Frontend

Frontend is the user-facing layer of a web application. It uses **HTML** for structure, **CSS** for presentation, and **JavaScript** for behavior. The frontend should be accessible, responsive, and interact with backend APIs to show dynamic data.

HTML — Structure, Forms, Semantics

Purpose

HTML provides the document structure and semantic meaning for content so browsers and assistive technologies (screen readers) can interpret it.

Key Concepts

- **Document structure**: <!DOCTYPE html>, <html>, <head>, <body>
- **Semantic tags**: Use <header>, <nav>, <main>, <section>, <article>, <aside>, <footer> to convey meaning.
- **Headings**: <h1>–<h6> used hierarchically.
- **Forms**: <form>, <input>, <label>, <select>, <textarea>, type attributes.
- **Accessibility**: alt on images, aria-* attributes, labels tied to inputs.

Example — Basic Page with Form

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,initial-scale=1" />
  <title>Example App</title>
</head>
<body>
  <header>
    <h1>Example App</h1>
    <nav>
      <a href="#home">Home</a> | <a href="#contact">Contact</a>
    </nav>
  </header>

  <main>
    <section id="contact">
      <h2>Contact Us</h2>
      <form id="contactForm" aria-label="Contact form">
        <label for="name">Name</label>
        <input id="name" name="name" type="text" required />

        <label for="email">Email</label>
        <input id="email" name="email" type="email" required />

        <label for="message">Message</label>
        <textarea id="message" name="message" rows="4"></textarea>

        <button type="submit">Send</button>
      </form>
    </section>
  </main>

  <footer>
    <p>&copy; 2025 Example</p>
```

```
  </footer>

  <script src="app.js"></script>
</body>
</html>
```

CSS

CSS (Cascading Style Sheets) defines how HTML content looks and behaves visually: layout, spacing, typography, colors, responsive behavior and small interactions (transitions/animations). Good CSS is maintainable, performant, accessible, and predictable across screen sizes.

---

1. Box Model (foundation)

Every element is a rectangular box composed of (from inside out):

- **content** — the element's text/children area
- **padding** — space between content and border
- **border** — outline around padding
- **margin** — space between this element's border and neighboring elements

Important properties:

```
element {
  width: 200px;       /* content width if box-sizing: content-box */
  padding: 16px;
  border: 2px solid #ccc;
  margin: 8px;
  box-sizing: border-box; /* recommended: includes padding+border in width/height */
}
```

Best practice: use box-sizing: border-box; globally so width/height include padding and border (simpler layout math).

```
*,
*::before,
*::after { box-sizing: border-box; }
```

Collapsing margins

Adjacent vertical margins between block elements can collapse — be aware when using margin-top / margin-bottom. Use padding or borders to avoid surprises.

---

2. Display Types

How elements participate in layout.

- **block** — takes full width, starts on a new line (e.g., , <p>).
- **inline** — flows within a line, width/height ignored (e.g., , <a>).
- **inline-block** — flows inline but respects width/height.
- **flex** — establishes a flex container; children become flexible items.
- **grid** — establishes a grid container for two-dimensional layout.

Example:

```
<div class="box">Block</div>
<span class="inline">Inline</span>
```

---

3. Flexbox — 1-D layout (rows/columns)

Use when you need to align items along one axis and distribute space.

Core container properties:

- display: flex;
- flex-direction — row | row-reverse | column | column-reverse
- justify-content — main-axis alignment (flex-start, center, space-between, ...)
- align-items — cross-axis alignment (stretch, center, flex-end, ...)
- gap — spacing between flex items

Core item properties:

- flex — shorthand for flex-grow flex-shrink flex-basis (e.g., flex: 1 1 0).
- order — control visual ordering.

Practical example — header with logo + nav + CTA that adapts:

```
.header { display: flex; align-items: center; justify-content: space-between; gap: 1rem; padding: 1rem; }
.logo { flex: 0 0 auto; }
.nav { display: flex; gap: 1rem; }
.cta { flex: 0 0 auto; }
```

Example — equal-width cards:

```
.card-container { display: flex; gap: 16px; }
.card { flex: 1; /* each card grows equally */ padding: 16px; }
```

When to use Flexbox:

- Nav bars, toolbars, horizontal lists
- Centering (both axes) of a single element
- Distributing items in a single row/column

---

4. CSS Grid — 2-D layout (rows & columns)

Grid is powerful for designing overall page layouts and complex components.

Core container properties:

- display: grid;
- grid-template-columns / grid-template-rows — define tracks (e.g., 1fr 300px)
- grid-template-areas — semantic named areas
- gap — spacing between rows/columns

Example — two-column layout with sidebar:

```
.page {
  display: grid;
  grid-template-columns: 1fr 320px;
  gap: 24px;
}
.main { grid-column: 1; }
.sidebar { grid-column: 2; }
```

Example — responsive grid using auto-fit / minmax:

```
.gallery {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(220px, 1fr));
  gap: 16px;
}
```

When to use Grid:

- Page-level layouts
- Complex components that require both axes control (e.g., dashboards)

Tip: combine Grid for the page and Flexbox for components inside grid items.

---

5. Responsive Design (mobile-first recommended)

Goal: UI adapts to many screen sizes and input methods.

Mobile-first approach

Write base styles for small screens, then add @media (min-width: …) for larger screens.

Common breakpoints (examples only — adapt to design):

- small phones < 480px
- tablets >= 600px
- desktops >= 1024px
- large desktops >= 1280px

Example:

```
/* Base (mobile) */
.container { padding: 1rem; }
```

```
/* Tablet and up */
@media (min-width: 768px) {
  .container { max-width: 720px; margin: 0 auto; }
}

/* Desktop */
@media (min-width: 1024px) {
  .container { max-width: 1024px; }
}
```

Responsive units

- rem/em — typography & spacing scaled to root/font-size
- % — relative width/height
- vw/vh — viewport units
- fr — grid fractional units

Prefer rem for consistent spacing, minmax() and auto-fit for fluid grids.

Responsive images

Use <img srcset> and sizes or <picture> to serve appropriate images for different viewports.

Example:

```
<img src="small.jpg"
    srcset="small.jpg 480w, medium.jpg 800w, large.jpg 1200w"
    sizes="(max-width: 600px) 100vw, (max-width: 1200px) 50vw, 33vw"
    alt="...">
```

Container Queries (modern)

Container queries let components adapt based on parent width. Useful for reusable components. (Check browser support before using.)

---

6. Utility Classes & Design System

Utility classes are small, reusable classes for spacing, typography, alignment.

Example utilities:

```
.mt-1 { margin-top: 0.25rem; }
.p-2  { padding: 0.5rem; }
.text-center { text-align: center; }
.flex { display: flex; }
.gap-4 { gap: 1rem; }
```

Benefits:

- Fast development, less repetition
- Encourages consistent spacing & scale

Combine with CSS variables (design tokens):

```
:root {
  --space-1: 4px;
  --space-2: 8px;
  --primary: #0b74de;
  --radius: 8px;
}
.btn { background: var(--primary); border-radius: var(--radius); padding: var(--space-2); }
```

---

7. CSS Architecture & Naming (maintainability)

- **BEM** (Block__Element--Modifier) for clear structure: .card, .card__title, .card--featured.
- **Atomic / Utility-first** (Tailwind style) for rapid UI.
- **Component-based**: style per component if using frameworks (scoped CSS, CSS Modules).
- Keep CSS modular and avoid deeply nested selectors.

---

8. Cascade, Specificity & Inheritance (rules you must know)

- Order of precedence: browser → author → user → inline styles → !important.
- Specificity: inline > id > class/attribute/pseudo-class > element/pseudo-element.

- Use low specificity selectors and avoid !important. Prefer class-based styles.

---

9. Transitions & Simple Animations

Small animations improve perceived quality — use transition for hover/focus, transform (GPU-accelerated) for movement.

Example:

```
.button {
  transition: transform 150ms ease, box-shadow 150ms ease;
}
.button:hover { transform: translateY(-2px); box-shadow: 0 6px 18px rgba(0,0,0,0.12); }
```

For complex sequences, prefer CSS @keyframes or animation libraries; keep animations short and optional (reduced-motion preference).

Respect prefers-reduced-motion:

```
@media (prefers-reduced-motion: reduce) {
  * { animation: none !important; transition: none !important; }
}
```

---

10. Accessibility & UX Considerations

- Ensure color contrast (WCAG AA minimum).
- Focus styles: don't remove outline — style it instead.
- Use :focus-visible for keyboard focus.
- Avoid relying solely on color to communicate meaning.
- Make tap targets >= 44px.
- Provide smooth resizing and readable font sizes (use rem, set html { font-size: 100% }).

---

11. Performance & Best Practices

- Minimize reflows/repaints by batching DOM reads/writes and avoiding expensive layout triggers (e.g., reading offsetHeight after modifying DOM).
- Use will-change sparingly.
- Compress and combine CSS (build step) and use critical CSS for above-the-fold content.
- Avoid huge CSS files — tree-shake unused rules (purge CSS with build tools).
- Use prefers-reduced-data/network hints when loading heavy assets.

---

12. Debugging Tools & Workflow

- Browser DevTools: Elements, Styles, Computed, Layout, Performance, Lighthouse.
- Use outline: 1px solid red; during layout debugging.
- Lint CSS (stylelint), auto-format (Prettier).
- Use source maps in development.

---

13. Example: Putting It All Together (Responsive Card Grid)

```
<section class="cards">
  <article class="card">
    <img src="..." alt="..." />
    <h3 class="card__title">Title</h3>
    <p class="card__desc">Description</p>
  </article>
  <!-- more cards -->
</section>
.cards {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(260px, 1fr));
  gap: 1rem;
  padding: 1rem;
}
```

```
.card {
  background: #fff;
  border-radius: 12px;
  box-shadow: 0 6px 16px rgba(0,0,0,0.06);
  overflow: hidden;
  display: flex;
  flex-direction: column;
}

.card img { width: 100%; height: 160px; object-fit: cover; }
.card__title { margin: 0.75rem 1rem 0 1rem; font-size: 1.125rem; }
.card__desc { margin: 0.5rem 1rem 1rem 1rem; color: #555; flex: 1; }
```

JavaScript — DOM, Fetch API & Events

JavaScript adds interactivity and logic to web pages. It connects your UI (HTML/CSS) with dynamic behaviour such as validating forms, calling APIs, updating UI automatically, or responding to user actions.

---

1. Purpose

JavaScript controls how the webpage behaves:

- Dynamically updates HTML/CSS
- Validates inputs and processes data
- Fetches information from servers (APIs)
- Handles events like clicks, typing, scrolling
- Builds Single Page Applications (SPA)
- Drives frontend logic

It is the “brain” of the frontend.

---

2. Key Concepts

2.1 DOM (Document Object Model)

The DOM is a tree-like structure that represents your HTML in JavaScript. It allows you to **access and manipulate** elements.

Why it’s important

- Update text, images, styles.
- Show/hide elements.
- Add or remove elements dynamically.

Common DOM Methods

Selecting Elements

- document.getElementById("id")
- document.querySelector(".class")
- document.querySelectorAll("div")

Manipulating Elements

- element.textContent = "Hello";
- element.innerHTML = " Hi ";
- element.style.color = "red";
- element.classList.add("active");
- element.classList.remove("hidden");

Creating Elements

```
const box = document.createElement("div");
box.textContent = "New Element";
document.body.appendChild(box);
```

Removing Elements

```
element.remove();
```

2.2 Events

Events allow JavaScript to **respond to user actions**.

Common Events

- click — button clicks
- input — typing inside input fields
- submit — form submission
- mouseover — hover
- change — dropdown change
- keydown — keyboard press

Event Listeners

```
button.addEventListener("click", function () {
  alert("Button clicked!");
});
```

Event Object

Provides details about the event:

```
input.addEventListener("input", (e) => {
  console.log(e.target.value);
});
```

Prevent Default Behavior

Useful for forms:

```
form.addEventListener("submit", (e) => {
  e.preventDefault();
  console.log("Form submitted without reloading!");
});
```

2.3 Fetch API

Used to communicate with backend APIs (GET, POST, PUT, DELETE)

1. GET Request — Fetching Data

```
fetch("https://api.example.com/users")
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

2. POST Request — Sending Data

```
fetch("https://api.example.com/login", {
  method: "POST",
  headers: {"Content-Type": "application/json"},
  body: JSON.stringify({username: "john", password: "1234"})
})
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

3. Handling Errors

```
try {
  const res = await fetch(url);
  if (!res.ok) throw new Error("Network error");
} catch (error) {
  console.error(error);
}
```

2.4 DOM + Fetch + Events (Real Example)

Example: Load list on button click

```
document.getElementById("loadBtn").addEventListener("click", async () => {
  const response = await fetch("/api/products");
  const products = await response.json();
```

```
  const list = document.getElementById("productList");
  list.innerHTML = "";

  products.forEach(p => {
    const li = document.createElement("li");
    li.textContent = p.name;
    list.appendChild(li);
  });
});
```

What this does:

- User clicks button
- JS fetches data from backend
- DOM updates with product list

BACKEND

REST API — Detailed Explanation

1. What is REST?

REST (**Representational State Transfer**) is an **architectural style** used for designing **scalable, maintainable, lightweight** web services.

It defines **how clients and servers should communicate** over the internet using simple, predictable rules.

Key Principles of REST

1. **Stateless**
   - Every request from the client must contain all information needed.
   - Server does *not* store session data.
   - Improves scalability.
2. **Client–Server Separation**
   - UI (Frontend) and logic/data (Backend) are independent.
   - Either can change without affecting the other.
3. **Uniform Interface**
   - Standard method of communication.
   - Same structure for all APIs.
4. **Resource-Based**
   - Everything is treated as a **resource**.
   - Resources are identified using **URIs** (Uniform Resource Identifiers).
5. **Use of Standard HTTP Methods**
   - GET, POST, PUT, PATCH, DELETE
6. **Stateless Communication**
   - Each request must be independent.

---

2. HTTP Methods and Their Meaning

Method	Meaning	Usage
GET	Retrieve resource	Fetch data, list, or details
POST	Create new resource	Add new user/data
PUT	Update entire resource	Replace the full object
PATCH	Partially update resource	Update some fields
DELETE	Remove resource	Delete user/item

1. GET (Read Data)
Used to fetch data from server
Example:
GET /api/products
2. POST (Create Data)
Used to create a new resource
Example:
POST /api/users
3. PUT (Full Update)
Replaces the entire object
Example:
PUT /api/users/5
4. PATCH (Partial Update)
Updates only specific fields
Example:
PATCH /api/users/5
5. DELETE (Remove Data)
Deletes a resource
Example:
DELETE /api/users/5

---

3. Resource URLs (Endpoints)
REST APIs follow a predictable URL pattern.
User Resource Example

Operation	Endpoint
List all users	GET /api/users
Get user by ID	GET /api/users/:id
Create new user	POST /api/users
Update user	PUT /api/users/:id
Delete user	DELETE /api/users/:id

Important:

- Use **nouns**, NOT verbs
  ❌ /api/getUsers
  ✓ /api/users
- Use **plural nouns**
  ✓ /api/products
  ✓ /api/orders

---

4. Request & Response Structure
REST APIs usually exchange data in **JSON** format.
Request Format
Headers
Content-Type: application/json
Authorization: Bearer <token> (optional)
Request Body (for POST/PUT/PATCH)

```
{
  "name": "Dinesh",
  "email": "dinesh@example.com"
}
```

---

5. HTTP Status Codes (Must Know)

Code	Meaning	Use Case
200 OK	Success	GET, PUT, PATCH
201 Created	New resource created	POST
204 No Content	Success without response body	DELETE
400 Bad Request	Invalid input	Missing fields
401 Unauthorized	Invalid/missing authentication	No token
403 Forbidden	Access denied	No permission
404 Not Found	Resource does not exist	Invalid ID
409 Conflict	Duplicate entry	Email already taken
500 Internal Server Error	Server crash/error	Logic failure

---

6. Example — JSON Response

A typical REST API response looks like this:


```
{
  "id": 123,
  "name": "Dinesh",
  "email": "dinesh@example.com",
  "createdAt": "2025-02-05T14:32:21Z"
}
```


---

7. Example — Complete REST API Request & Response

POST /api/users

Request:


```
{
  "name": "Dinesh",
  "email": "dinesh@example.com",
  "password": "12345"
}
```


Response:

Status: **201 Created**


```
{
  "id": 101,
  "name": "Dinesh",
  "email": "dinesh@example.com",
  "message": "User created successfully"
}
```


---

8. Example — GET /api/users/101

Response:

Status: **200 OK**


```
{
  "id": 101,
  "name": "Dinesh",
  "email": "dinesh@example.com"
}
```


---

9. Best Practices for REST API

✓ **Use nouns for resources**

/api/products, /api/users

✓ **Use proper status codes**

Avoid sending 200 for everything.

✓ **Make API predictable and consistent**

✓ **Use filtering, sorting, pagination**

Examples:

GET /api/products?page=1&limit=10

GET /api/users?sort=asc

✓ **Protect sensitive endpoints using authentication**

- JWT (Bearer token)
- OAuth (optional)

✓ **Validate user input before saving**