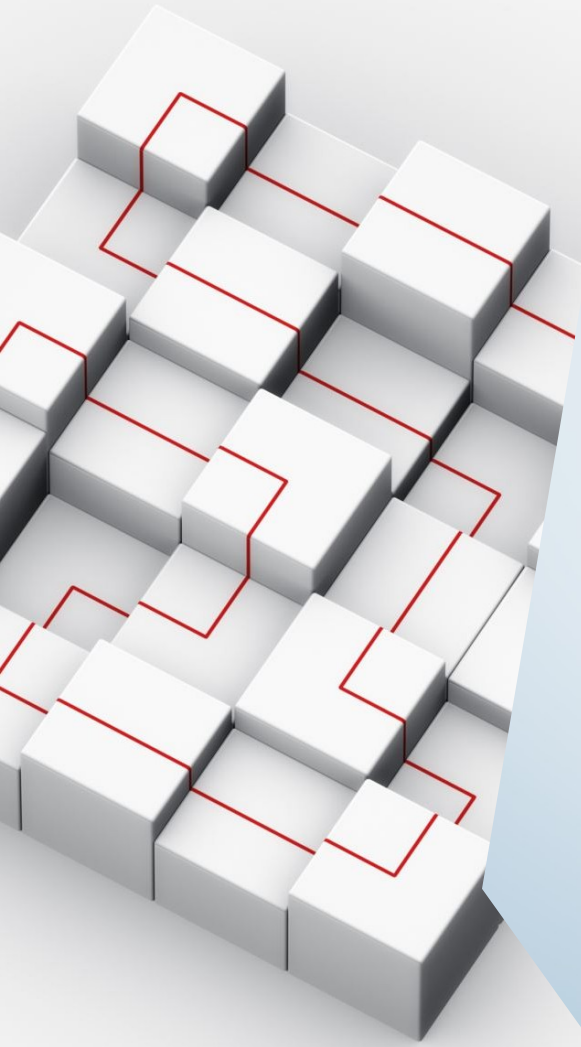
PYTHON FOR WEB DEVELOPMENT

2024-2025

Deschamps Marc



Course Objectives

1. Understand modern web architecture
2. Build applications with Flask and Django
3. Implement secure web applications
4. Deploy applications to cloud platforms
5. Work with REST APIs and microservices

Prerequisites

Python **programming knowledge**

Basic **understanding of HTML/CSS**

Basic database concepts

Git version control

Content

Introduction to Web Applications

Introduction to Flask

Introduction to Django

Data Binding and Communication

Practical Work -> group project

Web Application

What is a Web Application?

A **web application** is a software program that runs on a web server and is accessed through a web browser over a network.

Key Characteristics:

User interface delivered via a web browser.

No installation required on client devices.

Accessible from any device with internet connectivity.

Web Applications vs. Desktop Applications

Desktop Applications:

Installed locally on a user's computer.

Tied to the operating system.

Updates require manual installation.

Web Applications:

Accessed via web browsers.

Platform-independent.

Updates are deployed on the server side.

Feature	Desktop Applications	Web Applications
Installation	Required	Not required
Platform	OS-specific	Cross-platform
Updates	Manual	Automatic
Accessibility	Limited to device	Anywhere via web

COMPARISON TABLE

Web Applications vs. Websites

Website:

Informational, static or semi-static content.

Limited interactivity (reading, browsing).

Built with basic HTML, CSS, and JavaScript.

Usually public, minimal/no authentication.

Examples: News sites, blogs, company portfolios.

Web Application:

Functional, interactive, dynamic.

User-driven (input, actions, results).

Built with advanced technologies and frameworks.

Requires authentication for personalized features.

Examples: Google Docs, online banking, e-commerce platforms.

Criteria	Website	Web Application
Purpose	Information sharing	Task execution and interactivity
Interactivity	Low	High
Technology	HTML, CSS, JavaScript	Frameworks (React, Angular) + Backend
Authentication	Optional	Often required
Complexity	Simple architecture	Complex, multi-layered
Examples	Blogs, news sites	Gmail, Trello, Amazon

COMPARISON TABLE

Importance of Web Applications

Accessibility:

Available 24/7 from anywhere.

Ease of Maintenance:

Centralized updates reduce maintenance efforts.

Scalability:

Can handle growing user bases with appropriate infrastructure.

Cost-Effectiveness:

Reduces client-side hardware and software requirements.

The Role of Web Applications Today

Business Operations:

Online services, customer portals.

Communication:

Email, chat applications.

Entertainment:

Streaming services, online games.

Education:

E-learning, virtual classrooms.

Summary of Introduction

Web applications are essential in modern computing.

They offer flexibility and accessibility over traditional desktop applications.

Understanding their basics is crucial for web development.

The Client-Server Model

Understanding Clients and Servers

Client:

Requests services or resources.

Examples: Web browsers, mobile apps.

Server:

Provides services or resources.

Examples: Web servers, database servers.

Analogy:

Restaurant: Customers (clients) order food from the kitchen (server).

Client-Server Communication

Request-Response Cycle:

Client sends a **request** to the server.

Server processes the request.

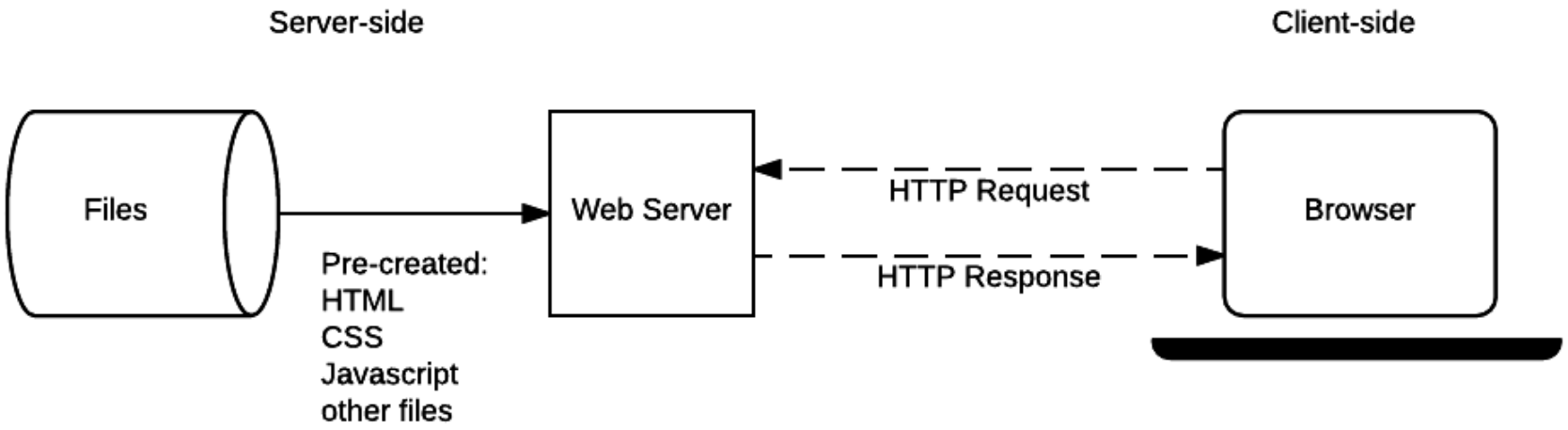
Server sends a **response** back to the client.

Communication Protocols:

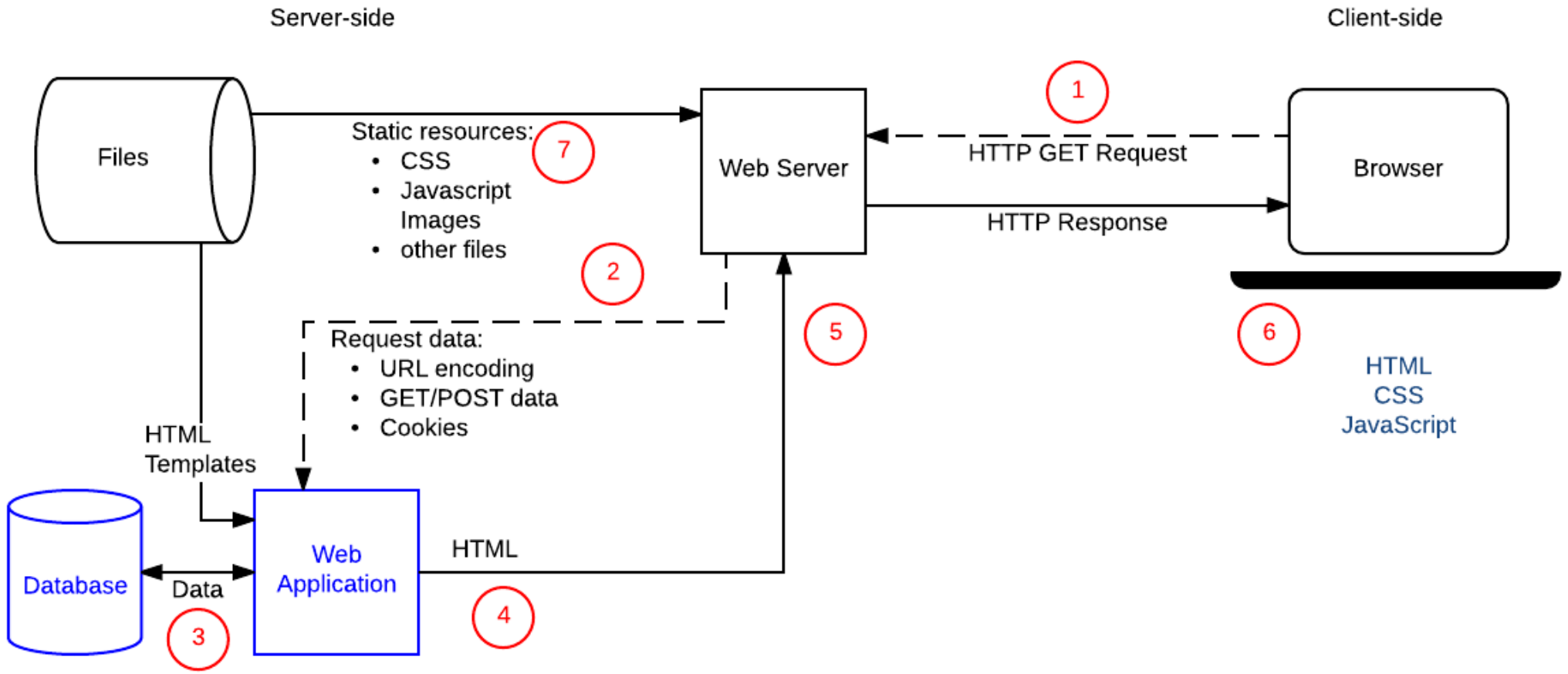
HTTP: HyperText Transfer Protocol.

HTTPS: Secure version of HTTP.

Static sites



Dynamic sites / Dynamic requests



HTTP Protocol Basics

HTTP Methods:

GET: Retrieve data from the server.

POST: Send data to create a resource.

PUT: Update a resource.

DELETE: Remove a resource.

HTTP Request Structure:

Request line, headers, optional body.

HTTP Response Structure:

Status line, headers, optional body.

HTTP Request and Response Example

```python
# -*- coding: utf-8 -*-
import http.client
import ssl

# Create an unverified SSL context
context = ssl._create_unverified_context()

conn = http.client.HTTPSConnection('www.example.com', context=context)

# For a GET request
conn.request('GET', '/')
response = conn.getresponse()
print(response.status, response.reason)
data = response.read()
print(data.decode('utf-8'))

conn.close()
```

HTTPS Protocol

HTTPS ensures:

Encryption: Data is encrypted to prevent eavesdropping.

Integrity: Data is not altered during transmission.

Authentication: The server is authenticated to ensure you're communicating with the intended recipient.

SSL/TLS Certificates:

SSL/TLS certificates secure data transmission between a web server and a browser, ensuring **encryption** and **authentication**.

Creates an encrypted connection using **public and private keys** after the server's certificate is verified.

Importance of HTTPS

Security Benefits:

Protects sensitive information (e.g., passwords, credit card numbers).

Trust and Credibility:

Users trust secure websites more.

SEO Advantages:

Search engines favor HTTPS websites.

Basic HTTPs Request Example

```python
# -*- coding: utf-8 -*-
import requests

# Make an HTTP GET request
response = requests.get("https://jsonplaceholder.typicode.com/posts/1")

# Check if the request was successful
if response.status_code == 200:
    data = response.json()  # Parse response to JSON format
    print(data)  # Print the response data
else:
    print(f"Request failed with status code {response.status_code}")

```

Summary of Client-Server Model

The **Client-Server Model** is an architecture where **clients** request services, and **servers** provide resources or responses.

Client: Initiates communication; sends **requests** for data.

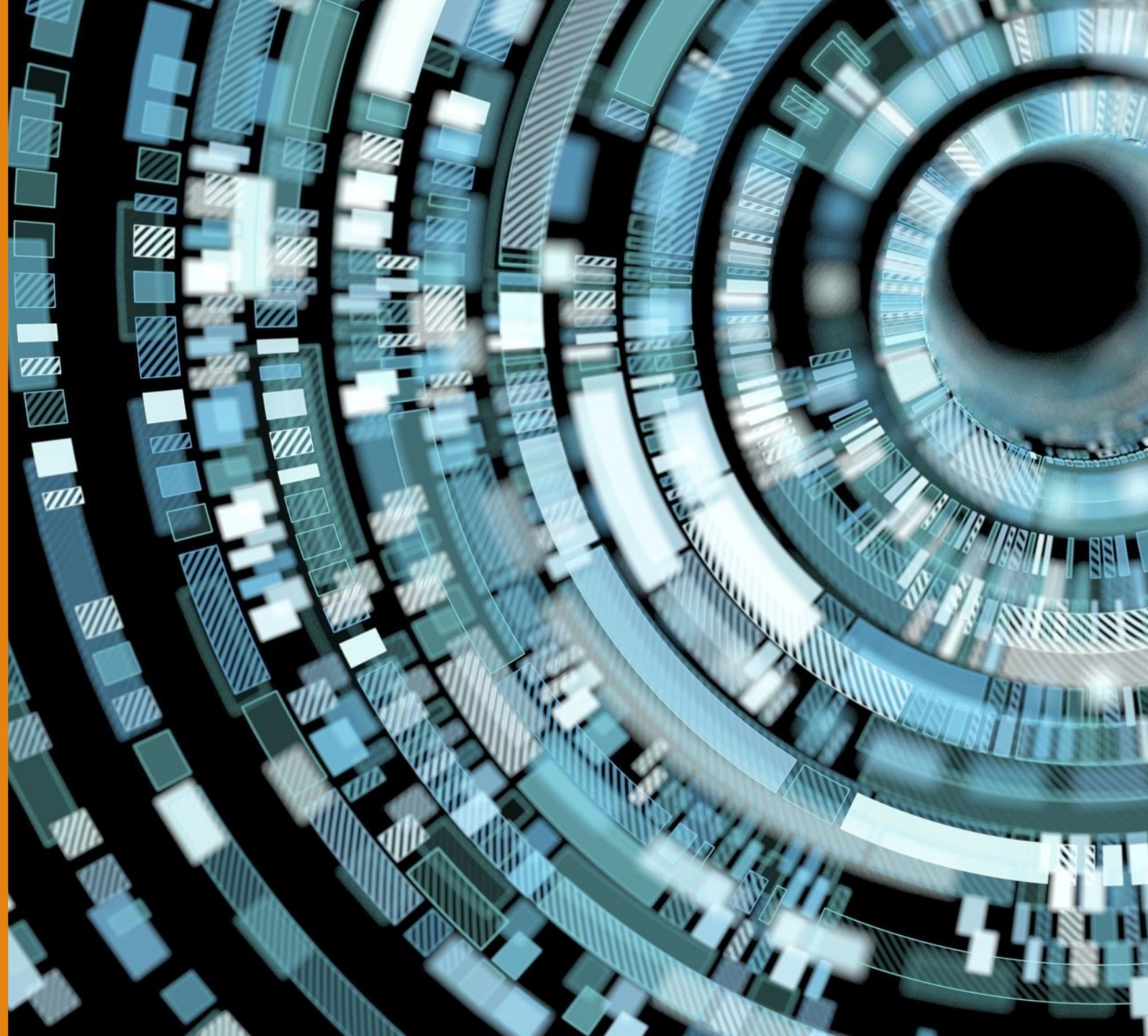
Server: Listens for and processes requests; provides **responses**

Communication: Typically uses **HTTP/HTTPS** or other protocols.

Benefits: Centralized control, easy **scalability**, better **security** for data and resources.

Drawbacks: **Server dependency** (failure leads to service unavailability), network congestion with too many clients.

Web Technologies Overview



Front-End vs. Back-End Development

Front-End Development:

Focuses on the user interface.

Technologies: **HTML**, **CSS**, **JavaScript**...

Back-End Development:

Handles server-side logic and databases.

Technologies: **Python**, **PHP**, **SQL**...

Interaction:

Front-end sends requests to the back-end.

Back-end processes requests and sends responses.

Not this again...

Core Technologies:

HTML (HyperText Markup Language): Defines the structure and content of web pages.

CSS (Cascading Style Sheets): Styles and layouts the visual presentation of web pages.

JavaScript: Adds interactivity and dynamic content to web pages.

Frameworks/Libraries:

React: JavaScript library by Facebook for building dynamic UIs.

Angular: Google's front-end framework for creating scalable SPAs.

Vue.js: Progressive JavaScript framework for easy integration and component-based development.

Tools:

Bootstrap: CSS framework for responsive design and pre-built UI components.

Sass/LESS: CSS preprocessors for better styling management.

Webpack: Module bundler to optimize and manage assets.

Focus:

Responsiveness, **performance**, and delivering engaging **user experiences** across devices.

Role of Python in Web Development

Backend Role: Python is popular for server-side web development, handling **logic, databases, script, APIs**...

Use Cases:

Backend Development: Business logic, user authentication.

API Creation: REST APIs for client-server communication.

Benefits:

Simplicity and **rapid prototyping**.

Rich **libraries** for HTTP, databases, and web tasks.

Focus:

Speed up backend development, ensure scalability, and easy integration with front-end technologies.

Introduction to Web Frameworks

Why Use Frameworks?

Simplify web development tasks.

Provide structure and tools.

Flask:

Suitable for small to medium applications.

Flexible and easy to use.

Django:

Follows the **MTV (Model-Template-View)** pattern.

Includes built-in admin interface.

FastAPI:

Modern framework for fast API creation.

Interaction Between Front-End and Back-End

Process Flow:

User interacts with the front-end trigger actions.

Front-end sends a request to the back-end.

Back-end processes the request.

Back-end sends a response.

Front-end updates the UI accordingly.

Example:

A user fills a form (front-end), data is sent to the server (back-end) for validation and storage, and a success response is returned.

State Management in Web Applications

Stateless Nature of HTTP

HTTP is a **stateless protocol**, meaning each request is independent, and the server does not retain information about previous interactions.

Implications:

The server doesn't retain information about clients between requests.

Benefits:

Scalability: Easier to distribute and load-balance as no session data is retained.

Simplicity: Reduces complexity of the server-side logic.

Focus: The stateless nature promotes efficiency but needs additional tools for maintaining user-specific data over multiple requests.

Why State Management is Needed

User Authentication: Keep users logged in across pages.

Data Persistence: Maintain data during browsing (e.g., shopping cart).

Complex Interactions: Manage user-specific data in web apps, such as form inputs or app state.

Tools:

Cookies, **Sessions**, **Local Storage** on the front-end.

Server-side Sessions and **State Stores** like **Redux** in front-end frameworks.

Benefits:

Provides **continuity** and an **enhanced user experience**.

Ensures the user can continue where they left off without restarting processes or re-entering data.

Cookies

Cookies are small pieces of data stored on the user's browser by a website to remember information about the user.

Purpose:

Session Management: Track user login sessions.

Personalization: Store preferences, like language or theme.

Tracking: Follow user activity for analytics or targeted advertising.

Types:

Session Cookies: Deleted after the browser closes.

Persistent Cookies: Stored for a set duration, used for remembering preferences or keeping users logged in.

Third-Party Cookies: Used by external services (e.g., ads, social media) for tracking across sites.

Limitations:

Size restrictions. Privacy concerns.

Sessions

A **session** is a server-side mechanism used to store information about a user across multiple HTTP requests, enabling persistent user interactions throughout their time on a website.

Purpose:

State Management: Keep track of logged-in users, preferences, and ongoing transactions.

User Identification: Typically linked to a **session ID** stored in a cookie, helping to identify a user uniquely across requests.

How it Works:

Session ID: Generated upon user login and stored in a cookie.

Server Storage: Server maintains data (e.g., user info, shopping cart) linked to the session ID.

Benefits:

Persistent Data: Maintains continuity (e.g., user logged in across different pages).

More Secure: Since data is stored server-side, it's more secure compared to storing sensitive information in cookies.

Focus: Sessions ensure a consistent user experience across pages, enabling web applications to maintain user data securely between different requests.

Basic Security Considerations

Authentication: Verify user identity via **passwords**, **multi-factor authentication (MFA)**, and **OAuth** to prevent unauthorized access.

Data Protection: Use **encryption** (HTTPS/SSL) to secure data in transit and **hashing** for sensitive data like passwords.

Input Validation: Protect against **injection attacks** (e.g., SQL injection, XSS) by validating and sanitizing user inputs.

Access Control: Implement **role-based access control (RBAC)** to restrict access to sensitive data based on user roles.

Session Management: Secure sessions with **session timeouts** and **regeneration of session IDs** after login to avoid **session hijacking**.

Error Handling: Avoid revealing system details in error messages; provide generic errors to prevent **information leakage**.

Regular Updates: Keep software, frameworks, and dependencies up-to-date to mitigate vulnerabilities and **security flaws**.

Focus: A comprehensive approach that combines **authentication**, **encryption**, and **proper coding practices** to protect both user data and the web application.

Web Application Architecture

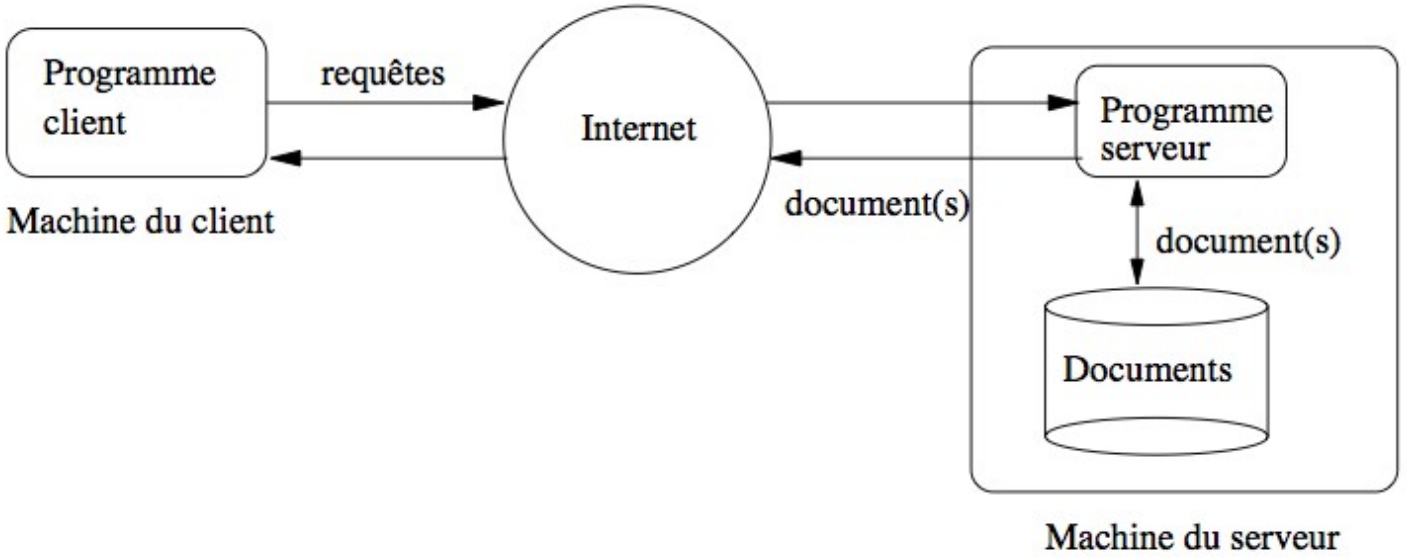
Overview of Web Application Architecture

Presentation Layer, Business Logic Layer, Data Layer (NOBDD).

Types:

Monolithic: Single unit containing all components.

Microservices: Decoupled services, each handling specific functionality, communicating via APIs.



Monolithic Architecture

Monolithic Architecture is a traditional design where the entire web application, including front-end, back-end, and database, is packaged and deployed as a **single unit**.

Characteristics:

All **features** (e.g., user management, order processing) are tightly integrated.

Single Codebase: A single repository for all components, simplifying development initially.

Benefits:

Simple Development & Deployment: Easy to develop, test, and deploy since all components are in one package.

Performance: Faster communication between components since they run within the same process.

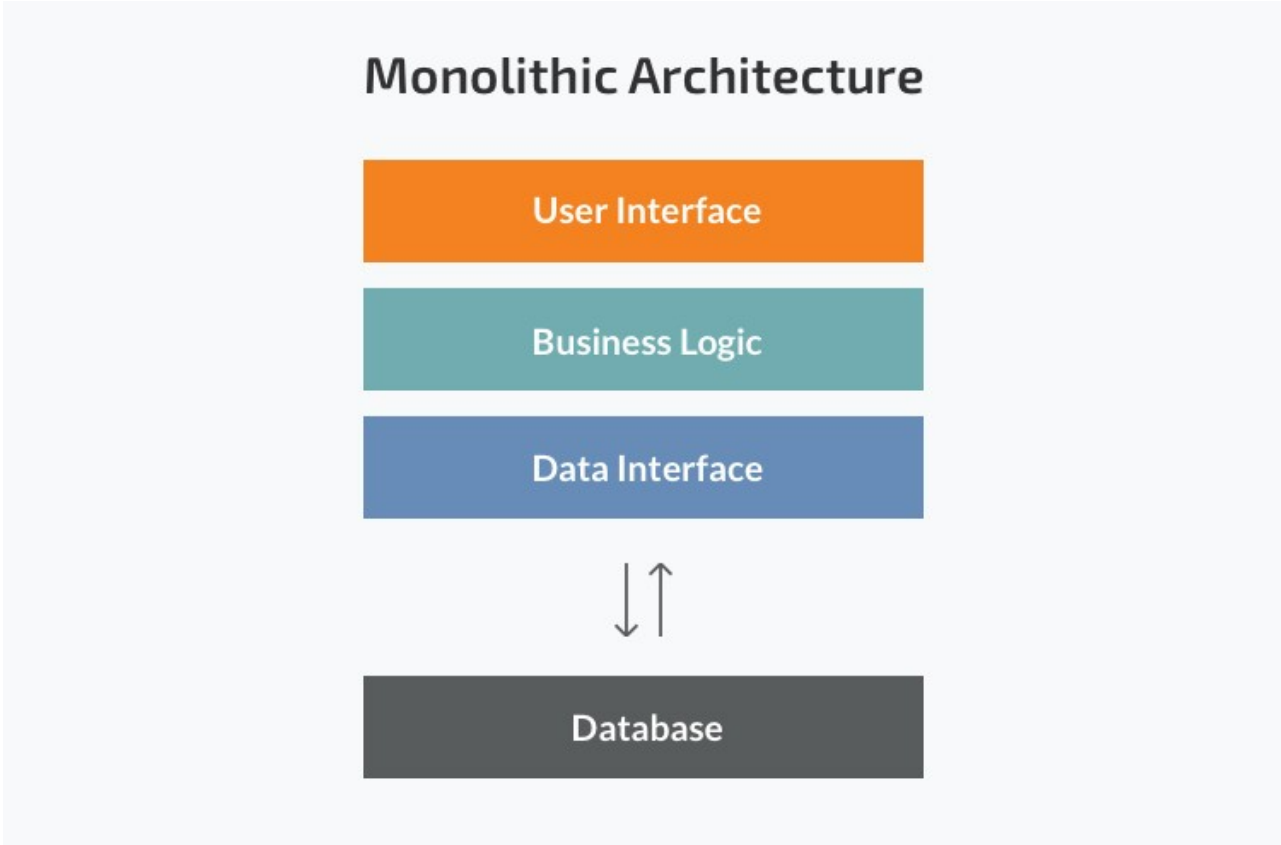
Drawbacks:

Scalability Issues: Hard to scale individual features; scaling requires scaling the entire application.

Tight Coupling: Changes in one component affect the entire system, making maintenance difficult.

Deployment Challenges: Any small change requires redeploying the whole application.

Focus: Suitable for small projects but can become **complex and inflexible** as the application grows in size and complexity.


Monolithic Architecture
User Interface
Business Logic
Data Interface
Database

Micro-services Architecture

Microservices Architecture is a design approach where a web application is built as a collection of **independent, loosely coupled services**, each focusing on a specific functionality.

Characteristics:

Decoupled Services: Each service (e.g., payment, authentication) runs independently and communicates with others via **APIs**.

Independent Development: Teams can develop, deploy, and scale each service independently.

Benefits:

Scalability: Scale individual services based on their needs.

Flexibility: Each service can use different technologies or programming languages.

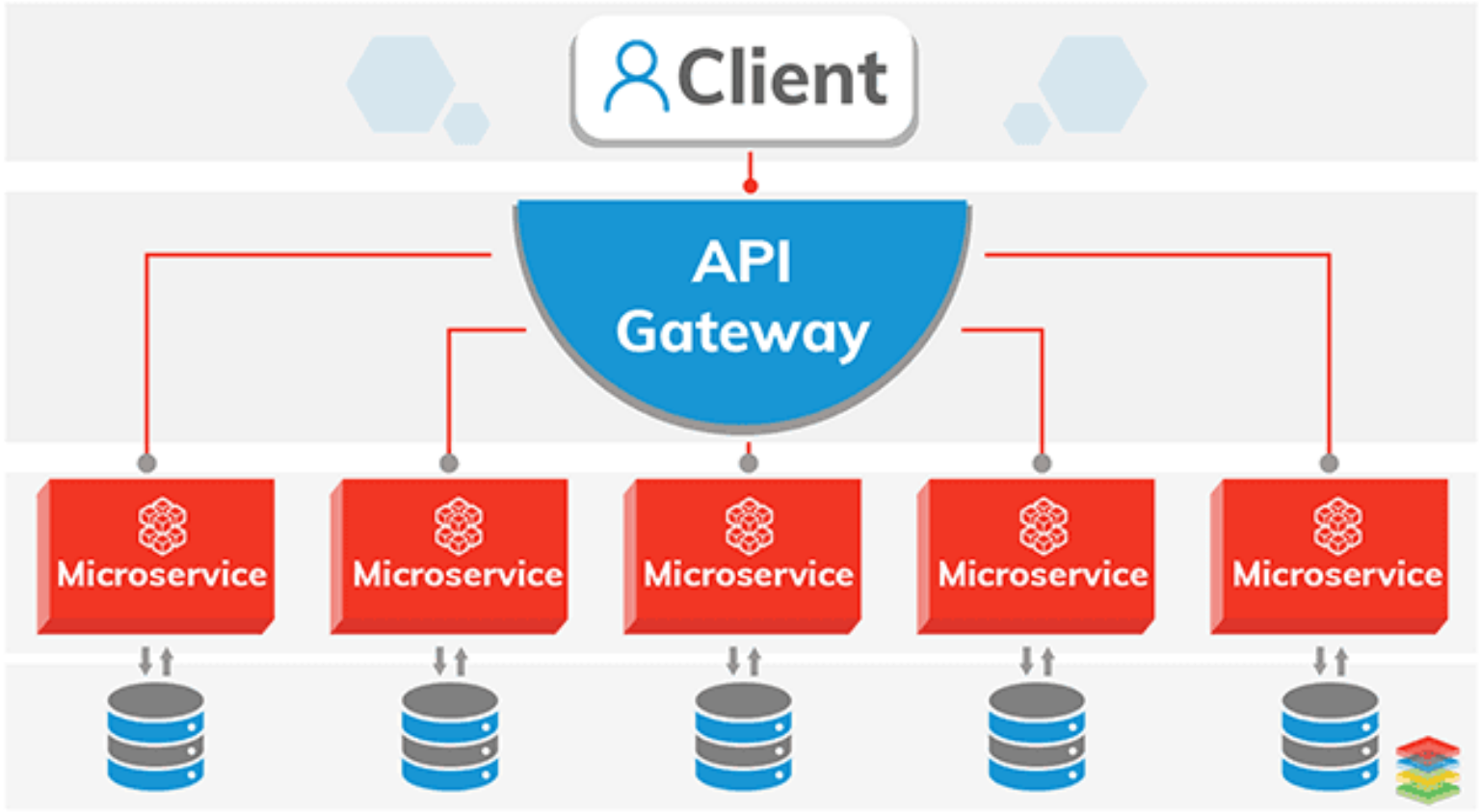
Resilience: Fault isolation ensures that failure in one service doesn't crash the entire system.

Drawbacks:

Complexity: More challenging to manage than monolithic systems due to distributed nature.

Communication Overhead: Services require efficient API communication, which can add latency.

Focus: Microservices improve scalability and agility, ideal for **large, complex applications** requiring frequent updates and distributed development teams.


Client
API Gateway
Microservice
Microservice
Microservice
Microservice
Microservice

Serverless Architecture

Serverless Architecture allows developers to build and run applications without managing server infrastructure. Services automatically scale, and developers only pay for actual usage.

How it Works:

Code is executed in the cloud using **functions-as-a-service (FaaS)** (e.g., AWS Lambda).

Third-party cloud providers manage server provisioning and scaling.

Benefits:

Cost-Efficiency: Pay only for resources consumed (no idle costs).

Scalability: Automatic scaling based on demand.

Reduced Management: No need to maintain or configure servers.

Drawbacks:

Vendor Lock-in: Dependent on specific cloud providers.

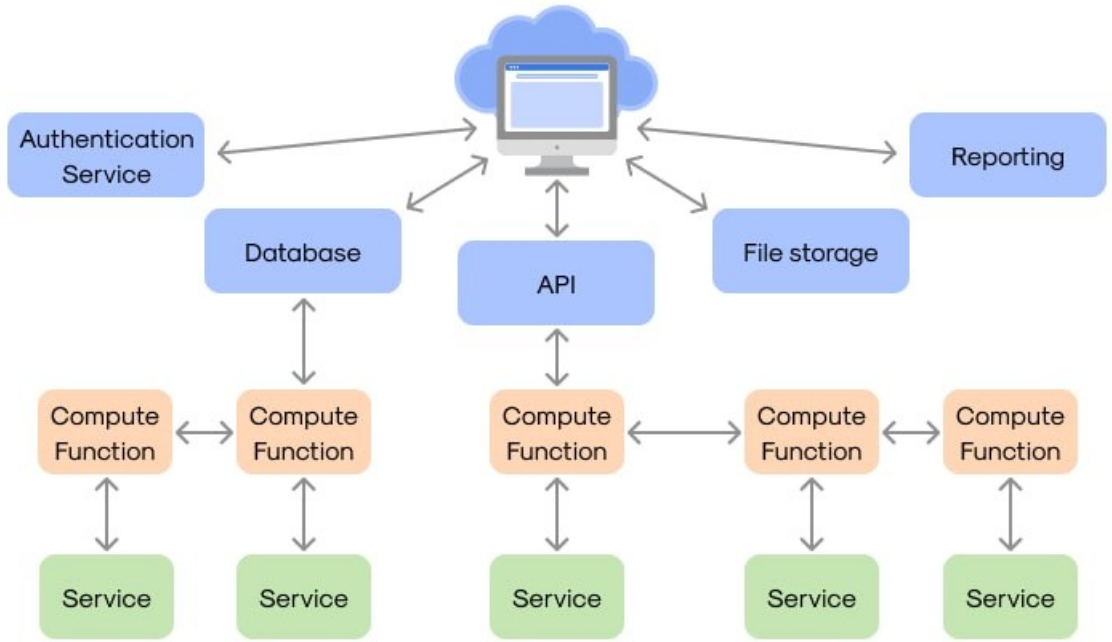
Cold Start Latency: Initial delay when a function is triggered after inactivity.

Use Cases: Real-time file processing, chatbots, APIs, and applications with unpredictable traffic.

Focus: Simplify development, allowing teams to focus on writing code rather than infrastructure management, making it ideal for event-driven, scalable applications.

wallarm

Serverless Architecture



Cloud-Native Architecture

Cloud-Native Architecture is an approach to building applications designed to run in **cloud environments**, leveraging the full benefits of cloud services such as **scalability**, **resilience**, and **automation**.

Characteristics:

Containerized Services: Uses containers (e.g., Docker) for packaging apps, ensuring consistency across different environments.

Microservices: Often organized as microservices for better scalability and flexibility.

Orchestration: Managed through tools like **Kubernetes** for automating deployment, scaling, and operations.

Benefits:

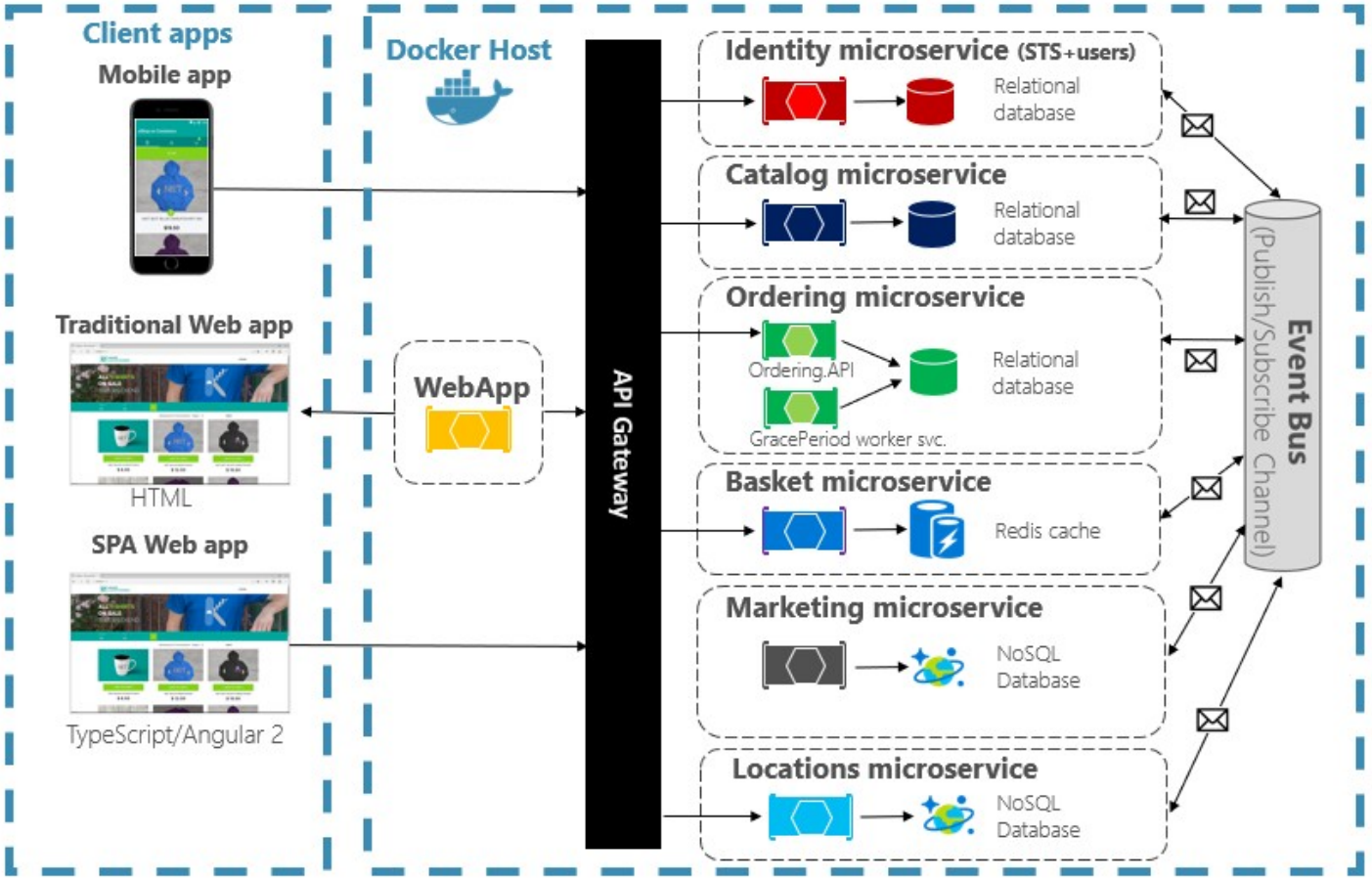
Scalability: Effortlessly scale services based on demand.

Resilience: Fault-tolerant, designed to handle failures without affecting the entire system.

Agility: Faster deployment and delivery cycles.

Use Cases: Modern SaaS platforms, large-scale web apps, e-commerce systems, and applications needing rapid iteration.

Focus: Harnessing the **full power of cloud infrastructure** to achieve agility, scalability, and efficiency in application development and deployment.


Client apps
Mobile app
Traditional Web app
HTML
SPA Web app
TypeScript/Angular 2
Docker Host
WebApp
API Gateway
Identity microservice (STS+users)
Relational database
Catalog microservice
Relational database
Ordering microservice
Ordering.API
GracePeriod worker svc.
Relational database
Basket microservice
Redis cache
Marketing microservice
NoSQL Database
Locations microservice
NoSQL Database
Event Bus
(Publish/Subscribe Channel)

Multi-Tier Architecture

Multi-Tier Architecture (or **N-Tier Architecture**) is a design pattern that separates the application into multiple layers, typically **Presentation**, **Business Logic**, and **Data** tiers.

Layers:

Presentation Layer: User interface, typically running on the client (e.g., browser).

Business Logic Layer: Handles application logic, data processing, and rules, usually running on a server.

Data Layer: Stores and manages data, often using databases like **MySQL** or **MongoDB**.

Benefits:

Separation of Concerns: Easier development and maintenance by isolating different functionalities.

Scalability: Different layers can be scaled independently based on demand.

Reusability: Logic and data layers can be reused across multiple applications.

Drawbacks:

Complexity: More layers can make the system more complex to manage.

Latency: Data must move between layers, which can introduce delays.

Use Cases: Business applications, ERP systems, banking software, where maintainability and separation of logic are key.

Focus: Clear separation of responsibilities, improving **maintenance**, **scalability**, and development efficiency for complex applications.

Presentation tier

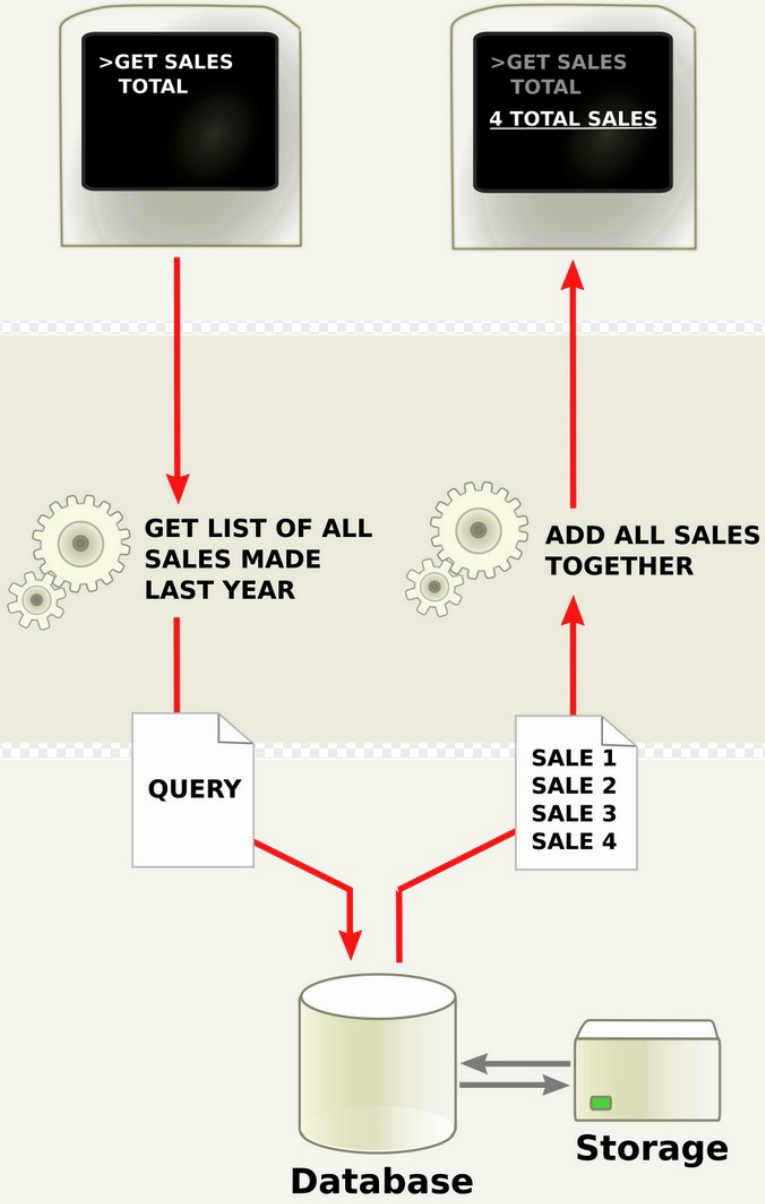
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.



Architecture Type	Structure	Scalability	Deployment	Maintenance	Use Cases
Monolithic	All components in a **single codebase**	**Limited** - scaling requires duplicating the entire app	Single unit deployment, easy at first	**Complex** - tightly coupled, changes affect entire system	Small to medium apps, early-stage startups
Microservices	**Independent services** for each feature	**High** - individual services scaled independently	Separate deployment for each service, requires orchestration	Easier to maintain each **service independently**	Large-scale apps (e.g., Netflix), frequent updates
Serverless	No server management, uses **functions**	**Automatic scaling** - triggered by events	Deployed as **functions** or event-driven units	Minimal - infrastructure managed by the provider	Real-time processing, APIs, small scalable workloads
Cloud-Native	Built specifically for **cloud environments**	**High** - uses containers and orchestration (Kubernetes)	Modular deployment, **containers** for services	Moderate to Easy - cloud tools simplify maintenance	Modern SaaS apps, scalable enterprise apps
Multi-Tier (N-Tier)	Separated into **logical layers** (Presentation, Logic, Data)	**Moderate** - can scale layers independently	Each tier can be deployed on different servers	Easier due to **separation of concerns**	Business apps, ERP systems, scalable enterprises

Introduction to APIs

API (Application Programming Interface) is a set of rules that allows different software applications to **communicate** with each other, acting as an intermediary to enable data exchange.

Purpose:

Integration: Connects different systems or services, allowing them to work together seamlessly.

Functionality Reuse: Allows developers to access existing functionalities (e.g., payment processing, maps) without writing new code.

Types of APIs:

REST (Representational State Transfer): Uses standard HTTP methods (GET, POST) for data exchange, widely used for web services.

SOAP (Simple Object Access Protocol): A protocol for exchanging structured information, more rigid compared to REST.

GraphQL: A query language for APIs that allows clients to request only the data they need.

Real-Life Examples:

Weather Apps: Use APIs to fetch up-to-date weather information.

Payment Processing: Websites use payment APIs like **Stripe** or **PayPal** to handle transactions.

Focus: APIs allow different applications to interact, enabling **integration**, **scalability**, and easier implementation of complex features without reinventing the wheel.

RESTful API Basics

REST is an **architectural style** for designing networked applications, emphasizing scalability, simplicity, and stateless communication between **clients** and **servers**.

Key Principles:

Stateless Communication: Each client request to the server must contain all the information needed; no client data is stored between requests.

Client-Server Separation: Clear separation of user interface (client) and data management (server), enhancing modularity.

Uniform Interface: Uses standard HTTP methods (**GET, POST, PUT, DELETE**) and **URI** to interact with resources.

Resource-Based: Everything is a **resource**, accessible via **URIs** (e.g., `/users`, `/products`).

HTTP Methods:

GET: Retrieve a resource.

POST: Create a new resource.

PUT: Update an existing resource.

DELETE: Remove a resource.

Benefits:

Scalability: Stateless nature makes it easy to scale the server.

Flexibility: Works with various data formats (often **JSON**).

Simplicity: Easy to use, leveraging HTTP protocols that are already well understood.

Use Cases: **Web services**, **APIs for mobile apps**, and any system requiring standardized communication between components. REST is widely adopted due to its simplicity and reliance on web standards.

SOAP Basics

SOAP is a **protocol** for exchanging structured information in the implementation of web services. It uses **XML** for message format and is designed to be **platform and language independent**.

Key Characteristics:

Protocol-Based: Unlike REST, which is an architectural style, SOAP is a strict protocol with built-in rules for messaging and communication.

XML Format: All messages are in **XML**, making them structured but more verbose compared to other formats like JSON.

Security & Reliability: Built-in standards for **security** (WS-Security) and **message reliability** (e.g., ACID transactions), making it suitable for critical business applications.

Core Elements:

Envelope: Defines the start and end of the message, providing a container for the information.

Header: Contains optional meta-information like authentication details.

Body: Contains the actual message or data for the request/response.

Benefits:

Platform Independent: Works across different platforms and programming languages.

High Security: Ideal for enterprise-level transactions, offering built-in security features and compliance standards.

Drawbacks:

Complexity: Requires strict rules and a lot of overhead, making it more complex and slower compared to REST.

Verbosity: XML-based messages are larger and require more bandwidth.

Use Cases: SOAP is often used in **banking**, **financial services**, and **enterprise applications** where **security** and **standardization** are crucial requirements, such as payment processing and government services.

MVC Pattern

What is the MVC Pattern?

Definition:

The **MVC pattern** is a software architectural pattern that separates an application into three interconnected components: **Model**, **View**, and **Controller**.

Purpose:

Encourages separation of concerns.

Makes applications easier to manage, maintain, and scale.

Benefits of Using MVC

Separation of Concerns: Divides application into **Model** (data), **View** (UI), and **Controller** (logic), making development and maintenance simpler.

Reusability: **Models** and **Views** can be reused across different parts of the application, reducing redundant code.

Parallel Development: Enables **independent development** of UI, logic, and data layers, speeding up team collaboration.

Easier Testing: Separation allows for focused unit tests on individual components, improving application reliability.

Scalability: Well-organized code structure improves **scalability** and adaptability as the project grows.

Focus: MVC improves maintainability, testability, and reusability, making it ideal for developing organized and scalable applications.

Overview of MVC Components

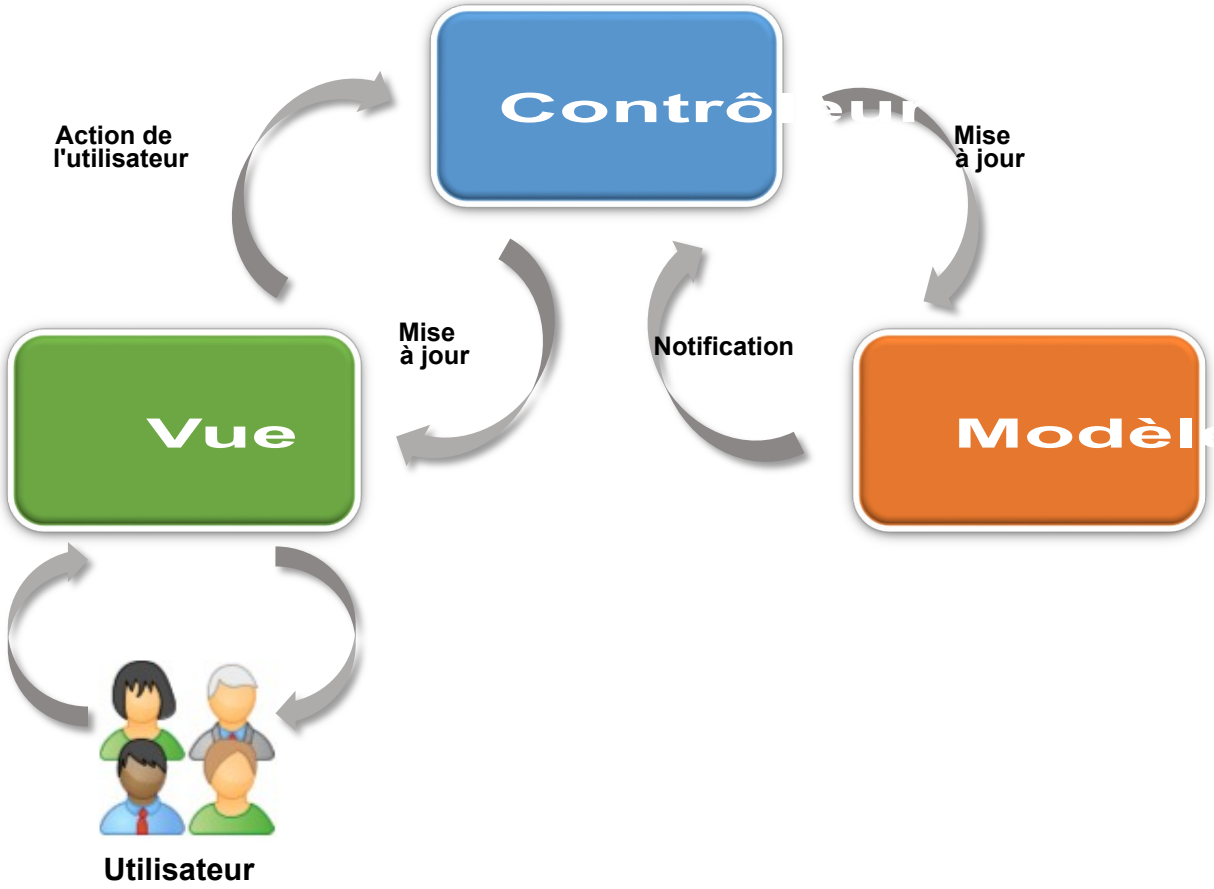
Model: Manages the **data** and **business logic**. Represents the state of the application and interacts with the database to fetch/store data.

View: Handles the **user interface** and presentation. Displays data from the model and captures user interactions, providing a visual representation.

Controller: Acts as an **intermediary** between Model and View. Receives user input, processes it (using the Model), and decides which View to render.

Interaction Flow:

- **User** interacts with the View.
- **Controller** processes user actions and updates the Model.
- **Model** notifies the View of changes.
- **View** updates accordingly.
- **Focus**: MVC splits application logic into distinct components—**Model, View, and Controller**—for better **organization**, **maintenance**, and **scalability**.



Overview of MVC Components

The Model Component

The **Model** in the MVC architecture is responsible for managing the **data**, **business logic**, and **state** of the application.

Responsibilities:

- **Data Management**: Represents data objects and interacts with the database to **fetch**, **update**, or **store** information.
- **Business Logic**: Implements the rules and processes that define how data is manipulated, validating data integrity.

Interaction:

- **Updates the View**: Notifies the View whenever there are changes in the data.
- **Communicates with Controller**: The Controller uses the Model to **process data** based on user input.

Examples:

- In a **shopping cart** app, the Model would represent products, prices, and manage business rules (e.g., calculating total cost).

Benefits:

- **Separation of Concerns**: Keeps data logic separate from UI, allowing easy modifications and data-focused testing.
- **Reusability**: The same Model can be reused across different Views.
- **Focus**: The Model handles data and business logic, ensuring that information is properly managed and kept consistent across the application.

The View Component

The **View** in the MVC architecture is responsible for managing the **user interface (UI)** and how information is presented to the user.

Responsibilities:

- **Display Data**: Renders data from the Model in a format that users can understand (e.g., forms, tables, charts).
- **Capture User Input**: Handles user actions like clicking buttons or filling out forms, and sends these inputs to the Controller for processing.

Interaction:

- **Receives Data**: Gets data from the Model to display.
- **Updates**: Refreshes the UI when there are changes in the Model.

Examples:

- In an **e-commerce** application, the View might display a list of products with prices and images for users to browse.

Benefits:

- **Separation of UI from Logic**: Keeps UI presentation separate from the business logic, making it easier to redesign or update the user interface.
- **Reusability**: Multiple Views can represent the same data differently (e.g., list view vs. grid view).
- **Focus**: The View is about **presentation**—how data is visually represented and how users interact with it, ensuring that UI and business logic are clearly separated.

The Controller Component

The **Controller** in the MVC architecture acts as an intermediary between the **Model** and **View**. It handles **user inputs**, processes them, and determines the appropriate responses.

Responsibilities:

- **Input Handling**: Captures user actions (e.g., clicks, form submissions) from the View.
- **Business Logic Processing**: Interprets these inputs, interacts with the Model to process data, and then decides what to display in the View.
- **Application Flow**: Determines the **flow of the application**, deciding which Views to render based on user interactions.

Interaction:

- **Communicates with Model**: Makes calls to the Model to **update data** or retrieve information.
- **Updates the View**: Decides which View should be rendered based on the current state of the Model.

Examples:

- In a **login system**, the Controller would take the user's credentials from the View, validate them with the Model, and then direct the user to the appropriate View (e.g., dashboard or error message).

Benefits:

- **Separation of Concerns**: Keeps user input logic separate from the UI and data, simplifying maintenance.
- **Reusability and Testability**: The Controller logic can be tested independently, and multiple Views can use the same Controller.
- **Focus**: The Controller manages **user input and application logic**, ensuring that data changes are properly propagated and that the user interface reflects those changes correctly.

Interaction Between MVC Components

Model-View-Controller (MVC) Interaction:

- **User Input**: The **Controller** handles user actions from the **View** (e.g., form submission).
- **Processing**: The Controller processes the input and interacts with the **Model** to **retrieve** or **update** data.
- **Model Updates**: The **Model** manages the data and business logic, making necessary changes and notifying the **View**.
- **Data Display**: The **View** updates its display based on changes in the Model, providing the user with an updated interface.
- **Focus**: MVC components interact seamlessly—**Controller** processes inputs, **Model** manages data, and **View** presents it—ensuring **separation of concerns** for better maintainability and scalability.

Implementing MVC in Python Without Frameworks

Why Implement MVC Without a Framework?

Educational Value:

- Understand the underlying mechanisms.

Flexibility:

- Learn how to structure applications from scratch.

Limitations:

- Not suitable for production environments.
- Lacks the features and security provided by frameworks.

WSGI in Python

WSGI (Web Server Gateway Interface):

A specification for a simple and universal interface between web servers and Python web applications.

Purpose:

Standardizes how Python apps interact with web servers, enabling compatibility and **flexibility** across servers and frameworks.

How It Works:

Web server forwards requests to the Python application via WSGI.

Benefits:

Promotes compatibility between web servers and applications.

Use:

Frameworks like **Flask** and **Django** leverage WSGI to run their applications.

WSGI Servers like **Gunicorn** or **uWSGI** are used to serve applications in production.

```python
# -*- coding: utf-8 -*-
from wsgiref.simple_server import make_server

# WSGI Application Function
def application(environ, start_response):
    status = '200 OK'  # HTTP Status Code
    headers = [('Content-Type', 'text/plain')]
    start_response(status, headers)
    return [b"Hello, WSGI!"]

# Running the WSGI server
if __name__ == '__main__':
    port = 8000
    httpd = make_server('', port, application)
    print(f"Serving WSGI app on http://localhost:{port} ...")

    httpd.serve_forever()
```

Limitations of No-Framework Approach

Scalability:

Manual handling becomes complex as the application grows.

Security:

Frameworks provide built-in security features.

Reinventing the Wheel:

Common functionalities are already implemented in frameworks.

Conclusion:

Good for learning, but frameworks are recommended for real applications.

Understanding Web Frameworks

Definition:

A **web framework** is a collection of packages or modules that allow developers to write web applications without handling low-level details like protocols and thread management.

Purpose:

Simplify web development tasks.

Provide tools and libraries for common web functionalities.

Benefits:

Increased productivity.

Security features.

Scalability and maintainability.

Preparing for Frameworks

Understanding MVC in Frameworks:

Frameworks like Django and Flask implement MVC (or similar) patterns.

Mapping Our Example to Frameworks:

Controllers map to views in frameworks.

Models can integrate with ORM systems.

Templates are managed by templating engines.

What Is Flask?

Definition:

Flask is a lightweight, micro web framework written in Python.

Characteristics:

Microframework: Minimalistic core, with optional extensions.

Flexibility: Does not enforce a specific project structure.

Easy to Learn: Ideal for beginners and small projects.

Key Features of Flask

Routing:

Map URLs to Python functions using decorators.

Templates:

Uses Jinja2 templating engine for dynamic HTML generation.

Request Handling:

Access request data via the `request` object.

Extensions:

Add functionality like database integration, authentication.

Built-in Development Server:

Quickly run applications locally for testing.

What Is Django?

Definition:

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design.

Characteristics:

Full-Featured: Comes with many built-in components.

"Batteries Included": Includes ORM, authentication system, admin interface, etc.

Convention over Configuration: Follows standard conventions to minimize configuration.

Key Features of Django

Object-Relational Mapper (ORM):

Interact with databases using Python classes instead of SQL.

Admin Interface:

Auto-generated, customizable interface for managing site content.

Authentication System:

Handles user accounts, groups, permissions, and more.

URL Routing:

Clean and readable URLs with a powerful URL dispatcher.

Template System:

Uses its own templating language for dynamic content.

Introduction to Flask

Key Features & Use Cases

Key Features:

Extensible: Add features using **Flask extensions**, such as Flask-SQLAlchemy for databases or Flask-WTF for forms.

Integrated Development Server: The **debug mode** allows live code reloading and detailed error reports.

RESTful API Ready: Flask's lightweight structure makes it ideal for **building REST APIs**.

Use Cases:

Prototypes: Ideal for quickly building **web app prototypes** and MVPs.

APIs: Frequently used for building **RESTful APIs** in backend services.

Simple to Medium Web Applications: Great for personal projects, small business websites, or tools.

Benefits:

Minimal Boilerplate: Easy to get started with very little code.

Flexibility: Does not enforce a particular project structure, giving developers the freedom to organize their projects.

```python
# -*- coding: utf-8 -*-
from flask import Flask
from flask import render_template

app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello, Flask!"

@app.route('/welcome')
def welcome():
    return render_template('index.html', name="Flask User")


if __name__ == '__main__':
    app.run(debug=True)
```