WEB DESIGN AND PROGRAMMING

ENTERPRISE JAVA FRAMEWORK
3

FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

ASST. PROF. DR. SAMET KAYA

skaya@fsm.edu.tr

kysamet@gmail.com

What is an Enterprise Java Framework?

- **Definition & Purpose**
  - An Enterprise Java Framework is a comprehensive collection of libraries, tools, and runtime components designed to facilitate the development of large-scale, complex business applications that are:
    - **Secure and scalable**
    - **Maintainable and standards-compliant**
    - **Production-ready with proven patterns**
- **Core Objective:**
  - Provide reusable, battle-tested solutions for common enterprise requirements including data access, transactions, security, messaging, integration, configuration, observability, resilience, testing, and deployment.

Why Do Enterprise Frameworks Exist?

- **Key Motivations**
  - **Abstraction of Infrastructure Problems:** Frameworks handle repetitive infrastructure concerns, allowing developers to focus on business logic rather than plumbing code.
  - **Standardized Programming Models:** Enable team collaboration and long-term maintainability through consistent patterns and conventions.
  - **Cloud-Native Readiness:** Provide built-in support for containers, Kubernetes, CI/CD pipelines, and observability requirements.
  - **Reduced Time-to-Market:** Pre-built solutions for common problems accelerate development cycles and reduce bugs.

Core Framework Capabilities (Part 1)

- **Dependency Injection & IoC**
  - Manages object lifecycle and dependencies (Spring DI, CDI)
  - Promotes loose coupling and testability
- **Data Access & Transactions**
  - JPA/Hibernate for ORM
  - JTA for ACID transactions
  - Query abstractions and repository patterns
- **Web & API Layer**
  - MVC or reactive web frameworks
  - REST/JAX-RS, GraphQL, gRPC support
  - Request/response handling and validation

Core Framework Capabilities (Part 2)

- **Messaging & Integration**
  - JMS, Kafka, AMQP support
  - Enterprise Integration Patterns
  - Batch processing capabilities
- **Security**
  - Authentication and authorization
  - OAuth2/OIDC, JWT tokens
  - TLS/SSL configuration
- **Configuration Management**
  - Profile and environment management
  - Externalized configuration
  - Property injection and validation

Core Framework Capabilities (Part 3)

- **Observability**
  - Metrics collection and exposure
  - Distributed tracing
  - Structured logging
  - OpenTelemetry/Micrometer integration
- **Resilience Patterns**
  - Circuit breaker for fault tolerance
  - Retry mechanisms and timeouts
  - Bulkhead isolation (Resilience4j, MicroProfile Fault Tolerance)
- **Testing & Deployment**
  - Unit and integration testing support
  - Testcontainers for external dependencies
  - Application servers, embedded servers, fat JARs
  - Container and Kubernetes deployment

Major Enterprise Java Frameworks

- **Spring Framework / Spring Boot**
  - Defacto standard with rich "starter" ecosystem
  - Auto-configuration and convention over configuration
  - Spring Data, Security, Cloud modules
  - Extensive community and documentation
  - Suitable for both monoliths and microservices
- **Jakarta EE (formerly Java EE)**
  - Specification-based approach
  - Runs on application servers (WildFly, Payara, WebSphere Liberty)
  - Strong portability and interoperability
  - MicroProfile extensions for microservices

Emerging Enterprise Frameworks

- **Quarkus**
  - Kubernetes-native with fast startup and low memory footprint
  - GraalVM native image support
  - Hibernate Panache, RESTEasy, SmallRye with MicroProfile
- **Micronaut**
  - Compile-time (AOT) dependency injection
  - Low footprint and fast cold-start
  - Ideal for serverless and cloud functions
- **Helidon**
  - Oracle-backed framework
  - Helidon SE (minimalist) and Helidon MP (MicroProfile) options
  - Lightweight and cloud-ready

Code Reusability - The Balance 1

- **The Fundamental Trade-off**
  - Code reuse is one of the most fundamental engineering strategies that reduces overall software costs and minimizes bugs. However, it introduces risks:
- **Key Principle:**
  - The right scale matters. Simple requirements may need only a minimal library; integrated workflows requiring observability and security demand framework-level orchestration.

Code Reusability - The Balance 2

- **Benefits:**
  - Reduced development time and cost
  - Fewer bugs through proven code
  - Consistent patterns across codebase
- **Risks:**
  - Increased complexity through dependencies
  - Expanded security surface
  - Version conflicts and dependency hell

Successful Code Reuse Strategy

- **Beyond "Calling Previously Written Code"**
- Successful reuse is measured by:
  - **Well-defined boundaries:** Clear module interfaces and responsibilities
  - **Explicit contracts:** API stability and versioning guarantees
  - **Lifecycle management:** Dependency updates and compatibility
  - **Documentation:** Usage patterns and examples
- **Warning:** Without proper boundaries and contracts, time-saving initiatives can reverse into dependency chaos and version incompatibilities.
- **Guiding Principles:**
  - High cohesion within modules
  - Loose coupling between modules

Structural Techniques for Code Reuse 1

- **Method Calls**
  - Encapsulate functional logic within methods
  - Call from multiple locations within the codebase
  - Simplest form of reuse
- **Classes**
  - Combine state and functional abstractions
  - Provide modularity to related method calls
  - Enable object-oriented design patterns

Structural Techniques for Code Reuse 1

- **Interfaces**
  - Define abstract contracts as placeholders
  - Enable different implementations
  - Support dependency inversion principle
  - Facilitate testing through mocking
- **Modules**
  - Package reusable constructs into physical modules
  - Enable flexible inclusion/exclusion in applications
  - Support versioning and dependency management

Code Reuse Example - Method Level

Java Example: Method Reuse

```java
// Reusable validation method
public class ValidationUtils {
    public static boolean isValidEmail(String email) {
        String regex = "^[A-Za-z0-9+_.-]+@(.+)$";
        return email != null && email.matches(regex);
    }
}

// Usage in multiple places
public class UserService {
    public void registerUser(String email) {
        if (!ValidationUtils.isValidEmail(email)) {
            throw new IllegalArgumentException("Invalid email");
        }
        // Registration logic
    }
}

public class NotificationService {
    public void sendEmail(String email, String message) {
        if (!ValidationUtils.isValidEmail(email)) {
            return; // Skip invalid emails
        }
        // Send email logic
    }
}
```

Code Reuse Example - Interface Level


```
// Abstract interface for payment processing
public interface PaymentProcessor {
    PaymentResult process(PaymentRequest request);
    boolean supportsPaymentMethod(PaymentMethod
method);
}

// Multiple implementations
public class CreditCardProcessor implements
PaymentProcessor {
    @Override
    public PaymentResult process(PaymentRequest request) {
        // Credit card processing logic
        return new PaymentResult(true, "CC-" +
UUID.randomUUID());
    }

    @Override
    public boolean supportsPaymentMethod(PaymentMethod
method) {
        return method == PaymentMethod.CREDIT_CARD;
    }
}
```

```
public class PayPalProcessor implements PaymentProcessor
{
    @Override
    public PaymentResult process(PaymentRequest request) {
        // PayPal API integration
        return new PaymentResult(true, "PP-" +
UUID.randomUUID());
    }
    @Override
    public boolean
supportsPaymentMethod(PaymentMethod method) {
        return method == PaymentMethod.PAYPAL;
    }
}
```

Code Reuse Styles - Libraries


```java
import com.fasterxml.jackson.databind.ObjectMapper;

public class DataProcessor {
    private ObjectMapper mapper = new ObjectMapper();

    public User parseUser(String json) throws IOException {
        // YOU control when to call the library
        return mapper.readValue(json, User.class);
    }
}
```

- **Libraries: You Are in Control**
  - Libraries are modules of reusable code invoked on-demand by your codebase. Your code maintains total control of the call flow.
- **Characteristics:**
  - You decide when and how to call library functions
  - Control flow remains in your application
  - Stateless or minimal state management
  - Examples: JSON parsers (Jackson, Gson), XML parsers, utility libraries (Apache Commons)

Code Reuse Styles - Frameworks


```java
@RestController
@RequestMapping("/api/users")
public class UserController {

   @Autowired
   private UserService userService;

   // Framework calls this method when HTTP
GET /api/users arrives
   @GetMapping
   public List<User> getUsers() {
      return userService.findAll();
   }

   // Framework manages lifecycle,
dependency injection, HTTP handling
}
```

- **Frameworks: Inversion of Control (IoC)**
  - Frameworks provide control and orchestration. Your codebase operates under the framework's control - this is called Inversion of Control (IoC).
- **Characteristics:**
  - Framework calls your code at specific extension points
  - Framework manages lifecycle and flow
  - You customize behavior through configuration and callbacks
  - Examples: Spring/Spring Boot, Jakarta EE, Quarkus

Libraries vs Frameworks - The Distinction

- **Control Flow**: The Key Differentiator
- **Important Note**: The distinction is not always binary. Libraries can contain mini-frameworks for customization and extension. Even a JSON/XML parser can act as a mini-framework when it provides extension points for custom serializers or deserializers.

Aspect	Library	Framework
Control	Your code controls flow	Framework controls flow
Invocation	You call library	Framework calls you
Flexibility	High - use as needed	Constrained by framework
Complexity	Lower	Higher
Setup	Minimal	Configuration required

Spring Framework Example - IoC in Action

Dependency Injection & Lifecycle Management

```java
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        // Framework manages this bean's lifecycle
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:postgresql://localhost/mydb");
        return ds;
    }

    @Bean
    public UserRepository userRepository(DataSource
dataSource) {
        // Framework injects dataSource automatically
        return new JdbcUserRepository(dataSource);
    }
}
```

```java
@Service
public class UserService {
    private final UserRepository repository;

    // Framework performs constructor injection
    @Autowired
    public UserService(UserRepository repository) {
        this.repository = repository;
    }

    public User findById(Long id) {
        return repository.findById(id);
    }
}
```

Framework Benefits - Real-World Impact

❖ Why Choose a Framework?

- ❖ **Standardization:**
  - ❖ Consistent project structure across teams
  - ❖ Reduced onboarding time for new developers
  - ❖ Industry-standard patterns and practices
- ❖ **Productivity:**
  - ❖ Auto-configuration reduces boilerplate
  - ❖ Built-in solutions for common problems
  - ❖ Rich ecosystem of extensions and integrations
- ❖ **Enterprise Features:**
  - ❖ Transaction management
  - ❖ Security infrastructure
  - ❖ Monitoring and observability
  - ❖ Scalability and performance optimizations

❖ Example:

- ❖ Spring Boot can create a production-ready REST API with database access, security, and monitoring in under 100 lines of code.

Choosing Between Library and Framework

- ❖ **Decision Criteria**
  - ❖ **Choose a Library When:**
    - ❖ You need a specific, isolated functionality
    - ❖ You want maximum control over application flow
    - ❖ The problem domain is well-understood and simple
    - ❖ You're building a custom architecture
    - ❖ Example: Adding JSON parsing to an existing system
  - ❖ **Choose a Framework When:**
    - ❖ Building a complete application from scratch
    - ❖ You need integrated solutions (security, data, web)
    - ❖ Team needs standardization and conventions
    - ❖ Enterprise features are required (transactions, monitoring)
    - ❖ Example: Building a new microservice or web application
- ❖ **Hybrid Approach:** Many modern applications use a framework as the foundation and add specialized libraries for specific needs.

Code Reuse Best Practices

- **Principles for Effective Reuse**
  - **Single Responsibility:** Each reusable component should have one clear purpose
  - **Stable Interfaces:** Define clear contracts that don't change frequently
  - **Version Management:** Use semantic versioning and maintain backward compatibility
  - **Documentation:** Provide clear usage examples and API documentation
  - **Testing:** Reusable components must have comprehensive test coverage
  - **Dependency Hygiene:** Minimize transitive dependencies to avoid conflicts

- **Anti-Patterns to Avoid:**
  - Over-engineering simple solutions
  - Creating dependencies for trivial functionality
  - Ignoring version compatibility
  - Tight coupling between modules

Framework Definition

- **What is a Framework?**
  - A successful software framework is a body of code developed from the skeletons of past and present successful (or unsuccessful) solutions within a specific common problem domain.
- **Key Characteristics:**
  - **Generalization of solutions:** Distills proven patterns from multiple implementations
  - **Provides core abstractions:** Defines the fundamental building blocks
  - **Enables specialization:** Allows customization for specific needs
  - **Offers default behavior:** Works out-of-the-box for common cases, simplifying entry and basic solutions
- **Philosophy:** "We've done this before. This is what we need, and this is how we do it."

Framework vs Pattern

- **Beyond Pattern Instantiation**
    - A framework is far more comprehensive than a pattern instantiation:
- **Pattern Completion:**
    - Operates at the level of specific object interactions
    - Completing a pattern creates or triggers something
    - "This is not enough — we're not done yet."
    - Still a long way to complete the overall solution goal
- **Framework Completion:**
    - Encompasses multiple patterns working together
    - Achieves a significant, difficult goal
    - "I would pay for this (or charge money for it)!"
    - Delivers production-ready, valuable functionality
- **Key Distinction:** A framework orchestrates many patterns into a cohesive, purposeful solution.

Framework as Orchestration

- **More Than a Collection of Patterns**
  - When many patterns come together without orchestration, you get a "sea of calls" — like a busy city street during rush hour:
  - People stop, turn, accelerate, yield
  - Individual tasks are accomplished
  - But stepping back, little holistic meaning emerges from all these interactions
  - **"Where is everyone going?"**

- **Framework Purpose:**
  - A framework has a complex, well-defined purpose. When we harness a framework to achieve a specific goal, we accomplish something significant or difficult.
- **Community Aspect:**
  - Framework users are not alone. Others with similar goals (though different requirements) face similar challenges.
- **Value Proposition:** "This has helped many people reach their goals. All you need to do is..."

Framework Scalability

- **Working at Different Scales**
  - Well-designed and popular frameworks can operate at different scales — not a one-size-fits-all approach:
- **Different Dimensions:**
  - **Production environments:** From small deployments to large-scale distributed systems
  - **Development workbench:** Learning, demonstration, or component development for specific areas
  - **Team sizes:** Solo developers to large enterprise teams
  - **Complexity levels:** Simple CRUD apps to complex event-driven architectures
- **Metaphor:** "Does the map need to be actual size?"
- A good framework provides the right level of abstraction for each use case without forcing unnecessary complexity.

Framework Characteristics Overview

- **Four Distinguishing Characteristics**
- According to Wikipedia and industry consensus, frameworks are characterized by:
  - **Inversion of Control (IoC):** The framework calls your code, not vice versa
  - **Default Behavior:** Sensible defaults that work out-of-the-box
  - **Extensibility:** Ability to customize and specialize for your domain
  - **Non-modifiable Framework Code:** Stable, well-defined structure and abstractions
- These characteristics work together to create a powerful, reusable foundation for application development

Inversion of Control (IoC)

❖ **"Don't Call Us, We'll Call You"**

```
//Traditional Procedural Approach:

public class TraditionalApp {
   public void processOrder(HttpServletRequest
request) {
       // Your code controls the flow
       Order order= parser.parse(request);
       ValidationLibrary.validate(order);
       DatabaseLibrary.save(order);
       }
}
```

```
//Framework Approach (IoC):

@Controller
public class OrderController {

   // Framework calls this method when request arrives
   @PostMapping("/orders")
   public Order createOrder(@Valid @RequestBody Order
order) {
       // Framework has already:
       // - Parsed HTTP request
       // - Validated input
       // - Started transaction
       // - Injected dependencies
       return orderService.save(order);
   }
}
```

❖ **Key Principle:** All complex but reusable logic is abstracted into the framework. The framework orchestrates the flow and calls your code at specific extension points.

Default Behavior

- **Sensible Defaults from Experience**
  - Framework users don't need to provide everything. One or more selectable defaults attempt to do the common and correct thing.
- **Remember:** Framework developers have solved this before and have harvested core abstractions and operations from the skeletons of previous solutions.
- **What You Get by Default (Spring Boot):**
  - Embedded Tomcat server on port 8080
  - JSON serialization/deserialization
  - Error handling and exception mapping
  - Logging configuration
  - Health check endpoints
  - Metrics collection

Default Behavior: Spring Boot Example


```java
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

- **Customization When Needed:**
- properties
    - # application.properties
        - server.port=9090
        - spring.datasource.url=jdbc:postgresql://localhost/mydb

Extensibility

- **Specialization for Your Domain**
  - Framework users must be able to provide specializations specific to their problem domain to solve concrete situations.
- **Framework Developer's Role:**
  - Understand the problem domain
  - Pre-identify abstractions that need to be specialized by users
  - If identified incorrectly, it's a sign of a poor framework

Extensibility: Spring Boot Example


```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    // Framework provides security infrastructure
    // You specialize authentication logic
    @Bean
    public UserDetailsService userDetailsService() {
        return username -> {
            // Your custom user lookup logic
            User user = userRepository.findByUsername(username);
            return new org.springframework.security.core.userdetails.User(
                user.getUsername(),
                user.getPassword(),
                getAuthorities(user)
            );
        };
    }
}
```

Non-modifiable Framework Code

- **Stable Structure and Abstractions**
  - A framework has a tangible structure; well-known abstractions fulfill well-defined responsibilities.
- **Benefits:**
  - **Consistency:** This concrete aspect is visible in each concrete solution
  - **Recognizability:** Framework products are immediately understandable to other framework users
  - **Stability:** Core framework code doesn't change with each application
  - **Maintainability:** Updates to framework benefit all applications
- **Example - Spring's Core Abstractions:**
  - @Controller - Always handles web requests
  - @Service - Always contains business logic
  - @Repository - Always handles data access
  - @Component - General-purpose managed bean
- **Developer Experience:** When a Spring developer joins a new Spring project, they immediately understand the structure and organization.

Framework Enablers - Introduction

- **What Makes Frameworks Work?**
  - "Framework Enablers" are the concepts, mechanisms, and tools that make a software framework practical, flexible, and manageable.
- These enablers work together to implement Inversion of Control (IoC) and enable frameworks to automatically add common functionality as layers.

- **Core Enablers:**
  - **Dependency Injection:** Assembles and wires objects
  - **POJO (Plain Old Java Object):** Simple, framework-agnostic objects
  - **Component:** Fully assembled, ready-to-use code units
  - **Bean:** Managed objects with lifecycle
  - **Container:** Assembler and manager of components
  - **Interposition:** Adding layers of functionality transparently

Dependency Injection (DI)

- Enabling Inversion of Control

  - Dependency Injection is a process that enables IoC: objects define their dependencies, and a manager ("Container") assembles and wires objects according to these definitions.

```java
//Without DI (Manual Wiring):

public class OrderService {
    private OrderRepository repository;
    private EmailService emailService;
    private PaymentProcessor paymentProcessor;

    public OrderService() {
        // Tight coupling - hard to test, hard to change
        this.repository = new JdbcOrderRepository();
        this.emailService = new SmtpEmailService();
        this.paymentProcessor = new
StripePaymentProcessor();
    }
}
```

```java
//With DI (Container Wiring):

@Service
public class OrderService {
    private final OrderRepository repository;
    private final EmailService emailService;
    private final PaymentProcessor paymentProcessor;

    // Container injects dependencies
    @Autowired
    public OrderService(OrderRepository repository,
                        EmailService emailService,
                        PaymentProcessor paymentProcessor) {
        this.repository = repository;
        this.emailService = emailService;
        this.paymentProcessor = paymentProcessor;
    }
}
```

POJO (Plain Old Java Object)


```java
// This is a POJO - no framework dependencies
public class Order {
    private Long id;
    private String customerName;
    private BigDecimal totalAmount;
    private LocalDateTime orderDate;

    // Assumes inputs are valid
    // Doesn't know validation rules
    // Doesn't know how to persist itself
    // Doesn't know business rules

    // Getters and setters
    public void setTotalAmount(BigDecimal amount) {
        this.totalAmount = amount;
    }
}
```

- **Simple Objects, Powerful Concept**
  - A POJO is exactly what the name says: an ordinary Java class instantiated without dependency on anything special.
- **Characteristics:**
  - Addresses the object's primary purpose
  - May lack some details or dependencies for full functionality
  - Missing pieces are typically for specialization and extensibility outside the object's main purpose
- Key Point: A POJO may assume inputs are valid but doesn't know the validation rules. The framework or container adds these concerns.

Component

- **Fully Assembled and Ready**
  - A component is a fully assembled set of code (one or more POJOs) that can perform its duties for clients.
- **Characteristics:**
  - Well-defined interface
  - Well-defined set of functions it can perform
  - Zero or more dependencies on other components
  - Once client code gains access, no further mandatory assembly required

Component Example


```
// POJO
public class UserRepositoryImpl implements
UserRepository {
    private DataSource dataSource;

    public UserRepositoryImpl(DataSource
dataSource) {
        this.dataSource = dataSource;
    }

    @Override
    public User findById(Long id) {
        // Implementation using dataSource
    }
}
```

```
// Component - fully assembled by container
@Repository
public class UserRepositoryComponent implements
UserRepository {

    @Autowired
    private DataSource dataSource; // Injected by
container

    @Autowired
    private CacheManager cacheManager; // Injected
by container

    @Override
    @Cacheable("users")
    public User findById(Long id) {
        // Fully functional with all dependencies
        // Client just calls this method
    }
}
```

Bean

- **Managed Objects with Lifecycle**
  - "Bean" is a generalized term between a POJO and a component, referring to an object that encapsulates something. A supplied "bean" handles some aspects on our behalf that we don't need to know about.
- **Spring Definition:**
  - In Spring, objects that form the backbone of your application and are managed by the Spring IoC container are called "beans."
- **Bean Lifecycle:**
  - Instantiated by Spring IoC container
  - Assembled with dependencies
  - Managed throughout its lifecycle
  - Destroyed when no longer needed

Bean Example

```java
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        // Container manages this bean's lifecycle
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:postgresql://localhost/mydb");
        ds.setUsername("user");
        ds.setPassword("pass");
        return ds;
    }

    @Bean
    public UserRepository userRepository(DataSource dataSource) {
        // Container injects dataSource bean
        return new UserRepositoryImpl(dataSource);
    }
}
```

Container

- **The Assembler and Manager**
  - A container assembles and manages components. It's responsible for instantiating, configuring, and assembling beans.
- **Spring Container:**
  - Assembles and packages software to run within a JVM
  - Instantiates, configures, and assembles beans
  - Reads configuration metadata (XML, Java annotations, or Java code)
  - Manages object lifecycle and dependencies

Spring Container Example


```java
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        // Creates and starts Spring container
        ApplicationContext context = SpringApplication.run(Application.class, args);

        // Container has assembled all beans
        OrderService service = context.getBean(OrderService.class);
    }
}
```

Interposition - The Magic Layer 1

- **Adding Functionality Transparently**
  - Framework/Spring containers do more than just configure and assemble simple POJOs. Containers can apply layers of functionality on top of beans when wrapping them into components.
- **What is Interposition?**
  - The container "interposes" itself into your method call, automatically performing additional operations.

Interposition - The Magic Layer 2

- **How It Works:**
  - Container wraps beans with a **proxy**
  - When calls go through this proxy, additional layers activate
  - If call goes through a container-managed bean reference → interposition works
  - If you call your own method within the same class (self-invocation) or create objects with new → call doesn't go through proxy, interposition doesn't work

- **Common Interposition Examples:**
  - Perform validation
  - Enforce security constraints
  - Manage transactions for backend resources
  - Execute a method in a separate thread
  - Log method execution
  - Cache results

POJO Calls - No Interposition

- When Container is Not Involved
- Two situations are "plain POJO calls" with no interposition:
  - Self-Invocation (Same Class Method Calls)
  - Creating Objects with **new**

POJO Calls - No Interposition -> Self-Invocation


```
@Service
public class OrderService {

    @Transactional
    public void processOrder(Order order) {
        // This works - called from outside
        validateOrder(order);
    }

    @Transactional
    public void validateOrder(Order order) {
        // Transaction annotation ignored when called from processOrder
        // Call goes directly, no proxy/container
    }
}
```

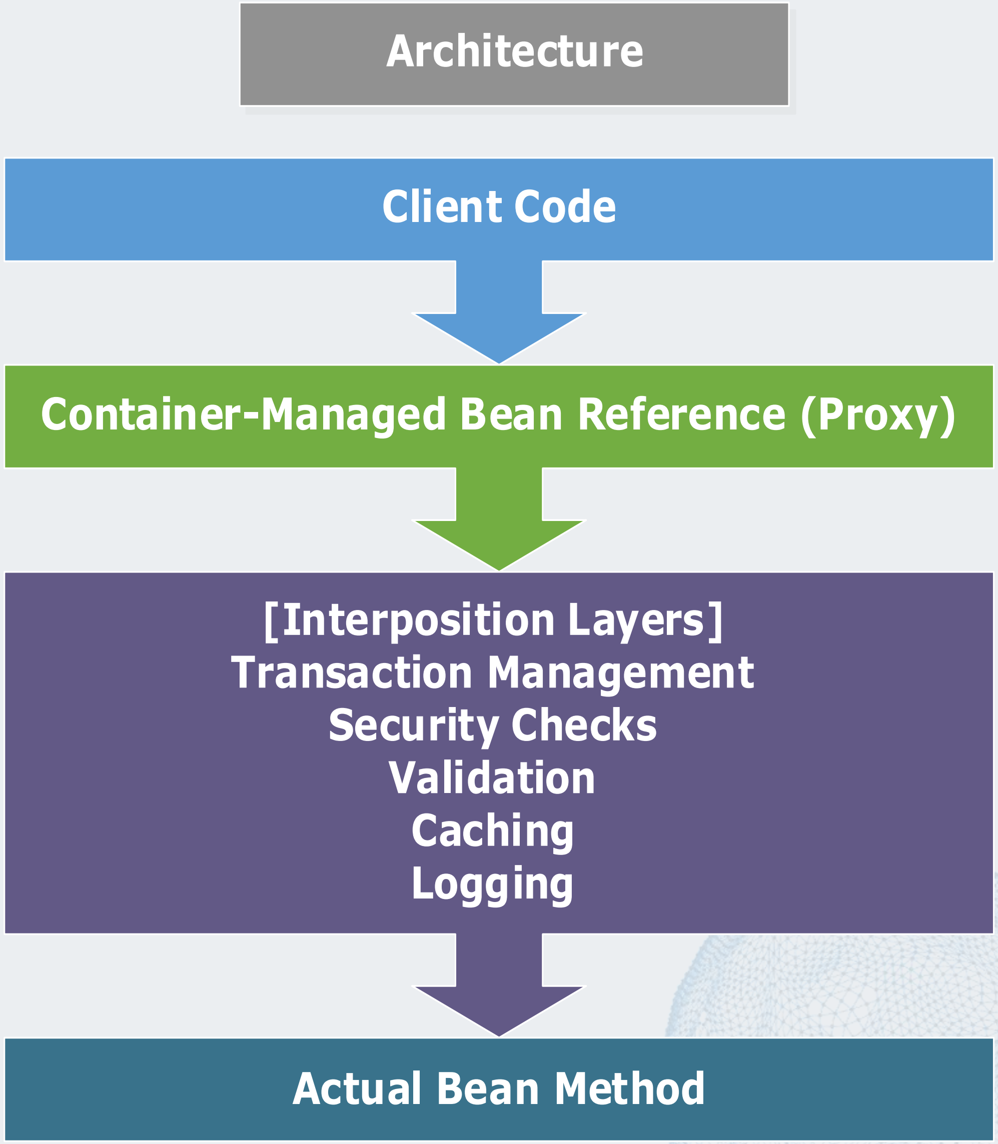
POJO Calls - No Interposition -> Creating Objects with new


```java
@Service
public class OrderService {

    public void processOrder(Order order) {
        // This EmailService is NOT managed by container
        EmailService emailService = new EmailService();
        emailService.sendConfirmation(order); // No interposition
    }
}
```

Bean Example

- **Proxy Pattern in Action**
  - When beans are created by the container and wrapped with a proxy, additional features activate when calls go through this proxy.



- **Key Principle:** The proxy intercepts the call, applies cross-cutting concerns, then delegates to the actual method.

Container

- **The Assembler and Manager**
  - A container assembles and manages components. It's responsible for instantiating, configuring, and assembling beans.
- **Spring Container:**
  - Assembles and packages software to run within a JVM
  - Instantiates, configures, and assembles beans
  - Reads configuration metadata (XML, Java annotations, or Java code)
  - Manages object lifecycle and dependencies

Transaction Management Example (Correct Way)


```
@Service
public class ReportService {

    @Autowired
    private DataSource dataSource;

    @Transactional // Container interposes transaction management
    public void generateReport(Long reportId) {
        // Transaction started automatically
        Report report = fetchReportData(reportId);
        processReport(report);
        saveReport(report);
        // Transaction committed automatically
        // Or rolled back if exception occurs
    }
}
```

```
// Client code
@RestController
public class ReportController {

    @Autowired
    private ReportService reportService; // Container-managed proxy

    @GetMapping("/reports/{id}")
    public void generate(@PathVariable Long id) {
        reportService.generateReport(id); // Goes through proxy ✓
    }
}
```

Transaction Management (Wrong Way)


```
@RestController
public class ReportController {

    @GetMapping("/reports/{id}")
    public void generate(@PathVariable Long id) {
        // Creating instance manually - NOT container-managed
        ReportService service = new ReportService();
        service.generateReport(id); // No proxy, no transaction! ✗
        // Transaction annotation is ignored
    }
}
```

- **Why It Fails:**
  - Object created with new is not managed by container
  - No proxy wrapping
  - @Transactional annotation has no effect
  - No automatic transaction management
- **Solution:** Always inject container-managed beans using @Autowired or constructor injection.

Self-Invocation Trap


```
@Service
public class DocumentService {

    public void generate(Long docId) {
        Document doc = createDocument(docId);
        export(doc); // Self-invocation - proxy
bypassed! ✗
    }

    @Transactional // This annotation is IGNORED
    public void export(Document doc) {
        // No transaction management when
called from generate()
        // Call goes directly via 'this', not through
proxy
        documentRepository.save(doc);
    }
}
```

- **Why It Fails:**
  - export() is called via this.export()
  - Call doesn't go through proxy
  - @Transactional annotation has no effect

Self-Invocation - Correct Pattern


```
@Service
public class DocumentExporter {

    @Autowired
    private DocumentRepository
documentRepository;

    @Transactional // Now this works!
    public void export(Document doc) {
        // Transaction management active
        documentRepository.save(doc);
    }
}
```

```
@Service
public class DocumentService {

    @Autowired
    private DocumentExporter exporter; // Inject
separate bean

    public void generate(Long docId) {
        Document doc = createDocument(docId);
        exporter.export(doc); // Goes through proxy ✓
    }
}
```

Framework Enablers - Complete Picture

How Everything Fits Together

Configuration Metadata (Annotations/XML/Java Config)

↓

Container Reads Metadata

↓

Creates POJOs → Wraps with Proxies → Produces Beans

↓

Dependency Injection (Wires Beans Together)

↓

Fully Assembled Components

↓

Client Code Calls Bean Methods

↓

Proxy Intercepts (Interposition)

↓

Cross-Cutting Concerns Applied

↓

Actual Method Executes

ASST. PROF. DR. SAMET KAYA

FATIH SULTAN MEHMET VAKIF UNIVERSITY

skaya@fsm.edu.tr

kysamet@gmail.com

Some images and icons in this presentation are sourced from Freepik.