Ministry of Higher Education and Scientific Research
University of Abdelhafid Boussouf - Mila
Institute of Mathematics and Computer Science
Department of Computer Science
Master 1 STIC - 2025/2026

Advanced Web Development

Chapter 3 : Server-Side Logic, Data & APIs.

Presented by: **Dr. Brahim Benabderrahmane**

Chapter 3 Overview

Section 1 : Theoretical Foundations – Architectures, REST & MVC

Objectives : High-level system design, API communication rules, and architectural patterns without code.

Section 2 : Technical Implementation – Node.js, Express & Routing

Objectives : Setting up the runtime environment, mastering the Express.js framework, and building RESTful endpoints.

Section 3 : Technical Implementation – MVC Architecture & PostgreSQL

Objectives : Refactoring the server into an MVC folder structure, connecting to PostgreSQL, and writing robust SQL/ORM logic.

01 Theoretical Foundations Architectures, REST & MVC

Objectives :

- High-level system design
- API communication rules
- Architectural patterns

The Client-Server Model

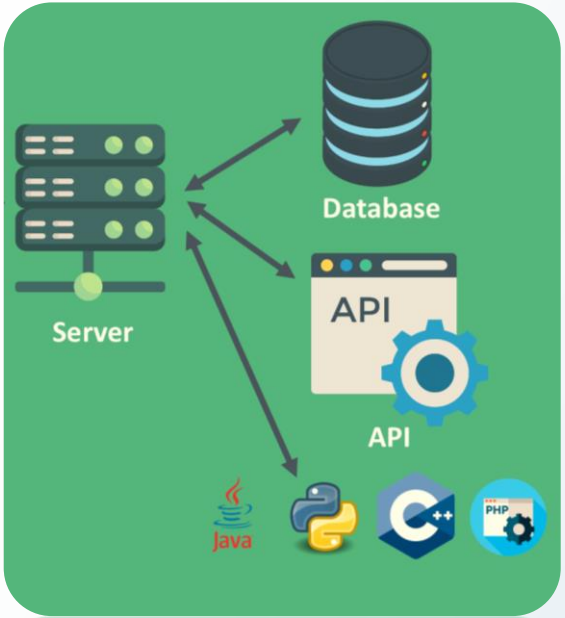
How Frontends and Backends Communicate over the Web

The Concept:
A distributed application structure that partitions tasks between the providers of a resource (Servers) and service requesters (Clients).
The Client (Frontend):
Your React SPA, mobile app, or IoT device. It handles the UI, user events, and local state. It is inherently untrusted.
The Server (Backend):
A remote, centralized computer that listens for requests, processes business logic, and manages the database. It is the trusted source of truth.
The Bridge:
They communicate over the internet using the HTTP protocol.

Anatomy of a Backend

Servers, Databases, and Application Logic

1. The Server Application (The Logic):
A program written in Node.js, Python, or Java that runs 24/7, listening for incoming network requests.
2. The Database (The Memory):
A structured storage system (like PostgreSQL) that securely saves users, products, and orders to a hard drive.
3. The API (The Messenger):
The set of rules that dictates exactly how the Client is allowed to ask the Server Application for data.



Evolution of Backend Architectures

Adapting to the Scale of the Web

1. **1990s - 2000s:**
   Server-Side Rendered (SSR) Monoliths (PHP, Java JSP).
   The server built the HTML and sent it to the browser.
2. **2010s:**
   The API Revolution.
   The rise of smartphones forced backends to stop sending HTML and start sending raw JSON data, leading to the SPA architecture we use today.
3. **2020s:**
   Cloud-Native & Distributed Systems.
   Backends are no longer single applications, but networks of cooperating services.

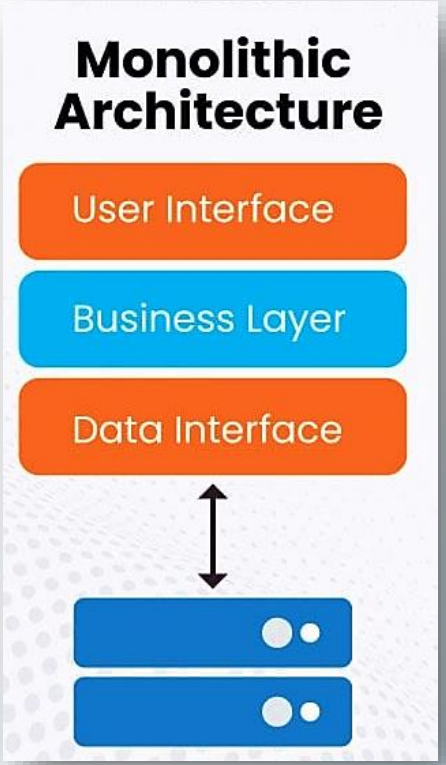
Monolithic Architecture

The Traditional "Everything in One Place" Approach

Definition: A single, massive codebase containing all the business logic; Authentication, Payments, Inventory, and User Management are all compiled into one application.

Pros: Extremely simple to develop, test locally, and deploy (just upload one folder to a server).

Cons: As the team and user base grow, a bug in the "Inventory" module can crash the "Payments" module, bringing the entire store down. Scaling is inefficient because you must duplicate the entire app even if only the "Checkout" feature is experiencing heavy traffic.



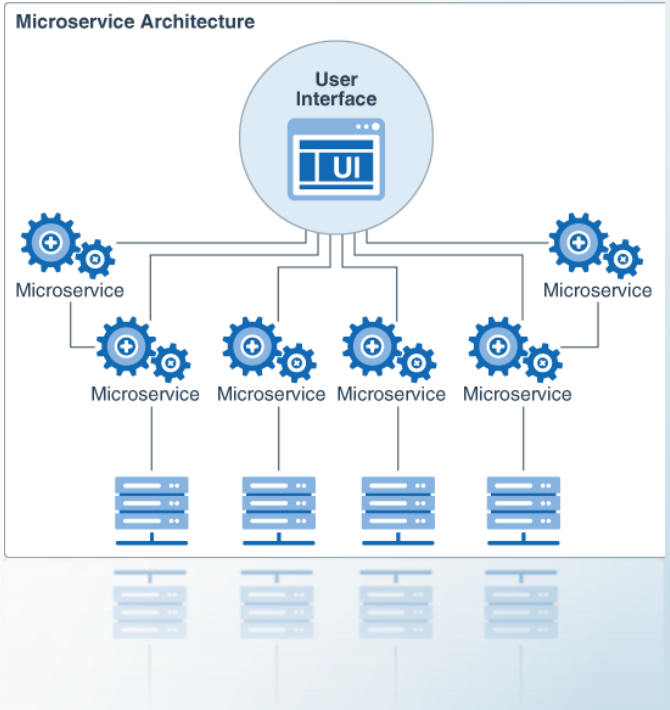
Microservices Architecture

Dividing and Conquering Complexity for Scale

Definition:
Breaking the Monolith down into dozens of tiny, independent backends. You have one server just for Users, one server just for Products, and one just for Payments.

Pros:
Incredible scalability. If it's Black Friday and payments are overloaded, you only scale up the Payment service. Teams can work independently using different programming languages.

Cons:
Enormous DevOps complexity. You now have to manage network latency between services and handle partial failures (e.g., Payment succeeds but Inventory fails).



Serverless and Edge Computing

Executing Code on Demand Without Server Management

The Concept:
You do not rent a server that runs 24/7. Instead, you write single functions and upload them to AWS or Vercel.

The Execution:
The cloud provider spins up a micro-container to run your function only when a request arrives, then destroys it milliseconds later.

The Benefit:
You pay exactly for what you use (per millisecond of compute time), and it scales infinitely automatically.

Our Scope:
For this Master's course, we will build a modern Monolithic API, as it is the prerequisite to understanding Microservices or Serverless.

What is an API?

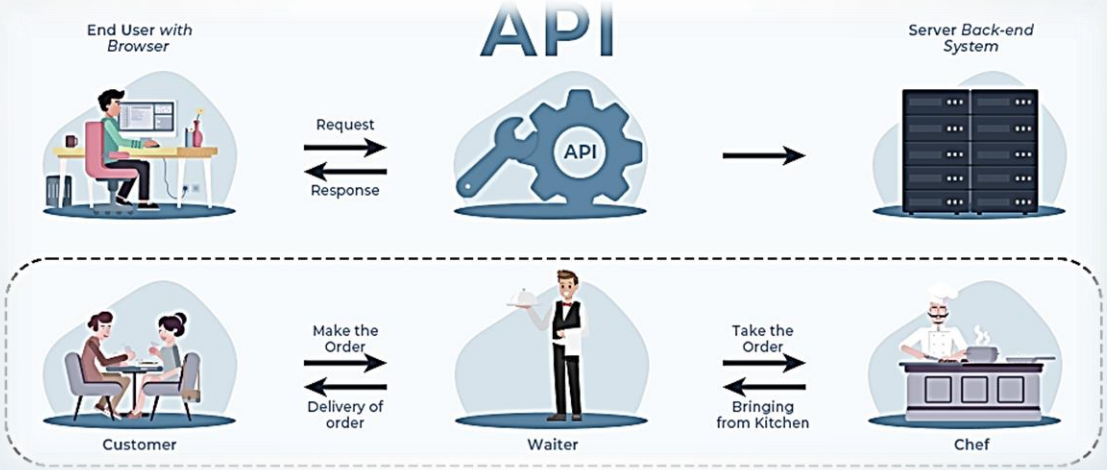
API: Application Programming Interface.

The Analogy:

An API is like a waiter in a restaurant. You (the Client) have a menu. You cannot go into the kitchen (the Database) to cook your own food. You give your order (the Request) to the waiter (the API), who takes it to the kitchen and brings back your food (the Response).

The Contract:

An API defines exactly what requests are valid, what data you must provide, and what data you will receive in return.



The HTTP Protocol

Verbs, Headers, and the Anatomy of a Network Request

HTTP: HyperText Transfer Protocol. It is a text-based protocol.

A Request contains:

1. **URL:** Where the message is going (e.g., api.dzshop.com/products).
2. **Method (Verb):** What we want to do (GET, POST).
3. **Headers:** Metadata (like Authorization tokens or declaring Content-Type: application/json).
4. **Body:** The actual data payload (e.g., the JSON string of the new product).

A Response contains:

1. **A Status Code**
2. **Headers**
3. **Body.**

Introduction to REST

REST:

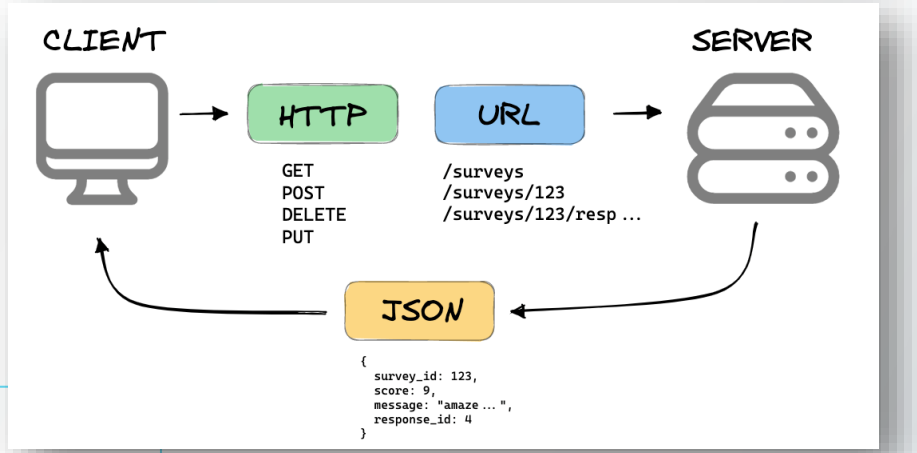
Representational State Transfer. It is not a technology, but a set of architectural rules for designing predictable web APIs.

The Core Idea:

Instead of thinking about "Actions" (like a function getAllUsers()), REST forces us to think about "Resources" (like the noun /users).

The Representation:

When the client requests a resource, the server sends back a representation of its current state, usually formatted as JSON.



Statelessness in REST

Amnesia on the Server: Why Every Request Must Stand Alone

The Rule:
The server must not store any information about the client's session between requests.
Every single HTTP request must contain all the information necessary to understand and process it.
Why? (Scalability):
If Server A remembers that User 1 is logged in, but the next request is routed to Server B (due to load balancing), Server B will reject the request.
The Solution:
The React frontend must send the **user's Authentication Token** inside the Headers of every single request. The server validates it, processes the request, and instantly forgets the user again.

Resources and URIs

Naming Things as Nouns (e.g., /products, not /getProducts)

The Rule:
URLs (Uniform Resource Identifiers) must represent resources (nouns), never actions (verbs).

Bad Design (RPC Style):
- api.com/getProducts
- api.com/createProduct
- api.com/deleteProductById?id=5

Good Design (REST Style):
- api.com/products
- api.com/products/5

The Question:
If the URL is just **/products**, how does the server know if we want to **read** them, **create** one, or **delete** one?

HTTP Verbs & CRUD Operations

The Action Verbs: GET (Read), POST (Create), PUT/PATCH (Update), DELETE

CRUD:
The four basic operations of persistent storage (Create, Read, Update, Delete).
REST maps these directly to HTTP Verbs applied to our Noun endpoints.

- **GET (Read):** Fetch data.
  (e.g., **GET** /products -> returns an array. **GET** /products/5 -> returns one item).
- **POST (Create):** Add new data.
  (e.g., **POST** /products + JSON body -> creates a new product).
- **PUT / PATCH (Update):** Modify data.
  (e.g., **PUT** /products/5 + JSON body -> updates item #5).
- **DELETE (Delete):** Remove data.
  (e.g., **DELETE** /products/5 -> destroys item #5).

Standard HTTP Status Codes

Communicating Success (2xx), Client Errors (4xx), and Server Errors (5xx)

- **2xx (Success):**
  200 OK: Request succeeded (used for GET, PUT, DELETE).
  201 Created: A new resource was successfully added (used for POST).

- **4xx (Client Error - The Frontend messed up):**
  400 Bad Request: The JSON body sent by React was malformed.
  401 Unauthorized: The user forgot to send their login token.
  404 Not Found: The requested URL or ID does not exist.

- **5xx (Server Error - The Backend messed up):**
  500 Internal Server Error: The database crashed or the Node.js code has a fatal bug.

The Spaghetti Code Problem

Why Backend Logic Needs Strict Structural Rules

The Trap:
Beginners often write the routing URL, the database connection, the validation logic, and the error handling all in one massive function block inside server.js.

The Consequence:
The code becomes impossible to read, test, or debug. If you change a database table, you have to rewrite the routing logic.

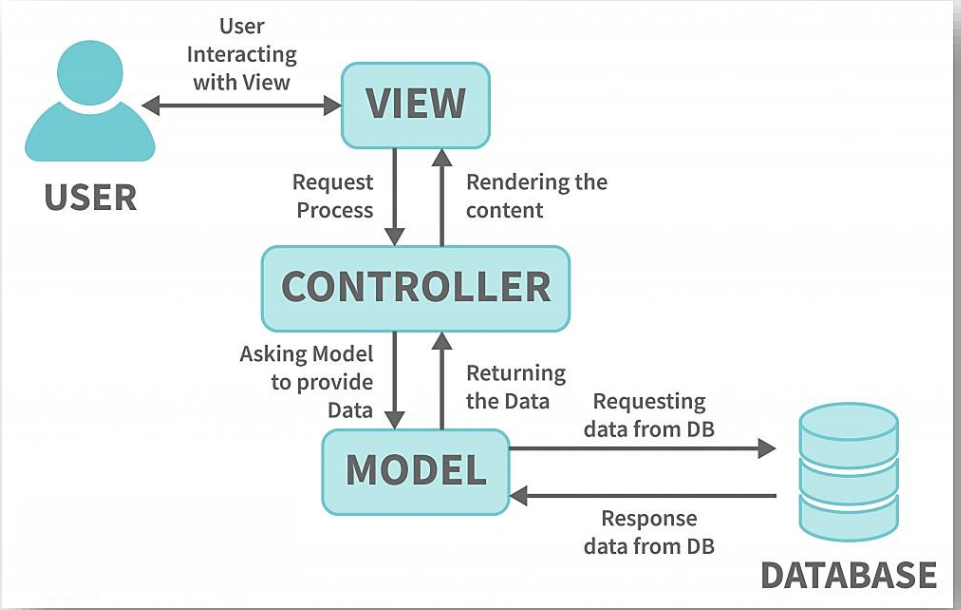
The Computer Science Principle: **"Separation of Concerns."**
We must split our application into distinct layers where each layer has exactly one job.

Introduction to MVC Architecture

Model-View-Controller: The Industry Standard Design Pattern

The Concept:
A design pattern that divides backend logic into three interconnected elements to separate internal representations of information from the ways information is presented to and accepted from the user.

The Flow:



The Model (M) & The View (V)

Managing Data Rules vs. Presenting the UI

The Model (Data Layer):
- It represents your database tables (e.g., a Product class).
- It holds the "Business Rules" (e.g., "A product price cannot be negative").
- It is the only part of the application allowed to talk directly to the database.

The View (Presentation Layer):
- Historically, this was HTML/EJS generated by the server.
- **In 2026 (SPA Era):** Our backend does not have a traditional View. Our "**View**" is the **React frontend**. The backend merely returns a **raw JSON** payload, and React decides how to paint it on the screen.

The Controller (C) & The Router

The Brains of the Operation and the Traffic Cop

The Router (Traffic Cop):
Listens to the URL and HTTP Verb.
Example: router.**get**('/products/:id', **getProductController**)

The Controller (The Manager):
O It receives the raw HTTP Request object.
O It extracts data (like the :id from the URL).
O It asks the Model to fetch the data.
O It packages the data into a JSON Response and sends it back to the client.

Crucial Rule:
The **controller** contains no database queries itself. It delegates that to the Model.

Relational Database Paradigms

Introduction to SQL, Tables, Rows, and ACID Compliance

SQL vs. NoSQL:
While MongoDB (NoSQL) stores data in flexible JSON documents, we will use PostgreSQL (SQL), a Relational Database Management System (RDBMS).

Tables and Relations:
Data is stored in strict rows and columns. We use Foreign Keys to link a User to their Orders.

ACID Compliance:
Essential for e-commerce. If an order is placed, the system guarantees that money is deducted and inventory is reduced simultaneously. If the server crashes mid-transaction, the entire database rolls back safely.

Strict Integrity:
You cannot accidentally insert a string into a decimal column, preventing catastrophic bugs.

Section 1 Summary & Next Steps

From Theory to Practice: Preparing for Node.js

Review:

1. We understand the Client-Server model.
2. We know the rules of REST (Stateless, Resources, HTTP Verbs, Status Codes).
3. We understand how to separate our code using MVC.
4. We chose PostgreSQL for relational integrity.

Next Steps (Section 2):

In our next session, we will open our IDEs, install Node.js and Express, and begin writing the actual JavaScript code to build a RESTful Router. Ensure your development environments are ready.