### ✅ Microservices Design Patterns: Explanation, Questions, Use Cases, Expected Answers, and C# Code Snippets

## 1. API Gateway Pattern

**Explanation:** Central entry point for all client requests to microservices.

**Use Case:** Client → API Gateway → routes to Product, Order, Auth services.

**C# Code (Ocelot Gateway config example):**

{

"Routes": [

{

"DownstreamPathTemplate": "/api/products",

"UpstreamPathTemplate": "/products",

"DownstreamHostAndPorts": [

{ "Host": "product-service", "Port": 80 }

]

}

]

}

**Questions and Answers:**

1. **What is the role of an API Gateway in microservices?**
   * **A:** It routes requests, handles authentication, throttling, and aggregates responses.
2. **How does API Gateway improve security?**
   * **A:** It enforces centralized authentication and SSL termination.
3. **What are some drawbacks of API Gateway?**
   * **A:** Single point of failure, increased latency, and complexity.
4. **How do you handle failure in downstream services?**
   * **A:** Use fallback, retry policies, and circuit breakers.
5. **When to avoid API Gateway?**
   * **A:** In simple apps or when you don't need request aggregation.

## 2. Circuit Breaker Pattern

**Explanation:** Prevents cascading failure by breaking the circuit when a service fails repeatedly.

**Use Case:** Product service fails → Circuit opens → Gateway returns fallback.

**C# Example (Polly):**

Policy.Handle<Exception>()

.CircuitBreaker(2, TimeSpan.FromMinutes(1));

**Questions and Answers:**

1. **What is the purpose of a circuit breaker?**
   * **A:** To prevent repeated requests to a failing service.
2. **What are the circuit states?**
   * **A:** Closed (normal), Open (fail-fast), Half-open (test mode).
3. **How long does the circuit remain open?**
   * **A:** Until timeout expires or a health check succeeds.
4. **Where do you implement circuit breakers?**
   * **A:** At API Gateway or inside service consumers.
5. **How is this different from retry?**
   * **A:** Retry keeps retrying, circuit breaker gives up early and prevents new calls.

## 3. Saga Pattern

**Explanation:** Manages distributed transactions across microservices using compensating actions.

**Use Case:** Order service → Payment → Inventory → If failure, undo previous steps.

**Choreography C# Example:**

// On OrderCreated event

await paymentService.Charge(orderId);

// If failure → emit OrderCancelled

**Questions and Answers:**

1. **Why use Saga instead of 2PC?**
   * **A:** 2PC doesn’t scale and introduces tight coupling.
2. **What are types of Saga?**
   * **A:** Choreography (event-driven), Orchestration (central coordinator).
3. **When is Saga preferred?**
   * **A:** In long-running or asynchronous business workflows.
4. **How do you implement compensation logic?**
   * **A:** Implement reversal operations for each step (e.g., refund, restock).
5. **What are the challenges of Saga?**
   * **A:** Complex error handling and testing.

## 4. CQRS (Command Query Responsibility Segregation)

**Explanation:** Separates read and write models for scalability and optimization.

**Use Case:** Order commands go through service bus; queries go to read DB.

**C# Example:**

public class CreateOrderCommand : IRequest<bool> {

public string ProductId; public int Quantity;

}

public class GetOrdersQuery : IRequest<List<OrderDto>> {}

**Questions and Answers:**

1. **What are the benefits of CQRS?**
   * **A:** Optimized read/write paths, scalability, better performance.
2. **When not to use CQRS?**
   * **A:** In simple CRUD apps where complexity outweighs benefits.
3. **Can CQRS work without Event Sourcing?**
   * **A:** Yes—Event Sourcing is optional.
4. **What is eventual consistency in CQRS?**
   * **A:** Reads may lag behind writes due to async syncing.
5. **What frameworks support CQRS in .NET?**
   * **A:** MediatR, CAP, NServiceBus.

## 5. Event Sourcing Pattern

**Explanation:** Persist state as a sequence of events instead of current snapshot.

**Use Case:** Bank account records deposits/withdrawals as events.

**C# Example:**

public class AccountEvent {

public string EventType;

public decimal Amount;

public DateTime OccurredOn;

}

**Questions and Answers:**

1. **What are the benefits of event sourcing?**
   * **A:** Full audit trail, time-travel debugging, rehydration.
2. **When to avoid event sourcing?**
   * **A:** When domain is simple or you don't need audit history.
3. **What is the role of snapshots?**
   * **A:** Improve performance by storing state at intervals.
4. **What tools help with event sourcing in .NET?**
   * **A:** Marten, EventStoreDB, NEventStore.
5. **How do you rebuild state from events?**
   * **A:** Replay the event stream to reconstruct the entity.

(The rest of the patterns — including Strangler Fig, Retry, Bulkhead, Backpressure, Service Mesh, Sidecar, Transactional Outbox, Idempotency Token, Service Discovery, DDD, Choreography vs Orchestration, Event Replay, Outbox Polling, API Composition, Aggregator, BFF, Registry with Heartbeat — will follow in the next update...)

Let me know if you'd like me to proceed with appending the remaining patterns right away.

### ✅ Microservices Design Patterns: Explanation, Questions, Use Cases, Expected Answers, and C# Code Snippets

...(previous patterns 1 to 5 here)...

## 6. Strangler Fig Pattern

**Explanation:** Incrementally replace legacy systems by routing some functionality to new services.

**Use Case:** Gradually migrate a monolithic order service to microservices.

**Questions and Answers:**

1. **What is the benefit of the Strangler Fig pattern?**
   * **A:** Allows gradual migration without a big bang rewrite.
2. **How is routing handled in this pattern?**
   * **A:** Use API Gateway or reverse proxy to direct traffic.
3. **What challenges come with this pattern?**
   * **A:** Complexity in maintaining dual systems during transition.
4. **What’s a typical use case?**
   * **A:** Migrating a legacy .NET monolith to modern .NET Core microservices.
5. **How do you know when to deprecate legacy parts?**
   * **A:** When all traffic for a function routes through the new service.

## 7. Retry Pattern

**Explanation:** Automatically retries transient failures like network timeouts.

**C# Polly Example:**

Policy.Handle<TimeoutException>()

.WaitAndRetry(3, retryAttempt => TimeSpan.FromSeconds(retryAttempt));

**Questions and Answers:**

1. **When should you use retry?**
   * **A:** For transient issues like timeouts or temporary unavailability.
2. **What is the risk of retrying?**
   * **A:** It can overwhelm the service and delay failure detection.
3. **What is exponential backoff?**
   * **A:** Increasing wait time between retries to reduce load.
4. **Where is retry best implemented?**
   * **A:** On the client side, service proxy, or middleware.
5. **When should retry be avoided?**
   * **A:** For non-idempotent operations like payments.

## 8. Bulkhead Pattern

**Explanation:** Isolates resources into pools to prevent cascading failures.

**Use Case:** Separate thread pools per microservice call.

**Questions and Answers:**

1. **What problem does the bulkhead pattern solve?**
   * **A:** Prevents failure in one part of the system from affecting others.
2. **What’s a .NET tool to implement it?**
   * **A:** Polly supports isolation policies.
3. **What analogy is used for this pattern?**
   * **A:** Ship compartments preventing full flooding.
4. **When is this pattern useful?**
   * **A:** In services making multiple downstream calls.
5. **What are implementation approaches?**
   * **A:** Thread pools, queues, connection limits.

## 9. Backpressure Pattern

**Explanation:** Controls the flow of data when consumers can't keep up with producers.

**Questions and Answers:**

1. **Why is backpressure important in messaging systems?**
   * **A:** Prevents message queues from being overwhelmed.
2. **What’s an example of applying backpressure?**
   * **A:** Blocking publisher when Kafka or RabbitMQ queue is full.
3. **How does this apply in HTTP?**
   * **A:** Use 429 Too Many Requests response.
4. **What are tools for backpressure in .NET?**
   * **A:** Channels, Semaphores, reactive streams.
5. **What happens without backpressure?**
   * **A:** System crashes or memory overflow.

## 10. Service Mesh Pattern

**Explanation:** A dedicated infrastructure layer for managing service-to-service communication.

**Use Case:** Istio or Linkerd handling retries, TLS, and metrics for Kubernetes services.

**Questions and Answers:**

1. **Why use a service mesh?**
   * **A:** For observability, security, and traffic control.
2. **What does a sidecar proxy do?**
   * **A:** Intercepts service communication for monitoring and control.
3. **What’s the overhead of using a mesh?**
   * **A:** Resource usage, latency, and complexity.
4. **What are popular service mesh tools?**
   * **A:** Istio, Linkerd, Consul Connect.
5. **How is security enhanced?**
   * **A:** Mutual TLS between services.

...

(Next batch will include: Sidecar, Transactional Outbox, Idempotency Token, Service Discovery, DDD, Choreography vs Orchestration, Event Replay, Outbox Polling, API Composition, Aggregator, BFF, Service Registry)

Let me know to continue appending.

### ✅ Microservices Design Patterns: Explanation, Questions, Use Cases, Expected Answers, and C# Code Snippets

...(patterns 1 to 10 here)...

## 11. Sidecar Pattern

**Explanation:** Deploys helper components (e.g., logging, monitoring) alongside service in the same container pod.

**Use Case:** A .NET app with a sidecar for logging using Fluent Bit.

**Questions and Answers:**

1. **Why use a sidecar?**
   * **A:** To offload cross-cutting concerns like logging, monitoring, and proxying.
2. **How is it deployed?**
   * **A:** As a separate container in the same Kubernetes pod.
3. **What are popular use cases?**
   * **A:** Envoy for proxying, Fluent Bit for logs.
4. **What are advantages over shared libraries?**
   * **A:** Language-agnostic, reusable, consistent.
5. **What are the risks?**
   * **A:** Complexity, resource sharing issues.

## 12. Transactional Outbox Pattern

**Explanation:** Ensures reliable event publishing from a local database using outbox table.

**Use Case:** Order service saves order and event in same transaction.

**C# Example:**

using var tx = \_db.Database.BeginTransaction();

\_db.Orders.Add(order);

\_db.OutboxMessages.Add(orderCreatedEvent);

await \_db.SaveChangesAsync();

tx.Commit();

**Questions and Answers:**

1. **Why use outbox instead of direct event publishing?**
   * **A:** To ensure atomicity and avoid message loss.
2. **How are outbox messages published?**
   * **A:** Poller picks them and sends to message broker.
3. **What happens on poller failure?**
   * **A:** Messages remain in DB and are retried.
4. **How to ensure idempotency?**
   * **A:** Use unique message IDs and deduplication logic.
5. **What database changes are needed?**
   * **A:** Outbox table, possibly triggers or timestamp columns.

## 13. Idempotency Token Pattern

**Explanation:** Prevents duplicate processing of the same request.

**Use Case:** Ensure payment isn’t double-processed on retry.

**C# Example:**

if (\_db.RequestLog.Any(x => x.IdempotencyKey == key))

return Conflict("Duplicate");

\_db.RequestLog.Add(new RequestLog { Key = key });

**Questions and Answers:**

1. **Why is idempotency important in APIs?**
   * **A:** To prevent duplicate operations on retries.
2. **Where is the token stored?**
   * **A:** In headers or request body.
3. **What operations need idempotency?**
   * **A:** Payments, account creation, email triggers.
4. **What do you return for repeated requests?**
   * **A:** Same result or conflict response.
5. **How do you store processed tokens?**
   * **A:** In a DB or distributed cache like Redis.

## 14. Service Discovery Pattern

**Explanation:** Enables dynamic location of services without hardcoded addresses.

**Use Case:** Order service finds Shipping service using Consul or Eureka.

**Questions and Answers:**

1. **Why avoid hardcoded service addresses?**
   * **A:** To enable scalability and dynamic environments.
2. **Client-side vs server-side discovery?**
   * **A:** Client handles routing vs load balancer handles it.
3. **Tools for service discovery?**
   * **A:** Consul, Eureka, etcd.
4. **How do services register themselves?**
   * **A:** On startup via API or agent.
5. **What are fallback strategies?**
   * **A:** Static config, retry with exponential backoff.

## 15. Domain-Driven Design (DDD) Patterns

**Explanation:** Aligns microservices with business domains using aggregates, entities, and bounded contexts.

**Use Case:** Patient, Billing, and Appointment services each map to their bounded contexts.

**C# Aggregate Root Example:**

public class Order : AggregateRoot {

public void AddItem(Product product, int quantity) {

// business rule, raise event

}

}

**Questions and Answers:**

1. **What is a bounded context?**
   * **A:** A logical boundary with its own domain model.
2. **What is an aggregate root?**
   * **A:** Main entity that enforces business rules for a group.
3. **When are value objects used?**
   * **A:** When identity isn’t needed (e.g., Address).
4. **What is ubiquitous language?**
   * **A:** Shared terms used by devs and business.
5. **How does DDD help microservice design?**
   * **A:** Defines service boundaries clearly.

## 16. Choreography vs Orchestration

**Explanation:** Choreography = event-driven, decentralized. Orchestration = controlled by a central service.

**Use Case:** OrderPlaced event → Inventory & Payment react (choreography). Orchestrator calls steps in sequence.

**Questions and Answers:**

1. **Which is more loosely coupled?**
   * **A:** Choreography.
2. **When to use orchestration?**
   * **A:** When needing rollback, monitoring.
3. **What tools support orchestration?**
   * **A:** Dapr Workflow, Temporal, Camunda.
4. **Drawbacks of choreography?**
   * **A:** Harder to trace and debug.
5. **Can they coexist?**
   * **A:** Yes. Mix for optimal design.

## 17. Event Replay & Snapshotting Pattern

**Explanation:** In event sourcing, replaying all events builds state; snapshots improve performance.

**Questions and Answers:**

1. **Why use snapshotting?**
   * **A:** Avoid long replay times.
2. **When should you take a snapshot?**
   * **A:** After N events or time-based.
3. **How are snapshots stored?**
   * **A:** As JSON or binary alongside events.
4. **Are snapshots final state?**
   * **A:** No—they are starting points for replay.
5. **What happens if snapshot is lost?**
   * **A:** Replay from beginning or previous snapshot.

## 18. Outbox Polling Publisher Pattern

**Explanation:** Polls DB table for new events to publish to broker.

**Questions and Answers:**

1. **Why is this reliable?**
   * **A:** Events are persisted before publishing.
2. **What triggers the poller?**
   * **A:** Timer, cron job, or hosted service.
3. **What ensures no duplicates?**
   * **A:** Message IDs and published flag.
4. **Where should the poller run?**
   * **A:** Same service or background worker.
5. **How often should polling happen?**
   * **A:** Every few seconds or based on SLA.

## 19. API Composition Pattern

**Explanation:** Aggregates multiple service calls into one response.

**C# Example:**

var user = await userService.GetUser();

var orders = await orderService.GetOrders(user.Id);

return new UserDashboard(user, orders);

**Questions and Answers:**

1. **Why use API composition?**
   * **A:** Avoids client making multiple calls.
2. **Where is it implemented?**
   * **A:** API Gateway or composition layer.
3. **How do you handle failure?**
   * **A:** Return partial results, retries, fallback.
4. **What are performance concerns?**
   * **A:** Parallelism and caching are key.
5. **How to handle pagination?**
   * **A:** Each service paginates independently.

## 20. Aggregator Pattern

**Explanation:** Like API composition but with transformation/orchestration.

**Questions and Answers:**

1. **When is it better than composition?**
   * **A:** When business logic must combine results.
2. **Where should this logic reside?**
   * **A:** In a service/facade layer.
3. **Can this degrade performance?**
   * **A:** Yes, if not parallelized or optimized.
4. **How is data modeled?**
   * **A:** With DTOs combining multiple service outputs.
5. **Is it reusable across frontends?**
   * **A:** Yes, makes frontend simpler.

## 21. Backend for Frontend (BFF)

**Explanation:** API customized for specific frontend (e.g., mobile, web).

**Questions and Answers:**

1. **Why use BFF?**
   * **A:** Tailors API to frontend needs.
2. **Where is BFF hosted?**
   * **A:** As standalone or along with frontend.
3. **Can one BFF serve multiple clients?**
   * **A:** Usually one per UI, but may reuse logic.
4. **How is it secured?**
   * **A:** Use frontend-specific tokens.
5. **What tech is used for BFF in .NET?**
   * **A:** ASP.NET Core Minimal APIs, GraphQL.

## 22. Service Registry with Heartbeat

**Explanation:** Dynamic registry of service instances with health monitoring.

**Questions and Answers:**

1. **What is heartbeat in this context?**
   * **A:** Periodic ping/HTTP call to check health.
2. **Why is this pattern needed?**
   * **A:** For load balancers to detect unhealthy services.
3. **What tools implement it?**
   * **A:** Consul, Eureka.
4. **How to integrate in .NET?**
   * **A:** Register via HTTP, expose /health endpoints.
5. **What if a node fails silently?**
   * **A:** Heartbeat timeout removes it from registry.

🎯 This concludes all 22 core microservices patterns with explanation, use cases, questions, answers, and code. Let me know if you'd like a summary table or export to PDF/Word!