh

HTML - Master HTML
CSS - CSS

Gamified Learning

Learn programming by playing games:

1. JavaScript → javascriptquiz.com
2. SQL → mystery.knightlab.com
3. Python →codedex.io
4. CSS →cssbattle.dev
5. Go → codingame.com
6. Java → tynker.com
7. Git → ohmygit.org

Requestly → POSTMAN Killer?