

# Introduction to programming

How to create a website with HTML, CSS, and JS

# Outline

- Computers
- Programming
- The Internet
- Web development
- Modern toolchains
- Website deployment

Slides available at [it-course.ch/Programming.pdf](https://it-course.ch/Programming.pdf).

License for these slides: [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/)

# Warning

- You will be frustrated.
- You will run out of patience.
- This is normal, there's nothing wrong with you.
- Programming is an emotional rollercoaster.
- Struggling makes the result more rewarding.

# Advice

- Treat it as a game and accept the challenge.
- Determine the source of confusion precisely.
- Ask, ask, ask:
  - If you don't understand something, some others don't understand it either.
  - You're not here for me, I'm here for you!

# Goal

- Programming languages have many features.
- Programmers google for solutions all the time.
- Learn how to find and adapt existing solutions.
- I can't teach you three languages in two days.
- I can show you only how to use a “dictionary”.

# Computers

Programmable machines

# Good news

- Computers do exactly what you tell them to do.
- They do it incredibly fast.

# Bad news

- Computers do exactly what you tell them to do.
- They do it incredibly fast.

# Hardware and software

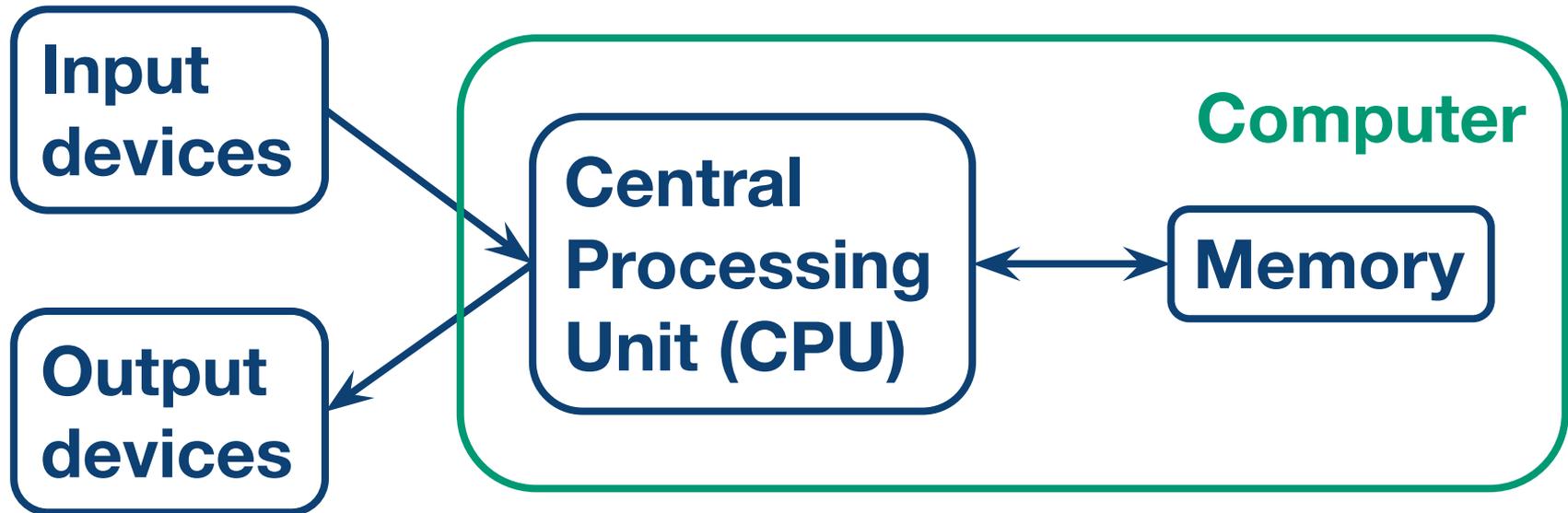
## Hardware:

- Computers (physical)
- General purpose machines
- Expensive to design
- Expensive to copy
- Subjected to wear
- Difficult to fix once shipped

## Software:

- Programs (intangible)
- Special purpose instructions
- Expensive to design
- Free to copy
- Wear-free
- Never really finished, requires maintenance

# Computer (heavily simplified)



- There is **volatile** and **non-volatile** memory.
- Performance is increased with **caches**.
- Also: **Graphics Processing Unit (GPU)**.

# Instructions (heavily simplified)

The **CPU** stores its data in so-called **registers**.

The CPU executes five types of instructions:

- Load data from memory to registers,
- Store data from registers to memory,
- Perform arithmetic operations on registers,
- Perform logical operations on registers, and
- Jump to a given instruction, which can depend on so-called **flags** set by the previous operation.

# Programming

Giving instructions to a computer

# Algorithm



An **algorithm** is a procedure for solving a specified problem in a finite number of steps, i.e. it has to produce an output eventually.

Examples: **Sorting**, **finding the shortest path**, etc.

A **program** is a list of instructions that can be run.

# Compiler



A **compiler** is a program which translates code from a source language into a target language.

A **codebase** can be compiled to different targets.

For now: **source code** → **machine code**, which is usually displayed in an **assembly language**.

# Example

Sum all numbers from 1 to 100  
in the **C programming language**.

Compiled with the **GNU compiler collection** `gcc -S source.c`  
to **x86 AT&T assembly syntax**.

```
#include <stdio.h>
int main() {
    int result = 0;
    int number = 1;
    while (number <= 100) {
        result = result + number;
        number = number + 1;
    }
    printf("Result: %d\n", result);
    return 0;
}
```



```
_main:
    pushq %rbp
    movq  %rsp, %rbp
    subq  $16, %rsp
    movl  $0, -4(%rbp)
    movl  $0, -8(%rbp)
    movl  $1, -12(%rbp)
LBB0_1:
    cmpl  $100, -12(%rbp)
    jg    LBB0_3
    movl  -8(%rbp), %eax
    addl  -12(%rbp), %eax
    movl  %eax, -8(%rbp)
    movl  -12(%rbp), %eax
    addl  $1, %eax
    movl  %eax, -12(%rbp)
    jmp   LBB0_1
LBB0_3:
    movl  -8(%rbp), %esi
    leaq  L_.str(%rip), %rdi
    movb  $0, %al
    callq _printf
    xorl  %eax, %eax
    addq  $16, %rsp
    popq  %rbp
    retq
L_.str:
    .asciz "Result: %d\n"
```

# Explanations

- **Labels** end with `:` (LBB: local block begin).
- **Literal values** start with `$`, **registers** with `%`.
- Suffix of commands determines the **bit-length**:  
**b** for byte (1 byte = 8 bits), **w** for word (2 bytes = 16 bits), **l** for double word (4 bytes = 32 bits), and **q** for quad word (8 bytes = 64 bits).
- **Program flow**: `jmp` means jump unconditionally;  
`jg` means jump on greater than based on the `cmp` (comparison) in the previous instruction.

# Abstraction

Managing registers and memory (allocation and deallocation of space for variables) is a hassle.

This is why almost all programs are written in [high-level programming languages](#), which abstract from the details of the current computing platform.

[Low-level aspects](#) are then handled by a compiler.

You find more information about assembly [here](#).

# The Internet

The network of networks

# The Internet Protocol (IP)

The Internet Protocol made independent networks compatible by introducing a common packet and address format, allowing routing across networks.

Communication over the Internet is unreliable:  
Packets can get lost or arrive out of order.

I wrote an introduction at [ef1p.com/internet](https://ef1p.com/internet).

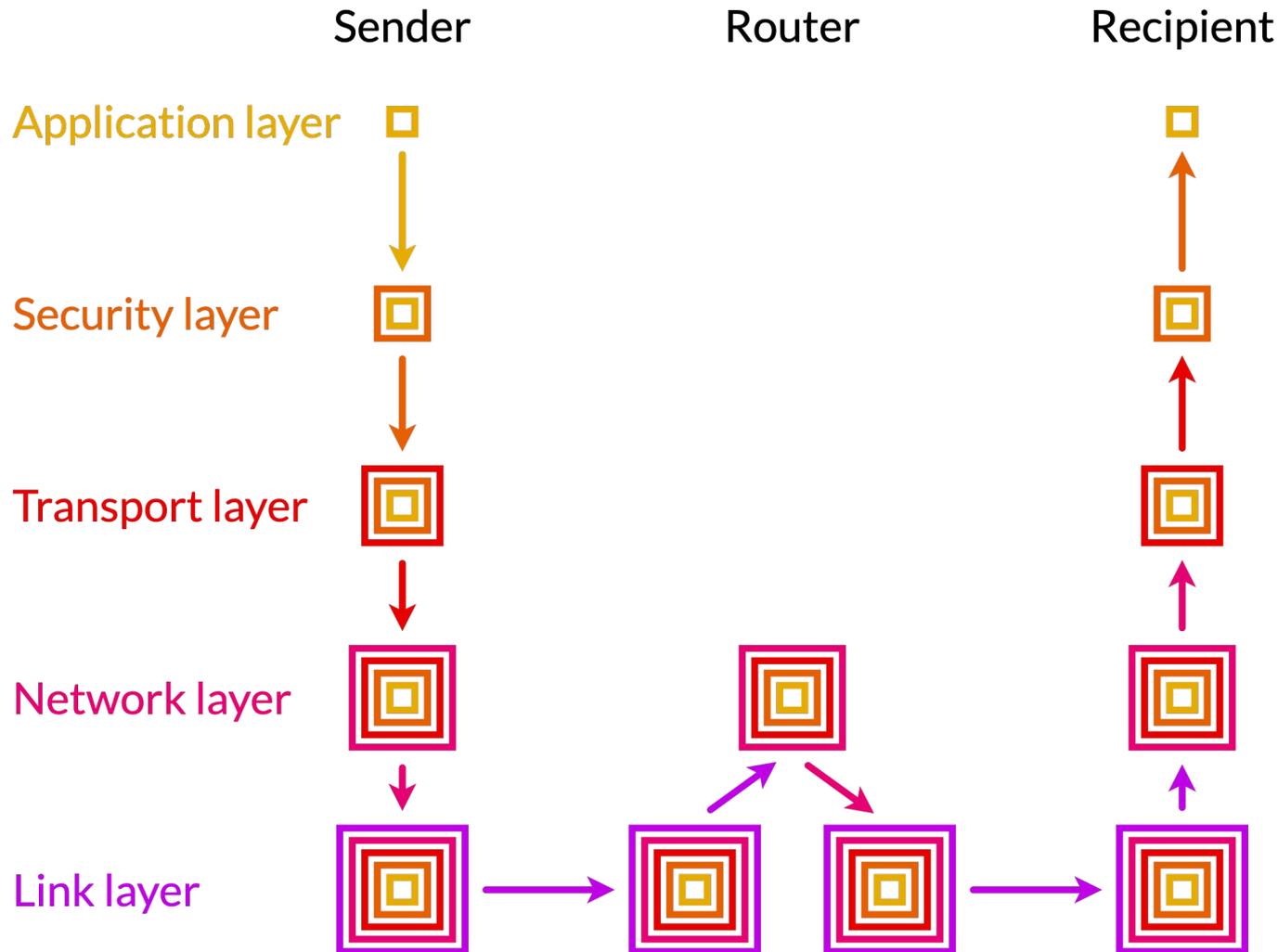
You find the tools at [ef1p.com/internet/tools](https://ef1p.com/internet/tools).

# Internet layers

The Internet operates in layers to be more flexible.

<b>Name</b>	<b>Purpose</b>	<b>Example</b>
Application layer	Application logic	HTTP
Security layer	Encryption and authentication	TLS
Transport layer	Typically reliable data transfer	TCP
Network layer	Packet routing across the Internet	IP
Link layer	Handling of the physical medium	Wi-Fi

# Internet layers visualized





# Client-server model

A client requests a service from a server:



The server's port number depends on the service; the client's port number can be chosen randomly.

Wikipedia has a [list of established port numbers](#).

# Transmission Control Protocol (TCP)

The [Transmission Control Protocol \(TCP\)](#) provides in-order data transfer between two computers.

The sender and the receiver buffer all packets.

The receiver reorders the incoming packets based on a sequence number and asks the sender to resend any missing packets.

TCP also provides [flow](#) and [congestion control](#).

# Transport Layer Security (TLS)

Transport Layer Security (TLS) is the main protocol to provide confidential and authenticated communication over the Internet.

It uses TCP on the transport layer and provides:

- Party authentication with [public-key certificates](#),
- Confidentiality w. [symmetric-key cryptography](#),
- Message authentication with a [hash function](#).

# Domain Name System (DNS)

The [Domain Name System \(DNS\)](#) is a hierarchical namespace of easily memorizable [domain names](#) and an application-layer protocol to access public information associated with such names.

It is most commonly used to look up the IP address of a server in order to connect to it.

You can register a domain name at a [registrar](#).

Example: `dig +short ef1p.com`

# HyperText Transfer Protocol (HTTP)

HTTP is the application-layer protocol of the [World Wide Web \(www\)](#) to transfer files over the Internet.

It's a [text-based protocol](#) with two versions:

- HTTP over TCP on port 80 and
- HTTPS over TLS on port 443.

```
$ openssl s_client -quiet -crlf -connect  
explained-from-first-principles.com:443  
GET /internet/ HTTP/1.0  
Host: explained-from-first-principles.com
```

# Uniform Resource Locator (URL)

A [Uniform Resource Locator \(URL\)](#) identifies a resource with the following syntax:

scheme://domain:port/path?querystring#fragment

Example: `https://ef1p.com/internet/#preface`

# Web development

How to create a website from scratch

# Web languages

A [web browser](#) retrieves a [web page](#) from a [web server](#) and renders its content for you. The server can send static files or generate them dynamically.

There are three languages for web pages:

- [HyperText Markup Language \(HTML\)](#) for content,
- [Cascading Style Sheets \(CSS\)](#) for design,
- [JavaScript \(JS\)](#) for interactivity.

Only JavaScript is a real [programming language](#).

# HyperText Markup Language (HTML)

HTML structures the content of a web page with tags. The content is put between an opening tag, e.g. `<p>`, and a corresponding closing tag, `</p>`.

Tags can have attributes: `<a href="d.html">`.

Elements can be nested: `<p>A <i>B</i></p>`.

... unless they're **void**: ``.

There are **many HTML tags**, but you don't have to know them all. Just google what you're looking for.

# HTML example

```
<!doctype html>
<html>
  <head>
    <title>Title of web page</title>
  </head>
  <body>
    <h1>Heading</h1>
    <p>Paragraph with a
      <a href="d.html">link</a>.</p>
    
  </body>
</html>
```

# The doctype

For legacy reasons, HTML5 documents must start with the following, case-insensitive **doctype**:

```
<!doctype html>
```

(This is not a proper HTML tag and isn't closed.)

# Code indentation

Indent code with spaces (or tabs) to make it easier to recognize the structure and spot missing tags.

```
<html>
  <head>
    <title>Title</title>
  </head>
  <body>
    ...
  </body>
</html>
```

# HTML link examples

```
<html>
  <body>
    <p>You can link to a:</p>
    <ul id="list">
      <li><a href="https://ef1p.com">website</a></li>
      <li><a href="about.html">local file</a></li>
      <li><a href="#list">element on page</a></li>
    </ul>
  </body>
</html>
```

p: paragraph

ul: unordered list

li: list item

a: anchor

# Cascading Style Sheets (CSS)

CSS describes how the content is to be styled.

By separating content from presentation, the same style can be used for many elements and across different pages of the same website.

There are more than 200 [CSS properties](#).

```
p {  
    font-size: 12px;  
    color: blue;  
}
```

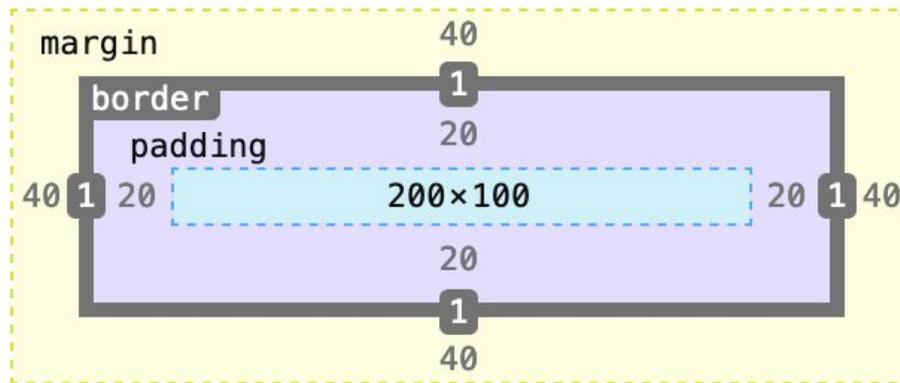
# CSS selectors

- **Universal:** \*
- **Type:** h1 (applies to `<h1></h1>`)
- **Class:** .test (e.g. `<p class="test"></p>`)
- **ID:** #name (for `<a id="name"></a>`)
- **List:** h1, .test, #name (matches all)
- **Descendant:** p a (all links inside a paragraph)
- **Child:** p > a (links belonging to a paragraph)
- **Adjacent sibling:** p + p (p preceded by p)
- And [some more](#)

# CSS box model

```
p {  
  width: 200px;  
  height: 100px;  
  padding: 20px;  
  border: 1px solid blue;  
  margin: 40px;  
}
```

Result:



# CSS embedded in HTML head

```
<html>
  <head>
    <style type="text/css">
      p {
        color: red;
      }
    </style>
  </head>
  <body>
    <p>Hello, World!</p>
  </body>
</html>
```

# CSS embedded in style attribute

```
<html>  
  <body>  
    <p style="color: red;">  
      Hello, World!  
    </p>  
  </body>  
</html>
```

# CSS loaded from separate file

```
<html>
  <head>
    <link rel="stylesheet" href="s.css">
  </head>
  <body>
    <p>Hello, World!</p>
  </body>
</html>
```

# JavaScript (JS)

JavaScript is a programming language to make web pages dynamic and interactive.

JavaScript can modify the content, structure, and layout of a web page and load and submit data.

The browser executes JavaScript in a [sandbox](#), preventing you from accessing the user's files.

Every browser offers powerful [developer tools](#).

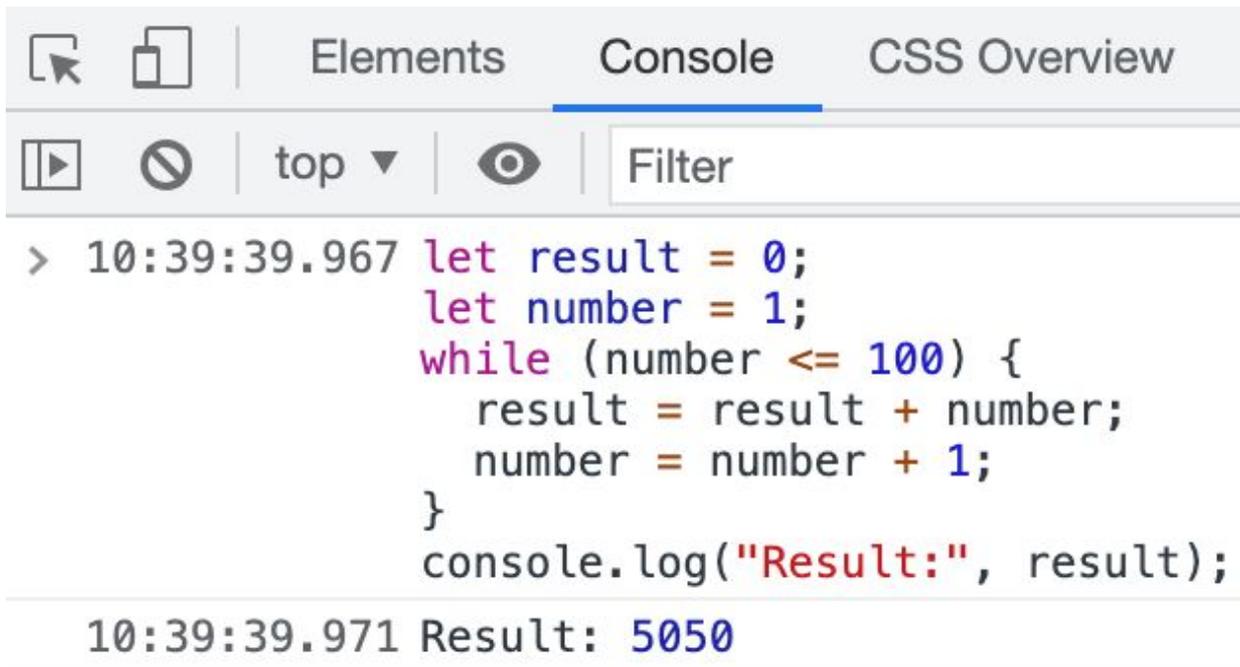
In particular, it has a [read-eval-print loop \(REPL\)](#).

# JavaScript example

```
let result = 0;
let number = 1;
while (number <= 100) {
    result = result + number;
    number = number + 1;
}
console.log("Result:", result);
```

# JavaScript console

In the console of the developer tools:



The screenshot shows the developer console with the 'Console' tab selected. The console contains a JavaScript code snippet that calculates the sum of numbers from 1 to 100. The code is as follows:

```
> 10:39:39.967 let result = 0;
    let number = 1;
    while (number <= 100) {
      result = result + number;
      number = number + 1;
    }
    console.log("Result:", result);
```

Below the code, the console shows the output of the `console.log` statement:

```
10:39:39.971 Result: 5050
```

# Essential syntax

- Delimit each statement with a **semicolon**
- **Variable declaration:** `let variable = 'a';`
- **Branching:** `if (condition) {...} else {...}`
- **Looping:** `while (condition) {...}`
- **Function:**

```
function sum(a, b) {  
    return a + b;  
}
```
- **Comments:** `// This is a line comment.`  
`/* Block comments can span lines. */`

# Example of an object (a data type)

```
// Declaration:
```

```
let coordinate = { x: 2, y: 3 };
```

```
// Accessing its properties:
```

```
console.log(coordinate.x);
```

```
// Assigning an additional property:
```

```
coordinate.z = 4;
```

# Example of an array (a data type)

```
let array = [6, 2, 4, 3, 5];  
  
// Accessing one of its elements:  
console.log(array[2]); // Output: 4  
  
// Loop through all its elements:  
let index = 0;  
while (index < array.length) {  
    console.log(array[index]);  
    index = index + 1;  
}
```

# Assignment versus equality

- **Assignment:** `let a = 2;` (2 assigned to a)
- **Loose equality:** `a == '2'` (results in `true`)
- **Strict equality:** `a === '2'` (results in `false`)

When using loose equality, JavaScript attempts to convert the type of one side to match the other side. This is confusing and should not be used.

Inequality with `!==`, for example `a !== 2`.

# Event handler example

```
function keyDownHandler(event) {  
    console.log(event.key);  
}
```

```
document.addEventListener('keydown',  
keyDownHandler);
```

Using an anonymous function instead:

```
document.addEventListener('keydown',  
function (e) {console.log(e.key);});
```

# JavaScript embedded in HTML

```
<html>
  <body>
    <p id="content"></p>
    <script>
document.getElementById('content')
.textContent = 'Hello, World!';
    </script>
  </body>
</html>
```

# JavaScript loaded from separate file

```
<html>
  <head>
    <script defer src="s.js"></script>
  </head>
  <body>
    <p id="content"></p>
  </body>
</html>
```

And in the file `s.js`:

```
document.getElementById('content')
.textContent = 'Hello, World!';
```

# Modern toolchains

How websites are actually built

# Integrated development environment

HTML, CSS, and JS are stored in simple [text files](#).

You could edit them with TextEdit.app on macOS or Notepad.exe on Windows.

However, an [integrated development environment \(IDE\)](#), such as [Visual Studio Code](#), makes development much easier thanks to [code completion](#), [documentation tooltips](#), [code linting](#), [source-level debugger](#), [unit testing](#), and [plugins](#).

# Libraries for CSS and JS

Don't start from scratch, build on code from others (also to improve browser compatibility).

Examples:

- **CSS:** [Bootstrap](#), [Tailwind](#), [Materialize](#), ...
- **JS (generic):** [jQuery](#), [Underscore](#), ...
- **JS (for state updates):** [React](#), [Angular](#), [Vue](#), ...
- **JS (for math equations):** [KaTeX](#), [MathJax](#), ...

# Package manager

Instead of installing and updating libraries yourself, you can use a package manager to manage your dependencies.

For JavaScript, the dominant code ecosystem is [npm](#) ([Node](#) package manager). Other tools, such as [yarn](#), rely on the same ecosystem.

# Module bundler

Instead of including all your files and dependencies separately in your web page, you can bundle them into a single file with a module bundler.

Bundlers can usually also [minify](#) your source code by removing unnecessary whitespace and by shortening the names of local variables.

Example: [webpack](#)

# Version control system

If you collaborate with others, you need a way to submit (“commit”) and merge changes. This is accomplished with a [version control system](#).

The most dominant system nowadays is [Git](#).

A popular storage provider for Git [repositories](#) is [GitHub](#), which is owned by Microsoft since 2018.

Before accepting a [pull/merge request](#), you may want to run all [unit tests](#) against the change. This is usually automated with [continuous integration](#).

# Preprocessor languages

HTML, CSS, and JavaScript are implemented by various browsers. These standards evolve slowly.

To meet their needs, developers and companies specify their own languages and compile (also known as [transpile](#)) them into HTML, CSS, or JS.

Examples:

- **HTML** (template engines): [Liquid](#), [Nunjucks](#), ...
- **CSS**: [Sass](#), [Less](#), ...
- **JS**: [TypeScript](#), [JSX](#), ...

# Runtime environment

Run JavaScript on the server instead of in the browser with [Node.js](#) or [Deno](#).

These runtime environments have [application programming interfaces \(APIs\)](#) for file system, network, and database access.

Many developers specialize in [frontend](#) (client or browser) or [backend](#) (server) development. If you can do both, you're a [full-stack developer](#).

# Website deployment

How to publish your website

# Website hosters

To publish your website, you need a server which serves your files and optionally a custom domain name if you don't want to use the provider's URL.

If your website is static (none of its content is generated on the server and no database is needed), you can use [GitHub Pages](#) for free.

If you want to run code on the server, many cloud providers, such as [Glitch](#) and [Render](#), have a free tier, where your instance hibernates when it is idle.