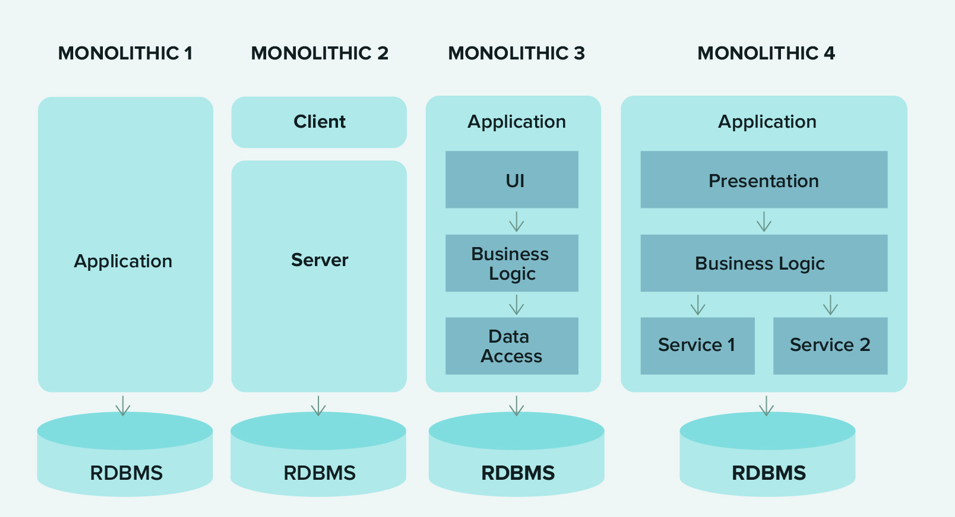
**Web Application Software Architecture**  
A **web application software architecture** defines the structural organization of a web‐based system: its components (clients, servers, databases, caches, APIs, etc.), their responsibilities, and how they interact. Proper architecture ensures scalability, maintainability, security, and performance.

* **Why it arose**
  + Early monolithic systems became brittle as web apps grew in complexity.
  + Evolving requirements (mobile clients, high traffic, rapid feature‐delivery) demanded clear “separation of concerns.”
  + DevOps practices (CI/CD, containerization) required modular, loosely coupled components.
* **Core rules & principles**
  + **Separation of Concerns** – UI, business logic, and data layers should be distinct.
  + **Single Responsibility** – each module/service does one thing and does it well.
  + **Loose Coupling & High Cohesion** – components interact over well‐defined interfaces (e.g., REST, messaging), minimizing interdependencies.
  + **Scalability & Resilience** – design for horizontal scaling (add more instances) and graceful failure (circuit breakers, retries).
  + **Security by Design** – authentication, authorization, input validation, encryption enforced at each layer.
* **When to use**
  + **Small apps** (few pages, low traffic): simple layered or even monolithic architecture may suffice.
  + **Mid‑size to large apps** (multiple teams, high availability): consider n‑tier, microservices, or service‑mesh architectures.
  + **Rapid iteration & frequent deployments**: favor modular or serverless designs.
* **Real‑world examples**
  + **Monolithic MVC (Model‑View‑Controller)**:
    - **ASP.NET MVC** apps where Views, Controllers, and the relational database live in one deployable package.
  + **Single‑Page App (SPA) + API backend**:
    - **React/Vue** front‑end served from CDN, communicating via REST/GraphQL to a Node.js or Django API.
  + **Microservices architecture**:
    - **Netflix** divides features (authentication, streaming, recommendations) into dozens of independently deployable services.

**Single‑Tier (Monolithic) Architecture**

In a **single‑tier** (or **monolithic**) setup, the entire application—UI, business logic, data access, and data store—runs as one process on a single machine.

* **Why it came**
  + Early desktop and small web apps needed simple deployment and minimal ops overhead.
  + Hardware was expensive; combining layers reduced resource consumption.
* **Rules & characteristics**
  + **All‑in‑one**: one codebase, one deployment unit.
  + **Shared memory/calls**: internal function calls, direct DB access.
  + **Simple operations**: one server to configure, monitor, and scale.
* **When to use**
  + **Prototypes**, **proofs of concept**, or **small teams**.
  + Applications with limited scaling needs (< hundreds of concurrent users).
* **Real‑world examples**
  + **Local CMS**: a lightweight WordPress instance running on a single VPS with MySQL embedded.
  + **Desktop database app**: Microsoft Access file with forms, logic, and data in one .accdb file.
  + **Early e‑commerce store**: legacy PHP app where product catalog, checkout, and admin panel share one codebase and database.



**Two‑Tier (Client‑Server) Architecture**

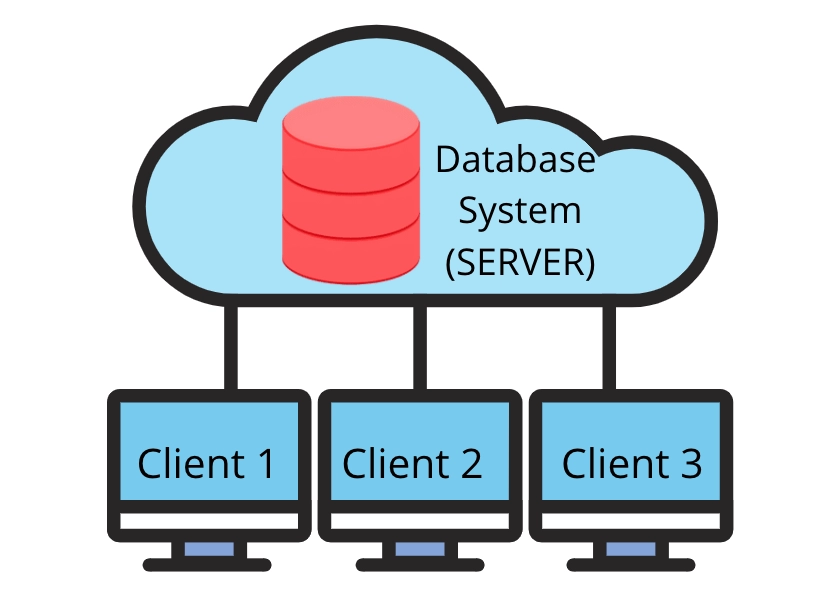
A **two‑tier** architecture splits the system into a **client** that handles presentation and a **server** that manages data and business logic.

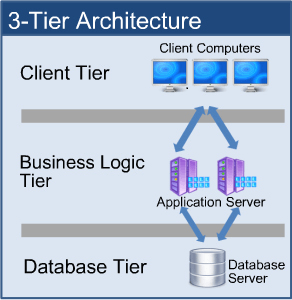
* **Why it came**
  + As user bases grew, running everything on the same machine became unsustainable.
  + Separating UI and data allowed teams to specialize and scale layers independently.
* **Rules & characteristics**
  + **Client tier**: renders UI, performs input validation, invokes server APIs.
  + **Server tier**: provides data services (e.g., SQL queries), enforces business rules, maintains state.
  + **Clear API boundary**: typically via REST, RPC, or proprietary protocols.
* **When to use**
  + **Internal business apps** where rich client features and centralized data management are needed but full n‑tier complexity isn’t justified.
  + **Desktop clients** talking to an enterprise database.
* **Real‑world examples**
  + **Microsoft Outlook + Exchange**: Outlook client handles UI; Exchange Server stores mailboxes and enforces policies.
  + **Browser + Web server for static sites**: browser requests HTML/CSS/JS from Apache or Nginx; server reads files from disk.
  + **ERP client on Windows + SQL Server**: a custom client application executes stored procedures on a central SQL Server database.

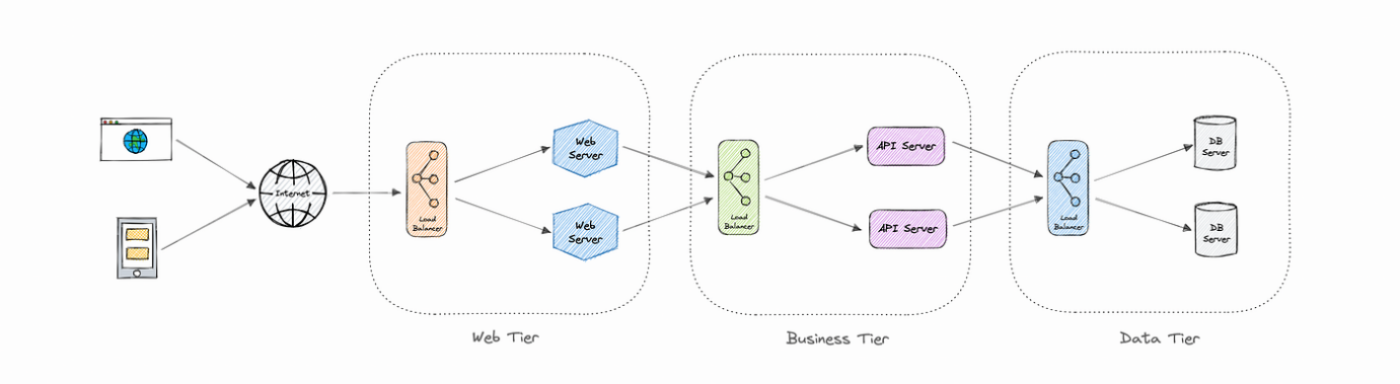
**Communication Between Client & Server**

Fundamental to client‑server (two‑tier and beyond) is **how** messages flow between client and server.

* **Why it matters**
  + Performance, reliability, and security hinge on choosing the right protocol and interaction pattern.
* **Core rules & patterns**
  + **Request–Response (synchronous)**
    - Client waits for server reply before proceeding.
    - Commonly uses HTTP/HTTPS over TCP.
  + **Asynchronous / Event‑Driven**
    - Client sends a message (e.g., via message queue) without blocking; server processes independently and may push results later.
    - Protocols: WebSockets, MQTT, AMQP.
  + **Statelessness vs. Stateful**
    - **Stateless** interactions (e.g., REST) improve scalability—each request carries all context.
    - **Stateful** connections (e.g., WebSocket sessions) enable real‑time updates.
  + **Security considerations**
    - Encrypt in transit (TLS).
    - Authenticate & authorize each call (OAuth, API keys).
    - Validate inputs at both ends.
* **When to use which**
  + **REST over HTTP**: general CRUD apps, public APIs, where caching and stateless scaling are priorities.
  + **WebSockets**: chat apps, live dashboards, collaborative editing requiring real‑time push.
  + **Message queues (AMQP, MQTT)**: IoT telemetry, background processing, pub/sub eventing.
* **Real‑world examples**
  + **RESTful API**: GitHub’s v3 API—clients issue HTTP GET/POST/PUT to manipulate repos; responses are JSON payloads.
  + **WebSockets**: Slack or Discord—persistent TCP connection for instant message delivery and presence updates.
  + **MQTT in IoT**: smart home sensors publish readings to an MQTT broker; mobile apps subscribe to topics for updates.







**Pros & Cons**

| **Pros** | **Cons** |
| --- | --- |
| ▶ Easy to develop and test: single codebase and single test harness. |  |
| ▶ Simple deployment: one artifact, one environment. |  |
| ▶ High performance internally (no RPC overhead). |  |
| ▶ Lower operational overhead: fewer servers/configurations. |  |
| ▶ Straightforward debugging and profiling. | ▶ Poor scalability: you must scale the whole app, not parts. |
| ▶ Slower release cycles: a small change forces redeploy of entire system. |  |
| ▶ Tight coupling: hard for multiple teams to work independently. |  |
| ▶ Technology lock‑in: can’t mix languages or frameworks per feature. |  |
| ▶ Risk of “big ball of mud”: codebase becomes unmanageable as it grows. |  |

## Microservices Architecture

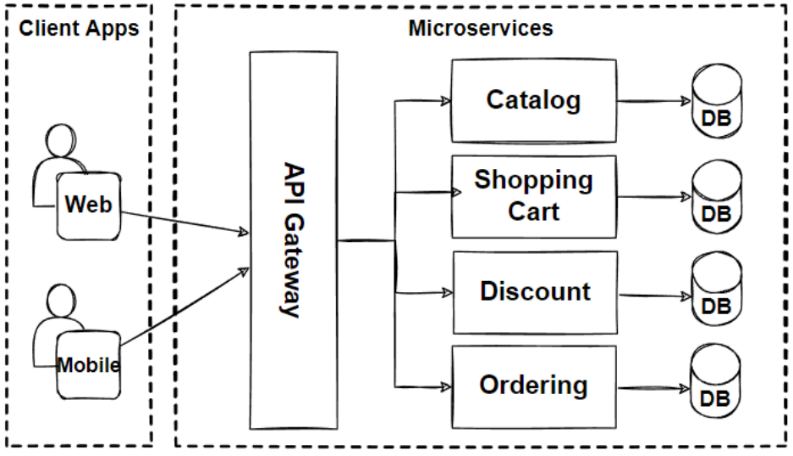
A **microservices architecture** decomposes an application into a suite of small, autonomous services, each running in its own process and communicating over lightweight protocols (e.g., HTTP/REST, gRPC, messaging).

**Why & How It Emerged**

* **Scale & Resilience Needs**: Companies like Netflix outgrew monoliths and needed finer‑grained scaling and isolation.
* **Conway’s Law & Team Autonomy**: Aligns code boundaries with organizational teams, enabling independent development and deployment.
* **Polyglot Flexibility**: Allows each service to choose the best language, framework, or data store for its problem.

**Where & When to Use**

* **Large, Complex Domains** with clear bounded contexts (e.g., e‑commerce, media streaming).
* **High Throughput Systems** requiring different scaling profiles per component.
* **Continuous Deployment Environments** where teams deploy independently dozens of times per day.
* **Multi‑tenant or SaaS Platforms** that need fault isolation between features or customers.



**Pros & Cons**

| **Pros** | **Cons** |
| --- | --- |
| ▶ Independent Deployment & Rollback: services can be updated without touching others. |  |
| ▶ Fine‑Grained Scalability: scale only high‑load services. |  |
| ▶ Fault Isolation: one service crash won’t necessarily take down the whole system. |  |
| ▶ Technology Heterogeneity: each team picks optimal tech stack. |  |
| ▶ Organizational Alignment: mirrors team structure for faster feature delivery. | ▶ Operational Complexity: dozens–hundreds of services to deploy, monitor, secure. |
| ▶ Distributed‑systems Challenges: network latency, retries, eventual consistency. |  |
| ▶ Testing Overhead: end‑to‑end integration tests become more complex. |  |
| ▶ Data Management Complexity: distributed transactions, data duplication, consistency models. |  |
| ▶ Higher Resource Usage: multiple runtimes, containers, and orchestration overhead. |  |

**Real‑World Examples**

1. **Netflix**
   * 700+ services for user accounts, recommendations, streaming, billing—all independently deployed.
2. **Amazon**
   * Originally a monolith, now hundreds of microservices for inventory, checkout, personalization, logistics.
3. **Uber**
   * Separate services for rider matching, pricing, maps, fare calculation, and notifications.
4. **Spotify**
   * “Squad” teams own services like playlist management, search, and social sharing.

**Choosing Between Monolith & Microservices**

| **Factor** | **Favor Monolith** | **Favor Microservices** |
| --- | --- | --- |
| **Team Size** | Small (1–5 developers) | Larger (> 10 developers, multiple teams) |
| **Domain Complexity** | Simple or well‑contained | Complex with clear bounded contexts |
| **Release Cadence** | Weekly or monthly | Multiple deploys per day |
| **Operational Maturity** | Minimal DevOps (single server/VM) | Mature DevOps/CI-CD, container orchestration |
| **Scalability Needs** | Modest, uniform scaling | Highly variable load across components |
| **Technology Flexibility** | Single stack is acceptable | Need for polyglot services |

By weighing these considerations—team structure, domain complexity, operational maturity, and scalability requirements—you can select the architecture style that best aligns with your project’s goals and constraints

## **Monolithic vs Microservices quiz**

**1. What is a monolithic architecture?**

A) An architecture with distributed services  
B) A single-tiered software application where all components are tightly coupled  
C) A cloud-native design using containers  
D) A peer-to-peer architecture  
  
**Answer:** B) A single-tiered software application where all components are tightly coupled  
**Explanation:** Monolithic architecture means the entire application is built as one unit.

**2. What is a key advantage of microservices over monolithic architecture?**

A) Simpler to develop initially  
B) Better scalability and independent deployment  
C) Requires fewer servers  
D) Less network overhead  
  
**Answer:** B) Better scalability and independent deployment  
**Explanation:** Microservices allow services to scale and deploy independently.

**3. In which architecture do you typically face *tight coupling* between components?**

A) Microservices  
B) Serverless  
C) Monolithic  
D) Cloud-native  
  
**Answer:** C) Monolithic  
**Explanation:** In monoliths, components are often tightly coupled and hard to isolate.

**4. Which of the following is a *disadvantage* of microservices architecture?**

A) Difficult to scale  
B) Complex to manage and deploy  
C) Single point of failure  
D) Cannot use modern tech stack  
  
**Answer:** B) Complex to manage and deploy  
**Explanation:** Microservices involve managing multiple services, which increases operational complexity.

**5. What is the typical communication method between microservices?**

A) Function calls  
B) File sharing  
C) REST APIs or messaging queues  
D) Shared memory  
  
**Answer:** C) REST APIs or messaging queues  
**Explanation:** Microservices communicate over network protocols using APIs or message brokers.

**6. What is a common challenge in monolithic architecture?**

A) Slow communication between components  
B) Difficult deployment and scaling of individual features  
C) Too many independent services  
D) Loose coupling  
  
**Answer:** B) Difficult deployment and scaling of individual features  
**Explanation:** In monoliths, all components are deployed together, limiting flexibility.

**7. Which architecture best supports Continuous Deployment and DevOps practices?**

A) Monolithic  
B) Client-server  
C) Microservices  
D) Batch processing  
  
**Answer:** C) Microservices  
**Explanation:** Microservices enable CI/CD by allowing teams to deploy services independently.

**8. In a microservices architecture, each service should be:**

A) Dependent on others  
B) Managed by the same team  
C) Independently deployable and loosely coupled  
D) Written in the same programming language  
  
**Answer:** C) Independently deployable and loosely coupled  
**Explanation:** Independence is a core principle of microservices.

**9. Which architecture is more suitable for small teams and simple applications?**

A) Event-driven  
B) Microservices  
C) Monolithic  
D) Serverless  
  
**Answer:** C) Monolithic  
**Explanation:** Monoliths are easier to build and manage for small applications.

**10. Which architecture allows using different technologies for different components?**

A) Monolithic  
B) Microservices  
C) Mainframe  
D) Two-tier  
  
**Answer:** B) Microservices  
**Explanation:** Each microservice can be developed using different languages and tools.

**11. Which of the following is easier to debug due to its simplicity?**

A) Microservices  
B) Monolithic  
C) Distributed  
D) Event-driven  
  
**Answer:** B) Monolithic  
**Explanation:** Monoliths have fewer components, making debugging simpler compared to distributed services.

**12. What is the term used when multiple microservices are managed and orchestrated together?**

A) Containerization  
B) Service Mesh  
C) Monolith  
D) Load Balancing  
  
**Answer:** B) Service Mesh  
**Explanation:** Service Mesh like Istio handles service-to-service communication in microservice architecture.

**13. Which architecture is more prone to a single point of failure?**

A) Microservices  
B) Monolithic  
C) Serverless  
D) Mesh  
  
**Answer:** B) Monolithic  
**Explanation:** A failure in one part of a monolith can crash the entire system.

**14. Which tool is commonly used to manage and deploy microservices at scale?**

A) Docker  
B) Nginx  
C) Kubernetes  
D) Apache  
  
**Answer:** C) Kubernetes  
**Explanation:** Kubernetes orchestrates and manages containerized microservices.

## **Introduction to REST**

**REST** (Representational State Transfer) is an architectural style for designing networked applications. Coined by Roy Fielding in his 2000 PhD dissertation, REST provides a set of guiding principles—rather than a strict protocol—on how web standards (particularly HTTP) should be used to build scalable, maintainable, and evolvable APIs.

**What Is a REST API?**

A **REST API** is an application programming interface that adheres to REST principles. It exposes “resources” (nouns) over HTTP, allowing clients to perform operations via standard HTTP methods (verbs). Each resource is identified by a URI, and clients manipulate these resources by exchanging representations (typically JSON or XML).

* **Resource**: Any object or concept, e.g., a user, order, or image.
* **Representation**: A document (JSON, XML, HTML) that captures the resource’s state.
* **Endpoint**: A URL where a given resource can be accessed, e.g.,
  + GET /users/123 → fetch user #123
  + POST /orders → create a new order

**Six Core REST Constraints**

1. **Uniform Interface**
   * Consistent URI structure and HTTP methods.
   * Decouples client and server via standardized interactions.
2. **Statelessness**
   * Each request contains all the information needed (no server‑side session).
   * Simplifies scaling: any server can handle any request.
3. **Cacheability**
   * Responses explicitly marked as cacheable or non‑cacheable.
   * Improves performance by reducing client‑server round trips.
4. **Client–Server Separation**
   * Clients handle UI/UX; servers handle data storage and business logic.
   * Enables independent evolution of each side.
5. **Layered System**
   * Architecture may include intermediaries (load balancers, proxies, gateways).
   * Clients aren’t aware whether they connect directly to the end server.
6. **Code on Demand (Optional)**
   * Servers can send executable code (e.g., JavaScript) to extend client functionality.

**Why REST?**

* **Simplicity & Familiarity**: Leverages HTTP methods (GET, POST, PUT, DELETE).
* **Scalability**: Statelessness and caching facilitate horizontal scaling.
* **Interoperability**: Uniform interface allows diverse clients (web, mobile, IoT) to interact.
* **Evolvability**: Loose coupling ensures servers can change without breaking clients.

**When & Where to Use REST APIs**

* **Public Web Services**: When you need a simple, HTTP‑based interface for third‑party developers.
* **Microservices Communication**: Lightweight RPC between services when eventual consistency is acceptable.
* **Mobile & Single‑Page Apps**: Fetching JSON payloads to dynamically update UIs.
* **IoT & Edge Devices**: Stateless, resource‑oriented interactions over constrained networks.

**Real‑World Examples**

1. **GitHub REST API**
   * **Resources**: Repositories, issues, pull requests.
   * **Typical call**: GET https://api.github.com/repos/octocat/Hello-World/issues retrieves issues for a repo.
   * **Use‑case**: Third‑party tools integrate with GitHub for automation, analytics, and CI/CD.
2. **Stripe API**
   * **Resources**: Customers, charges, subscriptions.
   * **Typical call**: POST https://api.stripe.com/v1/charges creates a new payment charge.
   * **Use‑case**: E‑commerce platforms embed Stripe for payment processing with minimal PCI‑DSS scope.
3. **Twitter API (v2)**
   * **Resources**: Tweets, users, timelines.
   * **Typical call**: GET https://api.twitter.com/2/tweets?ids=1453489038376132611 fetches specific tweets.
   * **Use‑case**: Analytics dashboards and social‑media management tools aggregate and post tweets.
4. **OpenWeatherMap API**
   * **Resources**: Current weather, forecasts, historical data.
   * **Typical call**: GET https://api.openweathermap.org/data/2.5/weather?q=London&appid={KEY} returns London’s weather.
   * **Use‑case**: Mobile weather apps and IoT devices display localized forecasts.

**Best Practices**

* **Use nouns, not verbs** in URIs:

GET /orders/123/items ≫ good

GET /getOrderItems?orderId=123 ≫ avoid

* **Leverage HTTP status codes** for clear outcomes:
  + 200 OK, 201 Created, 204 No Content
  + Client errors: 400 Bad Request, 401 Unauthorized
  + Server errors: 500 Internal Server Error
* **Support filtering, sorting, and pagination** via query parameters:

GET /products?page=2&limit=50&sort=price\_desc&category=books

* **Document thoroughly** using tools like Swagger/OpenAPI to enable discoverability.

## **🧱 What is a Java Servlet?**

**🔹 Definition**

A **Servlet** is a Java class used to build **dynamic web content**. It runs on a **Java-enabled web server (Servlet container)** and responds to HTTP requests (like form submissions, API calls, etc.).

Think of a **Servlet** as:

A Java program that runs on the server and generates web content (usually HTML or JSON) based on user input or request parameters.

**🔹 Why Servlets Came?**

* Before Servlets, dynamic content was created using **CGI (Common Gateway Interface)**, which launched a new OS process per request—inefficient and slow.
* Servlets were introduced to:
  + Use Java’s scalability and portability
  + Reuse threads instead of creating processes
  + Maintain persistent state across requests (via sessions)

**🛠 How Servlets Work: High-Level Flow**

1. **Client (Browser)** sends a request (e.g., via a form or URL).
2. **Servlet Container** (e.g., Tomcat) receives the request.
3. The **container locates and invokes the appropriate Servlet**.
4. The **Servlet processes the request**, performs logic (maybe DB calls), and generates a **response**.
5. The container sends this response back to the client.

**✳️ Servlet Lifecycle (Managed by the Container)**

1. **init()**
   * Called once when the servlet is loaded. Used for setup/config.
2. **service()**
   * Called every time a request is made. Handles GET, POST, etc.
3. **destroy()**
   * Called before the servlet is removed from memory. Used for cleanup.

**🧪 Real-World Examples of Servlets**

**Example 1: Login Servlet**

@WebServlet("/login")

public class LoginServlet extends HttpServlet {

protected void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

String user = request.getParameter("username");

String pass = request.getParameter("password");

if("admin".equals(user) && "admin123".equals(pass)) {

response.getWriter().println("Login successful!");

} else {

response.getWriter().println("Invalid credentials.");

}

}

}

**Example 2: Order Processing Servlet**

* E-commerce system calls /placeOrder which:
  + Reads cart data
  + Validates stock
  + Stores order in DB
  + Returns order confirmation page

**Example 3: Weather API Servlet**

* Client calls /weather?city=London
* Servlet:
  + Looks up data from DB or API
  + Returns JSON response to frontend

**🚀 What is a Java Servlet Container?**

**🔹 Definition**

A **Servlet Container (aka Web Container)** is a part of a web server or application server that:

* **Manages lifecycle** of servlets
* **Routes HTTP requests** to the correct servlet
* Provides **API support** for:
  + Sessions
  + Cookies
  + Security
  + Request/Response handling

It acts like a **runtime environment** for Servlets.

**🔹 Popular Servlet Containers:**

* **Apache Tomcat** (most widely used)
* Jetty
* WildFly (includes full Java EE stack)
* GlassFish

**📦 Servlet Container Responsibilities**

1. **Servlet Management** (init, destroy)
2. **Request Routing** (URL mapping to servlet)
3. **Multithreading** (handles concurrent requests with thread pool)
4. **Security** (authentication, HTTPS)
5. **Session Management** (via cookies or URL rewriting)

**🧾 Real-World Examples of Servlet Containers**

1. **Tomcat in a Banking App**  
   Hosts secure, stateless servlets that perform KYC validation and show account balances.
2. **Jetty in a Microservices Environment**  
   Lightweight and embeddable, ideal for internal service endpoints (e.g., /payment, /inventory).
3. **Tomcat in a University Portal**  
   Hosts servlets for login, course registration, grading, and file uploads.

**✅ When to Use Servlets**

* Building simple, dynamic web apps with custom logic.
* When you want **full control over HTTP request/response cycle**.
* Lightweight microservices needing Java’s performance but not full Spring stack.
* Educational purposes for understanding how web apps work under the hood.

**❌ When Not to Use Plain Servlets**

* For complex apps, plain servlets can become verbose and hard to manage.  
  **Use a framework** like:
  + **Spring MVC**
  + **Jakarta Faces (JSF)**
  + **Struts (legacy)**

These frameworks still use servlets **under the hood**, but offer abstraction, routing, dependency injection, and templating.

**Summary**

| **Concept** | **Description** |
| --- | --- |
| **Servlet** | Java class for handling HTTP requests & generating responses |
| **Servlet Container** | Manages lifecycle & routing of Servlets |
| **Popular Containers** | Tomcat, Jetty, WildFly |
| **Used In** | Login systems, REST APIs, dashboards, internal tools |

## **🎯 What are Software Design Patterns?**

**Design Patterns** are **proven solutions to common software design problems**. They are not code templates, but rather best practices or guidelines for solving particular challenges in software architecture and object-oriented design.

They provide:

* A **shared vocabulary** for developers
* **Reusable solutions** that are tried and tested
* **Maintainability and scalability** in growing systems

Think of design patterns like architectural blueprints—while the actual building may vary, the structure has common, reusable solutions (like placing support beams in key areas).

**💡 Why Design Patterns?**

* Help solve **recurring design problems**
* Improve **code reusability, readability, and maintainability**
* Support **object-oriented principles** like SOLID
* Encourage **loose coupling** and **high cohesion**
* Enable **team collaboration** with shared concepts

**🔗 Types of Design Patterns**

**1. Creational Patterns – how objects are created**

* Focus on object instantiation control

**2. Structural Patterns – how objects are composed**

* Help form large object structures (like UIs)

**3. Behavioral Patterns – how objects interact**

* Focus on communication between objects

**🛠️ Popular Design Patterns with Real-World Examples**

**🧱 1. Singleton Pattern (Creational)**

**Ensures only one instance of a class exists in the system.**

🧠 *Used when you want a global, shared object.*

**Real-World Examples:**

* **Logger Service**: One logger for the entire app.
* **Database Connection Pool**: Reuses the same DB instance.
* **Configuration Manager**: Loads app settings only once.

public class Config {

private static Config instance = null;

private Config() {} // private constructor

public static Config getInstance() {

if (instance == null) {

instance = new Config();

}

return instance;

}

}

**🧰 2. Factory Pattern (Creational)**

**Delegates object creation to a factory class, based on input or context.**

🧠 *Used when object creation is complex or varies based on input.*

**Real-World Examples:**

* **UI Toolkit**: Factory produces Button objects (Windows/Mac).
* **Vehicle Rental App**: Creates Car, Bike, Scooter objects based on user choice.
* **Document Parser**: XML, JSON, CSV parser objects from a factory.

public class VehicleFactory {

public static Vehicle getVehicle(String type) {

if (type.equalsIgnoreCase("car")) return new Car();

else if (type.equalsIgnoreCase("bike")) return new Bike();

return null;

}

}

**🧱 3. Adapter Pattern (Structural)**

**Wraps one interface to make it compatible with another.**

🧠 *Used when integrating with legacy code or external libraries.*

**Real-World Examples:**

* **Charging Adapter**: Converts USB-C to USB-A.
* **Payment Gateway Adapter**: Stripe vs PayPal.
* **Media Player App**: Wraps MP3, MP4, VLC into one interface.

class MediaAdapter implements MediaPlayer {

AdvancedMediaPlayer player;

public MediaAdapter(String audioType) {

if(audioType.equals("vlc")) player = new VlcPlayer();

}

public void play(String type, String file) {

player.playVlc(file);

}

}

**🔄 4. Observer Pattern (Behavioral)**

**One-to-many dependency: when one object changes, its dependents are notified.**

🧠 *Used in event-driven systems.*

**Real-World Examples:**

* **Notification System**: Email/SMS/Push alerts when an order is placed.
* **Stock Ticker**: Multiple subscribers to price changes.
* **GUI Framework**: Buttons notify listeners on click.

interface Observer {

void update(String message);

}

class User implements Observer {

public void update(String msg) {

System.out.println("Notification: " + msg);

}

}

**⛓ 5. Strategy Pattern (Behavioral)**

**Defines a family of algorithms, encapsulates each, and makes them interchangeable.**

🧠 *Used when you want to switch behavior at runtime.*

**Real-World Examples:**

* **Payment Methods**: Credit Card, UPI, PayPal.
* **Sorting Algorithms**: QuickSort, MergeSort, BubbleSort.
* **Travel Routes**: Driving, walking, biking.

interface PaymentStrategy {

void pay(int amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) { System.out.println("Paid via Card"); }

}

**🧾 Summary Table**

| **Pattern** | **Type** | **Real World Use Case** |
| --- | --- | --- |
| Singleton | Creational | Logger, Config Loader |
| Factory | Creational | UI Components, Vehicle Rental |
| Adapter | Structural | Plug Converters, Payment Gateway Integration |
| Observer | Behavioral | Notifications, Stock Updates, GUI Events |
| Strategy | Behavioral | Payment Options, Route Planner, Sorting Algorithms |

**📚 When to Learn/Use Design Patterns**

* **As you scale** from basic to intermediate projects
* **When code complexity increases**
* **For team projects** with multiple contributors
* **When maintaining/rewriting legacy code**