**1. Monolithic Architecture**

**Definition**: A monolithic architecture is a single, unified application where all components (UI, business logic, database access, etc.) are tightly coupled and deployed as one unit.

**Real-Life Example**: Imagine a traditional e-commerce website like an early version of Amazon (circa 2000). The entire system—product catalog, shopping cart, payment processing, and user accounts—is built as one application. If a developer updates the payment module, the whole app (including unrelated features like product search) must be redeployed.

**Characteristics**:

* Simple to develop and deploy initially.
* Single codebase, single deployment unit.
* Scaling means duplicating the entire app.

**Limitation**: As the app grows, it becomes harder to maintain, scale specific features, or adopt new technologies. A bug in one module (e.g., payment) can crash the whole system.

**2. Microservice Architecture**

**Definition**: Microservice architecture breaks an application into smaller, independent services that communicate via APIs. Each service handles a specific function and can be developed, deployed, and scaled separately.

**Real-Life Example**: Think of modern Netflix. Instead of one giant app, Netflix has separate services for user authentication, video streaming, recommendation engine, and billing. If the recommendation engine needs an update, it can be redeployed without touching the streaming service.

**Characteristics**:

* Decentralized, loosely coupled services.
* Independent deployment and scaling (e.g., scale only the streaming service during peak hours).
* Teams can use different tech stacks for different services.

**Comparison Scenario (Microservices Overcoming Monolithic Limitation)**:

* **Scenario**: An e-commerce platform during Black Friday.
* **Monolithic Limitation**: In a monolithic app, a surge in traffic to the payment module requires scaling the entire app (including product search and user login), wasting resources and slowing updates.
* **Microservices Advantage**: With microservices, the payment service can be scaled independently (e.g., more servers for payment processing) while keeping the product catalog service lightweight. This reduces costs and allows faster feature updates (e.g., adding a new payment gateway without redeploying the whole app).

**Limitation**: Microservices introduce complexity in communication (e.g., API latency) and data consistency across services. If the payment service notifies the inventory service too slowly, a product might oversell.

**3. Event-Driven Architecture**

**Definition**: Event-driven architecture focuses on producing, detecting, and reacting to events (state changes) asynchronously. Services communicate by publishing and subscribing to events via a message broker (e.g., Kafka, RabbitMQ).

**Real-Life Example**: Consider Uber’s ride system. When a driver accepts a ride (an event), the system triggers multiple actions: notifying the rider, updating the driver’s status, and recalculating traffic routes. These actions happen asynchronously via events, not direct service calls.

**Characteristics**:

* Asynchronous communication via events.
* Highly responsive and scalable for real-time systems.
* Decouples producers (event creators) from consumers (event reactors).

**Comparison Scenario (Event-Driven Overcoming Microservices Limitation)**:

* **Scenario**: An e-commerce platform processing a purchase.
* **Microservices Limitation**: In a microservices setup, the payment service might call the inventory service via an API to reserve stock, then call the shipping service. If the inventory service is slow or down, the payment process stalls, causing delays or overselling.
* **Event-Driven Advantage**: With event-driven architecture, the payment service publishes a “PaymentCompleted” event to a message broker. The inventory service listens for this event and reserves stock, while the shipping service independently prepares a label. If one service is slow, others aren’t blocked, improving resilience and speed. The system can also easily add new listeners (e.g., a marketing service sending a “thank you” email) without changing existing services.

**Limitation**: Event-driven systems can be harder to debug (tracking events across services) and require robust message brokers.

**Comparison Summary**

| **Aspect** | **Monolithic** | **Microservices** | **Event-Driven** |
| --- | --- | --- | --- |
| **Structure** | Single unit | Independent services | Event-based communication |
| **Scalability** | Scale entire app | Scale specific services | Scale based on event load |
| **Flexibility** | Rigid, all-or-nothing updates | Independent updates | Dynamic, decoupled reactions |
| **Complexity** | Simple initially, grows unwieldy | Moderate (API management) | Higher (event tracking) |
| **Example Use Case** | Small e-commerce site | Netflix’s modular platform | Uber’s real-time ride system |

* **Microservices overcome monolithic limitations** by enabling independent scaling and updates.
* **Event-driven overcomes microservices limitations** by reducing direct dependencies and enabling asynchronous, resilient workflows.

**OAuth (Explained)**

**Definition**: OAuth is an authorization framework that allows a third-party application to access a user’s resources (e.g., data) on another service without sharing the user’s credentials (e.g., password).

**How It Works**:

1. **User Action**: A user wants to log into a website (e.g., a photo editor) using their Google account.
2. **Redirect**: The photo editor redirects the user to Google’s login page.
3. **Consent**: The user logs into Google and agrees to share specific data (e.g., email, profile info) with the photo editor.
4. **Token Exchange**: Google issues an **access token** to the photo editor, which it uses to access the user’s Google data via APIs.
5. **Access**: The photo editor uses the token (not the user’s password) to fetch data securely.

**Real-Life Example**: When you “Sign in with Google” on a site like Medium, OAuth lets Medium access your Google profile info without Medium ever seeing your Google password.

**Key Features**:

* **Security**: Tokens are temporary and scoped (limited to specific permissions).
* **Convenience**: Users avoid creating new accounts/passwords.
* **Common Use**: OAuth 2.0 is widely used (e.g., Google, Facebook, GitHub logins).

**Relevance to Architectures**: OAuth fits well with microservices and event-driven systems, as authentication can be a standalone service issuing tokens that other services validate asynchronously.