Java Technologies

Lecture 8

Microservice Architectures

Fall, 2025

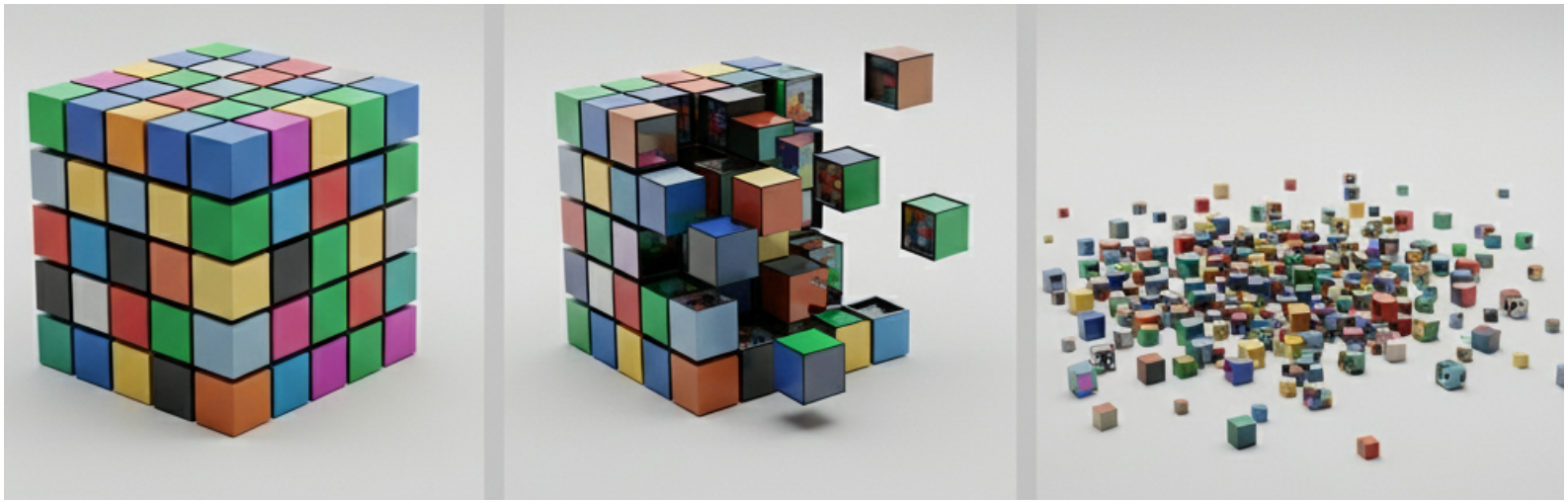
Agenda

- From Monolith to Microservices
- Characteristics of Microservices, Real-World Examples
- Eclipse MicroProfile vs. Spring Boot
- Challenges in Microservice Architectures
- Domain-Driven-Design (DDD), Decomposition of the Monolith
- Communication in a Microservices Architecture
- Fallacies of Distributed Computing
- REST Clients, Sync vs. Async, Reactive Programming
- Resilience, Fault Tolerance Patterns, Resilience4j
- Timeout, Retry, Fallback, Circuit Breaker, Bulkhead, RateLimiter
- Health Checks, Spring Actuator, Liveness, Readiness

The Context

- **The Monolith**
  - Big application, requiring a big server.
  - Contains all the components into a single, indivisible unit.
  - Uses the same programming language and technological stack.
  - The codebase is a giant, tangled web of dependencies.
  - No developer could possibly know the entire application.
  - Deployed as a single big (huge) package.
  - Changing a component $\rightarrow$ redeployment of the whole thing.
  - A bug or a memory leak in one module crashes the entire app.
  - Scalability is obtained using expensive server clustering ($$$).
  - ...
- How to create a more flexible architecture? 
- On the other hand:
  - Most applications in the world are still monoliths.
  - They are cheaper, easier, and faster to build.

From Monolith to Microservices

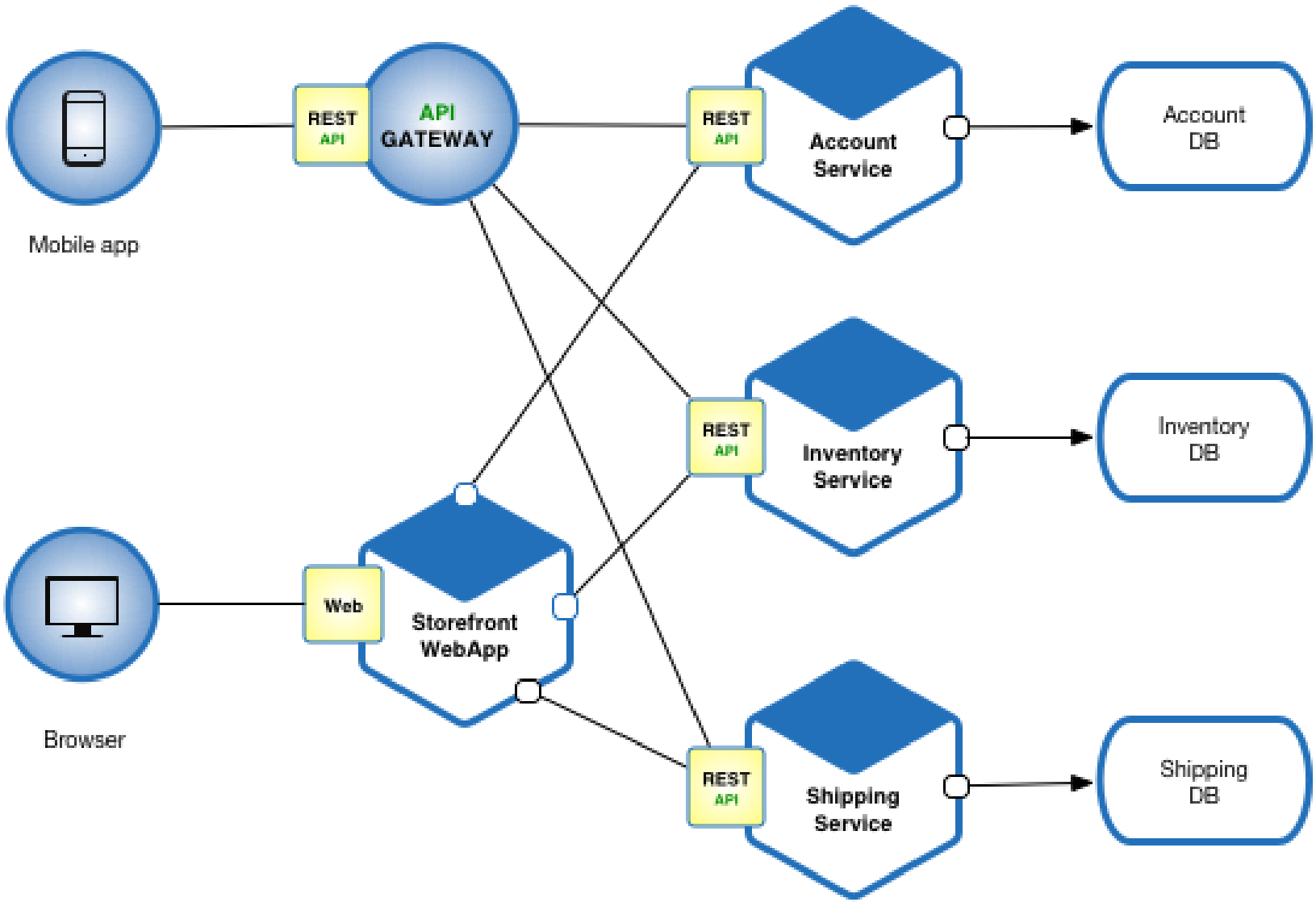


- **First**, there was the **monolith**. 😞
  A single large ship — if one part fails, the whole ship sinks.
- **Then**, there was **Service-Oriented Architecture (SOA)**
- **Now**, the era of **microservices**.
  A fleet of small ships — failure of one doesn't sink the others.

Characteristics of Microservices

- **Architectural Style**
- **Autonomy / Independence**
  - Each service runs independently, with its own lifecycle.
  - Teams can develop, deploy, and scale services separately.
- **Bounded Context (from Domain-Driven Design)**
  - Each service owns a distinct business domain.
  - Reduces coupling and improves maintainability.
- **Auto-Scalability**
  - Services can be scaled individually based on load.
- **Independent Deployability**
  - Deploy one service without redeploying the entire system.
  - Reduces downtime and accelerates delivery cycles.
- **Trade-off**: Flexibility and fine-grained scalability vs. Communication overhead and increased system complexity.

How It Should Look



Real-World Examples

- **Netflix**
  - Over 500 microservices.
  - Handles huge traffic from more than 250 million subscribers, using tools like Eureka (service discovery), Ribbon (load-balancing), and Hystrix (fault tolerance).
- **Amazon**
  - Introduced "two-pizza team" to handle a microservice.
  - "You build it, you run it" $\rightarrow$ Amazon Web Services (AWS).
  - However, Amazon Prime Video switched to a **hybrid monolith**.
- **Uber**: Decomposed its monolith into cloud-based microservices aligned with business domains.
- **Spotify**: Adopted a microservices architecture combined with a strong culture of autonomous, full-stack teams called "Squads".
- **Success Stories** vs. **Horror Stories** 💀

How to Create a Microservice?

"Just make some REST controllers and call them microservices." 😄

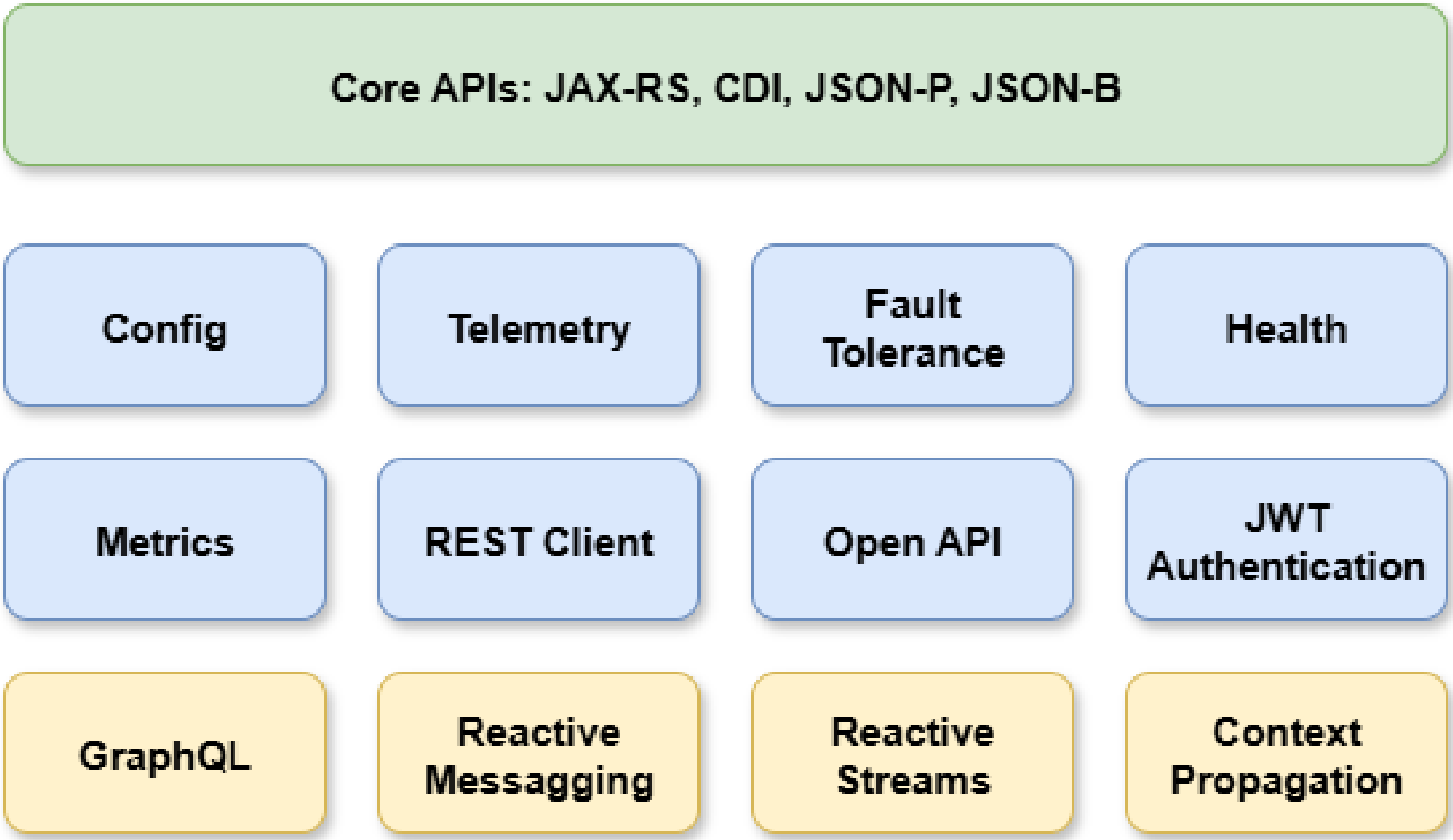
1. **Start with the design**: Define the service boundary.
   → Identify the business capability it should cover.
   → Single Responsability Pinciple
2. **Choose the tech stack**: That's easy: Java + Spring Boot
3. **Choose your database**: Each microservice has its own DB.
4. **Implement the API**: Expose the public endpoints.
5. **Make it resilient and observable**:
   - Implement retries and fallbacks.
   - Add logging and metrics.
6. **Externalize configuration**
7. **Package & Deploy**

Are there some standard specifications regarding microservices?

Eclipse MicroProfile

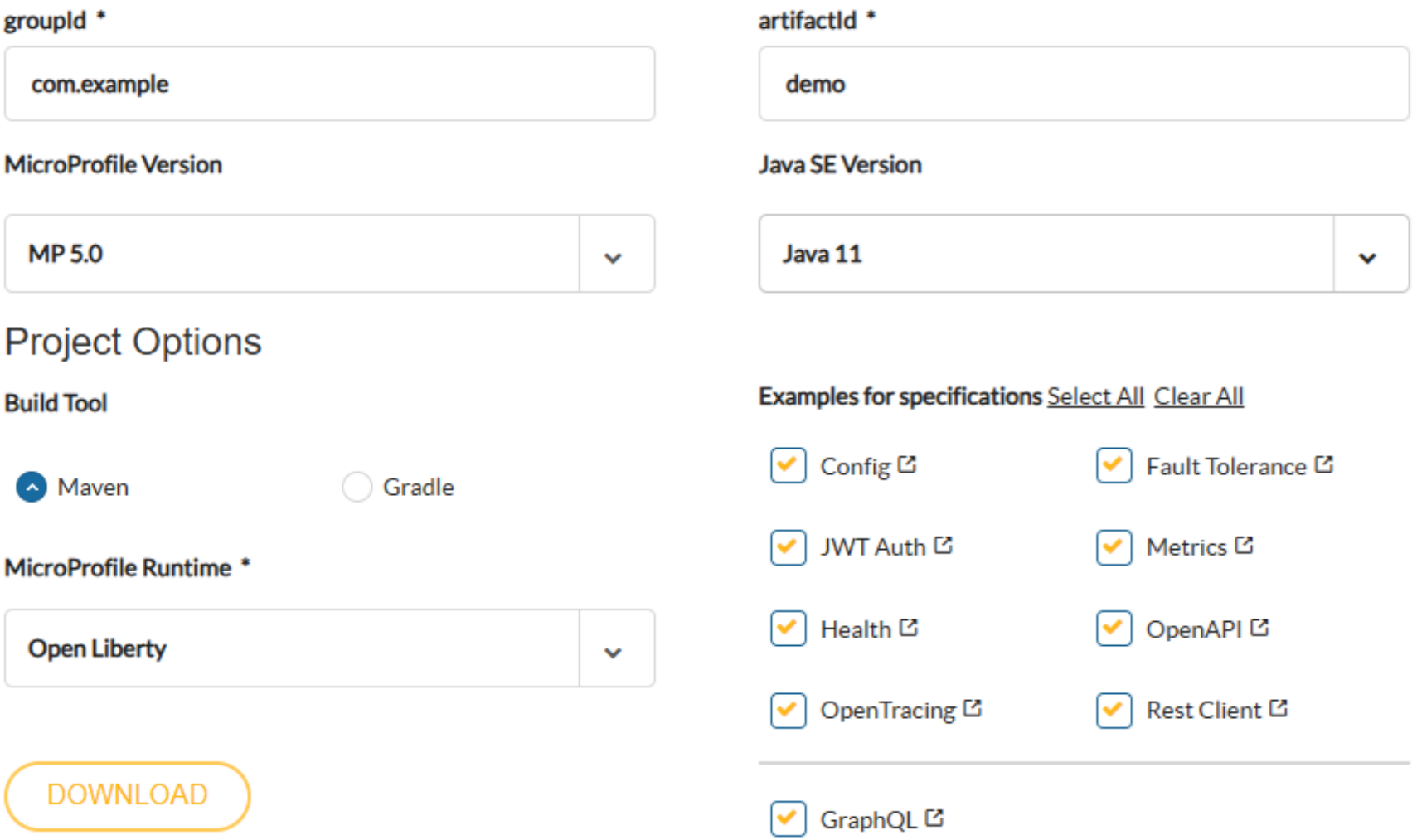
- Open-source project under the Eclipse Foundation that defines a set of APIs and specifications for building cloud-native, microservice-based Java applications.
- Based on a **subset of Java/Jakarta EE**.
- **Goals**
  - **Lightweight**: Keep runtimes small and efficient.
  - **Portable**: Apps run across multiple compatible runtimes.
  - **Cloud-native**: Built-in support for observability, health, metrics, and fault tolerance.
  - **Vendor-neutral**: Backed by multiple vendors and open governance. $\rightarrow$ Avoiding vendor lock-in.
- **Implementations**: Open Liberty (IBM), Payara Micro, WildFly, Quarkus  , Helidon (Oracle), TomEE.
  $\rightarrow$ Micro Servers

MicroProfile APIs



MicroProfile Starter

`https://start.microprofile.io`



Eclipse MicroProfile vs. Spring Boot

- **Eclipse MicroProfile**
  - Standardization and portability across multiple runtimes.
  - Specification-driven ecosystem.
  - Multi-vendor collaboration (IBM, Oracle, Red Hat, Payara).
  - Can use any compatible runtime (micro servers).
  - Standards, portability, cloud-native APIs,
    but smaller ecosystem.
- **Spring Boot**
  - Convention over configuration, batteries-included.
  - Huge ecosystem, faster developer productivity.
  - Backed by a single vendor and a large open-source community.
  - Embedded server (Tomcat/Jetty/Netty).
  - Productivity, ecosystem, opinionated,
    but vendor lock-in.

Challenges in Microservices Architecture

- Complexity of Distributed Systems
- Infrastructure (Chassis)
- Decomposition of the Monolith
- Communication and Discovery
- Observability and Reliability
- Data Management and Consistency
- Security
- Integration Testing, Logging, Tracing
- Configuration, Deployment and Management
- Performance

Domain-Driven Design (DDD)

- Introduced by Eric Evans in his 2003 book *Domain-Driven Design: Tackling Complexity in the Heart of Software*.
- It focuses on building software that closely reflects the **business domain (the problem space)** by creating a shared understanding between developers and domain experts.
- Instead of starting with technology or databases:
  $\rightarrow$ "Start with the business domain and model it in code."
- **Key Concepts**
  - **Domain**: The business problem area your software addresses.
  - **Ubiquitous Language**: A language for devs and experts.
  - **Bounded Context**: A boundary where a particular model applies $\rightarrow$ divides the system in small, manageable parts.
  - Entities, Value Objects, Aggregates, Repositories, Domain Events. Domain Services

Decomposition of the Monolith

- **Strangler Fig Pattern**: Gradually build a new system around the edges of the old monolith. Don't do a "Big Bang Rewrite".
- **Phase 1**: Analysis and Planning
  - Implement a **modular monolith** first: SoC
  - Define the **bounded contexts**: DDD is the key.
  - Identify the **scalability needs**: This is why we got here.
- **Phase 2**: The Decomposition Playbook
  - By **business capability**: *"what it does?"*, coarse grained
  - By **subdomain**: *"how to do it?"*, fine grained
    $\rightarrow$ Too fine-grained = distributed mess.
  - Decompose the **database** ⚠
- **Phase 3**: Operational Excellence
  - Invest in **automation and observability**: CI/CD, Monitoring
  - Repeat and Iterate

Start Making Microservices

- **High Cohesion**: A service should implement a small set of strongly related functions $\rightarrow$ it should do **one thing well**.
- **Common Closure Principle**: Things that change together should be packaged together.
- **Loose Coupling**: Each service encapsulates its implementation.
- **Small Enough**: "Two pizza" team, 5-10 people.
- **Autonomous Teams**: Minimal collaboration with other teams.
- **Free Choices**: Technology stack, database.
- **Testable**: It doesn't need the whole system.
- **Uniform Interface**: Should integrate in the grand scenario.

Communication in a Microservices Architecture

- A microservices-based application is a distributed system running on multiple processes or services, usually even across multiple servers or hosts.
- Each service instance is typically a process. Therefore, **services must interact using an interprocess communication protocol**, text-based or binary, such as HTTP, STOMP, AMQP, or simply TCP depending on the nature of each service.
- **Smart Endpoints & Dumb Pipes**
  - The microservices are "smart", they contain the business logic and decision-making: validation, routing, orchestration.
  - The communication infrastructure is lightweight and simple. Pipes just move data/messages.

Fallacies of Distributed Computing

Originally formulated by L. Peter Deutsch and other colleagues at Sun Microsystems (such as James Gosling) in the 1990s.

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

Communication Types in Microservices

- **Synchronous Communication**
  - HTTP/REST: Widely used, simple, human-readable, but can be slower due to text-based nature.
  - gRPC: High-performance, binary protocol, supports streaming, suitable for internal microservice communication.
  - Tight coupling, Cascading failures
- **Asynchronous Communication**
  - Message Queues: RabbitMQ, ActiveMQ. Ensures reliable delivery and load leveling.
  - Event Streaming: Kafka, Pulsar. Good for real-time data pipelines and event-driven architectures.
  - Increased complexity, Eventual Consistency
- **Hybrid Approaches**: Combines sync and async patterns based on use-case requirements.

REST Clients

- Two REST controller belonging **to the same a monolithic** application have no reason to invoke one another.
  $\rightarrow$ The logic should be in a service.
- If each controller is **in its own microservice**, they must communicate over the network using HTTP.
  - User Service: Provides user info at `/users/id`.
  - Order Service: Needs user info to create an order.
- The controller which makes the invocation is a **REST Client**.
- Spring offers several solutions for creating a REST Client
  - `RestTemplate`: The legacy, synchronous, blocking client.
  - `FeignClient`: The declarative client.
  - `WebClient`: The modern, non-blocking & reactive client.

Using RestTemplate

```
@RestController
@RequestMapping("/orders")
public class OrderController {

    private final RestTemplate restTemplate;

    public OrderController(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    } // Constructor based DI

    @PostMapping("/{userId}")
    public String createOrder(@PathVariable Long userId) {

        String userServiceUrl =
            "http://localhost:8081/users/" + userId;

        User user = restTemplate.getForObject( // blocking
            userServiceUrl, User.class);
        return "Order created for user: " + user.name();
    }
}
```

Using `FeignClient` from OpenFeign

- Add the **Spring Cloud** OpenFeign dependency.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

- Enable Feign clients.

```
@SpringBootApplication
@EnableFeignClients
```

- Create a Feign client interface.

```
@FeignClient(name="user-service", url="http://localhost:8081")
public interface UserClient {
    @GetMapping("/users/{id}")
    User getUser(@PathVariable("id") Long id);
} // more elegant, less boiler blate, integrates with other APIs
```

- Use the `UserClient` via dependency injection.

Using `WebClient` from WebFlux

- Add `spring-boot-starter-webflux`
- Configure the WebClient.

```
@Configuration
public class WebClientConfig {
  @Bean
  public WebClient webClient(WebClient.Builder builder) {
    return builder.baseUrl("http://localhost:8081").build();
  } // User Service runs at this URL
} // WebClient can be used in both MVC and WebFlux applications
```

- Use it in the `OrderController`.

```
  @PostMapping("/{userId}")
  public Mono<String> createOrder(@PathVariable Long userId) {
    return webClient.get() // non-blocking
                .uri("/users/{id}", userId)
                .retrieve()
                .bodyToMono(User.class)
                .map(user -> "Order created: " + user.name());
    } // What is a Mono? 😵
```

Reactive Programming

- Reactive programming is about handling asynchronous data streams in a non-blocking way.
- Instead of waiting (blocking) for data (like in RestTemplate), you **react to data as it arrives** $\rightarrow$ it scales better.
- Spring supports reactive programming via **Project Reactor**.
- `Mono` and `Flux` are **reactive types**. Both are **publishers**, they produce data asynchronously.

```
@GetMapping("/user/{id}")
public Mono<User> getUser(@PathVariable Long id) {
    return userService.findById(id); // returns Mono<User>
}
```

```
@GetMapping("/users")
public Flux<User> getAllUsers() {
    return userService.findAll();    // returns Flux<User>
}
```

Resilience

- We are in the context of a distributed system with **a large number of communicating nodes (microservices)**.
  - How to deal with unexpected failures?
  - How do we know if the services function properly?
  - How do we measure the performance of the services?
  - If something went wrong, how to trace the source of error?
- **Resilience**: The ability of a system to adapt, recover, and maintain functionality under stress or unexpected conditions.
  - **Fault Tolerance**: The ability of a system to continue functioning correctly even when some components fail.
  - **Health Checks**: The measurement of a service's internal state and its ability to function.
  - **Self-Healing**: The ability to automatically detect failures and take corrective actions (restart, "respawn").

Fault Tolerance Patterns

- **Retry**: Attempts execution again if it fails.
  → Recovers from transient errors.
- **Fallback**: Provides an alternative when the service fails.
  → Returns a cached data or a default response.
- **Timeout**: Fails fast if a service call takes too long.
  → Prevents threads from being blocked indefinitely.
- **Circuit Breaker**: Stops calling a failing service temporarily, when execution repeatedly fails.
  → Prevents services from overloading and cascading failures.
- **Bulkhead**: Limits concurrent calls to isolate failures.
  → Ensures failure in one part doesn't affect others.
- **Rate Limiter**: Controls request rate to prevent overloading.
  → Ensures fair usage of services.

Resilience4j Library

- **Resilience4j** is a lightweight fault tolerance library.

```
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
<!-- You also need spring-boot-starter-aop -->
```

- It provides specific starters for seamless integration with Spring Boot's annotation-based model and AOP. It is also designed to work well with reactive frameworks like Project Reactor.
- Similar to **Eclipse MicroProfile Fault Tolerance API**.

Timeout, Retry, Fallback

- Assuming a client sends a request to a service, **what happens if the service doesn't respond**?
  - Wait indefinitely... or
  - Leverage some sort of **timeout value**.
- But, **what happened on the service side**?
  - It never got the request.
  - It got the request, processed it, but the response didn't get to the client. The service assumes everything is ok.
  - The service crashed.
- What to do next?
  - **Send again the request**, if it's safe, i.e. idempotent service. ⚠
    - Duplicate side effects, order created twice.
    - Data inconsistency, stock decremented multiple times.
  - **Fallback** to a cached value, if that's acceptable.
  - Don't worry, ignore it, everything will be fine...

Retry & Fallback

- Annotate methods that may experience transient errors (for example, aceessing a resource being over commited).

```
@Retry(name = "paymentService",
        fallbackMethod = "fallbackPayment")
public String processPayment(String orderId) {
    if (Math.random() < 0.5) {
        throw new IOException("Temporary failure");
    }
    // call external service
}
public String fallbackPayment(String orderId, Throwable t) {
    return "Payment failed, please try later";
}
```

- All patterns are configurable in application.yml.

```
resilience4j.retry:
  instances:
    paymentService:
      max-attempts: 3    # total attempts (original + 2 retries)
      wait-duration: 2s  # wait between retries
```

TimeLimiter

- It works only with **async calls** (CompletableFuture).

```
@TimeLimiter(name = "inventoryService")
public CompletableFuture<String> getStock(String productId) {
  return CompletableFuture.supplyAsync(() -> {
      try {
          Thread.sleep(5000); // Simulate slow external call
          return "Stock available for product " + productId;
      } catch (InterruptedException e) {
          throw new RuntimeException(e);
      }
  });
} // Can also have a fallback method, just like @Retry
```

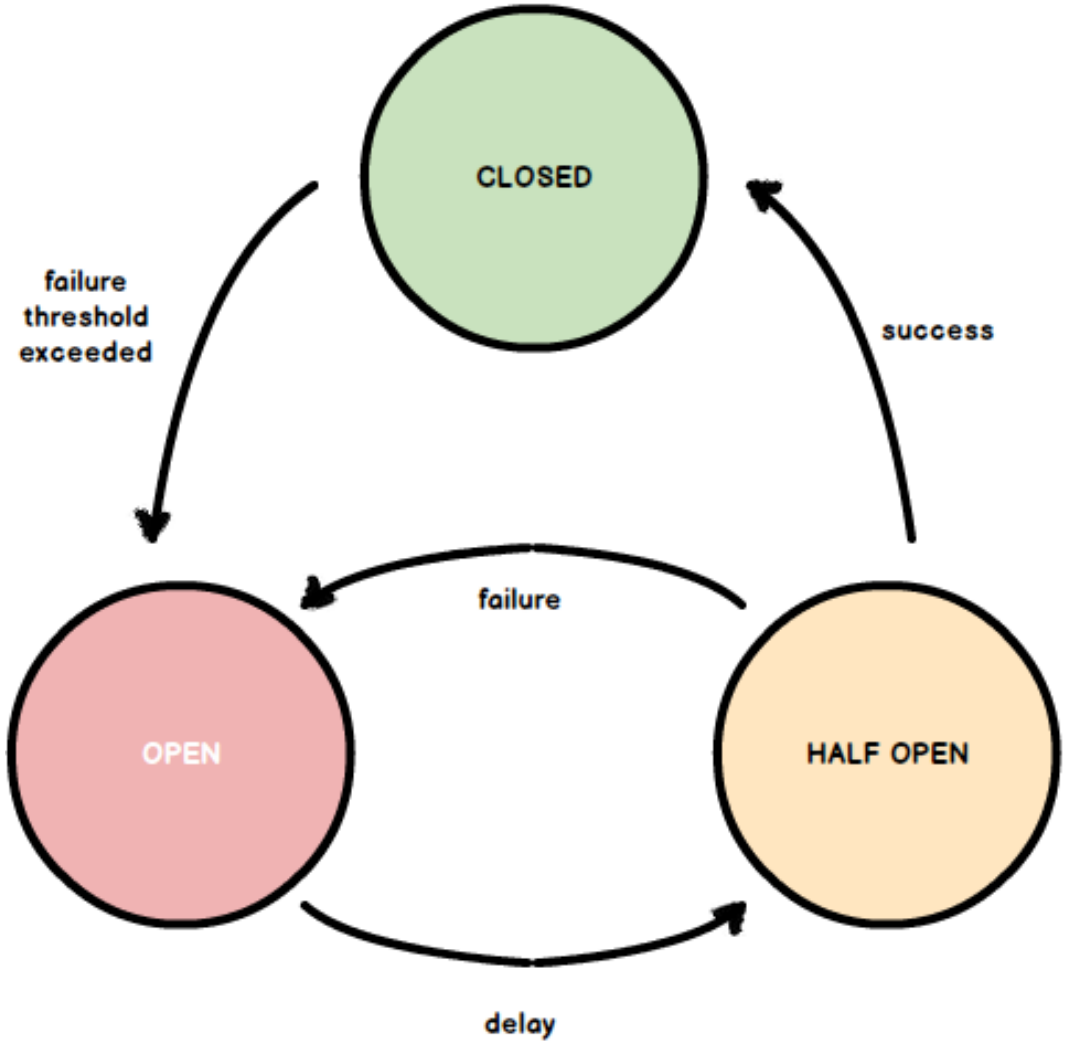
- It is also configurable in application.yml

```
resilience4j.timelimiter:
  instances:
    inventoryService:
      timeout-duration: 2s  # fail if it takes more than 2s
      cancel-running-future: true
```

The Circuit Breaker Pattern

- A circuit breaker aims to **prevent further damage** by not executing functionality that is doomed to fail.
- The circuit breaker analyzes if the number of failures exceeds a **predefined threshold** within a certain **time frame**.
- A circuit breaker can be in one of the following states:
  - Closed: In normal operation, the circuit is closed. If a failure occurs, the circuit will be opened.
  - Open: When the circuit is open, calls to the service operating under the circuit breaker will fail immediately. After a specified delay, the circuit transitions to half-open state.
  - Half-open: In half-open state, trial executions of the service are allowed. If these calls fail, the circuit will return to open state, otherwise the circuit will be closed.

Closed, Open, Half-Open

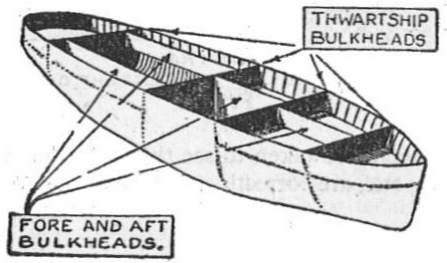


Using CircuitBreaker

```
@CircuitBreaker(name = "paymentService",
                fallbackMethod = "fallbackPayment")
public String processPayment(String orderId) {
    // Simulate failure
    if (Math.random() > 0.5) {
        throw new RuntimeException("Payment service failed!");
    }
    return "Payment processed for order " + orderId;
}
// fallback method called when breaker is open
// or failures exceed threshold
public String fallbackPayment(String orderId, Throwable ex) {
    return "Payment service unavailable for order " + orderId;
}
```

```
resilience4j.circuitbreaker:
  instances:
    paymentService:
      sliding-window-size: 5
      failure-rate-threshold: 50
      wait-duration-in-open-state: 10s
```

The Bulkhead Pattern



- Isolates **failures or high load in one part of the system**, preventing them from cascading to the entire service.
- The solution is to **limit the number of concurrent requests** accessing the service.
- Types of Bulkhead implementations:
  - **Semaphore**: Limits concurrent calls in the same thread pool.
  - **Thread Pool**: Uses a separate thread pool for calls.

Using Bulkhead

- **Semaphore**: Lightweight, sync or async.

```
@Bulkhead(name = "paymentService",
          type = Bulkhead.Type.SEMAPHORE)
public String processPayment(String orderId) { ... }
```

```
resilience4j.bulkhead:
  instances:
    paymentService:
      max-concurrent-calls: 5  # max simultaneous calls allowed
      max-wait-duration: 0     # wait 0 ms for semaphore
```

- **Thread Pool**: Heavyweight, only async (CompletableFuture).

```
resilience4j.bulkhead:
  instances:
    inventoryService:
      type: THREADPOOL
      core-thread-pool-size: 3  # min threads in the pool
      max-thread-pool-size: 5   # max threads allowed in the pool
      queue-capacity: 10        # max requests that can wait
```

RateLimiter

- Controls the **rate of requests** sent to a service.
- Prevents a service from being overloaded by bursts of requests.
- Works like a **token bucket**: only a fixed number of requests are allowed per time unit.

```
  @RateLimiter(name = "paymentService",
               fallbackMethod = "fallbackPayment")
  public String processPayment(String orderId) { ... }
```

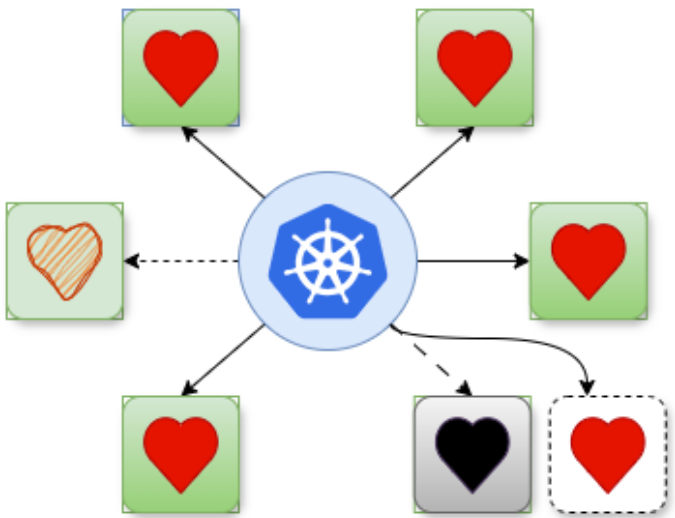
```
resilience4j.ratelimiter:
  instances:
    paymentService:
      limit-for-period: 5          # max 5 calls per refresh period
      limit-refresh-period: 1s     # refresh tokens every 1 second
      timeout-duration: 500ms      # max wait time for a token
                                   # if none available
```

Fault Tolerance Best Practices

- Apply retries only to idempotent operations.
- Combine Circuit Breaker, Retry, Timeout, and Bulkhead.
- Set realistic timeouts based on downstream SLA.
- Use fallbacks to provide graceful degradation.
- Monitor metrics and alerts for failures, timeouts, and rejections.
- Limit concurrent calls using Bulkhead to prevent resource exhaustion.
- Configure RateLimiter to prevent overwhelming services.
- Avoid over-retrying to prevent cascading failures.
- Test failure scenarios in staging to validate fault tolerance.
- Tune parameters (sliding window, thresholds, queue sizes) according to traffic patterns.

Health Checks

- In a microservices architecture, you often have **many services running and communication** with each other.
- If one instance crashes, it must be automatically **discarded and replaced** with another one by a service manager.
- To keep the system reliable and resilient, you need to monitor:
  - **Liveness**: Is the service process running?
  - **Readiness**: Is the service ready to serve traffic?
  - **Startup**: Has the service finished starting up properly?



Spring Boot Actuator

- Provides production-ready features to help **monitor and manage** an application. It exposes a number of built-in endpoints, accessible over HTTP or JMX.
  - *Is my app/service up and running?*
  - *How healthy are dependencies like DB, Redis, or Kafka?*
  - *What's happening inside (metrics, environment, logs)?*
  - *Can I expose info to monitoring tools?*
- The starter: `spring-boot-starter-actuator`
- By default, the **health endpoint** is enabled and available at:

```
http://localhost:8080/actuator/health
```

- The simplest response could be something like:

```
{
  "status": "UP"
}
```

Enabling the Health Endpoints

- Configuration in application.yml

```
management:
  endpoints:
    web:
      exposure:
        include: "health" # or "*" to expose all
  endpoint:
    health:
      probes:
        enabled: true     # enables liveness, readiness
      show-details: always
```

- The following endpoints are now enabled and available to external service orchestrators and managers (Kubernetes):

```
Liveness  → /actuator/health/liveness
            It reflects whether ApplicationContext is responsive
Readiness → /actuator/health/readiness
            Checks any auto-configured health indicators 💡
```

Creating a Custom Health Indicator

We'll create our own **startup health indicator** that flips to UP once `ApplicationContext` is ready and our own initialization is done.

```
@Component("init") // ← The name of our new health indicator
public class StartupHealthIndicator implements HealthIndicator {

  private final AtomicBoolean ready = new AtomicBoolean(false); //?

  @EventListener(ApplicationReadyEvent.class)
  public void onApplicationReady() {
    // Check other custom initialization conditions here
    ready.set(true);
  }
  @Override
  public Health health() {
    return ready.get()
      ? Health.up().withDetail("init", "Completed").build()
      : Health.outOfService()
              .withDetail("init", "Still initializing")
              .build();
    } // Could use also .down(), .unknown() or .status(your custom)
} // You may also extend AbstractHealthIndicator
```

Register a Health Group

- Register **a new health group** with the name `startup`.

```
management:
  endpoint:
    health:
      show-details: always
      group:
        startup:          # The group name defines the endpoint.
          include: init   # The health indicator(s) to check.
```

- We have **a new endpoint** available:

```
/actuator/health/startup
```

- A response could be `UP` or `OUT_OF_SERVICE`:

```
{
  "status": "UP",
  "components": {
    "init": {"status": "UP", "details": { "init": "Completed" }}
  }
}
```

Liveness & Readiness

```java
@Component
@Liveness
class MyLivenessProbe implements HealthIndicator {
  @Override
  public Health health() {
    return Health.up().withDetail("liveness", "Alive!").build();
  }
} // Adds an additional indicator to the "liveness" group
```

```java
@Component
@Readiness
class MyReadinessProbe implements HealthIndicator {

  @Override
  public Health health() {
    boolean databaseUp = true; // Check if DB is ready
    if (databaseUp) {
      return Health.up().withDetail("readiness", "Ready").build();
    }
    return Health.down().withDetail("readiness", "Not ready").build();
  }
} // Adds an additional indicator to the "readiness" group
```