CS544
Enterprise Application Architecture

Lesson 1 – Introduction

Frameworks and Best Practices Used in Designing Large-Scale Software Systems

Payman Salek, M.S.

Original Material: Prof. Rene de Jong – July 2022

© 2022 Maharishi International University

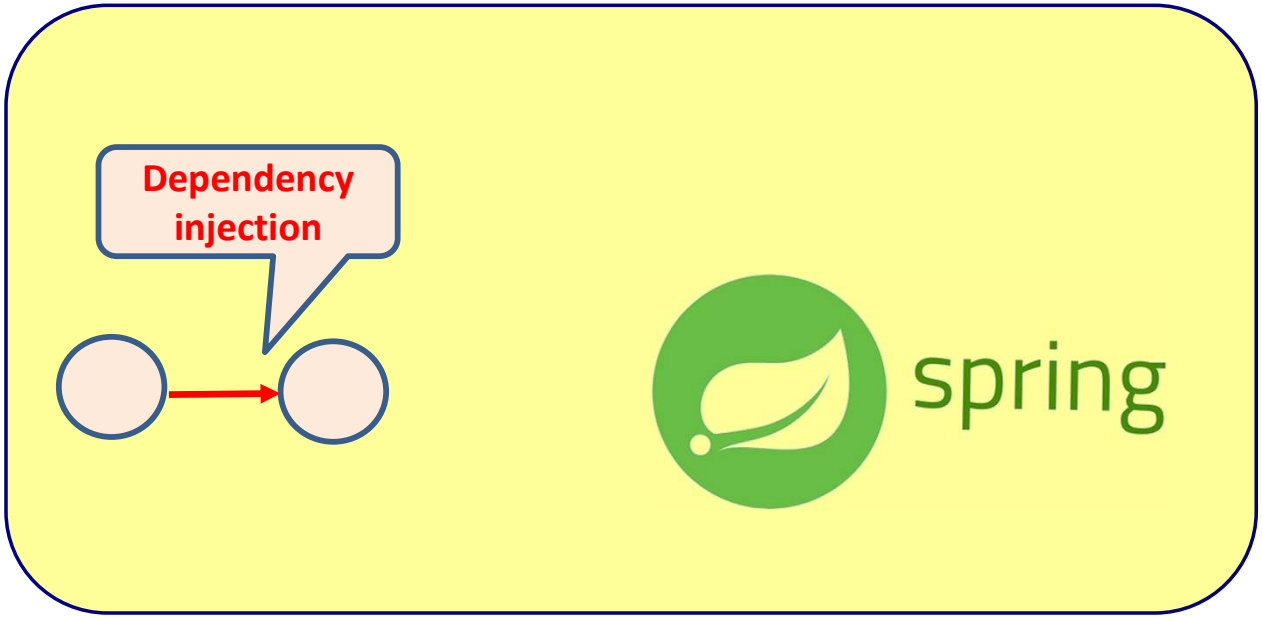
Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
November 28 **Lesson 1** Introduction Spring framework Dependency injection	November 29 **Lesson 2** Spring Boot AOP	November 30 **Lesson 3** JDBC JPA	December 1 **Lesson 4** JPA mapping 1	December 2 **Lesson 5** JPA mapping 2	December 3 **Lesson 6** JPA queries	December 4
December 5 **Lesson 7** Transactions	December 6 **Lesson 8** MongoDB	December 7 **Midterm Review**	December 8 **Midterm exam**	December 9 **Lesson 9** REST webservices	December 10 **Lesson 10** SOAP webservices	December 11
December 12 **Lesson 11** Messaging	December 13 **Lesson 12** Scheduling Events Configuration	December 14 **Lesson 13** Monitoring	December 15 **Lesson 14** Testing your application	December 16 **Final review**	December 17 **Final exam**	December 18
December 19 **Project**	December 20 **Project**	December 21 **Project**	December 22 **Presentations**			

© 2022 MIU

Enterprise Application Architecture

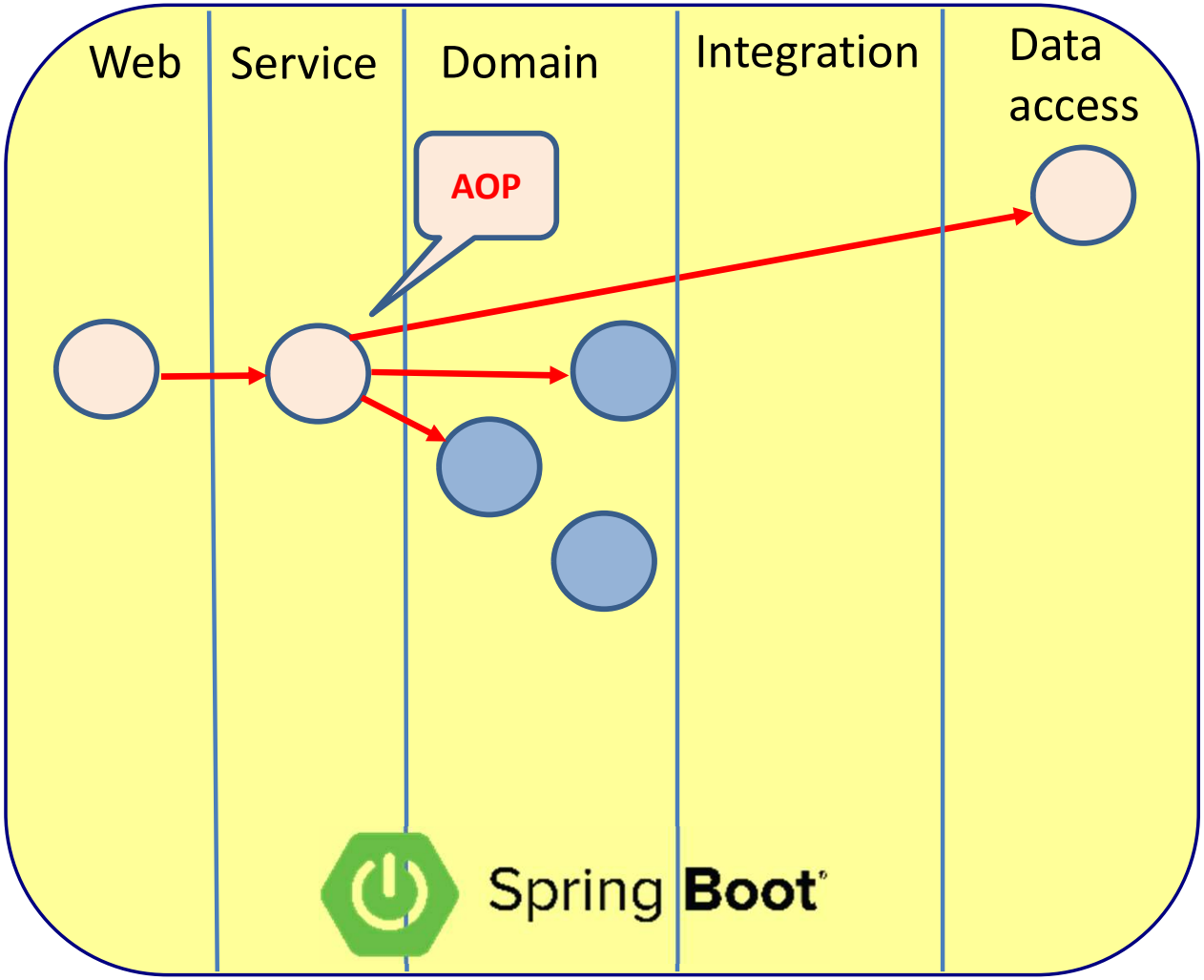
© 2022 MIU

Lesson 1: Introduction to Spring



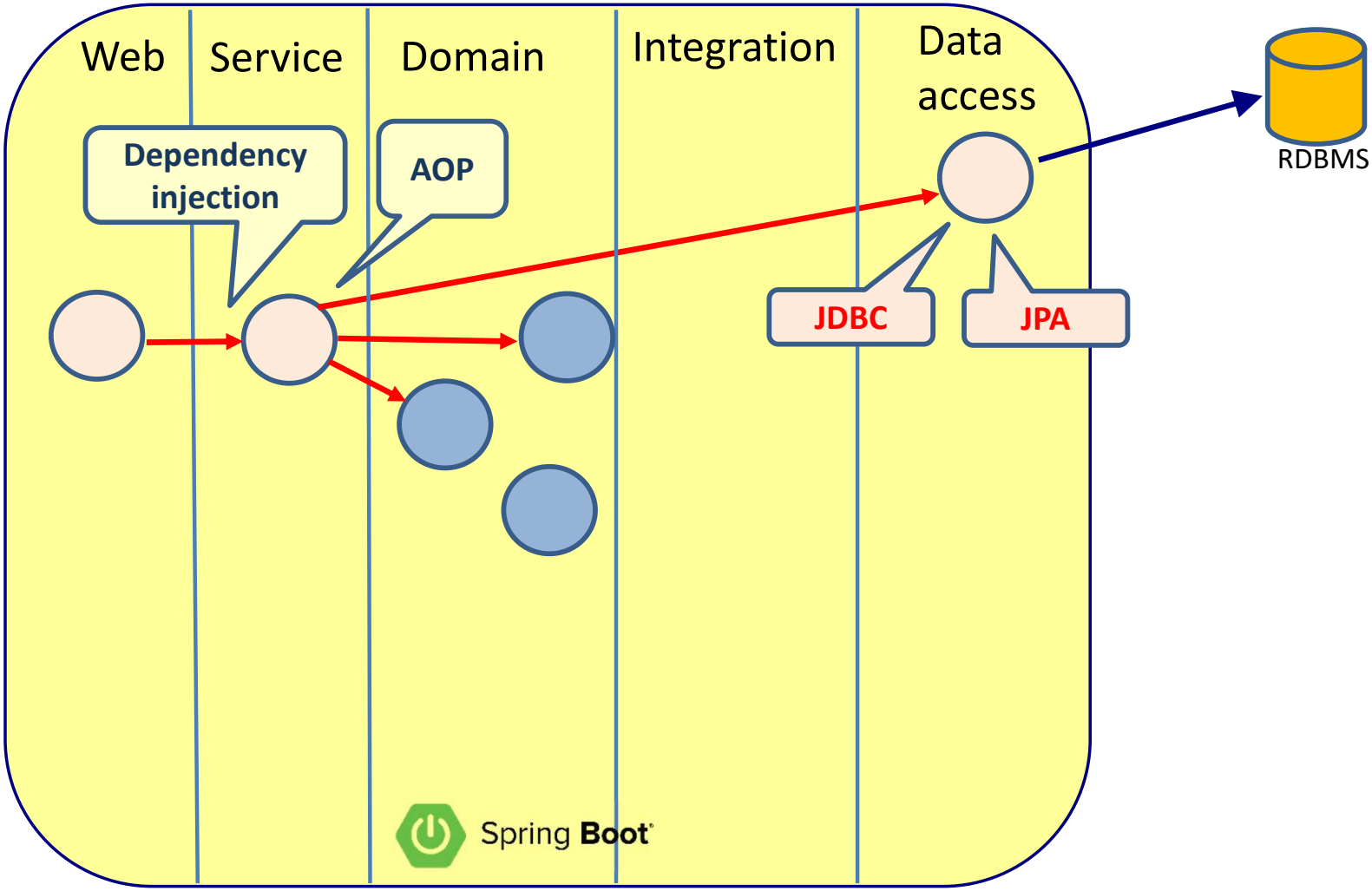
© 2022 MIU

Lesson 2



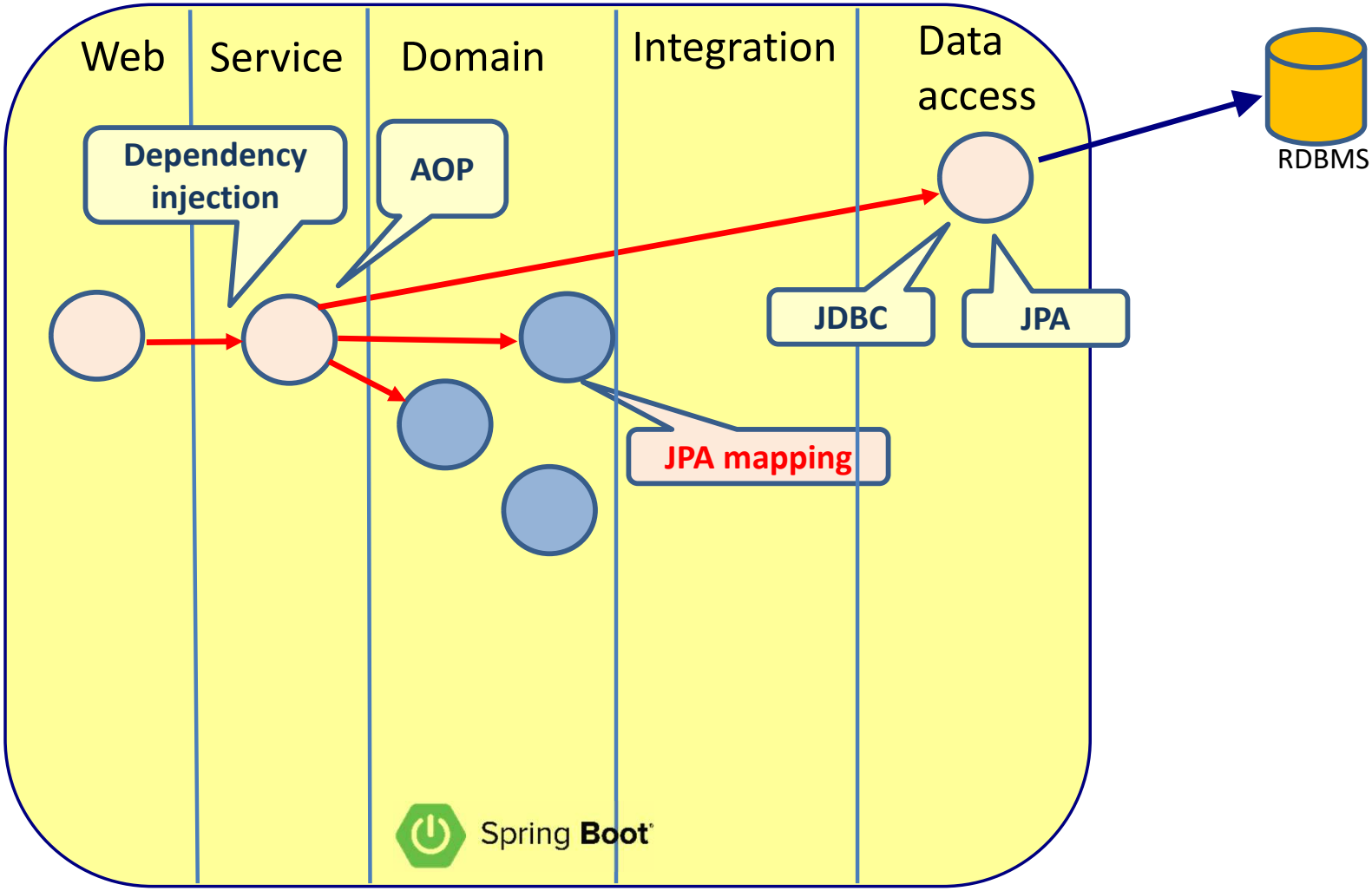
© 2022 MIU

Lesson 3



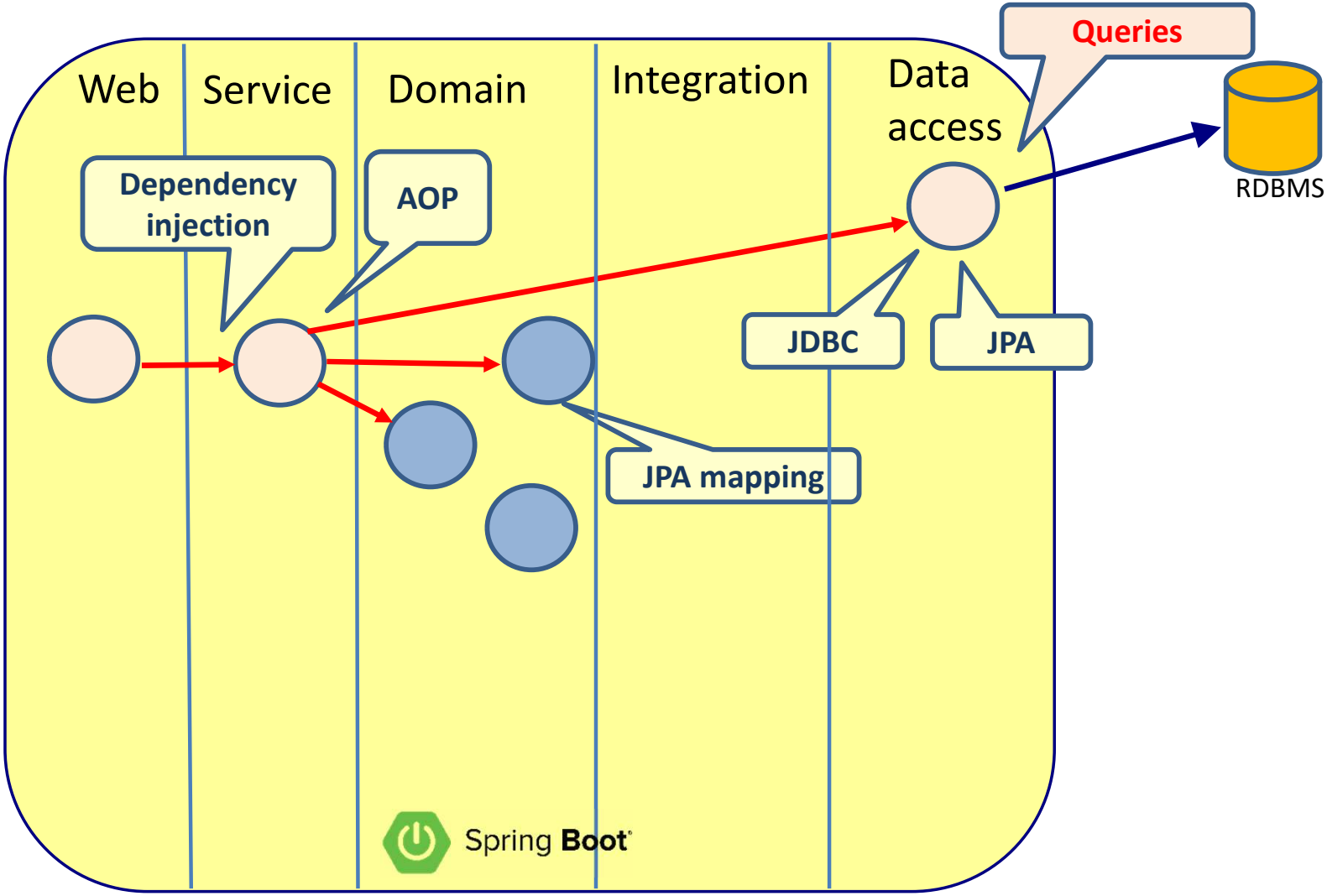
© 2022 MIU

Lesson 4+5



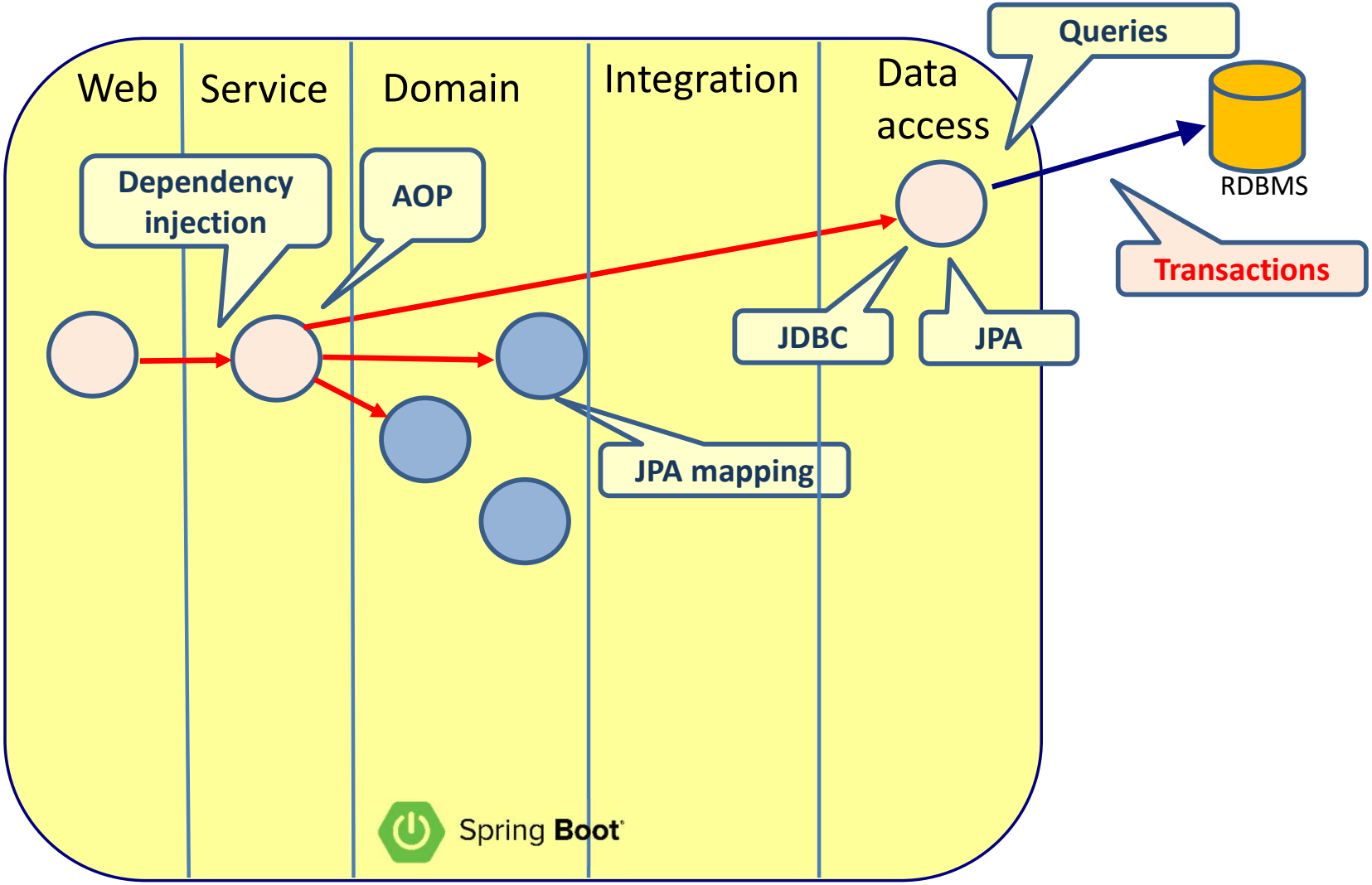
© 2022 MIU

Lesson 6



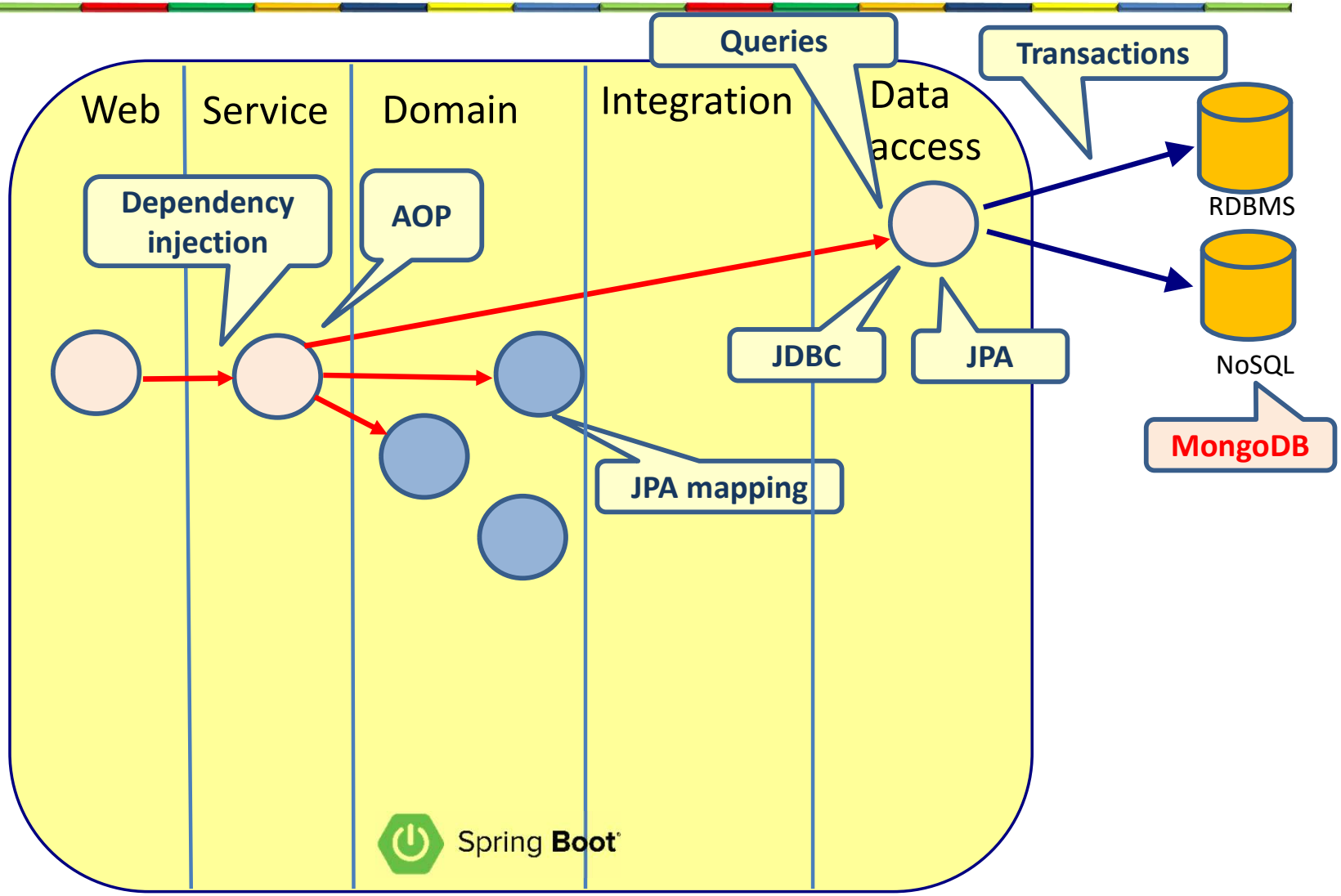
© 2022 MIU

Lesson 7



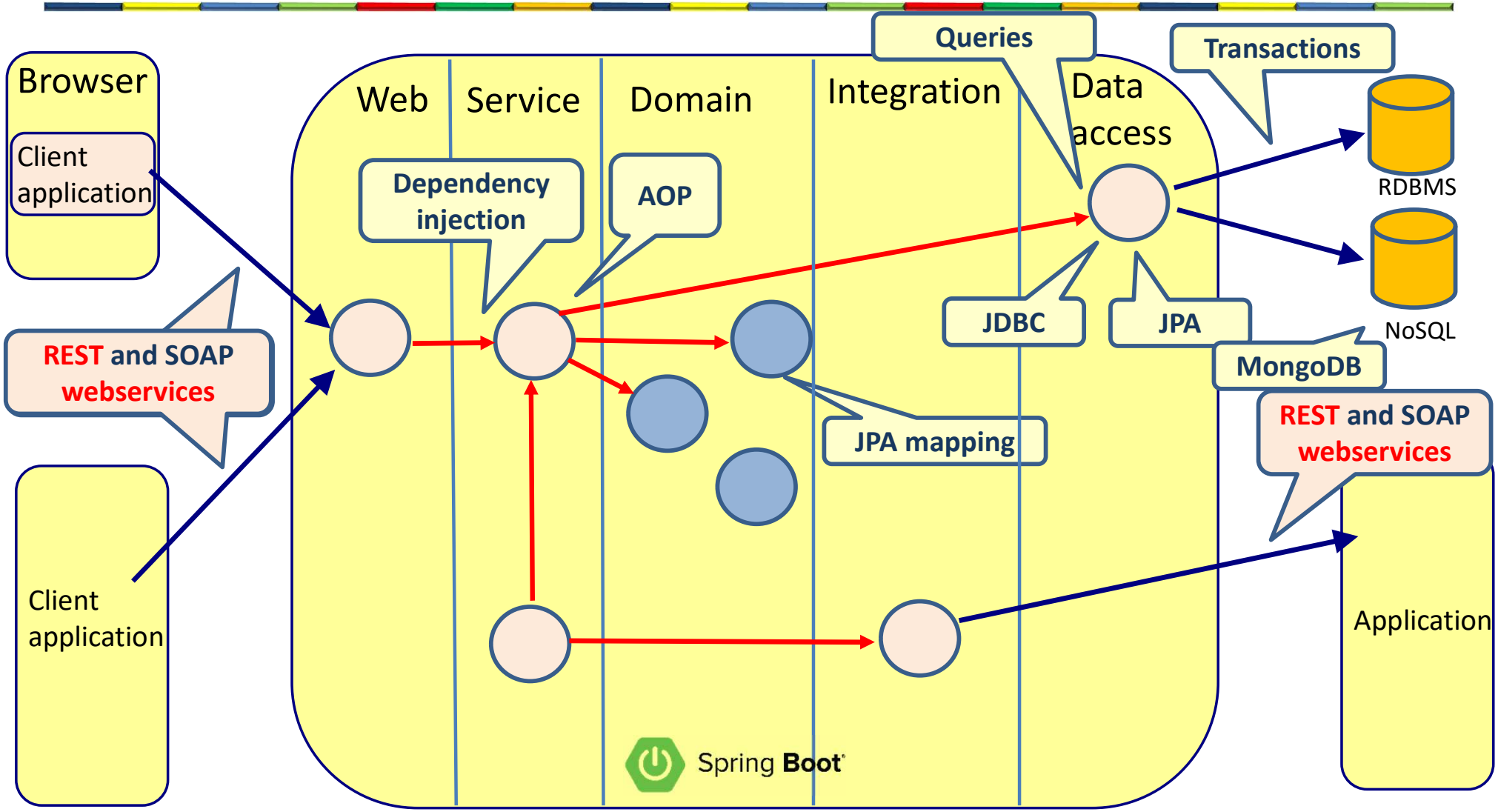
© 2022 MIU

Lesson 8



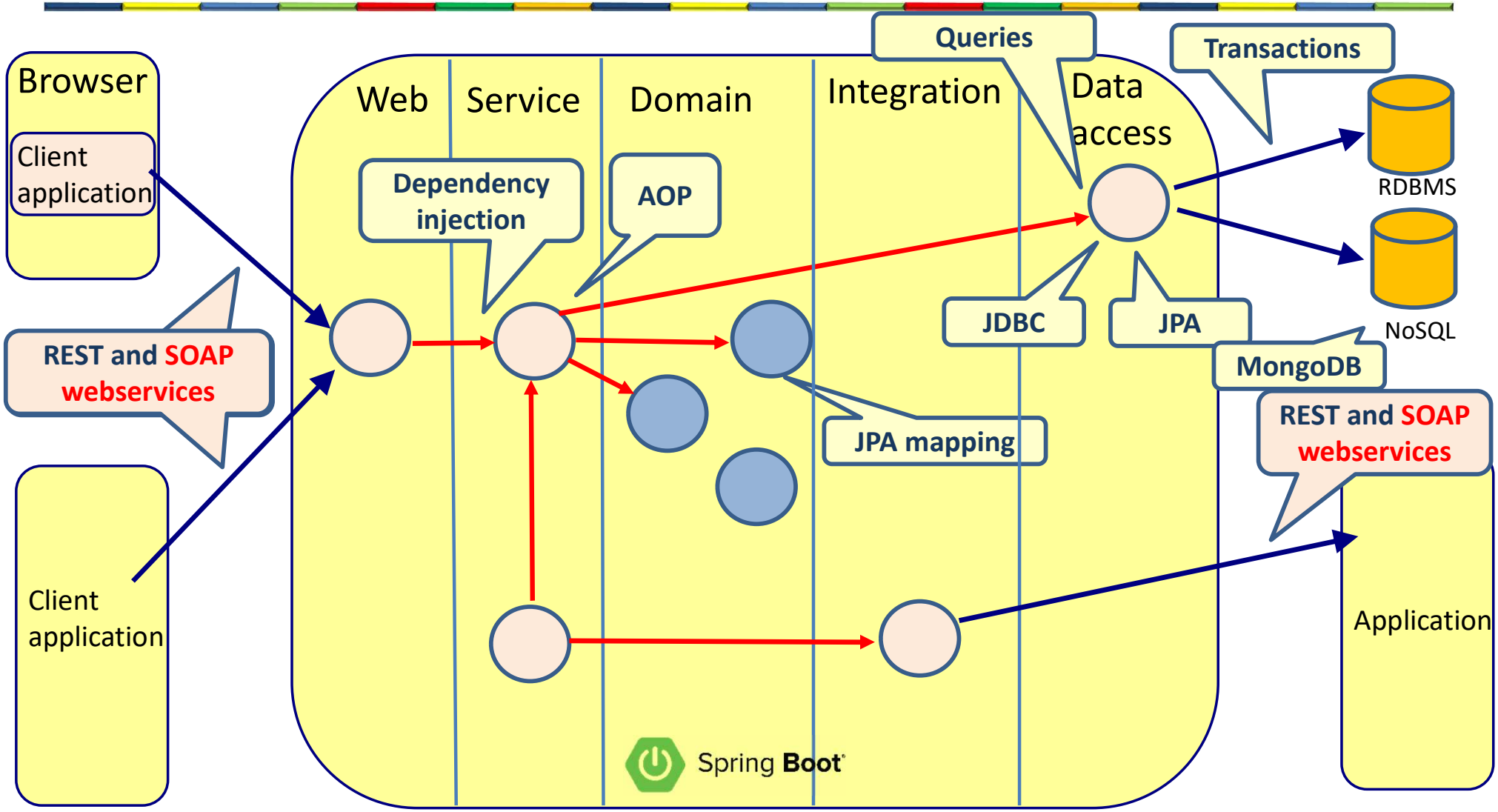
© 2022 MIU

Lesson 9



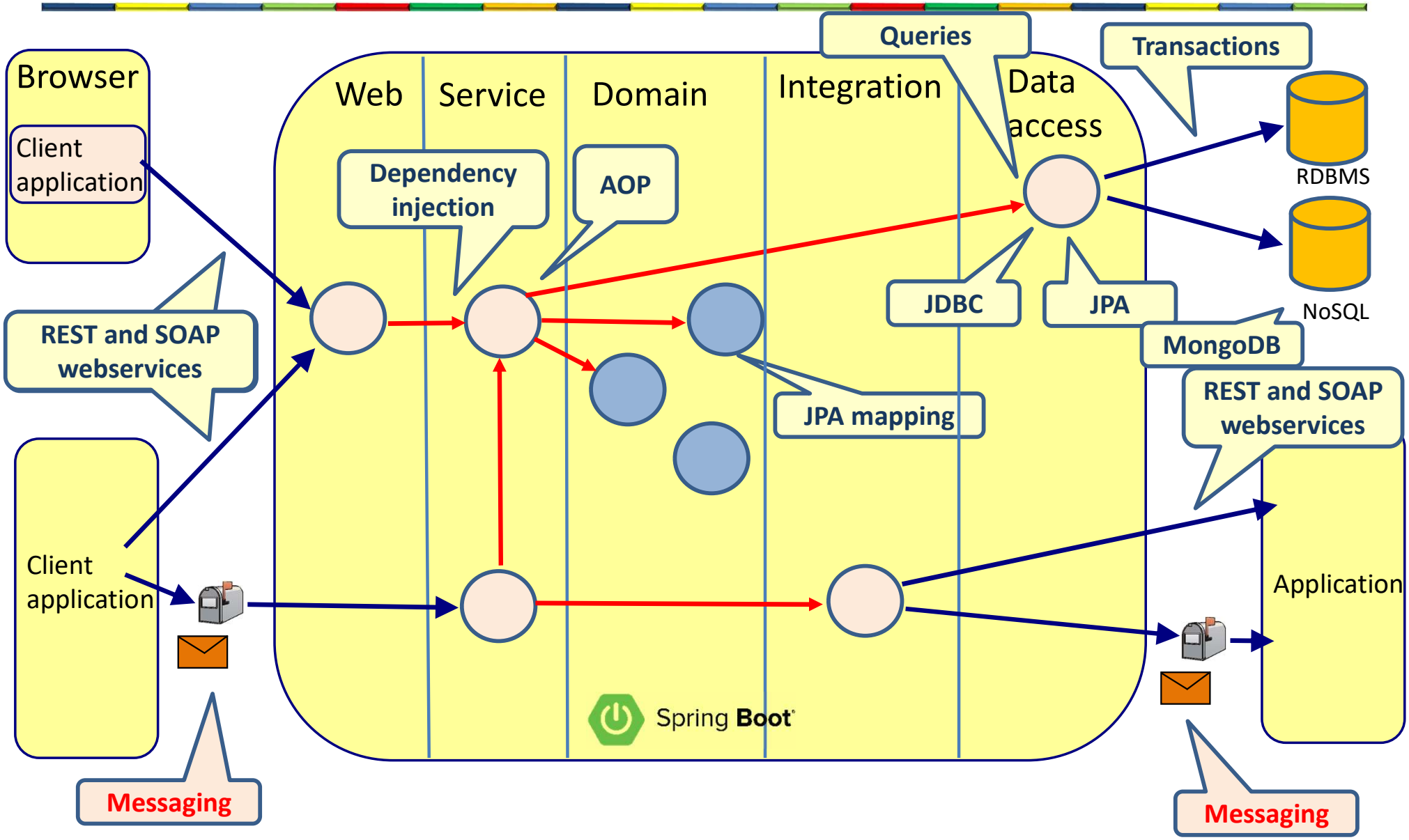
© 2022 MIU

Lesson 10



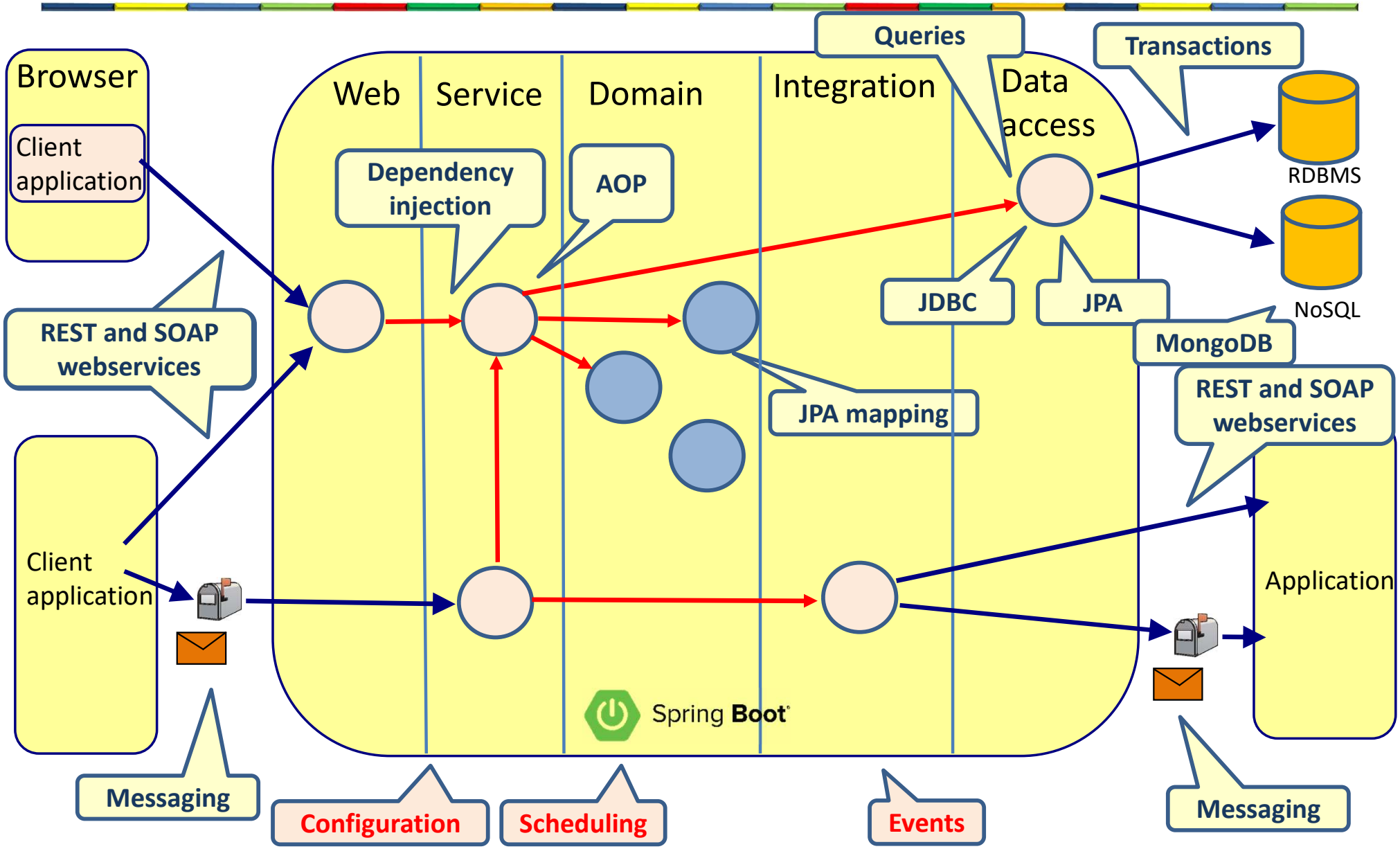
© 2022 MIU

Lesson 11



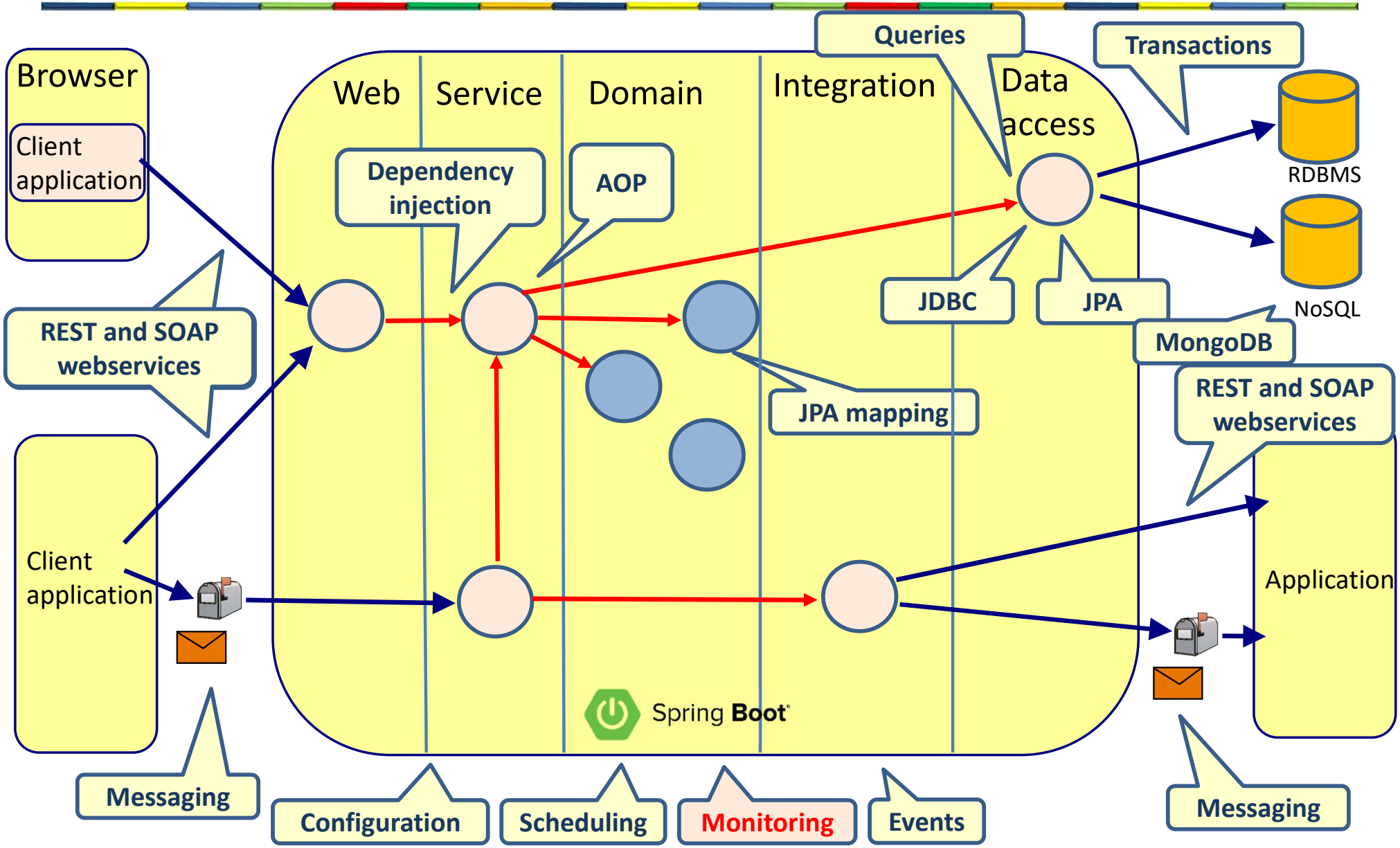
© 2022 MIU

Lesson 12



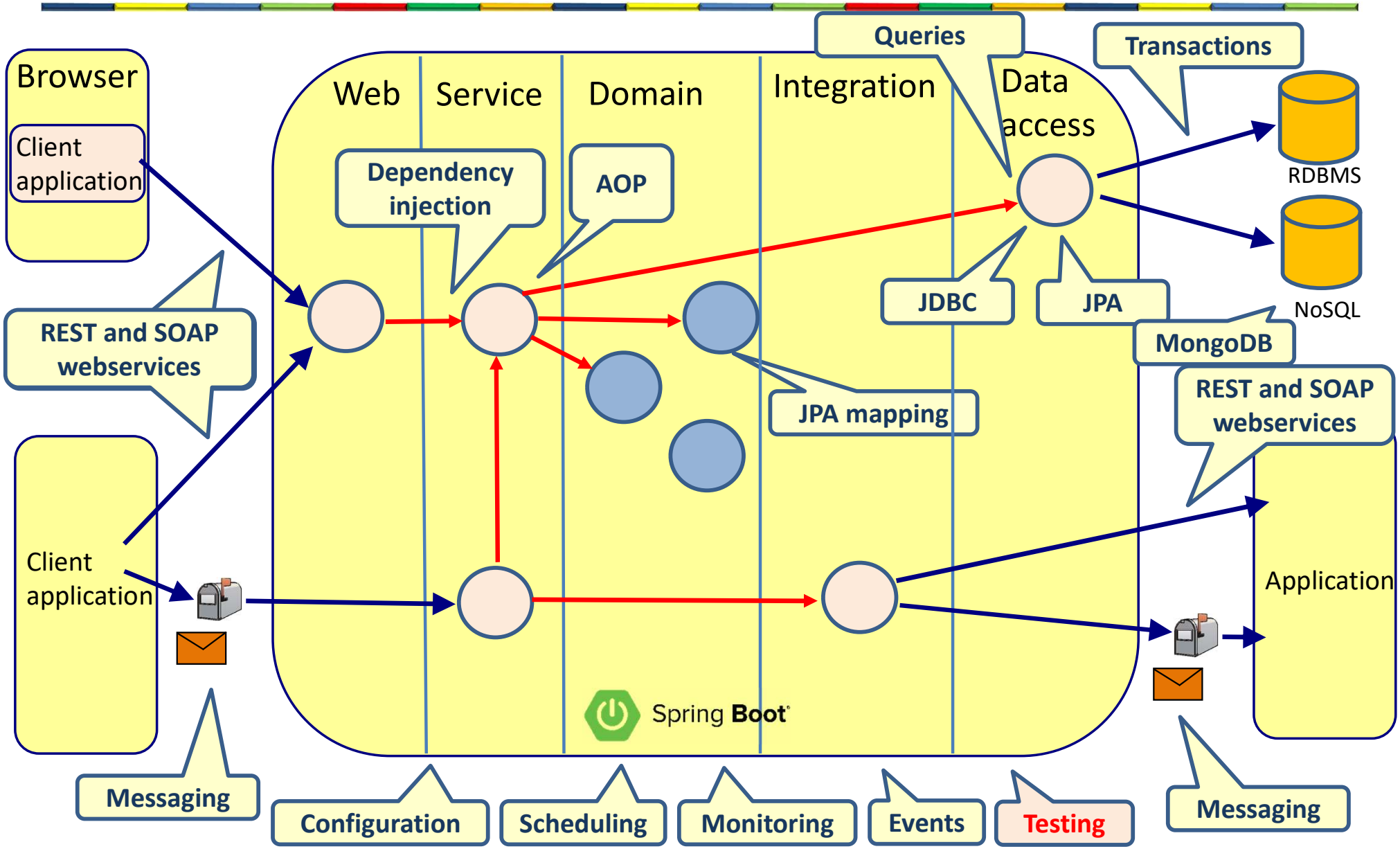
© 2022 MIU

Lesson 13



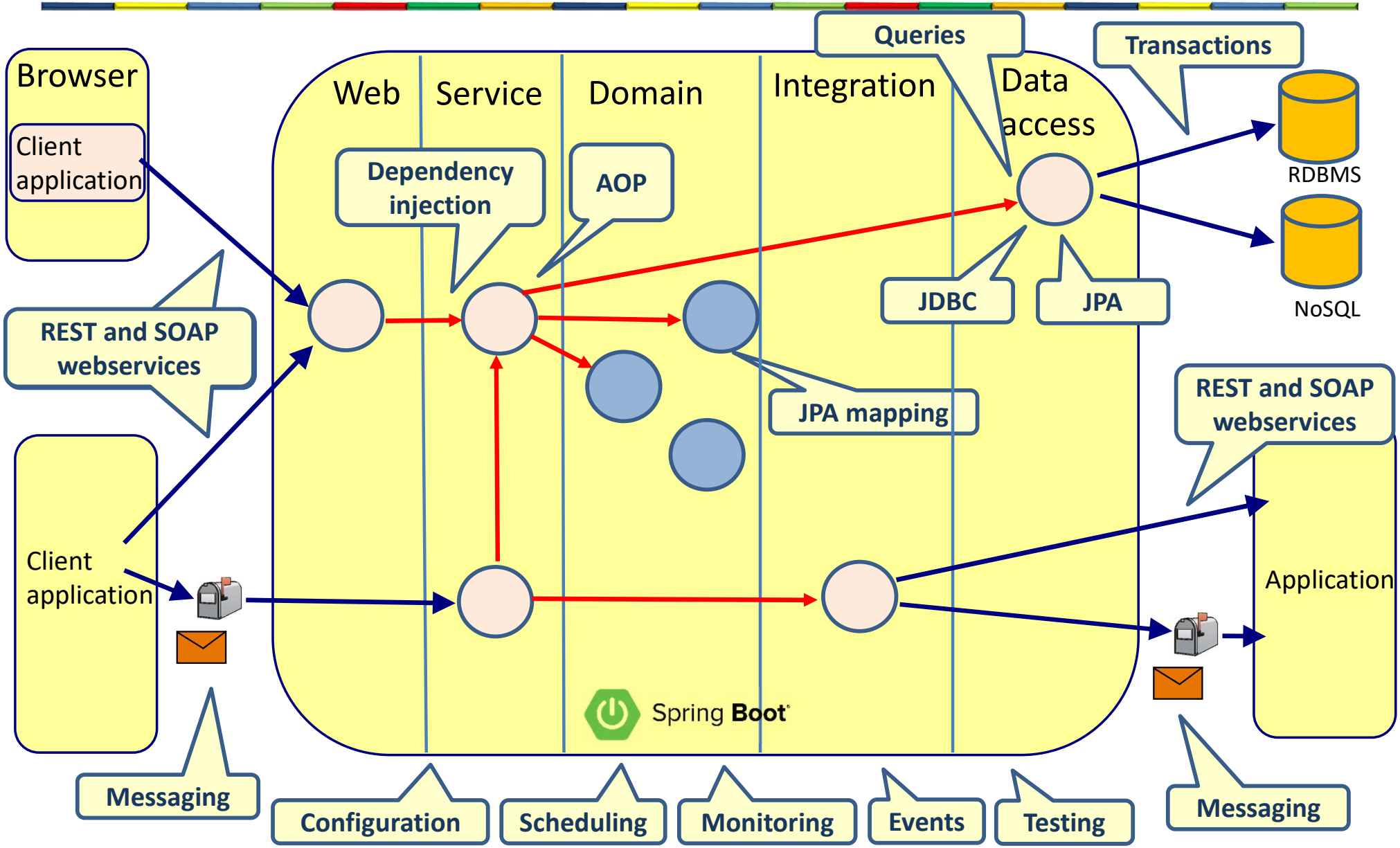
© 2022 MIU

Lesson 14



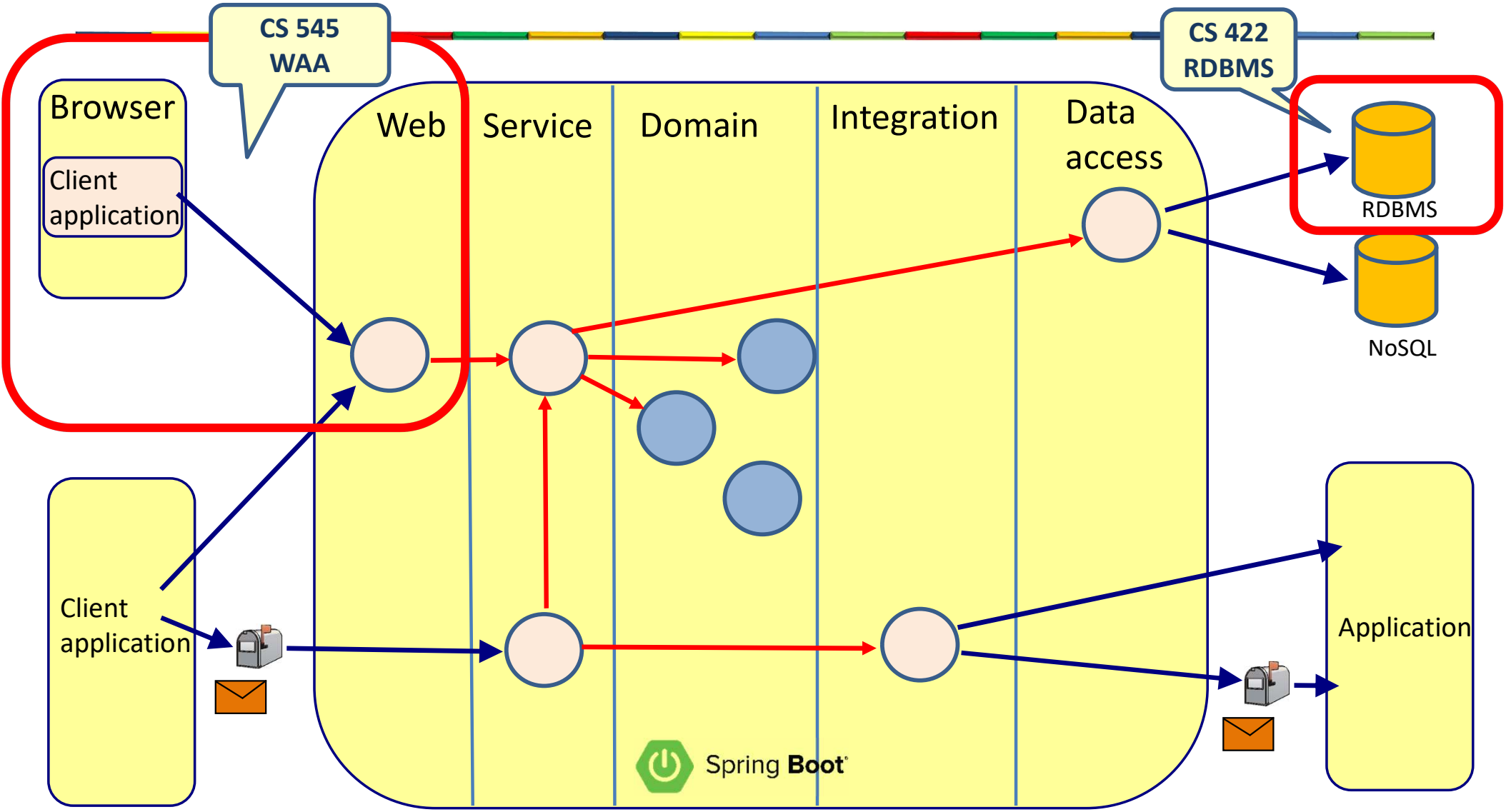
© 2022 MIU

Enterprise Application Architecture



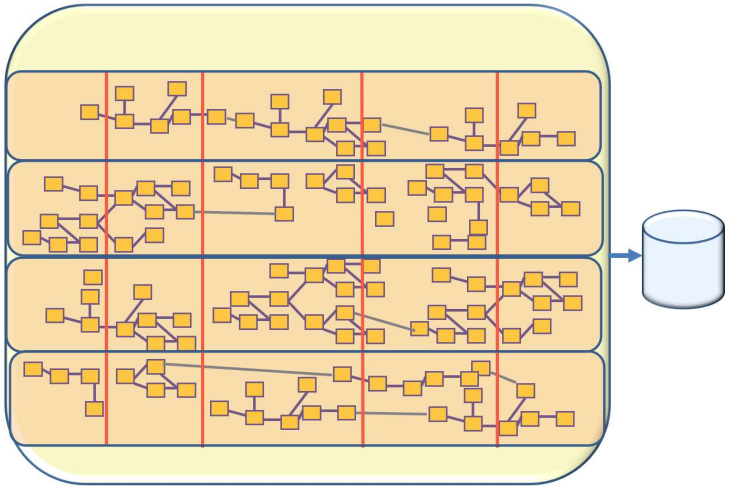
© 2022 MIU

Connection with other courses

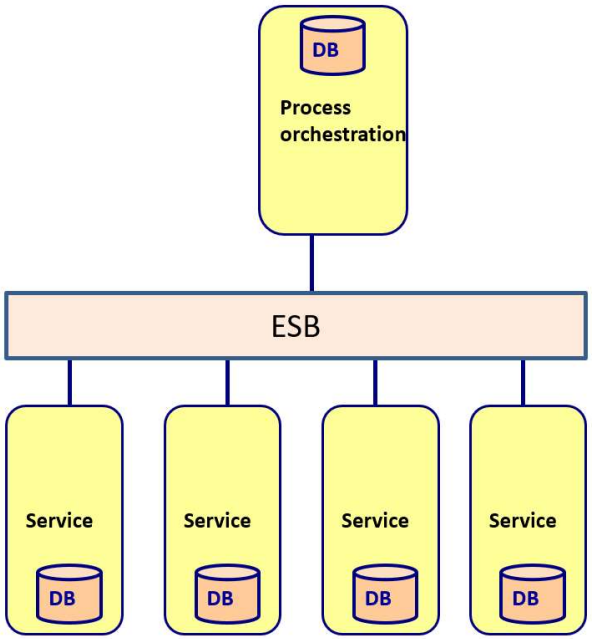


© 2022 MIU

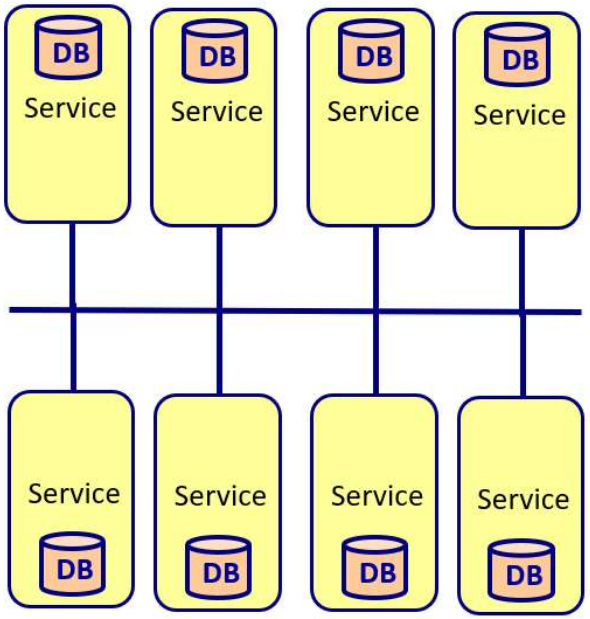
CS590 SWA



components



SOA



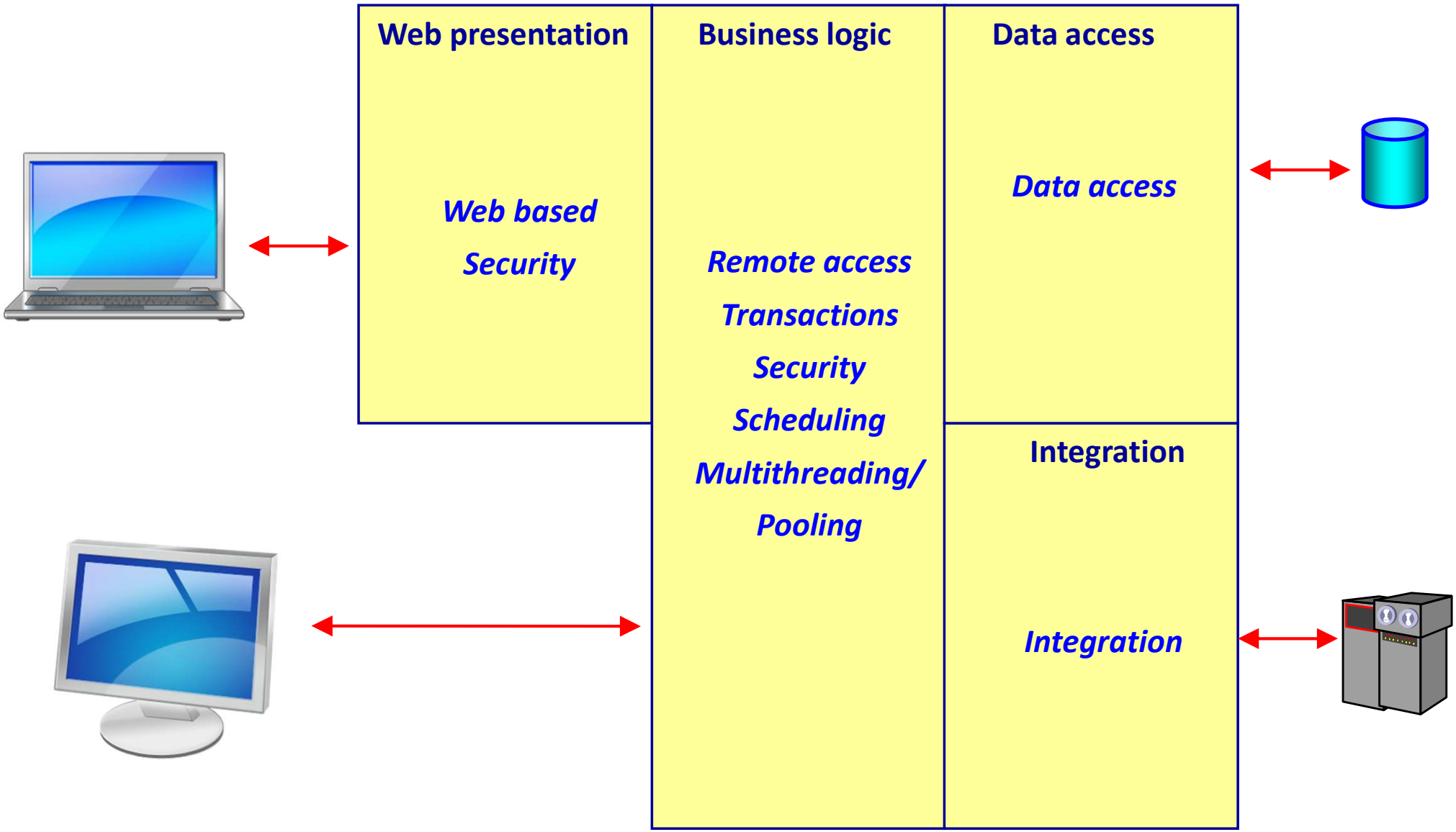
microservices

© 2022 MIU

INTRODUCTION TO SPRING

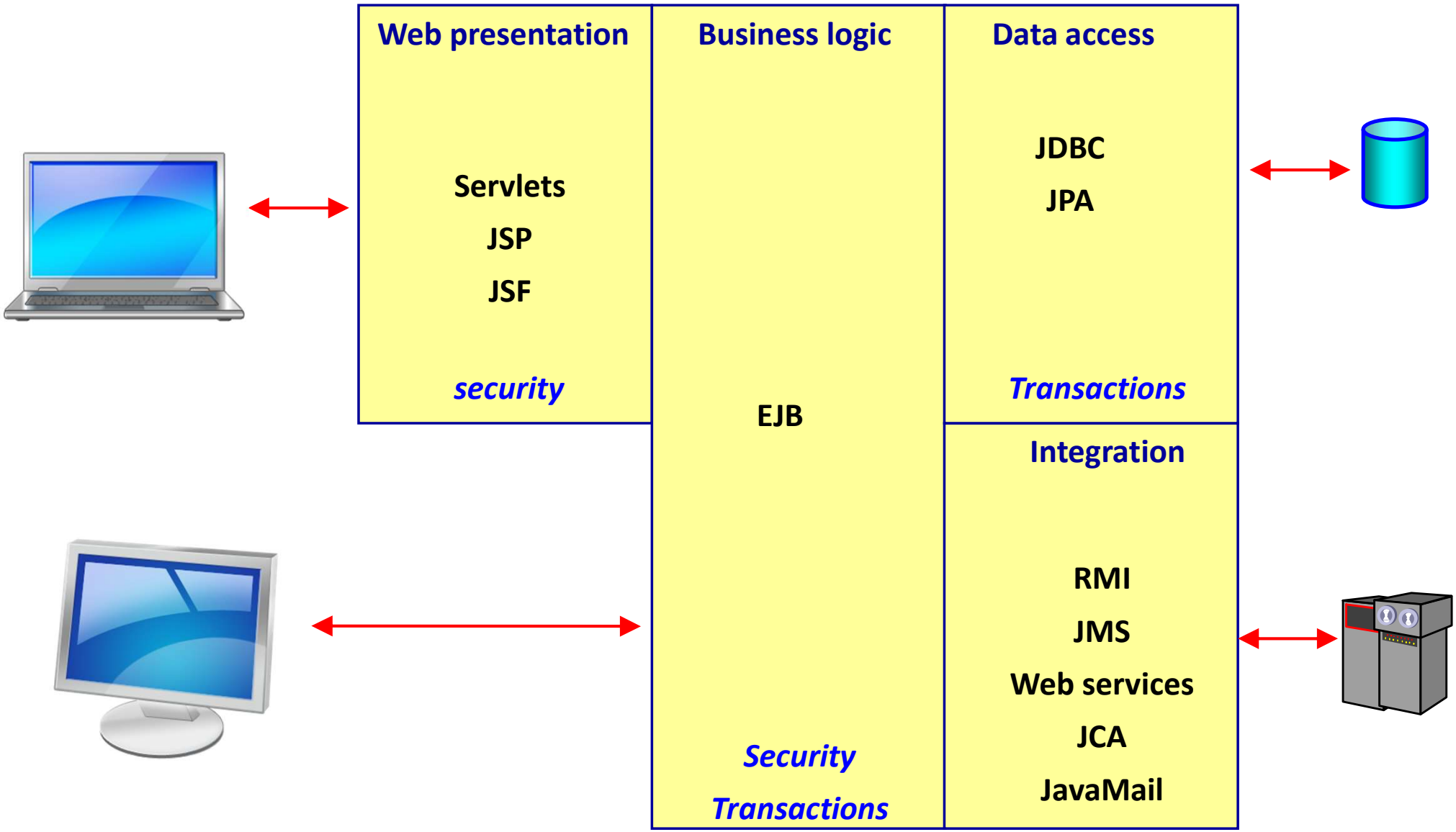
© 2022 MIU

What is an enterprise application?



© 2022 MIU

Java EE standard



© 2022 MIU

What is Spring?

- Lightweight enterprise Java framework
- Open source
- Goal: make developing enterprise Java applications easier

© 2022 MIU

Aim of the Spring framework

- Make enterprise Java application development as easy as possible following good programming practices
  - POJO-based programming
  - Separation of concern
  - Flexibility

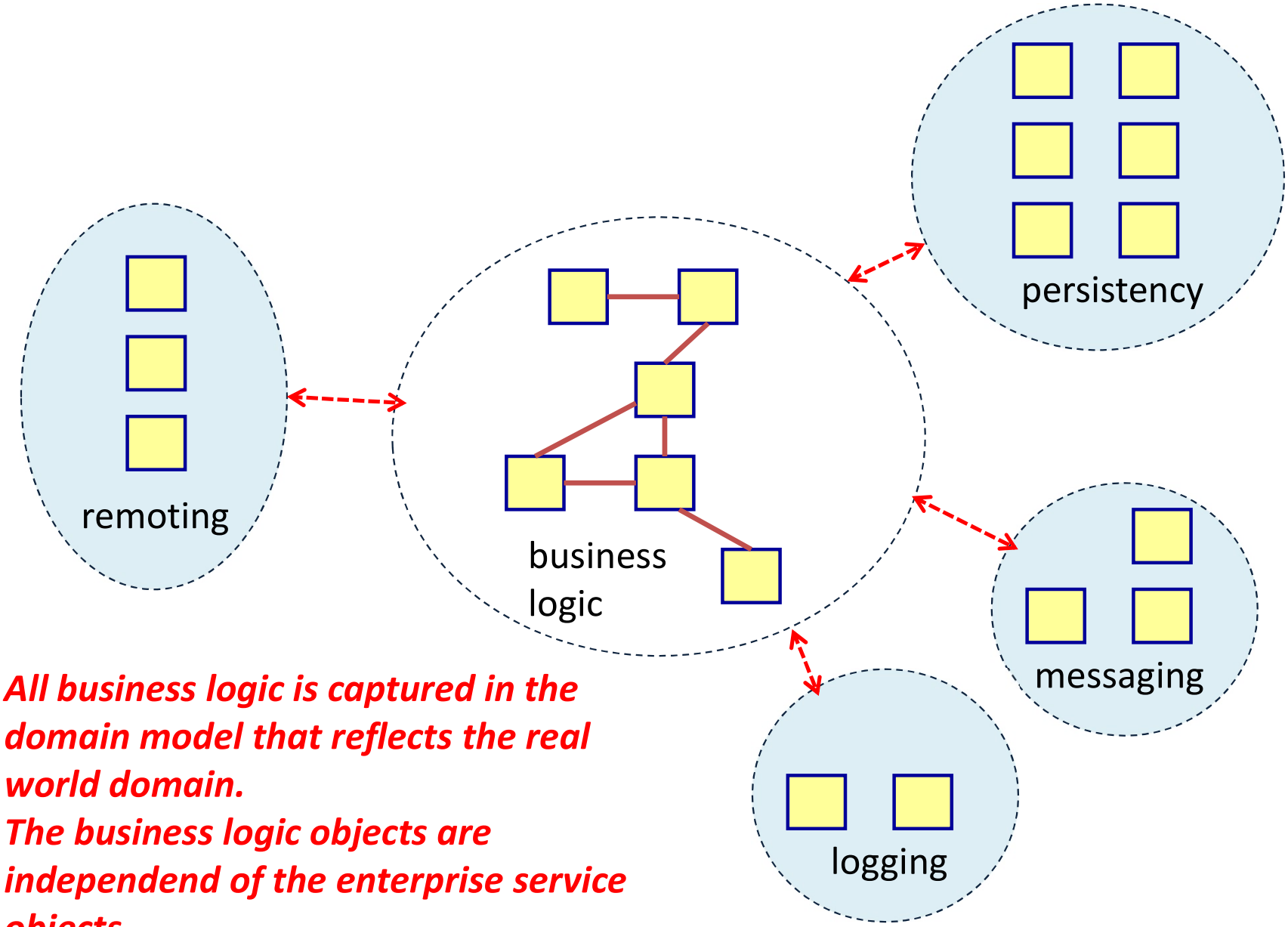
© 2022 MIU

POJO based programming

- All code is written in java objects
  - No EJB's
- Promotes Object-Oriented principles
- Simple to understand
- Simple to refactor
- Simple to unit test

© 2022 MIU

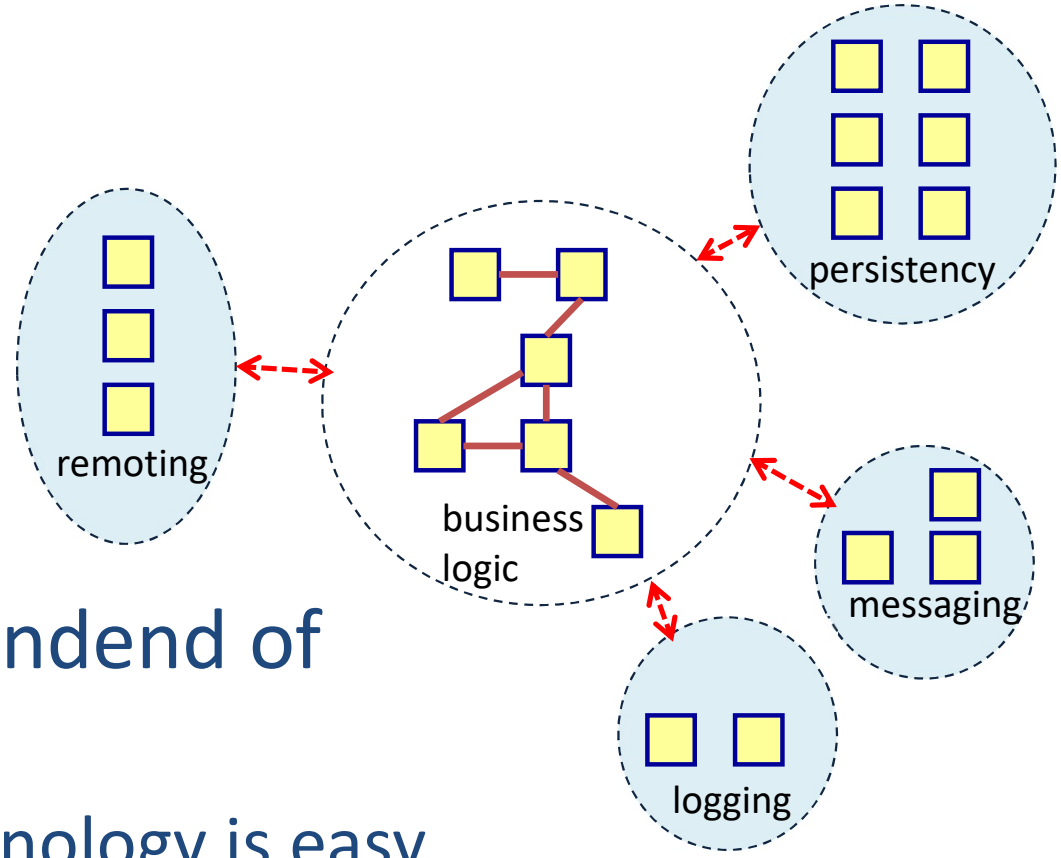
Domain-Driven Design (DDD)



1. ***All business logic is captured in the domain model that reflects the real world domain.***
2. ***The business logic objects are independend of the enterprise service objects***

© 2022 MIU

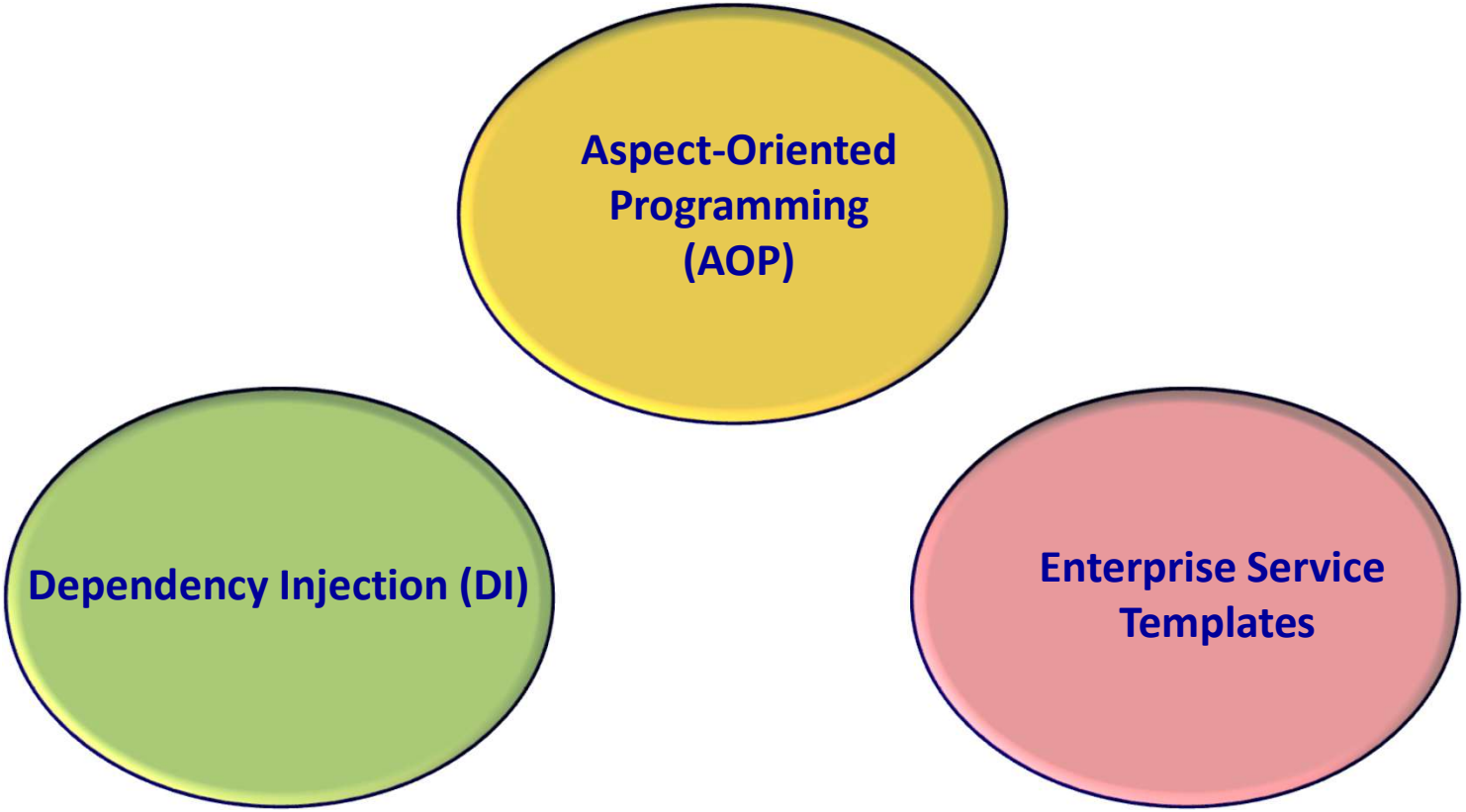
Advantages of DDD



- Business logic is independend of technology changes
  - Switching between technology is easy
- Business logic is easy to understand
  - Easy to write, test, modify

© 2022 MIU

Core of Spring



© 2022 MIU

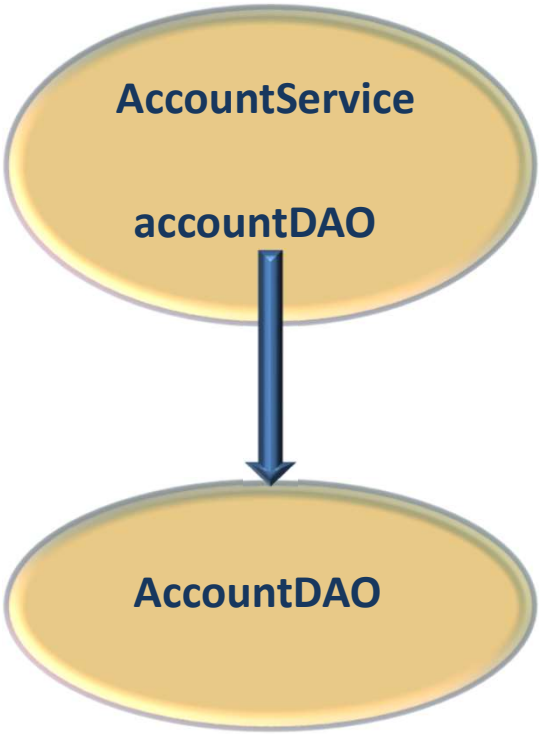
Dependency Injection

- Spring instantiates objects and wires them together

```
public class AccountService {
   private AccountDAO accountDAO;

    public void setAccountDAO(AccountDAO accountDAO) {
        this.accountDAO = accountDAO;
    }

    public Account getAccount(int accountNumber) {
        return accountDAO.loadAccount(accountNumber);
    }
}
```

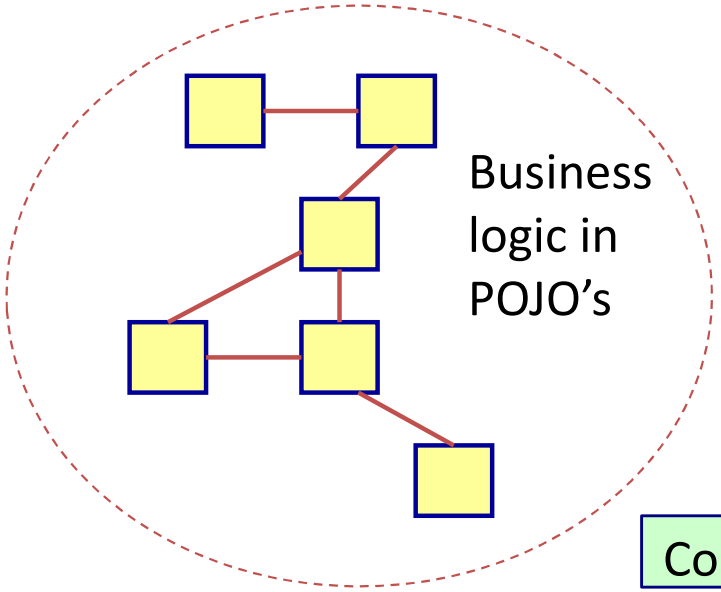



```
<bean id="accountService" class="bank.AccountService">
  <property name="accountDAO" ref="accountDAO" />
</bean>
<bean id="accountDAO" class="bank.dao.AccountDAO" />
```

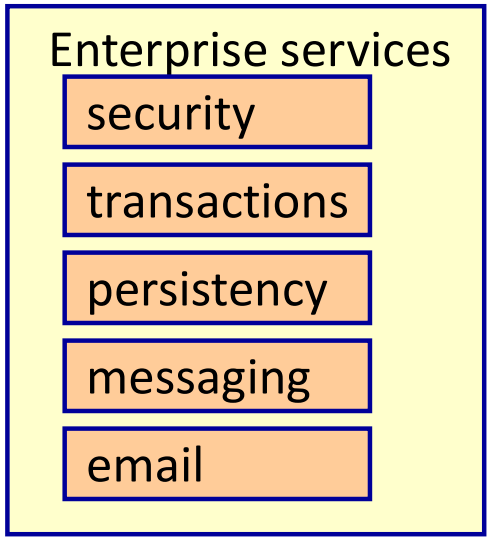

© 2022 MIU

Aspect-Oriented Programming (AOP)

- Separate the crosscutting concerns (plumbing code) from the business logic code
- AOP development
  1. **Write the business logic without worrying about the enterprise services (security, transactions, logging, etc)**
  2. **Write the enterprise services**
  3. **Weave them together**



Configuration file



© 2022 MIU

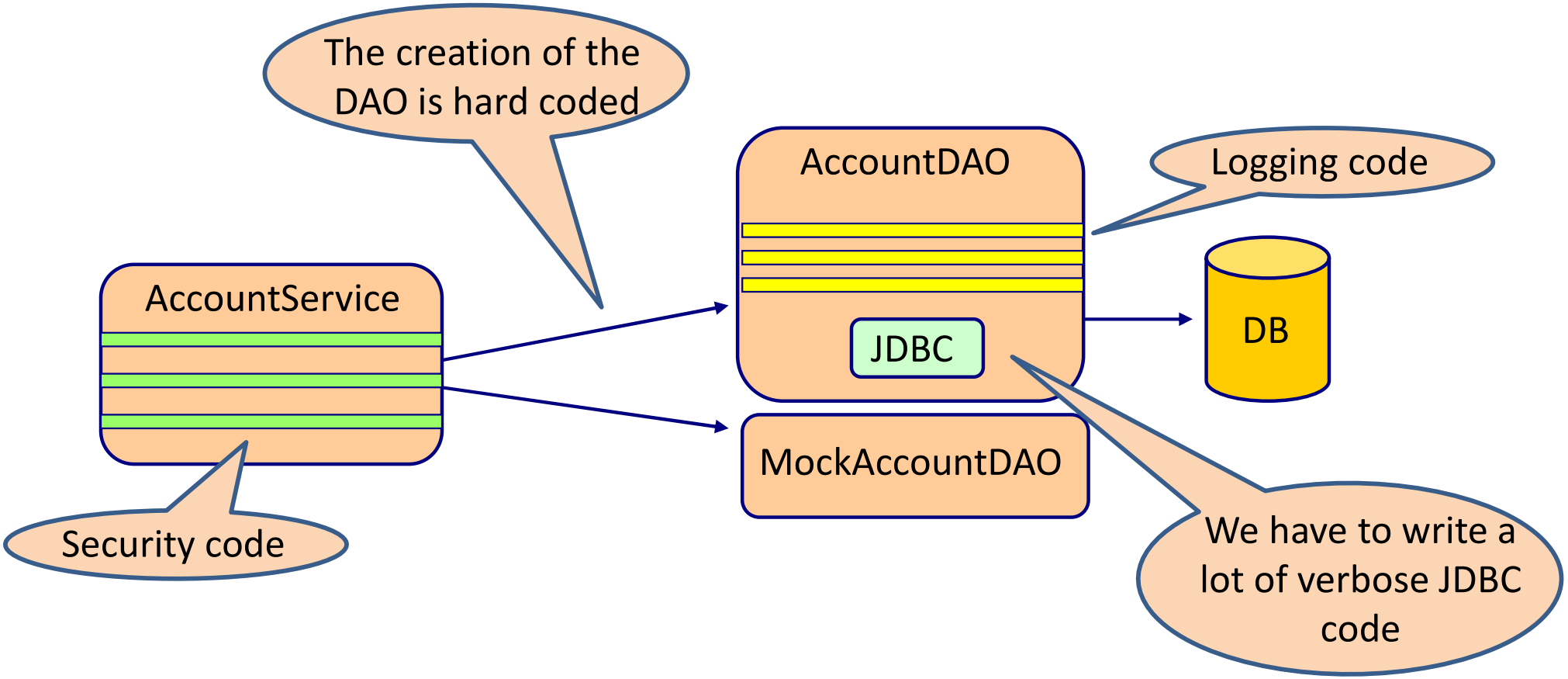
Enterprise Service Templates

- Makes programming the different enterprise service API's simpler.
  - JDBC template
  - JMS template
  - JavaMail template
  - Hibernate template

- Let the programmer focus on what needs to happen instead of complexity of the specific API
  - Resource management
  - Exception handling
  - Try-catch-finally-try-catch blocks

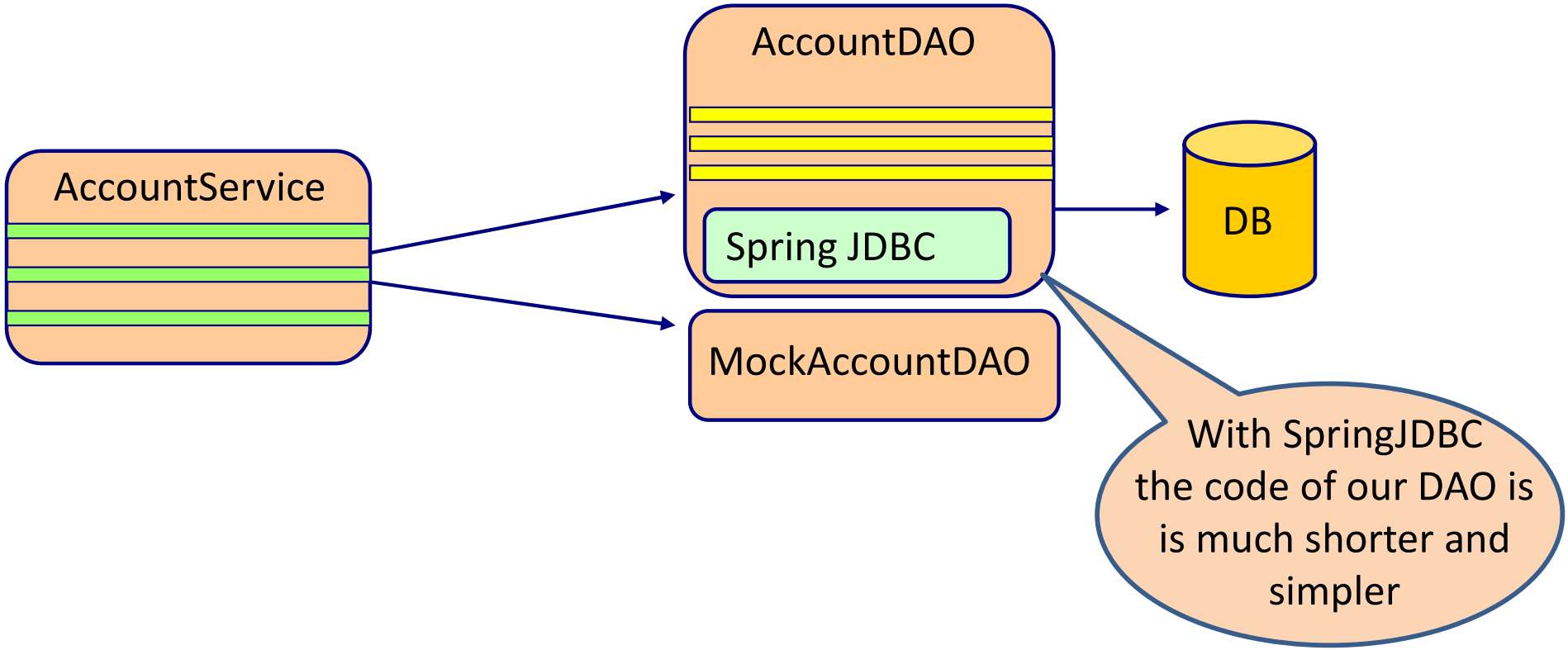
© 2022 MIU

Without Spring



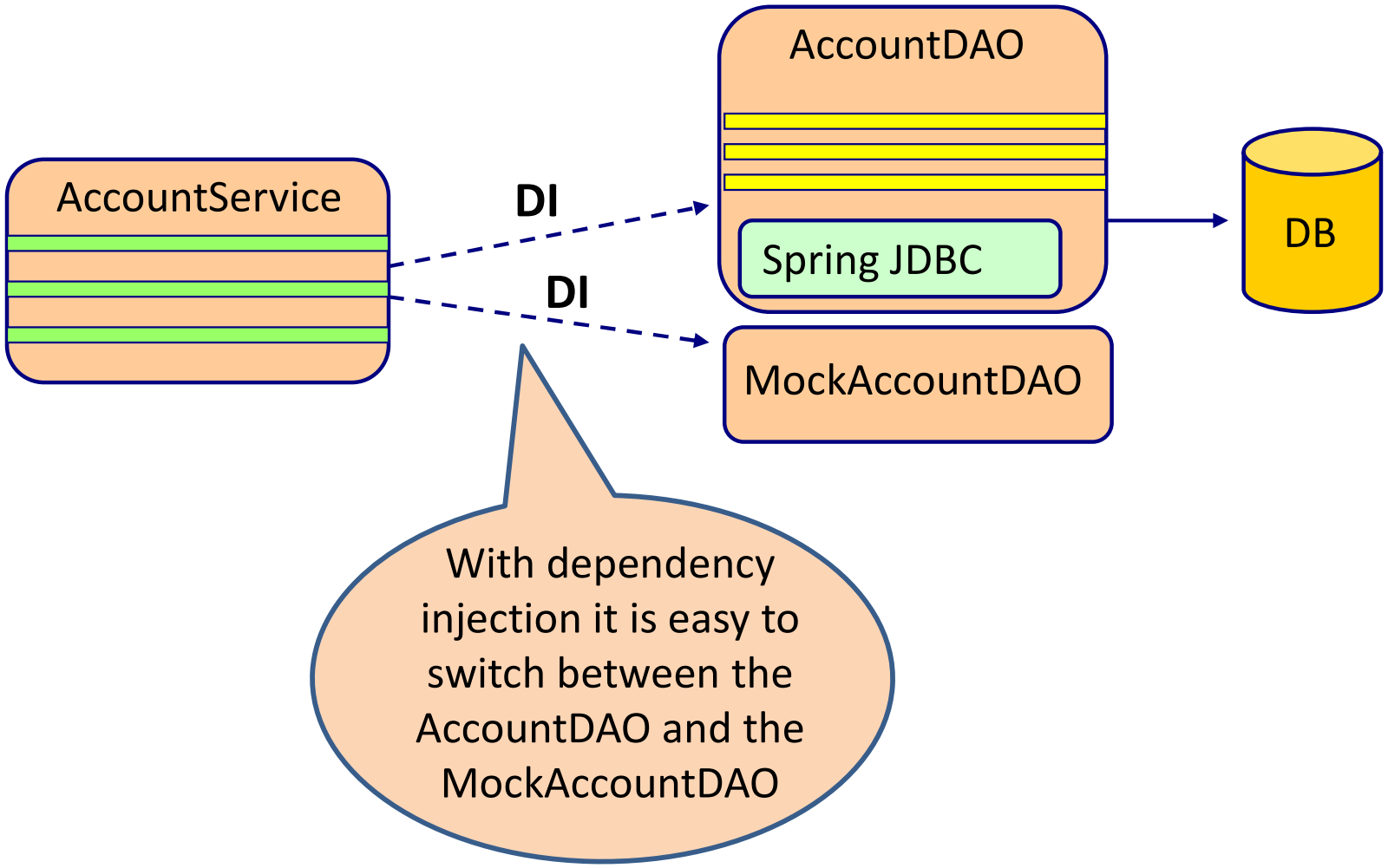
© 2022 MIU

Add SpringJDBC



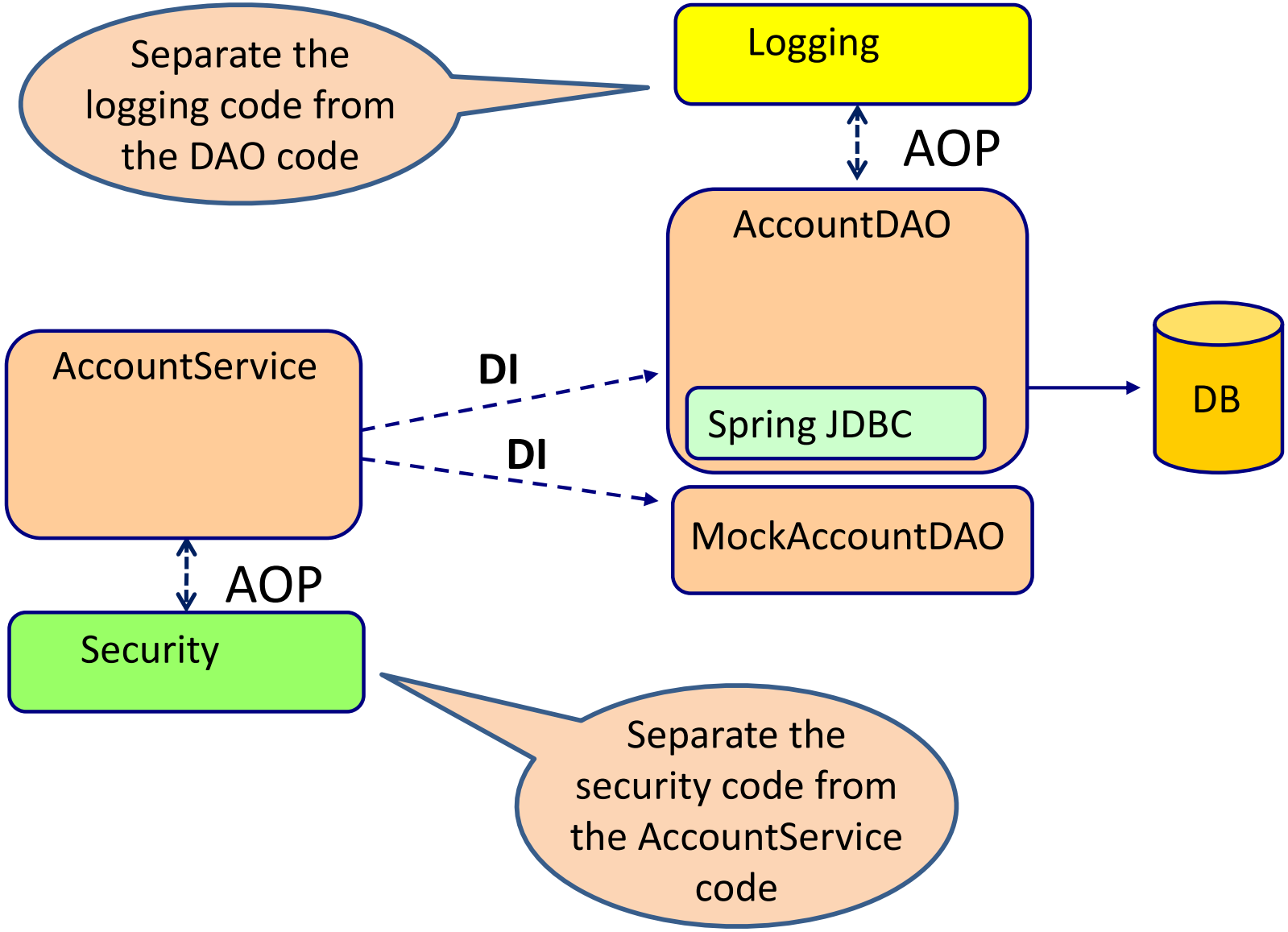
© 2022 MIU

Add Dependency Injection



© 2022 MIU

Use AOP



© 2022 MIU

Spring ecosystem

- Spring (core) framework
- Spring webflow
- Spring integration
- Spring batch
- Spring security
- Spring data
- Spring cloud
- Spring boot

© 2022 MIU

SPRING BASICS

© 2022 MIU

A basic Spring application

```
package module2.helloworld;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {
  public static void main(String[] args) {
    ApplicationContext context = new
                    ClassPathXmlApplicationContext("module2/helloworld/springconfig.xml");
    CustomerService customerService = context.getBean("customerService", CustomerService.class);
    customerService.sayHello();
  }
}
```

Create an ApplicationContext based on springconfig.xml

Get the bean with id="customerService" from the ApplicationContext

```
package module2.helloworld;

public class CustomerService {
 public void sayHello(){
   System.out.println("Hello from CustomerService");
 }
}
```

springconfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id="customerService" class="module2.helloworld.CustomerService" />
</beans>
```

Bean declaration

© 2022 MIU

The spring ApplicationContext

- Reads the Spring XML configuration file
- Instantiates objects declared in the Spring configuration file
- Wires objects together with dependency injection
- Creates proxy objects when needed

© 2022 MIU

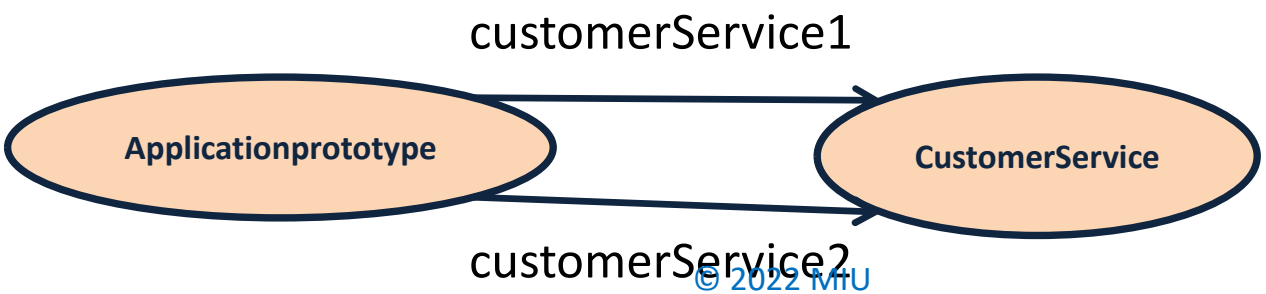
Spring beans are default singletons

```
public class Application{
  public static void main(String[] args) {
    ApplicationContext context = new
                        ClassPathXmlApplicationContext("module2/singleton/springconfig.xml");
    CustomerService customerService1 = context.getBean("customerService", CustomerService.class);
    CustomerService customerService2 = context.getBean("customerService", CustomerService.class);
    System.out.println("customerService1 ="+ customerService1);
    System.out.println("customerService2 ="+ customerService2);
  }
}
```

```
public class CustomerService {
  public CustomerService() {
  }
}
```

```
<bean id="customerService" class="module2.singleton.CustomerService" />
```

```
customerService1 =module2.singleton.CustomerService@29e357
customerService2 =module2.singleton.CustomerService@29e357
```



© 2022 MIU

Prototype beans

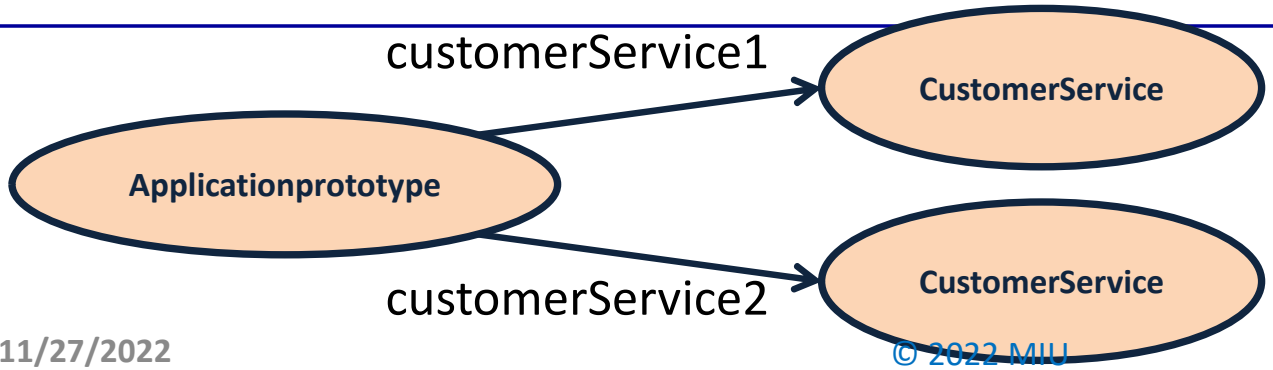
```
public class Application{
  public static void main(String[] args) {
    ApplicationContext context =
                  new ClassPathXmlApplicationContext("module2/prototype/springconfig.xml");
    CustomerService customerService1 = context.getBean("customerService", CustomerService.class);
    CustomerService customerService2 = context.getBean("customerService", CustomerService.class);
    System.out.println("customerService1 ="+ customerService1);
    System.out.println("customerService2 ="+ customerService2);
  }
}
```

```
public class CustomerService {
    public CustomerService() {
    }
}
```

```
<bean id="customerService" class="module2.prototype.CustomerService" scope="prototype" />
```

prototype

```
customerService1 =module2.prototype.CustomerService@1632847
customerService2 =module2.prototype.CustomerService@e95a56
```



© 2022 MIU

Eager-instantiation of beans

```
public class Application {
  public static void main(String[] args) {
    System.out.println("1");
    ApplicationContext context = new
           ClassPathXmlApplicationContext("/module2/eagerinstantiation/springconfig.xml");
    System.out.println("2");
    CustomerService customerService = context.getBean("customerService", CustomerService.class);
    System.out.println("3");
    customerService.addCustomer("Frank Brown");
    System.out.println("4");
  }
}
```

```
public class CustomerServiceImpl implements CustomerService {
  public CustomerServiceImpl() {
    System.out.println("calling constructor of CustomerServiceImpl");
  }

  public void addCustomer(String customername) {
    System.out.println("calling addCustomer of CustomerServiceImpl");
  }
}
```

```
<bean id="customerService" class="module2.eagerinstantiation.CustomerServiceImpl" />
```

```
1
calling constructor of CustomerServiceImpl
2
3
calling addCustomer of CustomerServiceImpl
4
```

The CustomerService bean is eagerly instantiated

© 2022 MIU

Lazy-instantiation of beans

```
public class Application {
  public static void main(String[] args) {
    System.out.println("1");
    ApplicationContext context = new
         ClassPathXmlApplicationContext("/module2/lazyinstantiation/springconfiglazy.xml");
    System.out.println("2");
    CustomerService customerService = context.getBean("customerService", CustomerService.class);
    System.out.println("3");
    customerService.addCustomer("Frank Brown");
    System.out.println("4");
  }
}
```

```
public class CustomerServiceImpl implements CustomerService {
  public CustomerServiceImpl() {
    System.out.println("calling constructor of CustomerServiceImpl");
  }

  public void addCustomer(String customername) {
    System.out.println("calling addCustomer of CustomerServiceImpl");
  }
}
```

```
<bean id="customerService" class="module2.lazyinstantiation.CustomerServiceImpl"
      lazy-init="true" />
```

Lazy instantiation

```
1
2
calling constructor of CustomerServiceImpl
3
calling addCustomer of CustomerServiceImpl
4
```

The CustomerService bean is lazy instantiated

© 2022 MIU

Lifecycle methods

```
public interface CustomerService {
  public void addCustomer(String customername);
  public void init();
  public void cleanup();
}
```

```
public class CustomerServiceImpl implements CustomerService {
  public CustomerServiceImpl() {
    System.out.println("calling constructor of CustomerServiceImpl");
  }
  public void addCustomer(String customername) {
    System.out.println("calling addCustomer of CustomerServiceImpl");
  }
  public void init() {
    System.out.println("calling init method of CustomerService");
  }
  public void cleanup() {
    System.out.println("calling cleanup method of CustomerService");
  }
}
```

```
<bean id="customerService" class="module2.xmllifecycle.CustomerServiceImpl"
      init-method="init" destroy-method="cleanup"/>
```

Method called just after the constructor

Method called when you close the ApplicationContext

© 2022 MIU

Lifecycle methods example

```
public class Application {
  public static void main(String[] args) {
    System.out.println("1");
    ConfigurableApplicationContext  context = new
                    ClassPathXmlApplicationContext("/module2/xmllifecycle/springconfig.xml");
    System.out.println("2");
    CustomerService customerService = context.getBean("customerService", CustomerService.class);
    System.out.println("3");
    customerService.addCustomer("Frank Brown");
    System.out.println("4");
    context.close();
  }
}
```

ConfigurableApplicationContext

Close the ApplicationContext

```
1
calling constructor of CustomerServiceImpl
calling init method of CustomerService
2
3
calling addCustomer of CustomerServiceImpl
4
calling cleanup method of CustomerService
```

init method

cleanup method

© 2022 MIU

Lifecycle methods with annotations

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class CustomerServiceImpl implements CustomerService {
  public CustomerServiceImpl() {
    System.out.println("calling constructor of CustomerServiceImpl");
  }
  public void addCustomer(String customername) {
    System.out.println("calling addCustomer of CustomerServiceImpl");
  }
  @PostConstruct
  public void init() {
    System.out.println("calling init method of CustomerService");
  }
  @PreDestroy
  public void cleanup() {
    System.out.println("calling cleanup method of CustomerService");
  }
}
```

@PostConstruct

@PreDestroy

```
1
calling constructor of CustomerServiceImpl
calling init method of CustomerService
2
3
calling addCustomer of CustomerServiceImpl
4
calling cleanup method of CustomerService
```

init method

cleanup method

© 2022 MIU

Lifecycle methods with annotations

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
          "http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:annotation-config/>
   <bean id="customerService" class="module2.annotationslifecycle.CustomerServiceImpl"/>
</beans>
```

Tells Spring to check for annotation configuration in the Spring beans

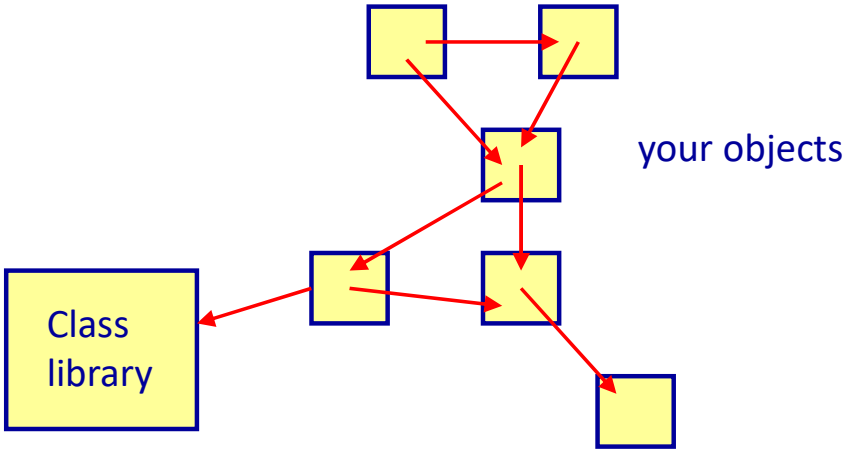
© 2022 MIU

DEPENDENCY INJECTION

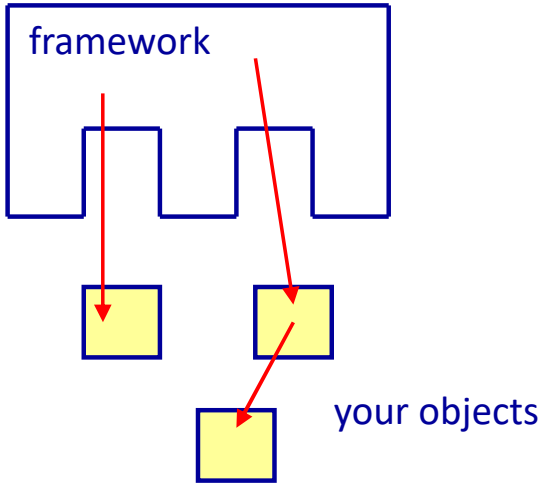
© 2022 MIU

Inversion of Control (IoC)

- Hollywood principle: Don't call us, we'll call you
- The framework has control over your code



Your code calls the class library



IoC: The framework calls your code

© 2022 MIU

Different way's to "wire"2 object together

1. Instantiate an object directly
2. Use an interface
3. Use a factory object
4. Use Spring Dependency Injection

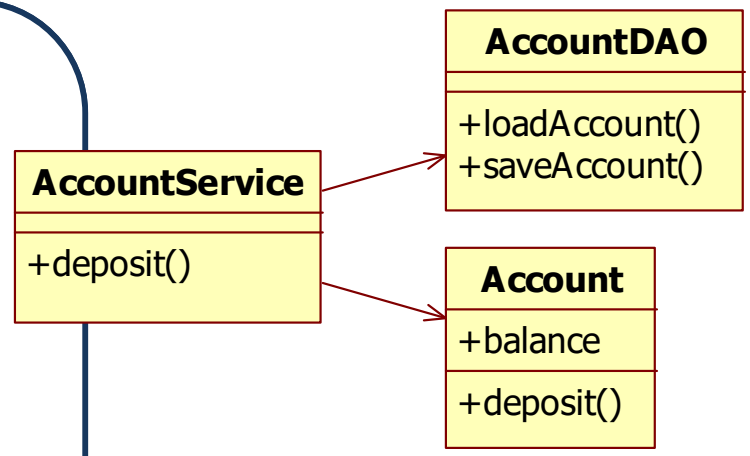
© 2022 MIU

1. Instantiate an object directly

```
public class AccountService {
  private AccountDAO accountDAO;

  public AccountService() {
    accountDAO = new AccountDAO();
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```



- The relation between AccountService and AccountDAO is hard coded
  - If you want to change the AccountDAO implementation, you have to change the code

© 2022 MIU

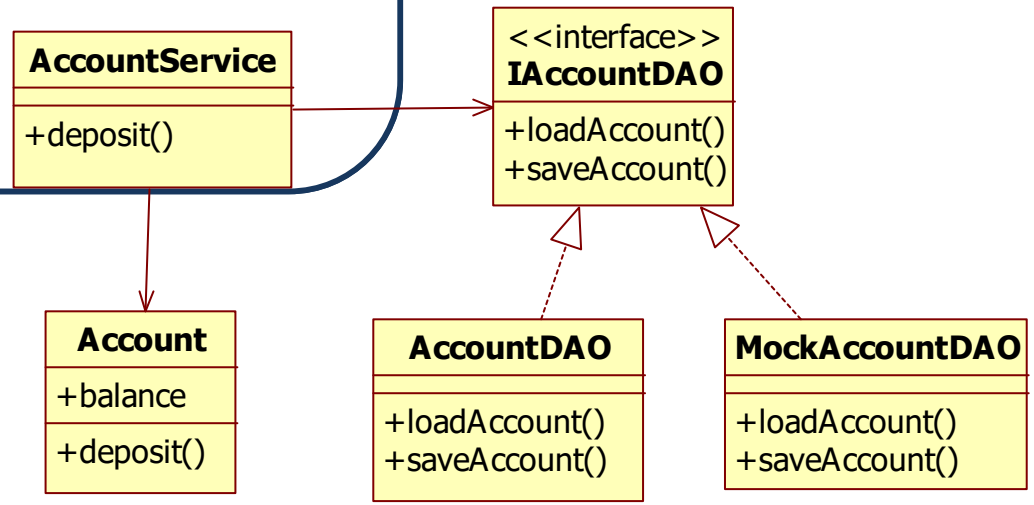
2. Use an Interface

```
public class AccountService {
  private IAccountDAO accountDAO;

  public AccountService() {
    accountDAO = new AccountDAO();
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

accountDAO is of type IAccountDAO



- The relation between AccountService and AccountDAO is still hard-coded
  - We have more flexibility, but if you want to change the AccountDAO implementation to the MockAccountDAO, you have to change the code

© 2022 MIU

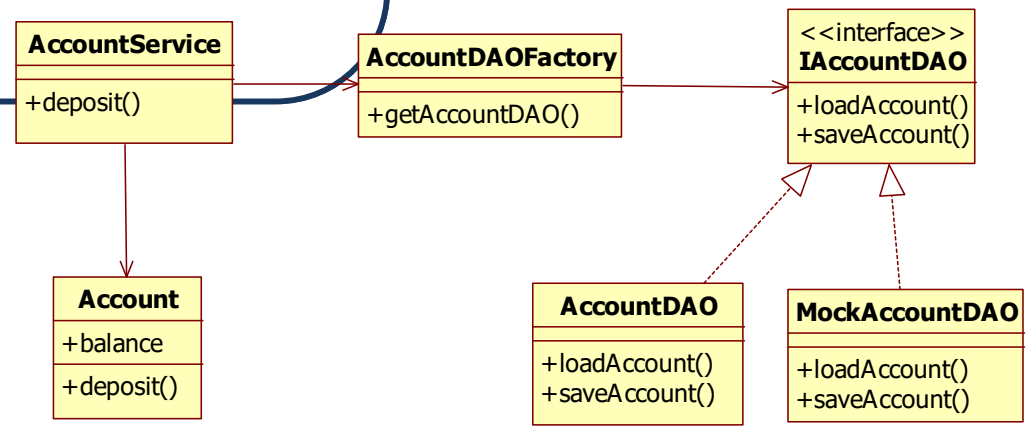
3. Use a factory object

```
public class AccountService {
  private IAccountDAO accountDAO;

  public AccountService() {
    AccountDAOFactory daoFactory = new AccountDAOFactory();
    accountDAO = daoFactory.getAccountDAO();
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

The factory creates
The accountDAO object



- The relation between AccountService and AccountDAO is still hard coded
  - We have more flexibility, but if you want to change the AccountDAO implementation to the MockAccountDAO, you have to change code in the factory

© 2022 MIU

4. Use Spring Dependency Injection

```
public class AccountService {
  private IAccountDAO accountDAO;

  public void setAccountDAO(IAccountDAO accountDAO) {
    this.accountDAO = accountDAO;
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

accountDAO is injected by the Spring framework

```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="accountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
<bean id="mockAccountDAO" class="MockAccountDAO" />
```

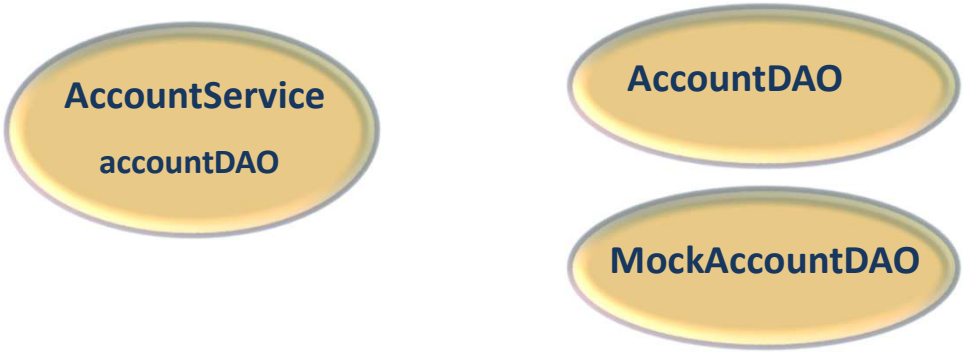
- The attribute accountDAO is configured in XML and the Spring framework takes care that accountDAO references the AccountDAO object.

© 2022 MIU

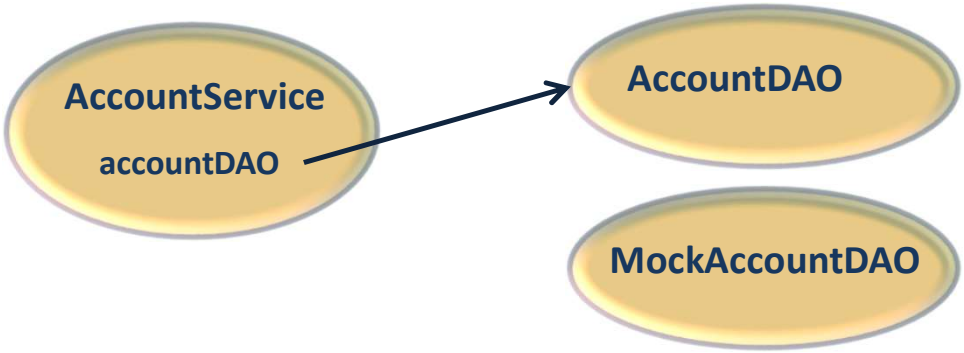
How does DI work?

```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="accountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
<bean id="mockAccountDAO" class="MockAccountDAO" />
```

1. Spring instantiates all beans in the XML configuration file



2. Spring then connects the accountDAO attribute to the AccountDAO instance

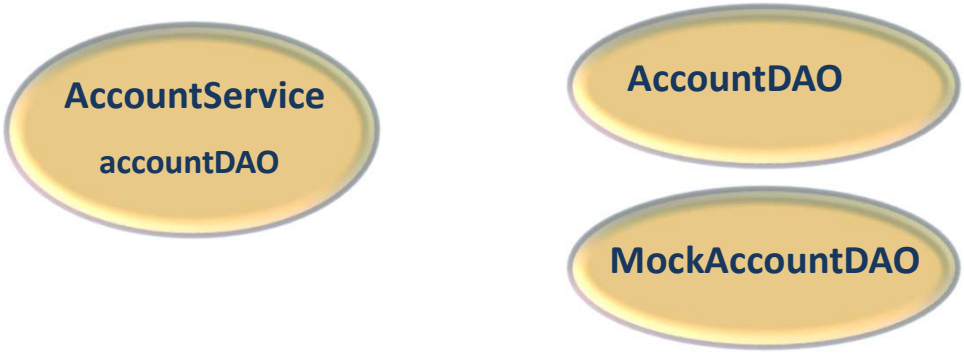


© 2022 MIU

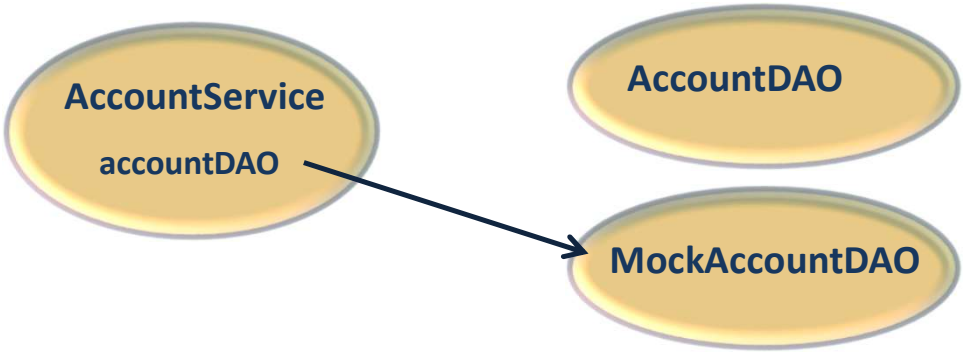
Change the wiring

```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="mockAccountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
<bean id="mockAccountDAO" class="MockAccountDAO" />
```

1. Spring instantiates all beans in the XML configuration file



2. Spring then connects the accountDAO attribute to the MockAccountDAO instance



© 2022 MIU

Advantages of Dependency Injection

```
public class AccountService {
  private IAccountDAO accountDAO;

  public void setAccountDAO(IAccountDAO accountDAO) {
    this.accountDAO = accountDAO;
  }
}
```

```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="accountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
```

- Flexibility: it is easy to change the wiring between objects without changing code
- Unit testing becomes easier
- Code is clean

© 2022 MIU

Main point

- With dependency injection the framework wires objects together.

  Science of Consciousness: Everything in creation is connected at the level of the Unified Field.

© 2022 MIU

DIFFERENT TYPES OF DI

© 2022 MIU

59

Types of DI

- Setter injection
- Constructor injection
- Autowiring

© 2022 MIU

Setter Injection

```
public class AccountService {
  private IAccountDAO accountDAO;

  public void setAccountDAO(IAccountDAO accountDAO) {
    this.accountDAO = accountDAO;
  }
}
```

A setter method is needed

```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="accountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
```

Use <property …/>

© 2022 MIU

Constructor Injection

```
public class AccountService {
  private IAccountDAO accountDAO;

  public AccountService(IAccountDAO accountDAO) {
    this.accountDAO = accountDAO;
  }
}
```

A constructor is needed to set accountDAO

```
<bean id="accountService" class="AccountService">
  <constructor-arg  ref="accountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
```

Use <constructor-arg …/>

© 2022 MIU

Constructor with multiple parameters

```
public class PaymentService implements IPaymentService{
  private IVisaVerifier visaVerifier;
  private IMastercardVerifier mastercardVerifier;

  public PaymentService(IVisaVerifier visaVerifier, IMastercardVerifier mastercardVerifier){
    this.visaVerifier=visaVerifier;
    this.mastercardVerifier=mastercardVerifier;
  }
}
```

Constructor has 2 arguments of a different type

```
<bean id="paymentService" class="products.PaymentService">
    <constructor-arg   ref="visaVerifier" />
    <constructor-arg   ref="mastercardVerifier" />
</bean>
<bean id="visaVerifier" class="products.VisaVerifier"/>
<bean id="mastercardVerifier"  class="products.MastercardVerifier"/>
```

Spring looks at the type of the argument to decide what to inject for the first and the second parameter

© 2022 MIU

Constructor with multiple parameters of the same type

```
public class PaymentService implements IPaymentService{
  private ICreditCardVerifier visaVerifier;
  private ICreditCardVerifier mastercardVerifier;

  public PaymentService(ICreditCardVerifier visaVerifier, ICreditCardVerifier mastercardVerifier){
    this.visaVerifier=visaVerifier;
    this.mastercardVerifier=mastercardVerifier;
  }
}
```

Constructor has 2 arguments of the same type

```
<bean id="paymentService" class="products.PaymentService">
    <constructor-arg   ref="visaVerifier" />
    <constructor-arg   ref="mastercardVerifier" />
</bean>
<bean id="visaVerifier" class="products.VisaVerifier"/>
<bean id="mastercardVerifier"  class="products.MastercardVerifier"/>
```

Spring looks at the order of declaration to decide what to inject for the first and the second parameter

© 2022 MIU

Constructor with multiple parameters of the same type

```
public class PaymentService implements IPaymentService{
  private ICreditCardVerifier visaVerifier;
  private ICreditCardVerifier mastercardVerifier;

  public PaymentService(ICreditCardVerifier visaVerifier, ICreditCardVerifier mastercardVerifier){
    this.visaVerifier=visaVerifier;
    this.mastercardVerifier=mastercardVerifier;
  }
}
```

Constructor has 2 arguments of the same type

```
<bean id="paymentService" class="products.PaymentService">
    <constructor-arg  index="0"  ref="visaVerifier" />
    <constructor-arg  index="1"  ref="mastercardVerifier" />
</bean>
<bean id="visaVerifier" class="products.VisaVerifier"/>
<bean id="mastercardVerifier"   class="products.MastercardVerifier"/>
```

Spring looks at the index to decide what to inject for the first and the second parameter

© 2022 MIU

Setter injection characteristics

- Order of execution:
  1. Instantiate the object
  2. Call the constructor
  3. Do the injection calling the setter method(s)
- Issues:
  - If the injection fails, you have an object in an invalid state
  - If you want to execute initialization code that uses the injected attributes, then you cannot place this code in the constructor, you need to write a separate init() method

© 2022 MIU

Constructor injection characteristics

- Order of execution:
  1. Instantiate the object
  2. Call the constructor and do the injection
- Issues:
  - You need constructor chaining with inheritance
  - In case of optional parameters you need multiple constructors

© 2022 MIU

Which one to choose?

- This is a more personal preference.
- If you need the injected attributes in the constructor, use constructor injection or use setter injection with an additional init() method.
- If constructor injection results in many different constructors, use setter injection for the optional arguments.

© 2022 MIU

Autowiring

- Spring figures out how to wire beans together
- 3 types of autowiring
  - By name
  - By Type
  - Constructor

© 2022 MIU

Autowiring by name

```java
public class CustomerService {
  private EmailService emailService;

  public void addCustomer(){
    emailService.sendEmail();
  }

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }
}
```

Autowire by name uses setter injection, so we need a setter method

```java
public class EmailService {

  public void sendEmail(){
    System.out.println("sendEmail");
  }
}
```

Spring will inject the bean with id="emailService" into the attribute 'emailService'

```xml
<bean id="customerService" class="mypackage.CustomerService" autowire="byName"/>
<bean id="emailService" class="mypackage.EmailService"/>
```


© 2022 MIU

Autowiring by type

```
public class CustomerService {
  private EmailService emailService;

  public void addCustomer(){
    emailService.sendEmail();
  }

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }
}
```

Autowire by type uses setter injection, so we need a setter method

```
public class EmailService {

  public void sendEmail(){
    System.out.println("sendEmail");
  }
}
```

Spring will inject the bean with type EmailService" into the attribute 'emailService'

```
<bean id="customerService" class="mypackage.CustomerService" autowire="byType"/>
<bean id="eService" class="mypackage.EmailService"/>
```


© 2022 MIU

Constructor autowiring

```java
public class CustomerService {
  private EmailService emailService;

  public CustomerService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void addCustomer(){
    emailService.sendEmail();
  }
}
```

The constructor has 1 attribute of type EmailService

```java
public class EmailService {

  public void sendEmail(){
    System.out.println("sendEmail");
  }
}
```

Spring will inject the bean with type EmailService" into the attribute 'emailService'

```
<bean id="customerService" class="mypackage.CustomerService" autowire="constructor"/>
<bean id="eService" class="mypackage.EmailService"/>
```


© 2022 MIU

Annotation based Autowiring by constructor

```java
public class CustomerService {
  private EmailService emailService;

  @Autowired
  public CustomerService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void addCustomer(){
    emailService.sendEmail();
  }
}
```

@Autowire indicates to Spring that the emailService attribute should be injected by type via the constructor

```java
public class EmailService {

  public void sendEmail(){
    System.out.println("sendEmail");
  }
}
```

This tag tells Spring to look for configuration annotations in the declared beans

```xml
<context:annotation-config/>
<bean id="customerService" class="mypackage.CustomerService"/>
<bean id="eService" class="mypackage.EmailService"/>
```


© 2022 MIU

Annotation based Autowiring by type

```
public class CustomerService {
  private EmailService emailService;

  @Autowired
  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void addCustomer(){
    emailService.sendEmail();
  }
}
```

@Autowire indicates to Spring that the emailService attribute should be injected by type via the setter method

```
public class EmailService {

  public void sendEmail(){
    System.out.println("sendEmail");
  }
}
```

This tag tells Spring to look for configuration annotations in the declared beans

```
<context:annotation-config/>
<bean id="customerService" class="mypackage.CustomerService"/>
<bean id="eService" class="mypackage.EmailService"/>
```


© 2022 MIU

Field injection

```java
public class CustomerService {
  @Autowired
  @Qualifier("myEmailService")
  private EmailService emailService;

  public void addCustomer(){
    emailService.sendEmail();
  }
}
```

autowire by name

```java
public class EmailService {

  public void sendEmail(){
    System.out.println("sendEmail");
  }
}
```

```xml
<context:annotation-config/>
<bean id="customerService" class="mypackage.CustomerService"/>
<bean id="myEmailService" class="mypackage.EmailService"/>
```


© 2022 MIU

Field injection

```
public class CustomerService {
  @Autowired
  private EmailService emailService;

  public void addCustomer(){
    emailService.sendEmail();
  }
}
```

```
public class CustomerService {
  @Inject
  private EmailService emailService;

  public void addCustomer(){
    emailService.sendEmail();
  }
}
```


© 2022 MIU

Autowiring

- Advantage
  - Makes configuration of bean wiring simpler
- Disadvantages
  - The Spring XML file does not contain all the explicit details on how the beans are wired together
  - Autowire by type gives the restriction that you can have only 1 bean of the given type

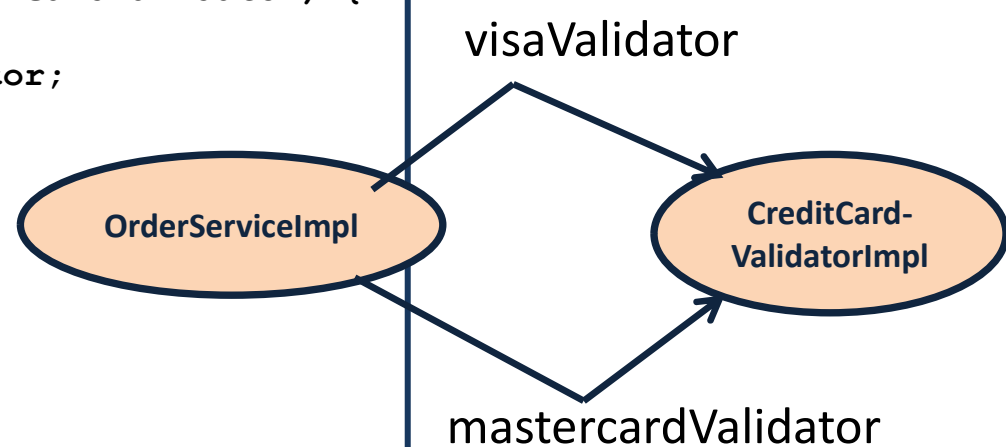
© 2022 MIU

DI and singletons

```
public class OrderServiceImpl implements OrderService {
  private CreditCardValidator visaValidator;
  private CreditCardValidator mastercardValidator;

  public OrderServiceImpl(CreditCardValidator visaValidator,
                          CreditCardValidator mastercardValidator) {
   this.visaValidator = visaValidator;
   this.mastercardValidator = mastercardValidator;
  }

  public void payOrder(CreditCard card){
    if (card.getType().equals("visa")){
      visaValidator.validate(card);
    }
    else{
      if (card.getType().equals("mastercard")){
        mastercardValidator.validate(card);
      }
    }
  }
}
```




```
<bean id="orderService" class= "OrderServiceImpl">
  <constructor-arg  index="0"  ref="creditcardVerifier" />
  <constructor-arg  index="1"  ref="creditcardVerifier" />
</bean>
<bean id="creditcardVerifier" class="CreditCardValidatorImpl"/>
```

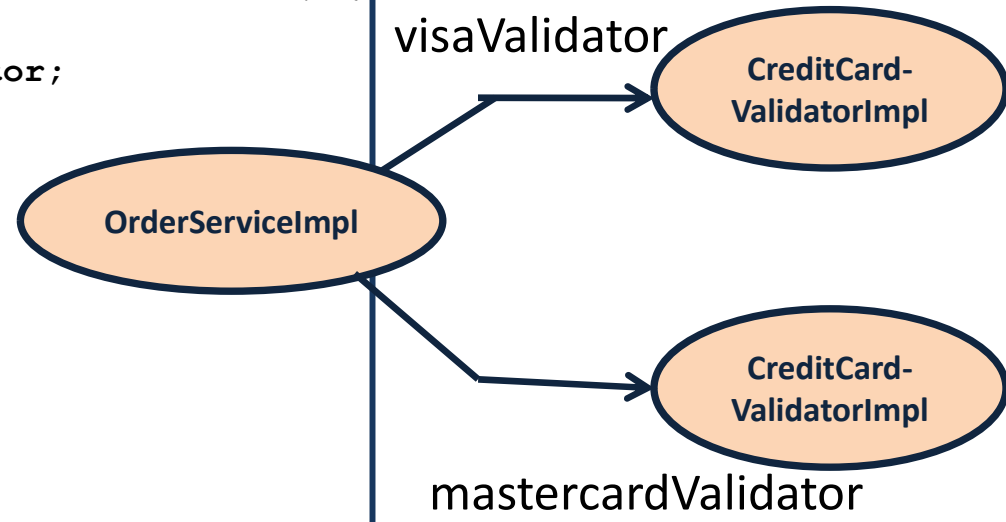

© 2022 MIU

DI and prototypes

```
public class OrderServiceImpl implements OrderService {
  private CreditCardValidator visaValidator;
  private CreditCardValidator mastercardValidator;

  public OrderServiceImpl(CreditCardValidator visaValidator,
                          CreditCardValidator mastercardValidator) {
   this.visaValidator = visaValidator;
   this.mastercardValidator = mastercardValidator;
  }

  public void payOrder(CreditCard card){
    if (card.getType().equals("visa")){
      visaValidator.validate(card);
    }
    else{
      if (card.getType().equals("mastercard")){
        mastercardValidator.validate(card);
      }
    }
  }
}
```




```
<bean id="orderService" class= "OrderServiceImpl">
  <constructor-arg  index="0"  ref="creditcardVerifier" />
  <constructor-arg  index="1"  ref="creditcardVerifier" />
</bean>
<bean id="creditcardVerifier" class="CreditCardValidatorImpl" scope="prototype"/>
```

prototype

© 2022 MIU

Injection of primitive values

```
public class CustomerServiceImpl implements CustomerService {
  private String defaultCountry;
  private long numberOfCustomers;

  public void setDefaultCountry(String defaultCountry) {
    this.defaultCountry = defaultCountry;
  }
  public String getDefaultCountry() {
    return defaultCountry;
  }
  public long getNumberOfCustomers() {
    return numberOfCustomers;
  }
  public void setNumberOfCustomers(long numberOfCustomers) {
    this.numberOfCustomers = numberOfCustomers;
  }
}
```

```
<bean id="customerService" class="mypackage.CustomerServiceImpl">
  <property name="defaultCountry" value="USA"/>
  <property name="numberOfCustomers" value="56982"/>
</bean>
```

Automatic conversion from String to long

© 2022 MIU

DEPENDENCY INJECTION WITH CLASSPATH SCANNING

© 2022 MIU

81

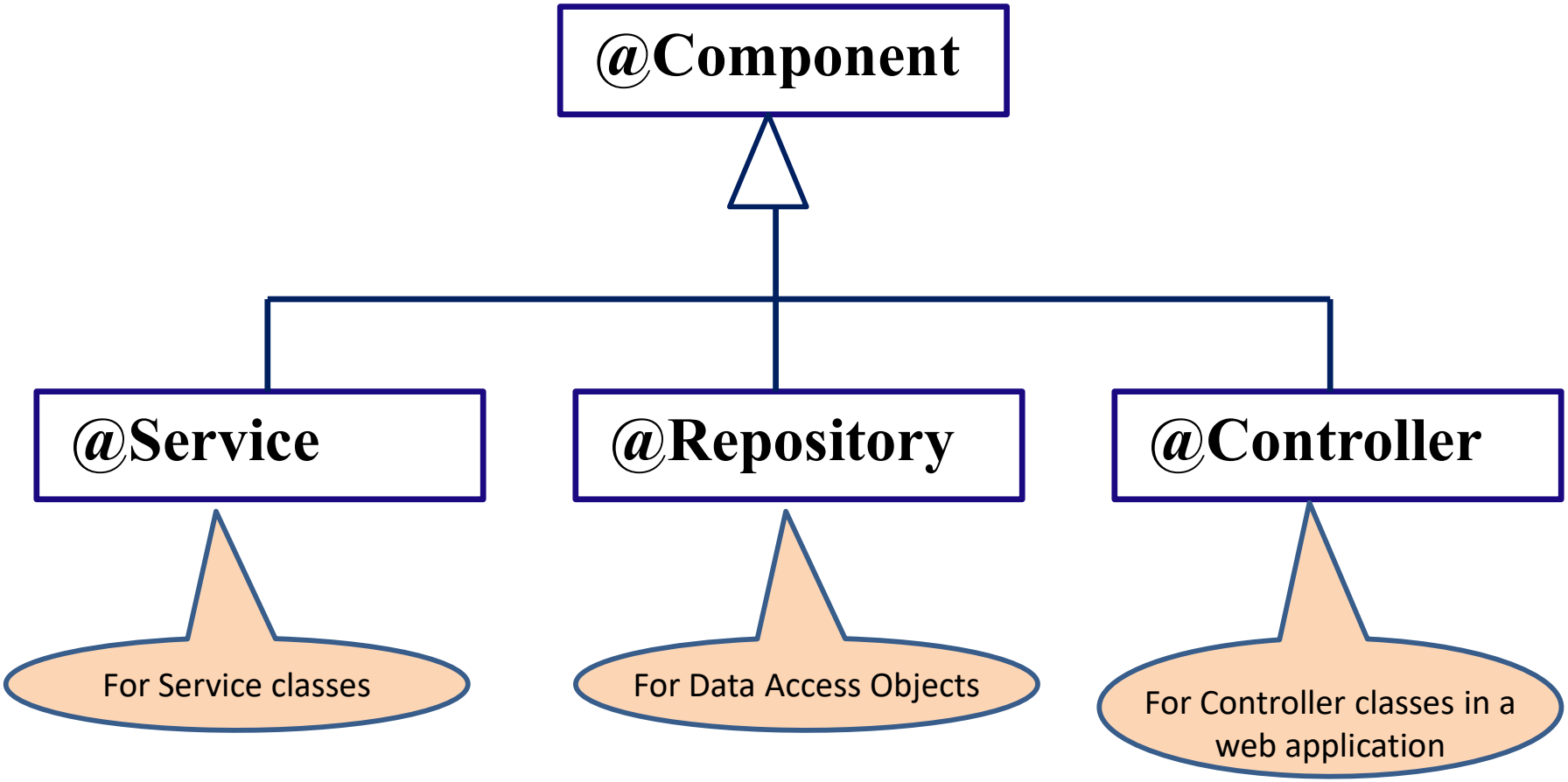
Classpath scanning

- Define beans with annotations instead of defining them with XML
- All classes with the annotations
  - @Component
  - @Service
  - @Repository
  - @Controller

  become spring beans

© 2022 MIU

Classpath scanning annotations



© 2022 MIU

Classpath scanning example (1/2)

```
@Service ("customerService")
public class CustomerServiceImpl implements CustomerService{
private EmailService emailService;

  @Autowired
  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

@Service annotation

The EmailService is injected

```
@Service ("emailService")
public class EmailService implements IEmailService {

  public void sendEmail() {
    System.out.println("sendEmail");
  }
}
```

@Service annotation

© 2022 MIU

Classpath scanning example (2/2)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <context:component-scan base-package="module3.classpathscanning.basic"/>

</beans>
```

No beans declared

```java
public class Application {
  public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");
    CustomerService customerService = context.getBean("customerService", CustomerService.class);
    customerService.addCustomer();
  }
}
```


© 2022 MIU

@Scope for autodetect components

- The default scope is “singleton”

```
@Service ("emailService")
@Scope("prototype")
public class EmailServiceImpl implements EmailService{
public void sendEmail() {
    System.out.println("sendEmail");
  }
}
```

Set the scope to prototype

© 2022 MIU

@Value

```
@Service ("emailService")
public class EmailServiceImpl implements EmailService{
  @Value("smtp.mailserver.com")
  private String emailServer;

  public void sendEmail() {
    System.out.println("send email to server: "+ emailServer);
  }
}
```

Set the Value of an attribute

© 2022 MIU

DEPENDENCY INJECTION WITH JAVA CONFIGURATION

© 2022 MIU

88

Java Configuration

- Spring beans can also be configured in Java (and annotations) instead of XML

```
@Configuration
public class AppConfig {
  @Bean
  public CustomerService customerService(){
    CustomerService customerService = new CustomerServiceImpl();
    customerService.setEmailService(emailService());
    return customerService;
  }
  @Bean
  public EmailService emailService(){
    return new EmailServiceImpl();
  }
}
```

Similar configuration

```
<bean id="customerService" class="module3.di.CustomerServiceImpl">
  <property name="emailService" ref="emailService"/>
</bean>
<bean id="emailService" class="module3.di.EmailServiceImpl" />
```


© 2022 MIU

Java configuration example (1/2)

```
public class CustomerServiceImpl implements CustomerService{
private EmailService emailService;

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

```
public class EmailService implements IEmailService {

  public void sendEmail() {
    System.out.println("sendEmail");
  }
}
```


© 2022 MIU

Java configuration example (2/2)

```
@Configuration
public class AppConfig {
  @Bean
  public CustomerService customerService(){
    CustomerService customerService = new CustomerServiceImpl();
    customerService.setEmailService(emailService());
    return customerService;
  }
  @Bean
  public EmailService emailService(){
    return new EmailServiceImpl();
  }
}
```

Create a bean with the name "customerService"

Set the property emailService

AnnotationConfigApplicationContext

```
public class Application {
  public static void main(String[] args) {
    ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
    CustomerService customerService =
         context.getBean("customerService", CustomerService.class);

    customerService.addCustomer();
  }
}
```


© 2022 MIU

@Lazy

```
@Configuration
public class AppConfig {
  @Bean
  @Lazy(true)
  public CustomerService customerService(){
    return new CustomerServiceImpl();
  }
  @Bean
  @Lazy(true)
  public EmailService emailService(){
    return new EmailServiceImpl();
  }
}
```

Lazy initialization

© 2022 MIU

@Scope

```
@Configuration
public class AppConfig {
  @Bean
  public CustomerService customerService(){
    return new CustomerServiceImpl();
  }

  @Bean
  @Scope(value="prototype")
  public EmailService emailService(){
    return new EmailServiceImpl();
  }
}
```

Set scope to prototype

© 2022 MIU

Configuration in Configuration file(s) and in the Spring beans

```
@Configuration
public class AppConfig {
  @Bean
  public CustomerService customerService(){
    return new CustomerServiceImpl();
  }
  @Bean
  public EmailService emailService(){
    return new EmailServiceImpl();
  }
}
```

Definition of 2 Spring beans

```
public class CustomerServiceImpl implements CustomerService{
  private EmailService emailService;

  @Autowired
  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

The EmailService is injected

© 2022 MIU

3 WAYS TO CONFIGURE SPRING APPLICATIONS

© 2022 MIU

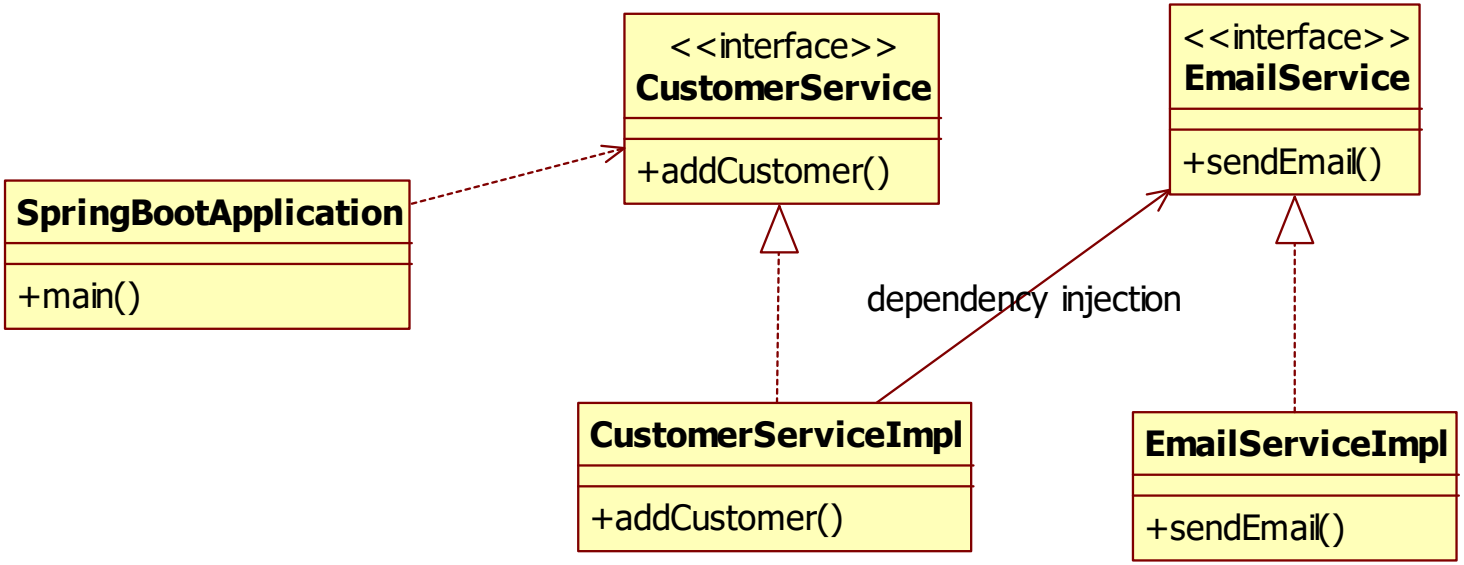
95

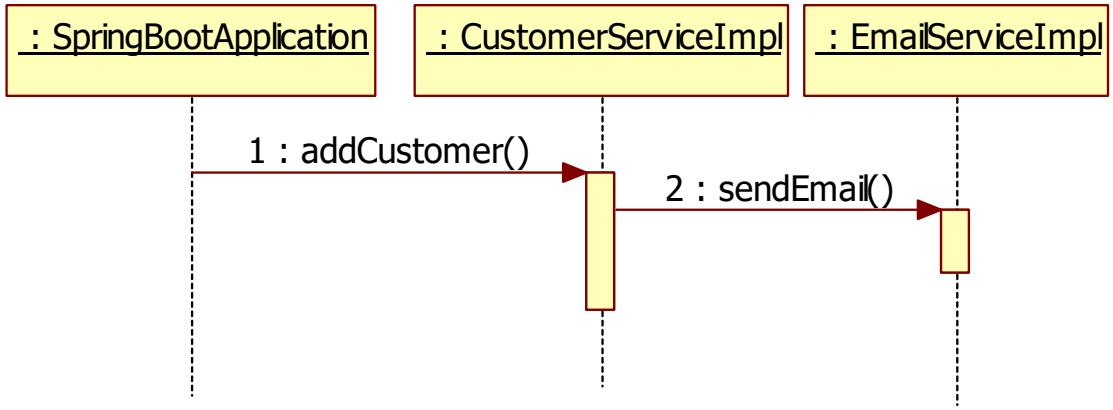
3 ways of Spring configuration

- XML configuration
- Classpath scanning and Autowiring
- Java configuration

© 2022 MIU

Example application





© 2022 MIU

The implementation

```java
public interface EmailService {
  void sendEmail();
}
```

```java
public class EmailServiceImpl implements EmailService{
  public void sendEmail() {
    System.out.println("Sending email");
  }
}
```

```java
public interface CustomerService {
  void addCustomer();
}
```

```java
public class CustomerServiceImpl implements CustomerService {

  private EmailService emailService;

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }
  public void addCustomer() {
    emailService.sendEmail();
  }
}
```


© 2022 MIU

Option 1: XML configuration

```xml
<bean id="customerService" class="xml.CustomerServiceImpl">
  <property name="emailService" ref="emailService" />
</bean>
<bean id="emailService" class="xml.EmailServiceImpl" />
```


```java
public class CustomerServiceImpl implements CustomerService {

  private EmailService emailService;

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }
  public void addCustomer() {
    emailService.sendEmail();
  }
}
```


© 2022 MIU

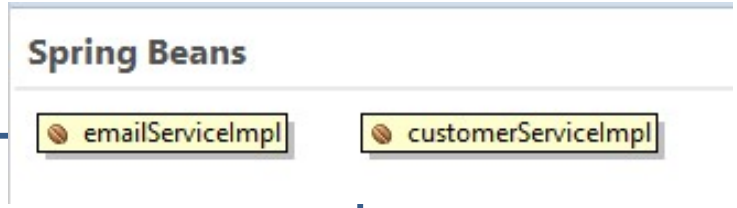
XML configuration

- Advantages
  - Configuration separate from Java code
  - All configuration in one place
  - Tools can use the XML for graphical views
  - Easy to change the configuration

- Disadvantages
  - Large verbose XML file(s)
  - No compile time type safety
  - Less refactor-friendly

© 2022 MIU

Option 2: Classpath scanning and Autowiring




```
<context:component-scan base-package="scanning"/>
<context:annotation-config />
```

```
@Service
public class CustomerServiceImpl implements CustomerService {
  @Autowired
  private EmailService emailService;

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```


© 2022 MIU

Classpath scanning and Autowiring

- Advantages
  - All information (configuration and logic) in one place: the Java code
  - Simpler as XML
  - More type safe
- Disadvantage
  - Configuration in the Java code
  - Configuration is harder to change
    - Not a clear overview
    - You have to recompile

© 2022 MIU

Option 3: Java configuration

```java
@Configuration
public class AppConfig {
  @Bean
  public CustomerService customerService(){
    CustomerService customerService = new CustomerServiceImpl();
    customerService.setEmailService(emailService());
    return customerService;
  }
  @Bean
  public EmailService emailService(){
    return new EmailServiceImpl();
  }
}
```

```java
public class CustomerServiceImpl implements CustomerService {

  private EmailService emailService;

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }
  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

Java configuration

- Advantages
  - Configuration separate from Java code
  - Simpler as XML
  - Type safe
- Disadvantage
  - Requires a little bit more code
  - Configuration is harder to change
    - Not a clear overview
    - You have to recompile

© 2022 MIU

Simpler configuration

- Java config + autowiring

```
@Configuration
public class AppConfig {
  @Bean
  public CustomerService customerService(){
    return new CustomerServiceImpl();
  }
  @Bean
  public EmailService emailService(){
    return new EmailServiceImpl();
  }
}
```

```
public class CustomerServiceImpl implements CustomerService {
  @Autowired
  private EmailService emailService;

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```


© 2022 MIU

Simplest Configuration!

- Java config + classpath scanning + autowiring

```
@Configuration
@ComponentScan
public class AppConfig {
}
```

```
@Service
public class CustomerServiceImpl implements CustomerService {
  @Autowired
  private EmailService emailService;

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

```
@Service
public class EmailServiceImpl implements EmailService{
  public void sendEmail() {
    System.out.println("Sending email");
  }
}
```

Main point

- The Spring configuration tells the Spring framework which classes to instantiate and which classes to connect to each other with dependency injection.

  Science of Consciousness: Nature is configured in such a way that it always takes the path of least action.

© 2022 MIU

Connecting the parts of knowledge with the wholeness of knowledge

1. Spring instantiates all Spring beans and wires them together with dependency injection
2. The simplest way to configure a Spring application is with Java config + classpath scanning + autowiring

3. **Transcendental consciousness** is the direct experience of pure consciousness, the unified field of all the laws of nature.
4. **Wholeness moving within itself:** In unity consciousness, one appreciates the inherent underlying unity that underlies all the diversity of creation.

© 2022 MIU