CS544
Enterprise Architecture Final
Exam 2 May 2016

Name____

Student ID ____

NOTE: This material is private and confidential. It is the property of MUM and is not to be disseminated.

1. [10 points]Determine which of the following are TRUE/FALSE concerning Security :

T **F** Spring security only supports the authentication model HTTP Basic defined by RFC 1945 which is the most popular authentication mechanism in the web.

EXPLAIN:

Spring security supports a wide range of authentication models such as HTTP Basic, HTTP Digest, HTTP X.509, etc.

T **F** Spring ACL is used to give permissions to access methods (i.e. specify who can execute which method)

EXPLAIN:

That is what the Authorization does. ACL instead is used to give permissions for a specific domain model INSTANCE.

T **F** Digest authentication uses Base64 encoding to transmit encrypted usernam/password

EXPLAIN:

Basic authentication transmits username/password as Base64 encoding.

T **F** Authorization refers to unique identifying information from each system user, generally in the form of a username and password.

EXPLAIN:

That is authentication. Authorization refers to the process of allowing or denying individual access to resources.

T F Access control list (ACL) refers to domain specific authorizations and CRUD operations

EXPLAIN:

Access Control List(ACL) defines access levels on each separate object

2. [15 points] AOP is a Spring Core Technology. It is used in numerous places with the Spring Framework, itself. Explain the fundamentals of Spring's AOP implementation; how it works, how it relates to AspectJ, examples of its usage within Spring.

   To help in your explanation of how it works consider the following use case:

   A client application needs to access a server application over the network. For monitoring purposes, it is necessary to log calls to save [save(Object object)] methods at the service tier.
   For example:

```
Class FooServiceImpl {

        public void save (Foo foo) {
           fooDao.save (foo);
        }

        Public List<Foo> findAll() {
           return fooDao.findAll();
        }

        Public Foo findOne(Long id) {
           return fooDao.findAll(id);
        }
}
```

   Using AOP terminology, describe what would need to be implemented. Be as specific as you can with respect to syntax.

   ANSWER:

```
@Pointcut("execution(* edu.mum.service..save(..))")
public void applicationMethod() {}

@Pointcut("args(object)")
public void argsMethod(Object object) {}

@Before(applicationMethod() && argsMethod(object)")
public void  doLogging( Object object) throws Throwable {
```

AOP Value Added

- Separation of Concerns
- Increased Modularity
- Reduces "spaghetti" code
- Code reduction
- Removes "hard" dependencies

- **USE CASES:**
  - **Boilerplate/repetitive code** – unable to be refactored using normal OO techniques
  - **Transaction**
  - **Security**
  - **Logging**

The General AOP Use Case: Scattering & Tangling

- A functional implementation is **scattered** if its code is spread out over multiple modules. Its implementation is not modular.

  Logging is "scattered" throughout an application

- A functional implementation is **tangled** if its code is intermixed with code that implements other functionality. The module in which tangling occurs is not cohesive.

  [Programmatic] Transaction Management is "tangled" within a method

Aspect-oriented approach identifies code scattering and tangling as the indicators of crosscutting concerns.

AOP Definitions

Cross-cutting Concern

Another name for an **Aspect.** An Aspect "crosscuts" core functionality – basically, violates Separation of Concerns [**unless "isolated"**]

- **Aspect**

  Functionality fundamental to application
  BUT not the primary business function.
  Aspect is to AOP as Class is to OOP.

AOP Definitions [Cont.]

- **Advice**
  - Implementation code of the aspect.
  - [executed Around, Before or After **Join point**]
  - [Associated with **Join Point** through a **Pointcut**]
- **Join point**
  - Where **Advice** code in applied
  - [Always class methods in Spring AOP]
- **Pointcut**
  - An expression that defines a set of **Join points**

- **Aspect** - implemented by applying **Advice** (additional behavior) at various **Join points** (methods in Spring application) specified by a **Pointcut** (criteria to match **Join points** to **Advice**).

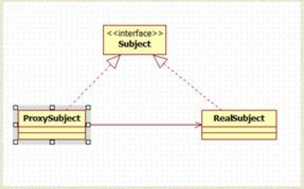
Static & Dynamic AOP in Spring

- **Static [AspectJ]**
  - Cross-cutting logic is applied at compile time
  - Byte code modification [better performance]
  - Aspect can be applied to fields [more join points]
  - Any Java code

Dynamic [Spring AOP]
  - Cross-cutting logic applied at run time
  - **Proxy based** approach [simple to use]
  - Aspects applied to methods only
  - Spring Managed beans

NOTE: @AspectJ is a subset of AspectJ that declares AOP annotations. We will use this subset with Spring AOP without FULL AspectJ implementation

Spring AOP - Proxy Pattern



- **Subject** - Interface implemented by the RealSubject
- **Proxy** - Controls access to the RealSubject
- **Real Subject** - the real object that the proxy represents.

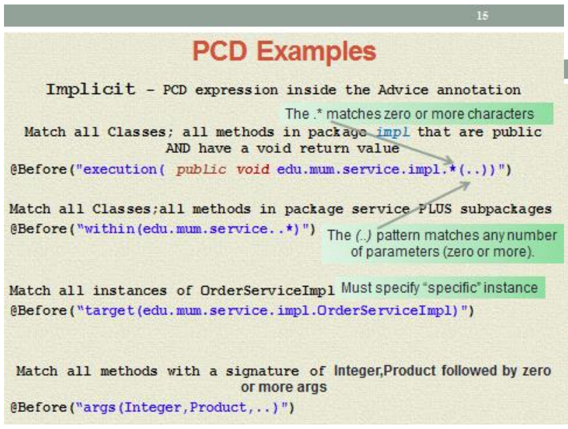
ADVICE TYPES

ADVICE	DESCRIPTION
@Before	executes before a join point
@AfterReturning	Executes if a join point completes normally
AfterThrowing	executes if a join point throws an exception
@After	executes if a join point executes normally OR throws an exception
@Around	Before AND after the join point. Also can end execution or throw exception

See AspectJ Programming Guide

Point Cut Designators [PCD]

ADVICE	DESCRIPTION	SYNTAX
execution	Matches methods Including visibility,return & parameters	`("execution(public * *.*.*(..))")`
within	matches join points within certain types	`("within(*.*.*.*)")`
target	Matches where the target object is an **instance** of the given type	`("target(pkg.pkg.pkg.class)")`
args	Matches where the arguments are instances of the given types	`("args(..)")`
@annotation	Matches methods where the given annotation exists	`@annotation(anotationName)`



PCD Examples

Implicit – PCD expression inside the Advice annotation

The .* matches zero or more characters

Match all Classes; all methods in package impl that are public AND have a void return value

```
@Before("execution( public void edu.mum.service.impl.*(..))")
```

Match all Classes;all methods in package service PLUS subpackages

```
@Before("within(edu.mum.service..*)")
```

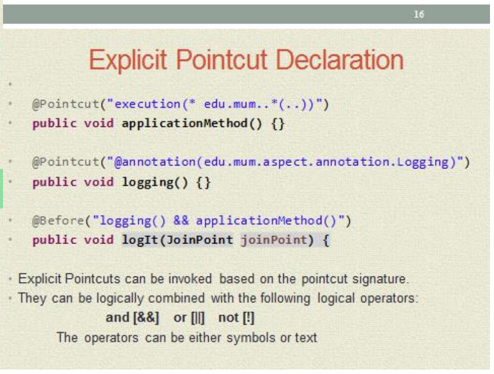
The (..) pattern matches any number of parameters (zero or more).

Match all instances of OrderServiceImpl Must specify "specific" instance

```
@Before("target(edu.mum.service.impl.OrderServiceImpl)")
```

Match all methods with a signature of Integer,Product followed by zero or more args

```
@Before("args(Integer,Product,..)")
```



Explicit Pointcut Declaration

```
@Pointcut("execution(* edu.mum..*(..))")
public void applicationMethod() {}

@Pointcut("@annotation(edu.mum.aspect.annotation.Logging)")
public void logging() {}

@Before("logging() && applicationMethod()")
public void logIt(JoinPoint joinPoint) {
```

- Explicit Pointcuts can be invoked based on the pointcut signature.
- They can be logically combined with the following logical operators:

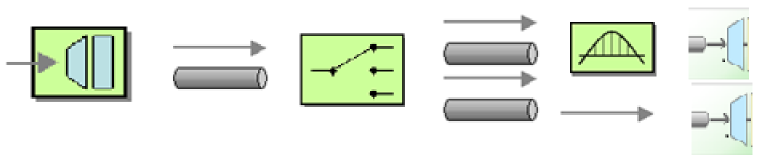
and [&&] or [||] not [!]

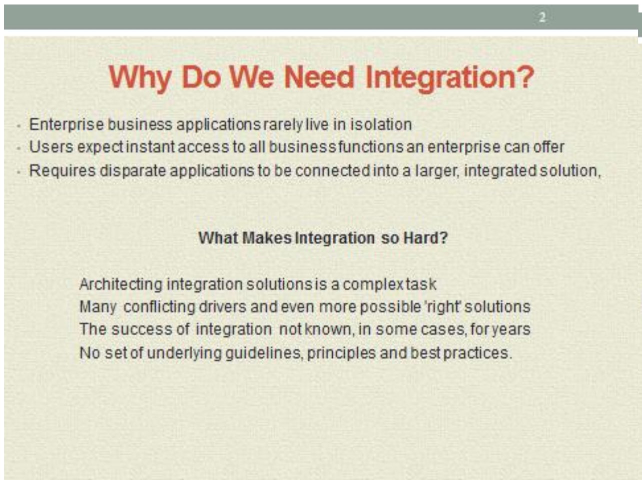
The operators can be either symbols or text

3. [20 points] Enterprise Integration Patterns [EIP] are a fundamental definition of how to do integration in a company of any significant size. Spring Integration implements those patterns.

   Explain the fundamental aspects of Spring Integration. Why is it necessary & valuable? Describe the 3 main components. Drawing on the demo from class [Routing an order through the "enterprise"], give details on some of the EIP components.

   Be specific. Give examples. Diagrams are good but be sure to explain them.
   Here is a diagram that you should use to describe [some] components and an ESB type flow:



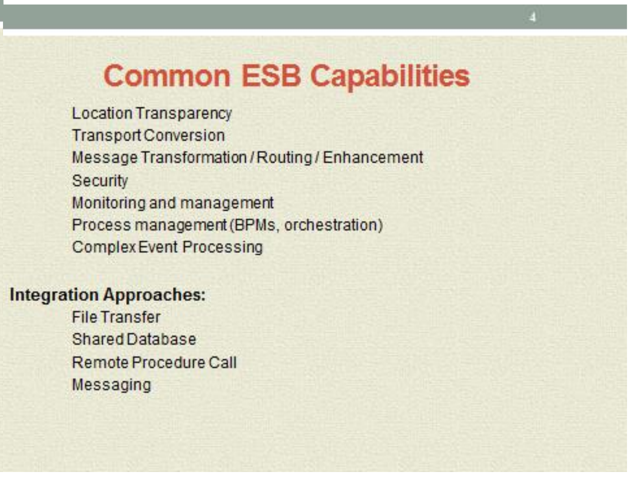


Why Do We Need Integration?

- Enterprise business applications rarely live in isolation
- Users expect instant access to all business functions an enterprise can offer
- Requires disparate applications to be connected into a larger, integrated solution,

What Makes Integration so Hard?

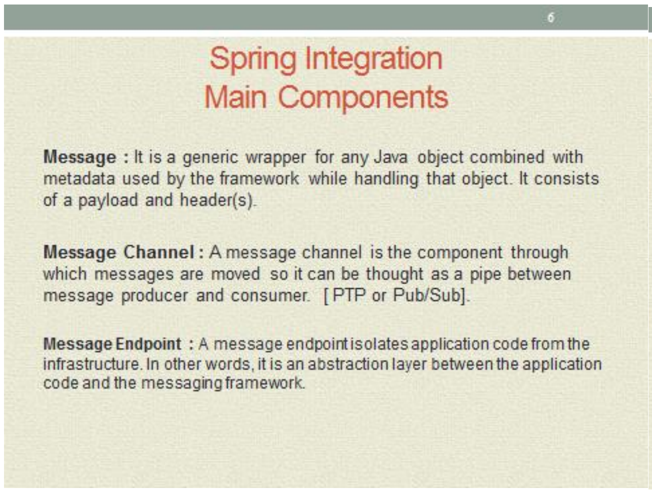
Architecting integration solutions is a complex task
Many conflicting drivers and even more possible 'right' solutions
The success of integration not known, in some cases, for years
No set of underlying guidelines, principles and best practices.



Common ESB Capabilities

Location Transparency
Transport Conversion
Message Transformation / Routing / Enhancement
Security
Monitoring and management
Process management (BPMs, orchestration)
Complex Event Processing

Integration Approaches:
File Transfer
Shared Database
Remote Procedure Call
Messaging

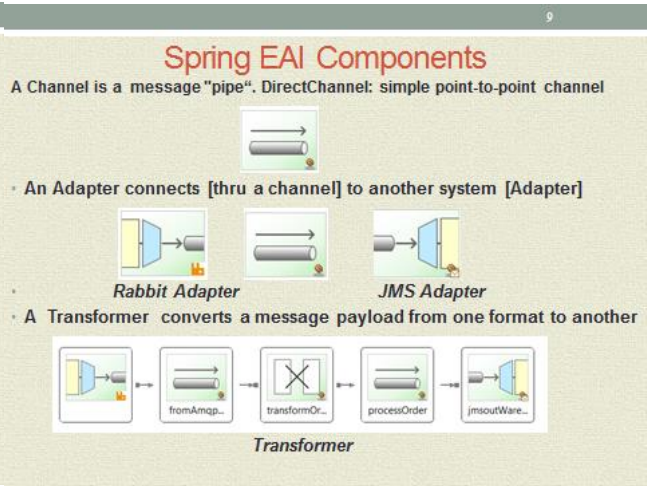


Spring Integration Main Components

Message : It is a generic wrapper for any Java object combined with metadata used by the framework while handling that object. It consists of a payload and header(s).

Message Channel : A message channel is the component through which messages are moved so it can be thought as a pipe between message producer and consumer. [PTP or Pub/Sub].

Message Endpoint : A message endpoint isolates application code from the infrastructure. In other words, it is an abstraction layer between the application code and the messaging framework.



Spring EAI Components

A Channel is a message "pipe". DirectChannel: simple point-to-point channel

- An Adapter connects [thru a channel] to another system [Adapter]

Rabbit Adapter *JMS Adapter*

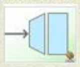
- A Transformer converts a message payload from one format to another

fromAmqp...
transformOr...
processOrder
jmsoutWare...

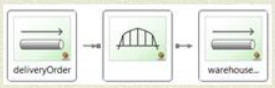
Transformer

[More] Spring EAI Components

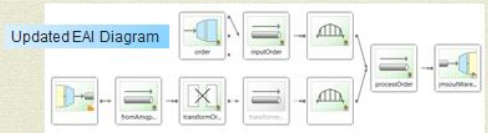
- A Gateway hides Messaging API from Application



- A Bridge connects two channels

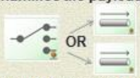


Bridge



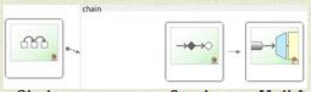
[More] Spring EAI Components

- A Content Router examines the payload to determine message channel



- A recipient-list-router distributes the message to all message channels

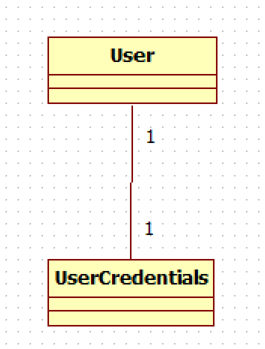
- A chain is a convenience reduce channels between endpoints
- A Service Activator triggers (or activates) a Spring-managed bean



Chain Service Activator Mail Adapter

4. [15 Points] Annotate for validation both the User and UserCredentials from the Auction System. The action that triggers validation is an invocation of - userService.save(user);

   It is NOT necessary to invoke validation, just annotate the domain Models AND externalize the messages in errorMessages.properties

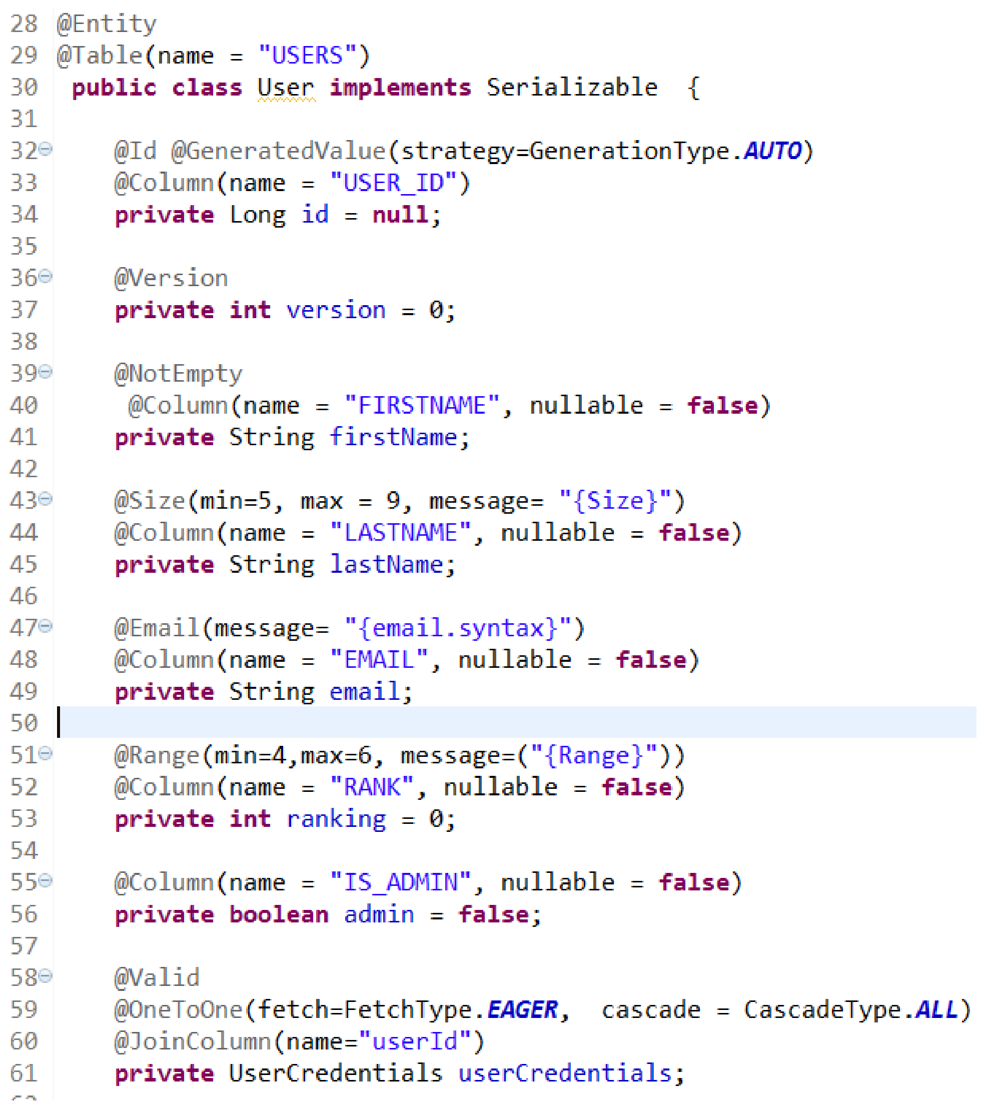


   Here are the generated error messages when validation fails:

```
First Name field must have a value
Size of the Last Name must be between 5 and 9
Email must have valid syntax
Ranking must be between 4 and 6

Password must have at least 6 characters
Size of the Login User Name must be between 6 and 16
```

Here is the relevant part of the User Domain Class:

```
@Entity
@Table(name = "USERS")
 public class User implements Serializable  {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "USER_ID")
    private Long id = null;



    @Version
    private int version = 0;



    @Column(name = "FIRSTNAME", nullable = false)
    private String firstName;



    @Column(name = "LASTNAME", nullable = false)
    private String lastName;



    @Column(name = "EMAIL", nullable = false)
    private String email;



    @Column(name = "RANK", nullable = false)
    private int ranking = 0;



    @Column(name = "IS_ADMIN", nullable = false)
    private boolean admin = false;




       @OneToOne(fetch=FetchType.EAGER,  cascade = CascadeType.ALL)
       @JoinColumn(name="userId")
       private UserCredentials userCredentials;
```

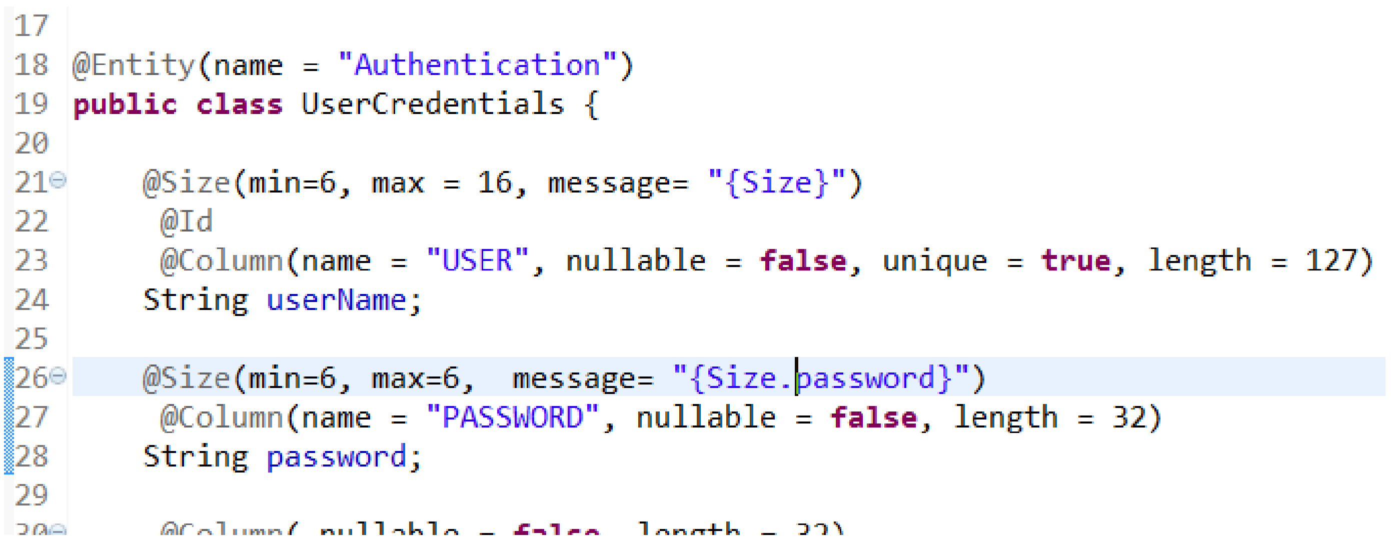


```java
@Entity
@Table(name = "USERS")
 public class User implements Serializable  {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "USER_ID")
    private Long id = null;

    @Version
    private int version = 0;

    @NotEmpty
     @Column(name = "FIRSTNAME", nullable = false)
    private String firstName;

    @Size(min=5, max = 9, message= "{Size}")
    @Column(name = "LASTNAME", nullable = false)
    private String lastName;

    @Email(message= "{email.syntax}")
    @Column(name = "EMAIL", nullable = false)
    private String email;

    @Range(min=4,max=6, message=("{Range}"))
    @Column(name = "RANK", nullable = false)
    private int ranking = 0;

    @Column(name = "IS_ADMIN", nullable = false)
    private boolean admin = false;

    @Valid
    @OneToOne(fetch=FetchType.EAGER,  cascade = CascadeType.ALL)
    @JoinColumn(name="userId")
    private UserCredentials userCredentials;
```

Here is the UserCredentials:

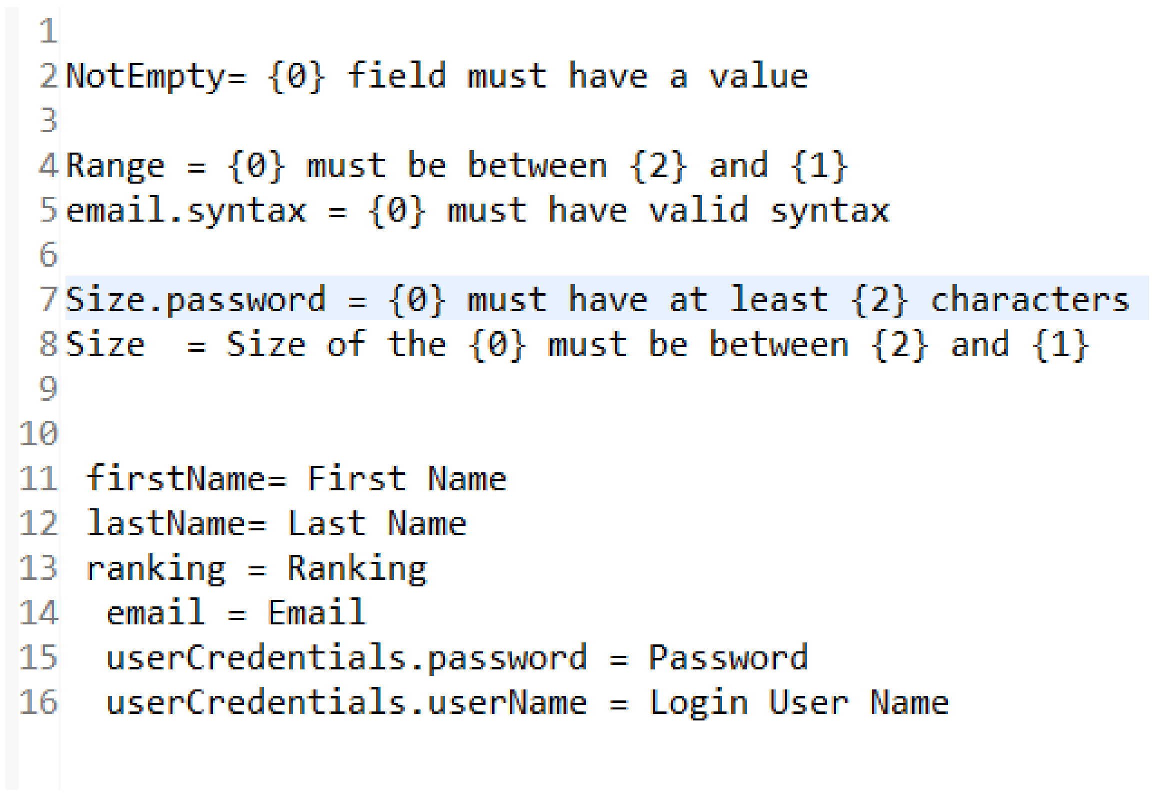
```java
@Entity(name = "Authentication")
public class UserCredentials {


         @Id
         @Column(name = "USER", nullable = false, unique = true, length = 127)
        String userName;


         @Column(name = "PASSWORD", nullable = false, length = 32)
        String password;


         @Column( nullable = false, length = 32)
        String verifyPassword;
        Boolean enabled;


        @OneToOne(mappedBy="userCredentials", cascade = {CascadeType.PERSIST, CascadeType.MERGE})
```

```java
    private User user;
```



```java

@Entity(name = "Authentication")
public class UserCredentials {

    @Size(min=6, max = 16, message= "{Size}")
     @Id
     @Column(name = "USER", nullable = false, unique = true, length = 127)
    String userName;

    @Size(min=6, max=6,  message= "{Size.password}")
     @Column(name = "PASSWORD", nullable = false, length = 32)
    String password;

     @Column( nullable = false, length = 32)
```

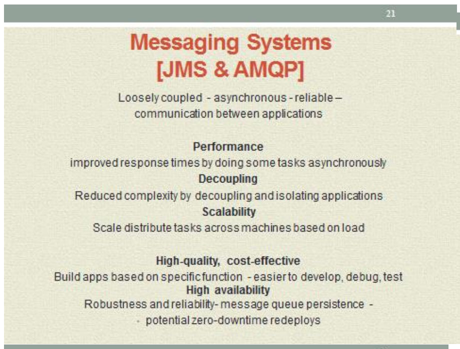
ErrorMessage.properties



```properties

NotEmpty= {0} field must have a value

Range = {0} must be between {2} and {1}
email.syntax = {0} must have valid syntax

Size.password = {0} must have at least {2} characters
Size   = Size of the {0} must be between {2} and {1}


 firstName= First Name
 lastName= Last Name
 ranking = Ranking
  email = Email
  userCredentials.password = Password
  userCredentials.userName = Login User Name
```

5. [15 points]
Messaging is basic to scalable enterprise architectures. We covered two messaging technologies, JMS & AMQP. Explain the fundamentals of messaging.

Be sure to cover: the types of messaging, the messaging architecture, and the differences between the two, JMS & AMQP and how they are implemented.
Be specific. Give examples. Diagrams are good but be sure to explain them.

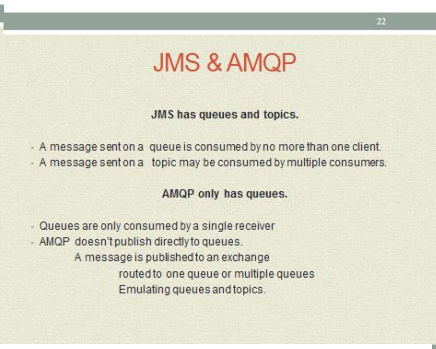


Messaging Systems [JMS & AMQP]

Loosely coupled - asynchronous - reliable – communication between applications

Performance
improved response times by doing some tasks asynchronously
Decoupling
Reduced complexity by decoupling and isolating applications
Scalability
Scale distribute tasks across machines based on load

High-quality, cost-effective
Build apps based on specific function - easier to develop, debug, test
High availability
Robustness and reliability- message queue persistence -
- potential zero-downtime redeploys



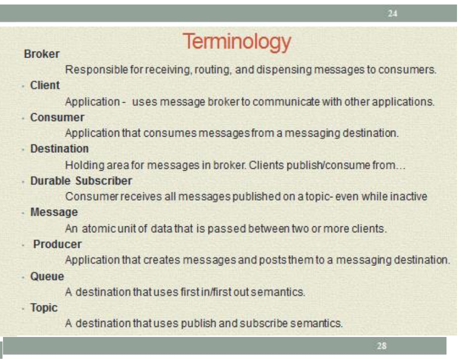
JMS & AMQP

JMS has queues and topics.

- A message sent on a queue is consumed by no more than one client.
- A message sent on a topic may be consumed by multiple consumers.

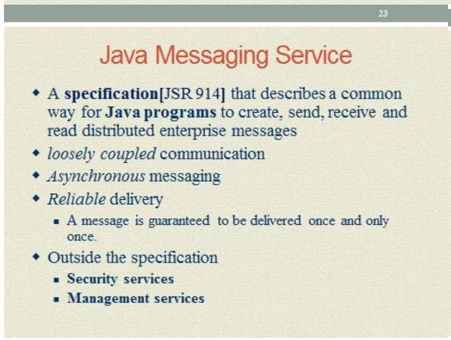
AMQP only has queues.

- Queues are only consumed by a single receiver
- AMQP doesn't publish directly to queues.
  A message is published to an exchange
  routed to one queue or multiple queues
  Emulating queues and topics.



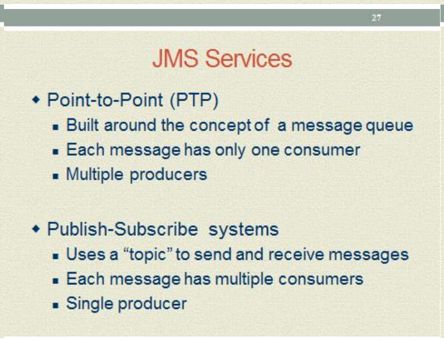
Terminology

Broker
Responsible for receiving, routing, and dispensing messages to consumers.
- **Client**
Application - uses message broker to communicate with other applications.
- **Consumer**
Application that consumes messages from a messaging destination.
- **Destination**
Holding area for messages in broker. Clients publish/consume from…
- **Durable Subscriber**
Consumer receives all messages published on a topic- even while inactive
- **Message**
An atomic unit of data that is passed between two or more clients.
- **Producer**
Application that creates messages and posts them to a messaging destination.
- **Queue**
A destination that uses first in/first out semantics.
- **Topic**
A destination that uses publish and subscribe semantics.



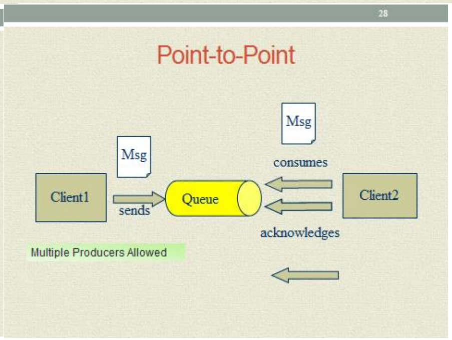
Java Messaging Service

- A **specification**[JSR 914] that describes a common way for **Java programs** to create, send, receive and read distributed enterprise messages
- *loosely coupled* communication
- *Asynchronous* messaging
- *Reliable* delivery
  - A message is guaranteed to be delivered once and only once.
- Outside the specification
  - **Security services**
  - **Management services**

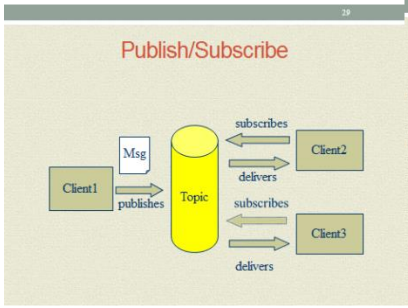


JMS Services

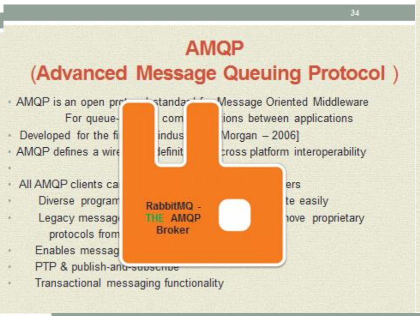
- Point-to-Point (PTP)
  - Built around the concept of a message queue
  - Each message has only one consumer
  - Multiple producers
- Publish-Subscribe systems
  - Uses a "topic" to send and receive messages
  - Each message has multiple consumers
  - Single producer



Point-to-Point



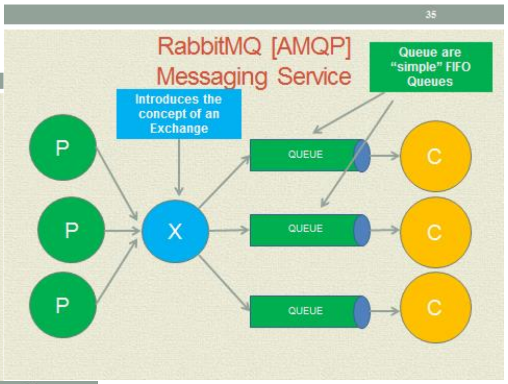
Publish/Subscribe



AMQP (Advanced Message Queuing Protocol)

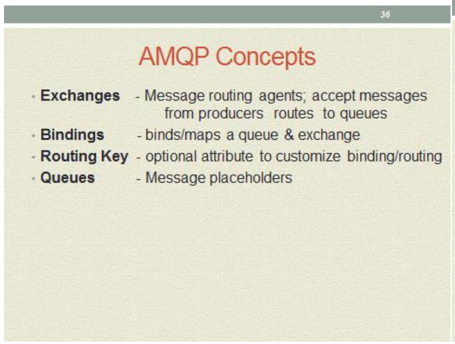
- AMQP is an open pro[illegible] standa[illegible] Message Oriented Middleware
  For queue-[illegible] com[illegible]ions between applications
- Developed for the fi[illegible] indus[illegible] Morgan – 2006]
- AMQP defines a wire[illegible] definit[illegible] cross platform interoperability
-
- All AMQP clients ca[illegible] ers
-   Diverse program[illegible] te easily
-   Legacy messag[illegible] nove proprietary protocols from
-   Enables messag[illegible]
-   PTP & publish-and-subscribe
-   Transactional messaging functionality

RabbitMQ - THE AMQP Broker



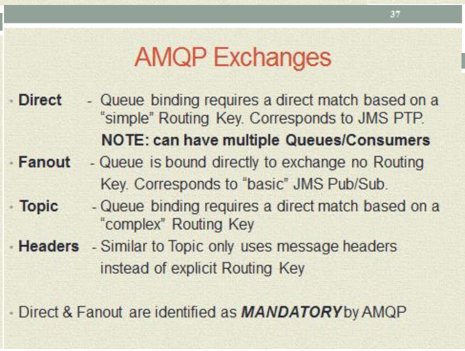
RabbitMQ [AMQP] Messaging Service

Queue are "simple" FIFO Queues
Introduces the concept of an Exchange
P
P
P
X
QUEUE
QUEUE
QUEUE
C
C
C



AMQP Concepts

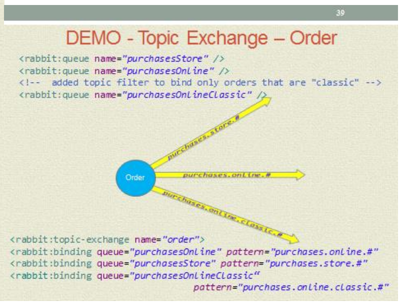
Exchanges	- Message routing agents; accept messages from producers routes to queues
Bindings	- binds/maps a queue & exchange
Routing Key	- optional attribute to customize binding/routing
Queues	- Message placeholders



AMQP Exchanges

Direct	- Queue binding requires a direct match based on a "simple" Routing Key. Corresponds to JMS PTP. **NOTE: can have multiple Queues/Consumers**
Fanout	- Queue is bound directly to exchange no Routing Key. Corresponds to "basic" JMS Pub/Sub.
Topic	- Queue binding requires a direct match based on a "complex" Routing Key
Headers	- Similar to Topic only uses message headers instead of explicit Routing Key

- Direct & Fanout are identified as ***MANDATORY*** by AMQP

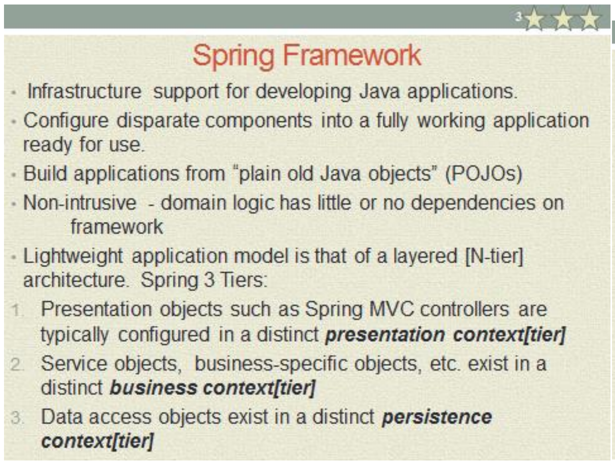


DEMO - Topic Exchange – Order

```
<rabbit:queue name="purchasesStore" />
<rabbit:queue name="purchasesOnline" />
<!--  added topic filter to bind only orders that are "classic" -->
<rabbit:queue name="purchasesOnlineClassic" />
```

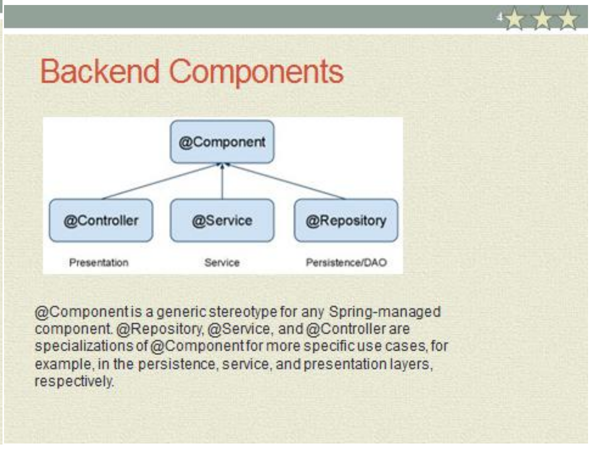
```
<rabbit:topic-exchange name="order">
<rabbit:binding queue="purchasesOnline" pattern="purchases.online.#"
<rabbit:binding queue="purchasesStore" pattern="purchases.store.#"
<rabbit:binding queue="purchasesOnlineClassic"
                                  pattern="purchases.online.classic.#"
```

6. [20 points] The Spring framework is the "example" architecture that we used in this course. It emphasizes good design, best practices and use of design patterns. Explain the value of the framework. Things to consider:
N-Tier; Separation of Concerns; Different types of N-tier; Distributed capabilities; The characteristics & value of a framework
Be specific. Give examples. Diagrams are good but be sure to explain them.


3

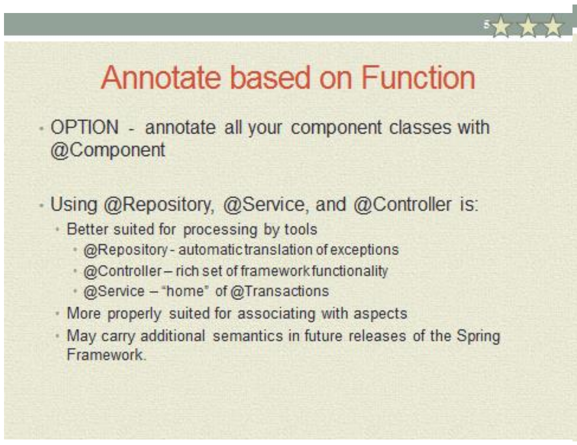
Spring Framework

- Infrastructure support for developing Java applications.
- Configure disparate components into a fully working application ready for use.
- Build applications from "plain old Java objects" (POJOs)
- Non-intrusive - domain logic has little or no dependencies on framework
- Lightweight application model is that of a layered [N-tier] architecture. Spring 3 Tiers:
  1. Presentation objects such as Spring MVC controllers are typically configured in a distinct ***presentation context[tier]***
  2. Service objects, business-specific objects, etc. exist in a distinct ***business context[tier]***
  3. Data access objects exist in a distinct ***persistence context[tier]***


4

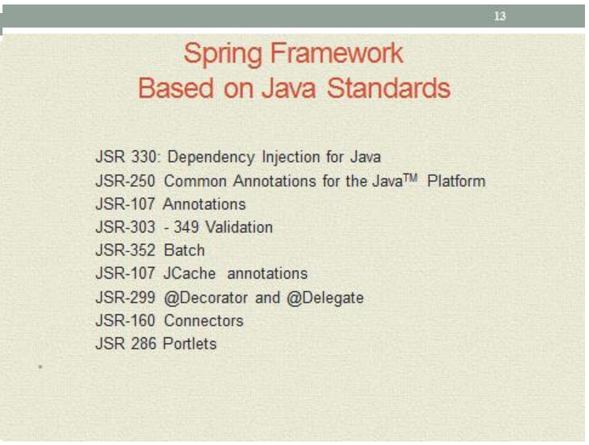
Backend Components

@Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.


5

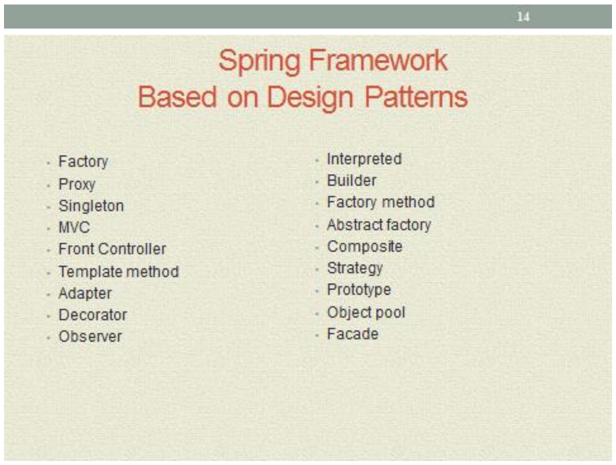
Annotate based on Function

- OPTION - annotate all your component classes with @Component
- Using @Repository, @Service, and @Controller is:
  - Better suited for processing by tools
    - @Repository - automatic translation of exceptions
    - @Controller – rich set of framework functionality
    - @Service – "home" of @Transactions
  - More properly suited for associating with aspects
  - May carry additional semantics in future releases of the Spring Framework.


13

Spring Framework
Based on Java Standards

JSR 330: Dependency Injection for Java
JSR-250 Common Annotations for the Java^{TM} Platform
JSR-107 Annotations
JSR-303 - 349 Validation
JSR-352 Batch
JSR-107 JCache annotations
JSR-299 @Decorator and @Delegate
JSR-160 Connectors
JSR 286 Portlets


14

Spring Framework
Based on Design Patterns

- Factory
- Proxy
- Singleton
- MVC
- Front Controller
- Template method
- Adapter
- Decorator
- Observer
- Interpreted
- Builder
- Factory method
- Abstract factory
- Composite
- Strategy
- Prototype
- Object pool
- Facade