**1. Monolithic Architecture**

**Definition**: A monolithic architecture is a single, all-in-one application where every component (frontend, backend, database logic) is bundled together and runs as a single process.

**Real-Life Example**: Imagine a small local restaurant’s online ordering system built as a single app. The menu display, order placement, payment processing, and kitchen notification are all part of one program. If the owner wants to add a new feature (e.g., loyalty points), the entire system must be updated and redeployed.

**Characteristics**:

* Easy to build and test for small-scale apps.
* One codebase, one server.
* Changes affect the whole system.

**Limitation**: As the restaurant grows (e.g., more orders, more features), the app becomes slow to update, and a failure in one part (e.g., payment) can bring down the whole site.

**2. Microservice Architecture**

**Definition**: Microservice architecture splits an application into small, independent services that each handle a specific task and communicate over a network (e.g., via APIs).

**Real-Life Example**: Think of a modern food delivery app like DoorDash. It’s split into services: one for browsing restaurants, one for placing orders, one for tracking deliveries, and one for payments. If DoorDash wants to improve delivery tracking, they update only that service without touching the payment system.

**Characteristics**:

* Each service can scale or fail independently.
* Teams can work on different services simultaneously.
* Flexible tech choices (e.g., Python for one service, Node.js for another).

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

* **Scenario**: A busy holiday season for the restaurant’s ordering system.
* **Monolithic Limitation**: In the monolithic restaurant app, a spike in orders overwhelms the payment module, slowing down the entire system (menu browsing, order submission, everything). Scaling requires duplicating the whole app, even unused parts.
* **Microservices Advantage**: With microservices, the payment service can be scaled up (e.g., more servers) to handle the holiday rush, while the menu service stays lightweight. Adding a new feature (like a holiday discount) can be deployed to the order service without redeploying everything, saving time and resources.

**Limitation**: Microservices rely on synchronous API calls, which can create bottlenecks. If the payment service takes too long to confirm with the kitchen service, orders pile up.

**3. Event-Driven Architecture**

**Definition**: Event-driven architecture uses events (notifications of changes) to trigger actions across systems. Services react to events asynchronously through a message broker, rather than waiting for direct responses.

**Real-Life Example**: Consider a smart home system like Amazon Alexa. When you say “turn on the lights,” it generates an event. The lighting service hears this event and turns on the lights, while a separate logging service records the action—all without one service directly calling another.

**Characteristics**:

* Loose coupling: Services don’t need to know each other.
* High responsiveness for real-time needs.
* Fault-tolerant: If one service lags, others keep running.

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

* **Scenario**: The food delivery app processing an order.
* **Microservices Limitation**: In a microservices setup, the order service calls the payment service, which then calls the delivery service to assign a driver. If the delivery service is overloaded, the whole chain waits, delaying the customer’s confirmation.
* **Event-Driven Advantage**: With an event-driven approach, the order service publishes an “OrderPlaced” event. The payment service processes it and publishes a “PaymentConfirmed” event. The delivery service listens for this and assigns a driver—all independently. If the delivery service is slow, the customer still gets a payment confirmation instantly, and the driver is assigned later. New features (e.g., a notification service texting the customer) can subscribe to events without rewriting existing code.

**Limitation**: Complexity increases with event management (e.g., ensuring events aren’t lost or duplicated).

**Comparison Summary**

| **Aspect** | **Monolithic** | **Microservices** | **Event-Driven** |
| --- | --- | --- | --- |
| **Structure** | All-in-one app | Separate services | Event-triggered actions |
| **Scalability** | Entire app scales together | Per-service scaling | Scales with event volume |
| **Flexibility** | Limited by single codebase | Modular updates | Adapts to new event listeners |
| **Complexity** | Low initially, high later | Moderate (networking overhead) | High (event orchestration) |
| **Example Use Case** | Small restaurant website | DoorDash’s delivery platform | Alexa’s smart home control |

* **Microservices overcome monolithic limitations** by isolating and scaling specific functions.
* **Event-driven overcomes microservices limitations** by enabling asynchronous, decoupled workflows.

**OAuth (Explained)**

**Definition**: OAuth is a secure way for one app to access a user’s data from another service without needing their password, using tokens instead.

**How It Works**:

1. **User Action**: Your friend wants to log into a music playlist app using their Spotify account.
2. **Redirect**: The playlist app sends them to Spotify’s login page.
3. **Consent**: They log in and approve sharing their Spotify playlists with the app.
4. **Token Exchange**: Spotify gives the playlist app an **access token**, which it uses to fetch playlist data via Spotify’s API.
5. **Access**: The app pulls the playlists without ever knowing your friend’s Spotify password.

**Real-Life Example**: When you “Sign in with Facebook” on a game app, OAuth lets the game access your Facebook name and friends list securely.

**Key Features**:

* **Security**: Tokens expire and limit access (e.g., only playlists, not messages).
* **Ease**: No need for new usernames/passwords.
* **Popular Use**: OAuth 2.0 powers logins for Spotify, Facebook, Twitter, etc.

**Relevance to Architectures**: In a microservices or event-driven system, an authentication service can issue OAuth tokens, which other services validate independently—perfect for distributed apps.