### **Introduction to Backend of Software**

The **backend of software** refers to the server-side part of an application that is responsible for handling business logic, managing databases, authenticating users, and ensuring secure communication between the client (frontend) and the server. While the frontend focuses on what users see and interact with, the backend powers the application’s core functionality—processing requests, applying rules, storing and retrieving data, and delivering results back to the client.

A well-structured backend is critical for performance, scalability, and security. It ensures that applications can handle multiple users at once, safeguard sensitive information, and integrate seamlessly with external services and APIs.

### **Possible Programming Languages for Backend Development**

Several programming languages are widely used to build robust backend systems. Some of the most common include:

* **JavaScript/TypeScript** – popular with Node.js for building fast, scalable, event-driven applications.
* **Python** – known for frameworks like Django and Flask, excellent for rapid development and machine learning integration.
* **PHP** – a traditional choice for backend, with frameworks like Laravel and Symfony.
* **Java** – strong in enterprise-level applications, with frameworks like Spring Boot.
* **C#** – commonly used with the .NET framework for building scalable web applications.
* **Ruby** – known for its Ruby on Rails framework, excellent for quick prototyping.
* **Go (Golang)** – lightweight, fast, and efficient for microservices and high-performance applications.
* **Rust** – gaining popularity for secure, high-performance systems.

### **Common Backend Frameworks**

Frameworks simplify backend development by providing structure, built-in tools, and reusable components. Some widely used backend frameworks include:

* **Node.js frameworks**: Express.js, AdonisJS, NestJS, Hapi.js
* **Python frameworks**: Django, Flask, FastAPI
* **PHP frameworks**: Laravel, Symfony, CodeIgniter
* **Java frameworks**: Spring Boot, Micronaut
* **C# frameworks**: ASP.NET Core
* **Ruby framework**: Ruby on Rails
* **Go frameworks**: Gin, Echo

### **Our Choice: JavaScript, Node.js, and AdonisJS**

For this project, we will focus on **JavaScript (JS)** as the primary programming language due to its flexibility and ubiquity across both frontend and backend development. By using **Node.js**, we gain the advantage of a non-blocking, event-driven runtime that is ideal for building high-performance and scalable server-side applications.

On top of Node.js, we will use **AdonisJS**, a modern and opinionated MVC framework that provides structure, productivity, and out-of-the-box features like authentication, validation, and database ORM. AdonisJS allows us to build applications quickly without sacrificing scalability or maintainability, making it an excellent choice for building secure, enterprise-grade backend systems.

Couse Outlline

1. Review of javascript:

OOP in js.

Import and export module

1. NodeJs
2. Typescript
3. Adonisjs

OOP Concept:

Object-Oriented Programming (OOP) is a paradigm based on objects that encapsulate data (attributes/properties) and behavior (methods). JavaScript supports OOP through classes and objects, enabling modular and reusable code.

In another word, Object-Oriented Programming is a programming style based on classes and objects. These group data (properties) and methods (actions) inside a box.

### **1. Classes and Objects**

A **class** is a blueprint for creating objects, while an **object** is an instance of a class.

#### **Example:**

class Car {

constructor(brand, model, year) {

this.brand = brand; // Property

this.model = model;

this.year = year;

}

displayInfo() { // Method

console.log(`${this.brand} ${this.model} was made in ${this.year}.`);

}

}

const myCar = new Car("Toyota", "Corolla", 2022);

myCar.displayInfo(); // Output: Toyota Corolla was made in 2022.

**How Do We Actually Design a Class?**

There is no perfect answer to this question. But we can get help from some OOP principles when designing our classes.

There are 4 main principles in OOP, and they are:

* Abstraction
* Encapsulation
* Inheritance
* Polymorphism

**What Does Abstraction Mean in OOP?**

Abstraction means hiding certain details that don't matter to the user and only showing essential features or functions.

For example, take a cell phone. We don't show details like verifyTemperature(), verifyVolt(), frontCamOn(), frontCamOff() and so on. Instead we provide essential features which matter to user like camera(), volumeBtn(), and others.

**What Does Encapsulation Mean in OOP?**

Encapsulation means keeping properties and methods private inside a class, so that they are not accessible from outside that class.

class Car {

constructor(brand) {

this.\_brand = brand;

}

get brand() {

return this.\_brand;

}

set brand(newBrand) {

this.\_brand = newBrand;

}

}

let myCar = new Car('Ford');

console.log(myCar.brand); // Ford

myCar.brand = 'BMW';

console.log(myCar.brand); // BMW

This will keep code that's outside the class from accidentally manipulating internal methods and properties.

**What Does Inheritance Mean in OOP?**

Inheritance makes all properties and methods available to a child class. This allows us to reuse common logic and to model real-world relationships.

class Animal {

constructor(name) {

this.name = name;

}

speak() {

console.log(`${this.name} makes a noise.`);

}

}

class Dog extends Animal {

speak() {

console.log(`${this.name} barks.`);

}

}

let d = new Dog('Mitzie');

d.speak(); // Mitzie barks.

**What Does Polymorphism Mean in OOP?**

Polymorphism allows objects of different classes to be treated as objects of a common superclass.

class Animal {

speak() {

console.log('Animal speaks');

}

}

class Cat extends Animal {

speak() {

console.log('Meow');

}

}

class Dog extends Animal {

speak() {

console.log('Woof');

}

}

function makeAnimalSpeak(animal) {

animal.speak();

}

makeAnimalSpeak(new Cat()); // Meow

makeAnimalSpeak(new Dog()); // Woof

### 

**Other Terminologies**

### **Constructor**

A **constructor** is a special method that initializes an object when a class is instantiated.

#### **Example:**

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

}

}

### **2. Methods (Static & Non-Static)**

Methods define the behavior of objects.

* **Static Methods:** Belong to the class, not an instance.
* **Non-Static Methods:** Belong to an instance of the class.

#### **Example:**

class MathUtils {

static add(a, b) {

return a + b;

}

subtract(a, b) {

return a - b;

}

}

console.log(MathUtils.add(5, 3)); // Output: 8

const calc = new MathUtils();

console.log(calc.subtract(5, 3)); // Output: 2

### **✅ Advantages of Static Methods**

✔ **Utility Functions** – Ideal for helper functions that don’t need instance-specific data.  
 ✔ **Memory Efficient** – Shared across all instances, reducing memory usage.  
 ✔ **Encapsulation** – Prevents modification from instances.  
 ✔ **Faster Execution** – Accessed directly from the class, avoiding instance overhead.

### **📌 When to Use Static Methods**

* Utility/helper functions (e.g., Math.random(), Object.keys(), Array.isArray()).
* When behavior does **not** depend on instance properties.
* Factory methods to create instances.

### **Advantages of Non-Static Methods**

✔ **Instance-Specific Behavior** – Can access and modify instance properties.  
 ✔ **Encapsulation** – Keeps related data and behavior together.  
 ✔ **Inheritance-Friendly** – Can be overridden in subclasses.

### **📌 When to Use Non-Static Methods**

* When behavior depends on **instance data**.
* When dealing with **object-oriented principles like encapsulation**.
* For methods that will be **overridden in subclasses**.

### **Getters (Accessors) & Setters (Mutators)**

* **Getters:** Retrieve object property values.
* **Setters:** Modify object property values.

#### **Example:**

class Employee {

constructor(name, salary) {

this.\_name = name;

this.\_salary = salary;

}

get salary() {

return this.\_salary;

}

set salary(amount) {

if (amount > 0) {

this.\_salary = amount;

} else {

console.log("Invalid salary");

}

}

}

const emp = new Employee("John", 5000);

console.log(emp.salary); // Output: 5000

emp.salary = 6000; // Modifies salary

console.log(emp.salary); // Output: 6000

### **3. Access Modifiers (Public, Private, Protected)**

* **Public:** Can be accessed anywhere.
* **Private (#):** Can only be accessed inside the class.
* **Protected (\_ - Convention):** Should only be used inside the class and subclasses.

#### **Example:**

class BankAccount {

constructor(balance) {

this.\_balance = balance; // Protected (by convention)

this.#pin = 1234; // Private property

}

showBalance() {

console.log(`Balance: $${this.\_balance}`);

}

}

const myAccount = new BankAccount(500);

myAccount.showBalance(); // Output: Balance: $500

console.log(myAccount.\_balance); // Works but should not be accessed directly

console.log(myAccount.#pin); // Error: Private field cannot be accessed outside class

### **Private Methods (#)**

In JavaScript, private methods are defined using the # symbol. These methods **cannot** be accessed outside the class.

#### **Example:**

javascript

CopyEdit

class User {

constructor(name) {

this.name = name;

}

#privateMethod() { // Private method

return "This is a private method!";

}

showPrivateMethod() {

return this.#privateMethod(); // Can be accessed within the class

}

}

const user = new User("Alice");

console.log(user.showPrivateMethod()); // ✅ Output: This is a private method!

console.log(user.#privateMethod()); // ❌ Error: Private field cannot be accessed outside the class

### **Protected Methods (Convention: \_)**

JavaScript **does not have a built-in protected modifier**, but by convention, developers use \_ before method names to indicate that they **should not be accessed outside the class or its subclasses**.

#### **Example:**

javascript

CopyEdit

class Person {

constructor(name) {

this.name = name;

}

\_protectedMethod() { // Protected by convention

return "This is a protected method!";

}

}

class Employee extends Person {

accessProtectedMethod() {

return this.\_protectedMethod(); // ✅ Accessible within subclass

}

}

const emp = new Employee("John");

console.log(emp.accessProtectedMethod()); // ✅ Output: This is a protected method!

console.log(emp.\_protectedMethod()); // ⚠️ Works, but should not be accessed directly

**Object-Oriented vs. Functional Programming in JavaScript**

**Differences Between OOP and Functional Programming**- State and Immutability: OOP handles state within objects, which can change over time. In contrast, functional programming prefers immutable data structures and pure functions without side effects.  
- Methodology: OOP is about modeling real-world entities using objects and classes, whereas functional programming focuses on the computation process and avoids changing state.  
- Code Reusability: In OOP, reusability comes through inheritance and polymorphism. Functional programming achieves reusability through functions and higher-order functions.

When to Use OOP or Functional Programming

**Use OOP when:**- You’re dealing with a complex system with clearly defined types and relationships.  
- Your application’s state changes frequently and needs to be managed cohesively.  
- You prefer a modular and structured approach to organizing code.

**Use Functional Programming when:**- You need a robust system with operations that don’t depend on or alter the state.  
- Your focus is on the flow of data and transformations applied to it.  
- You aim for code that’s easy to test and reason about due to its immutability and purity.

**Introduction to Node.js**

Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to execute JavaScript code outside a web browser. It is widely used for building scalable and high-performance applications, making it a preferred choice for modern web development. With Node.js, JavaScript can be used for both frontend and backend development, ensuring a unified programming language across the entire application stack.

### **What is Node.js?**

Node.js is a powerful runtime environment that executes JavaScript code on the server side. Here are some key features:

* **JavaScript Runtime:** Built on the V8 JavaScript engine (also used by Google Chrome), Node.js ensures fast execution of JavaScript outside the browser.
* **Single-Threaded Model:** Node.js operates within a single process, eliminating the overhead of creating multiple threads for handling requests.
* **Asynchronous I/O:** Supports non-blocking operations, enabling efficient handling of tasks like database access and network requests.
* **Concurrency Handling:** Manages thousands of concurrent connections efficiently using an event-driven architecture.
* **ECMAScript Compatibility:** Supports modern JavaScript standards, independent of browser updates.

### **Why Choose Node.js?**

Node.js is widely adopted by companies like PayPal, Uber, Netflix, and Walmart for backend services. Here’s why:

1. **Easy to Learn & Use:** JavaScript developers can quickly adapt to Node.js without learning a new language.
2. **Scalability:** Easily scales both horizontally (adding more machines) and vertically (upgrading resources).
3. **Real-Time Capabilities:** Ideal for building chat applications, collaborative tools, and live notifications.
4. **Speed & Efficiency:** Handles multiple operations efficiently without blocking the main execution thread.
5. **Rich Ecosystem:** npm provides a vast collection of reusable modules and libraries.

### **How Node.js Works?**

Node.js follows an event-driven, non-blocking architecture. It processes incoming requests efficiently using a single thread, known as the event loop. Here’s how it handles requests:

1. **Receives a request from the client.**
2. **Processes the request asynchronously using callbacks or Promises.**
3. **Returns the response without blocking the execution of other tasks.**

Unlike traditional models such as PHP and ASP.NET, where the server waits for each task to complete before processing the next one, Node.js can handle multiple requests simultaneously.

### **Advantages of Node.js**

* **Easy Scalability:** Supports both vertical and horizontal scaling for large applications.
* **High Performance:** Uses a non-blocking, event-driven architecture for efficient request handling.
* **Fast Execution:** Leverages the V8 engine to execute JavaScript at high speed.
* **Caching Advantage:** Stores modules in memory for quick access without reloading.
* **Unified Development:** JavaScript can be used across frontend and backend, simplifying development workflows.

### **Common Use Cases of Node.js**

Node.js is widely used for:

* **Real-Time Applications:** Chat applications, live notifications, and collaborative tools.
* **Single-Page Applications (SPAs):** Supports seamless frontend-backend interaction.
* **RESTful APIs and Microservices:** Frameworks like Express.js make API development easy.
* **Streaming Services:** Ideal for handling video/audio streaming and real-time data processing.

### **Node.js Ecosystem**

* **npm (Node Package Manager):** The default package manager with thousands of reusable modules.
* **Express.js:** A lightweight web framework for building APIs and web applications.
* **Socket.io:** Facilitates real-time communication using WebSockets.
* **Mongoose:** A library for interacting with MongoDB databases.
* **NestJS:** A progressive Node.js framework for building scalable and maintainable server-side applications.
* **PostgreSQL:** A powerful, open-source relational database that integrates well with Node.js applications.
* **MySQL:** A widely used relational database, often used in combination with Node.js for web applications.

### **Installing Node.js**

To install Node.js:

1. Visit [Node.js official website](https://nodejs.org/).
2. Download and install the LTS version for your operating system.
3. Verify the installation using:  
   **node -v**
4. If it returns a version number, Node.js is successfully installed.

### **Running a Web Server with Node.js in VS Code**

#### **Step 1: Install VS Code**

1. Download and install [Visual Studio Code](https://code.visualstudio.com/).

#### **Step 2: Install Node.js Extension**

1. Open VS Code.
2. Go to the Extensions tab and search for "Node.js Extension Pack".
3. Install the extension.

#### **Step 3: Create a New Project**

1. Open VS Code and create a new folder for your project.
2. Open the terminal in VS Code and run:  
   npm init -y  
   This creates a package.json file.

#### **Step 4: Create a Web Server File**

Create a new file called app.js.

Add the following code:

const http = require('http');

const server = http.createServer((req, res) => {

res.writeHead(200, { 'Content-Type': 'text/html' });

res.end('<h1>Hello, Node.js!</h1>');

});

const PORT = 3000;

server.listen(PORT, () => {

console.log(`Server running at http://localhost:${PORT}/`);

});

#### **Step 5: Run the Server**

1. Open the terminal in VS Code.
2. Run the command:  
   node app.js
3. Open a browser and visit http://localhost:3000/ to see the output.

/oj

Readmore about the nodejs here

https://www.freecodecamp.org/news/get-started-with-nodejs/

Typescript:

### **Introduction to TypeScript**

TypeScript is a **strongly typed superset of JavaScript** that compiles to plain JavaScript. It introduces **static typing, interfaces, and modern ECMAScript features**, making it ideal for large-scale applications.

## **Installing TypeScript**

Ensure you have **Node.js** installed. Then, install TypeScript globally using **npm**:

npm install -g typescript

Verify installation:

tsc --version

## **Initializing TypeScript in a Project**

To set up TypeScript in a project, navigate to your project folder and run:

tsc --init

This generates a tsconfig.json file, which allows you to configure TypeScript options.

## **Setting Up TypeScript Configuration**

The tsconfig.json file includes various settings. Here are some key configurations:

{

"compilerOptions": {

"target": "ES6", // JavaScript version to compile to

"module": "CommonJS", // Module system for Node.js

"strict": true, // Enable strict type checking

"outDir": "./dist", // Output folder for compiled files

"rootDir": "./src", // Source folder for TypeScript files

"esModuleInterop": true // Enables compatibility with CommonJS and ES modules

},

"include": ["src"], // Include the 'src' folder for compilation

"exclude": ["node\_modules"] // Exclude unnecessary folders

}

## **Writing and Compiling Your First TypeScript Code**

**Create a Project Structure**  
mkdir my-typescript-app

cd my-typescript-app

npm init -y

npm install -g typescript

tsc --init

mkdir src

**Create a hello.ts file inside src/**  
let message: string = "Hello, TypeScript!";

console.log(message);

**Compile the TypeScript Code**  
tsc

1. This compiles all .ts files from src/ to dist/.

**Run the JavaScript File**  
node dist/hello.js

**Output:** Hello, TypeScript!