Enterprise JavaBeans (EJB 3.x)

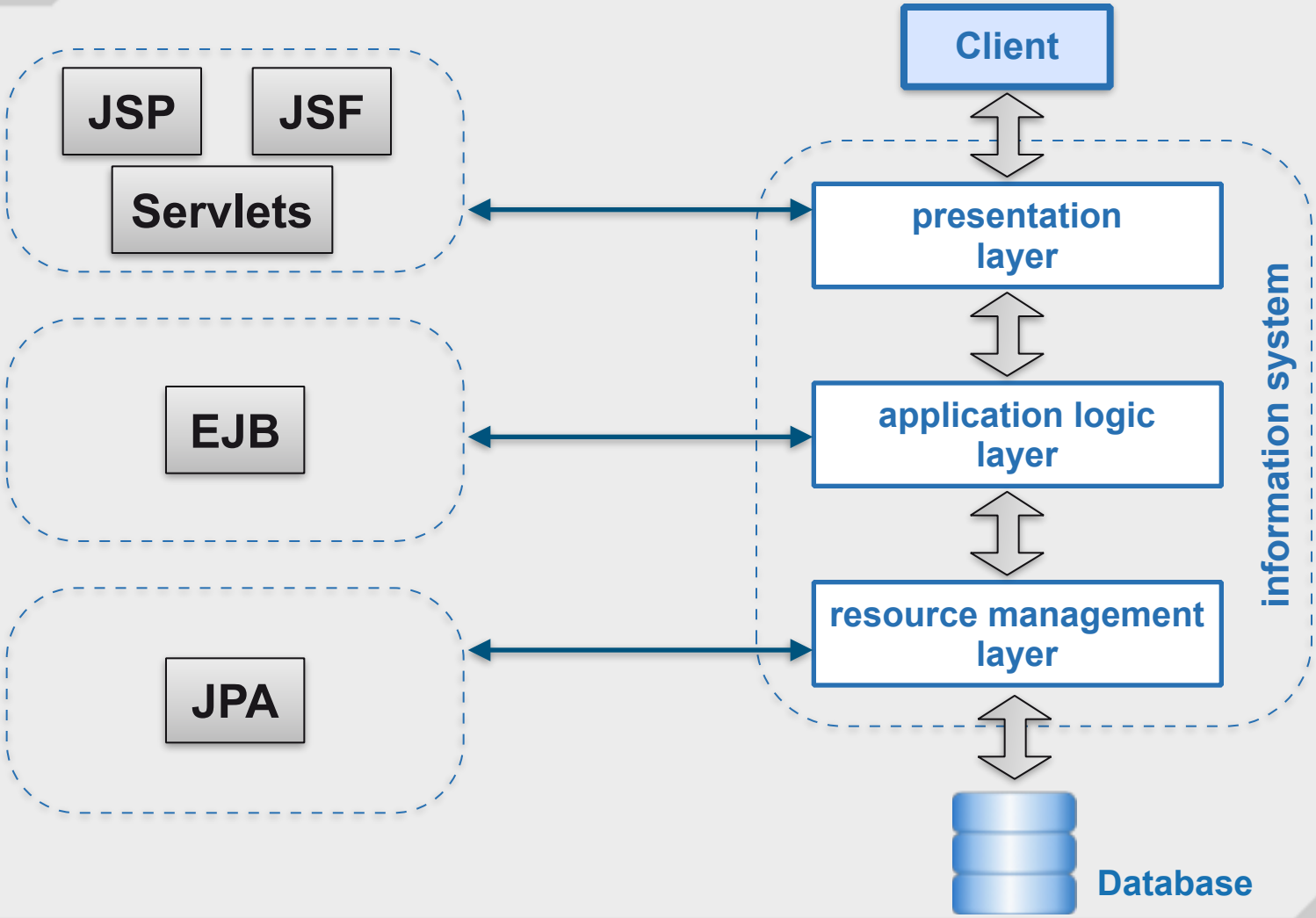
Fahad R. Golra

ECE Paris Ecole d'Ingénieurs - FRANCE

Lecture 8 - Enterprise Java Beans (EJB)

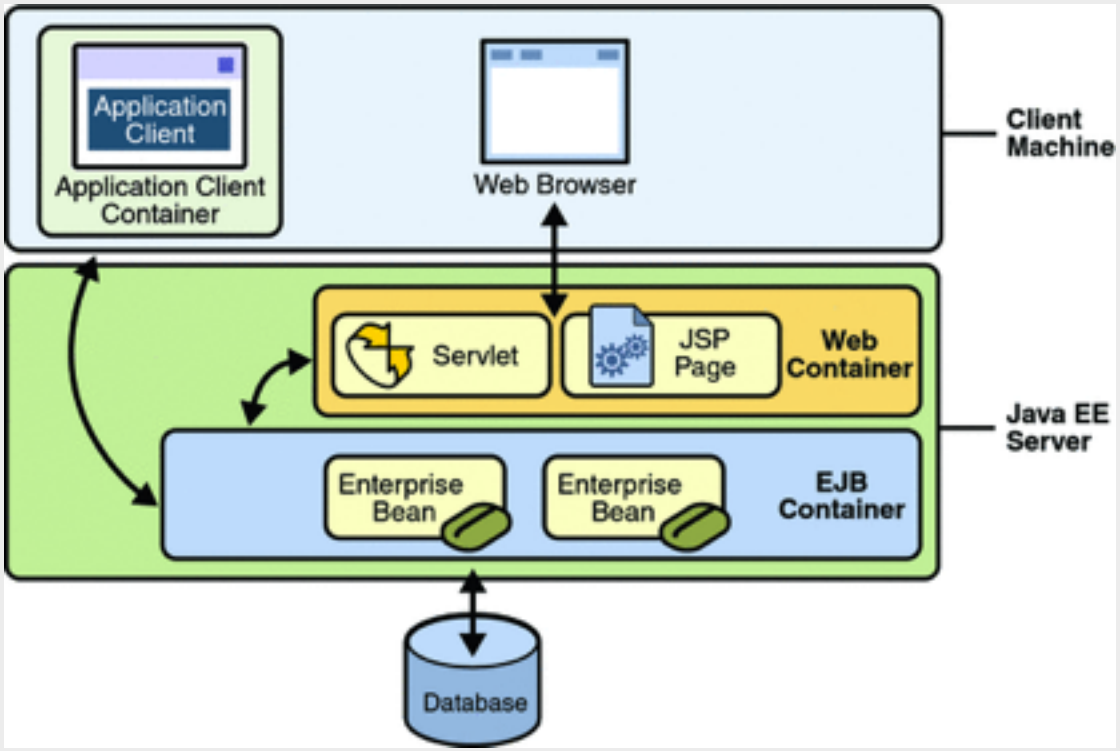
- Introduction to EJBs
- Types of EJBs
  - Session Beans
  - Message Driven Beans
- Transaction Management

3 Layers of Information System



Java EE Server

- The runtime portion of a Java EE product. A Java EE server provides EJB and web containers.



When to use EJB?

- If the application needs
  - to be scalable
  - a transactional context
  - diversity of clients

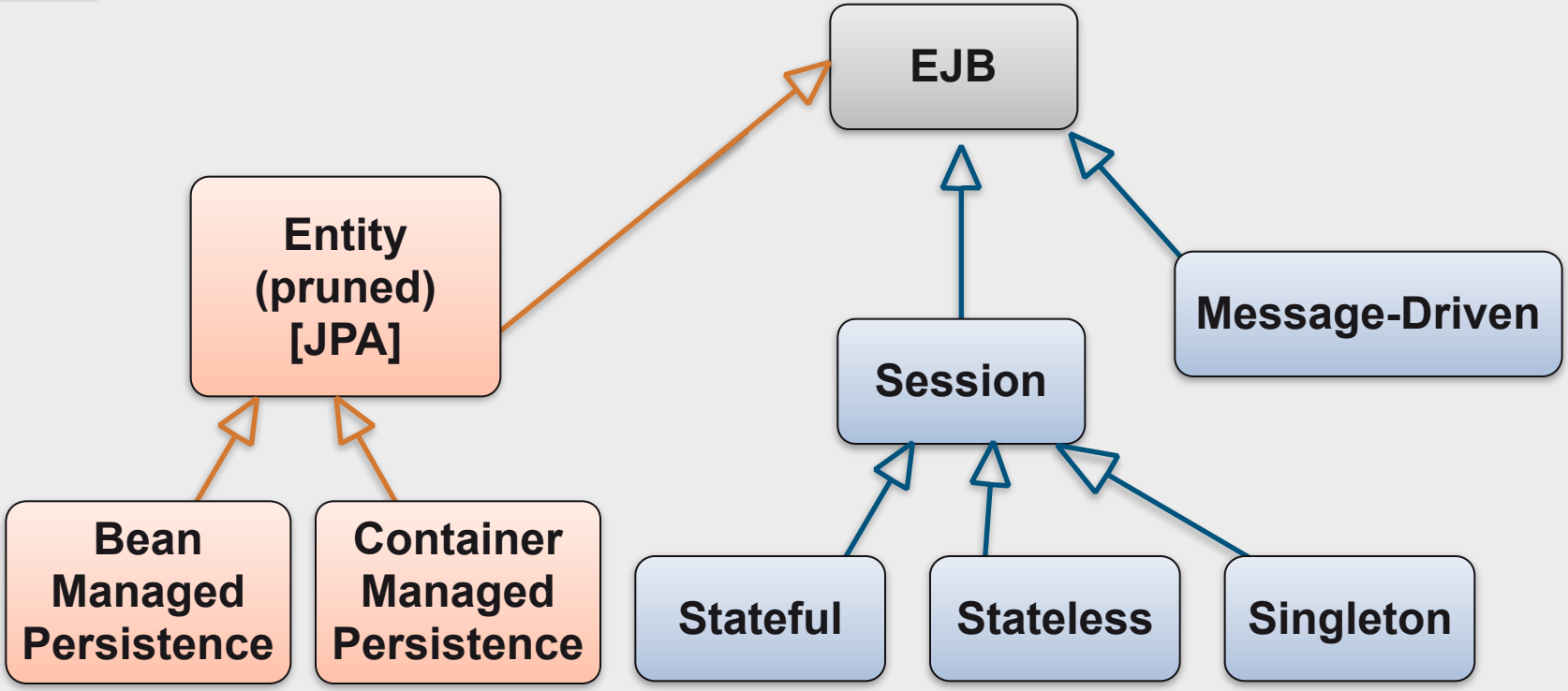
Why to use EJBs

- Encapsulate business logic
  - multi-tier architecture
- Remote access
  - apps on different servers can access them
- Simplicity
  - simpler than other remote object systems
- Broad vendor support
  - JBoss, Oracle AS, WebLogic, WepSphere
- Scalability
  - support for clustering, load-balancing and failover

Why EJB 3.x

- Network connections between the clients and the EJBs
- Naming services (JNDI)
- Transactions
- Persistence and the management of DB pool of connections
- Distribution
- Security
- Management of component's life cycle

Types of EJB

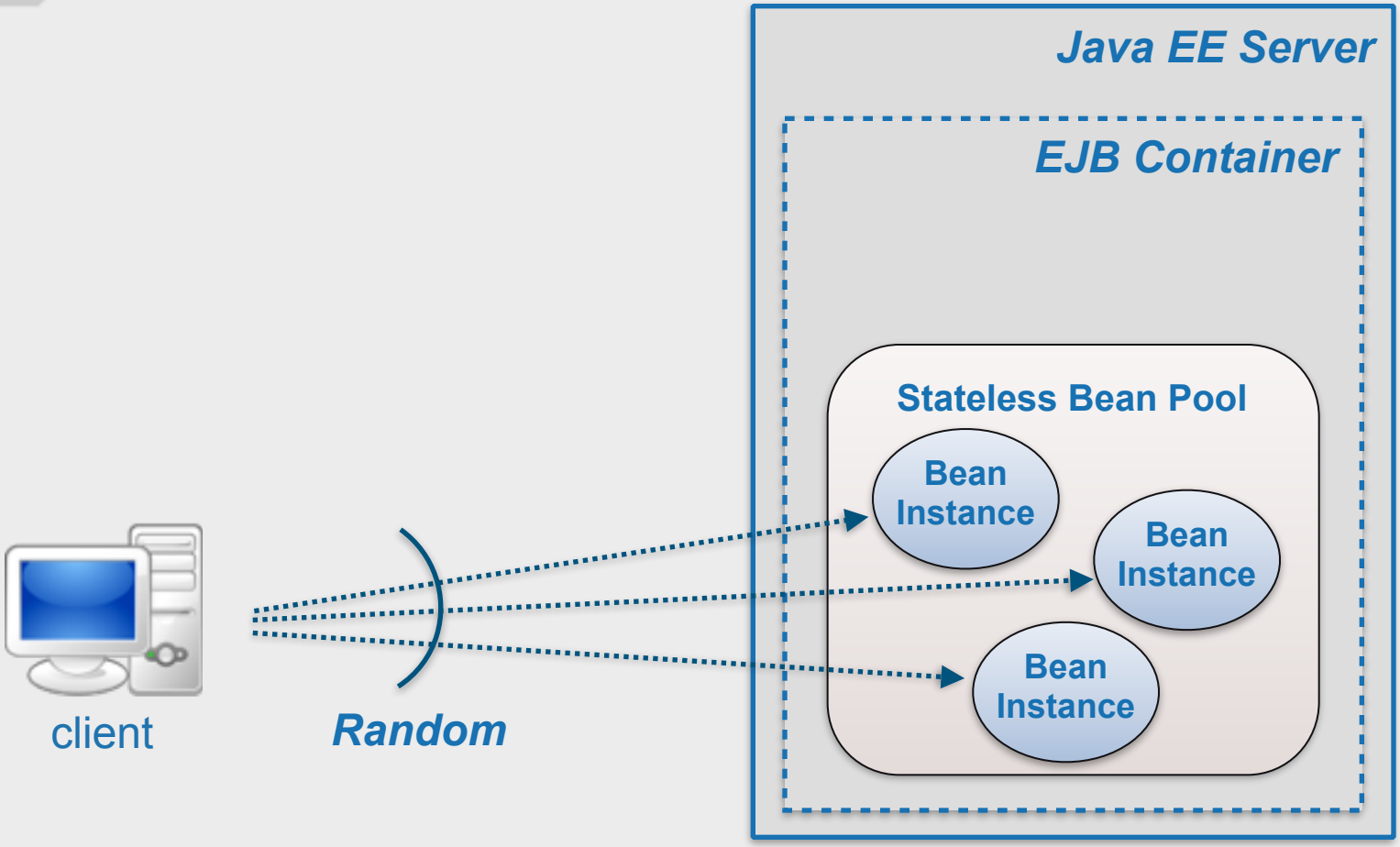


Bean Type	*Annotation*
Session Bean	*Performs a task for a client; optionally, may implement a web service*
Message-driven Bean	*Acts as a listener for a particular messaging type, such as the Java Message Service API*

Stateless Session Beans

- Used when
  - The bean's state has no data for a specific client.
  - In a single method invocation, the bean performs a generic task for all clients. For example, you might use a stateless session bean to send an email that confirms an online order.
  - The bean implements a web service.

Stateless Session Beans

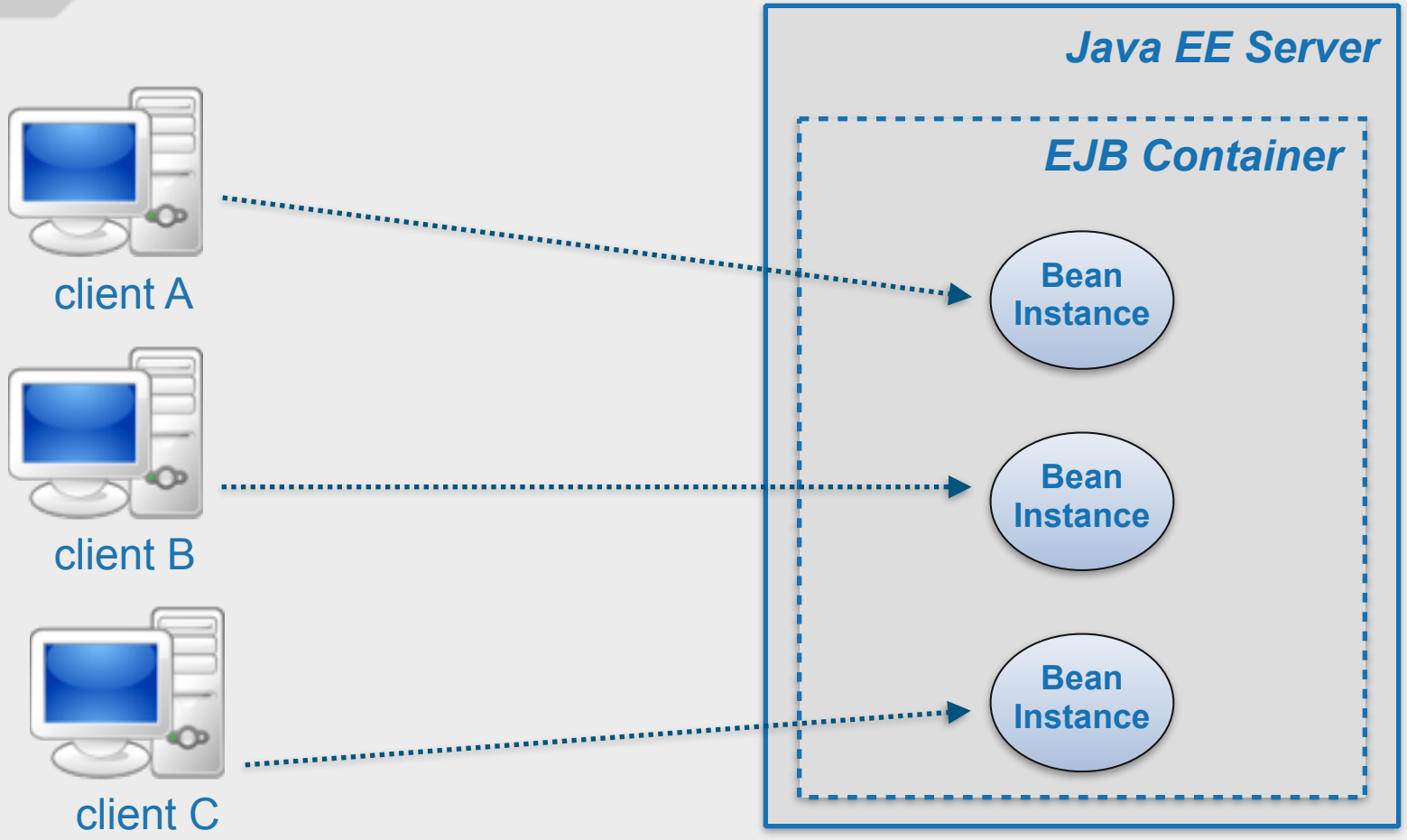


A random bean instance is selected from the pool

Stateful Session Beans

- Used when the following conditions are true
  - The bean's state represents the interaction between the bean and a specific client.
  - The bean needs to hold information about the client across method invocations.
  - The bean mediates between the client and the other components of the application, presenting a simplified view to the client.
  - Behind the scenes, the bean manages the work flow of several enterprise beans.

Stateful Session Beans

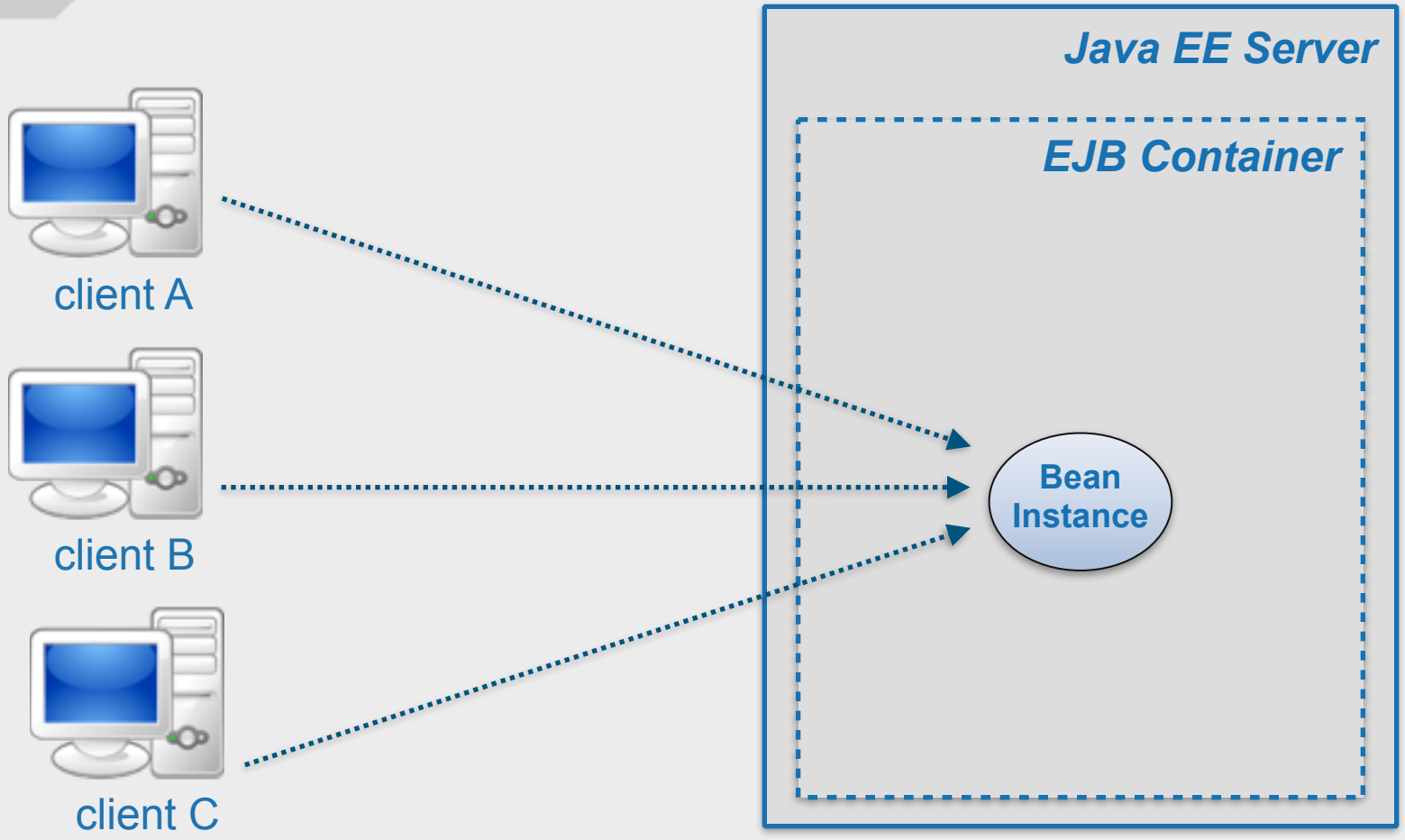


One bean instance per client

Singleton Session Beans

- They are used when
  - State needs to be shared across the application.
  - A single enterprise bean needs to be accessed by multiple threads concurrently.
  - The application needs an enterprise bean to perform tasks upon application startup and shutdown.
  - The bean implements a web service.

Singleton Session Beans



One bean instance per application server instance

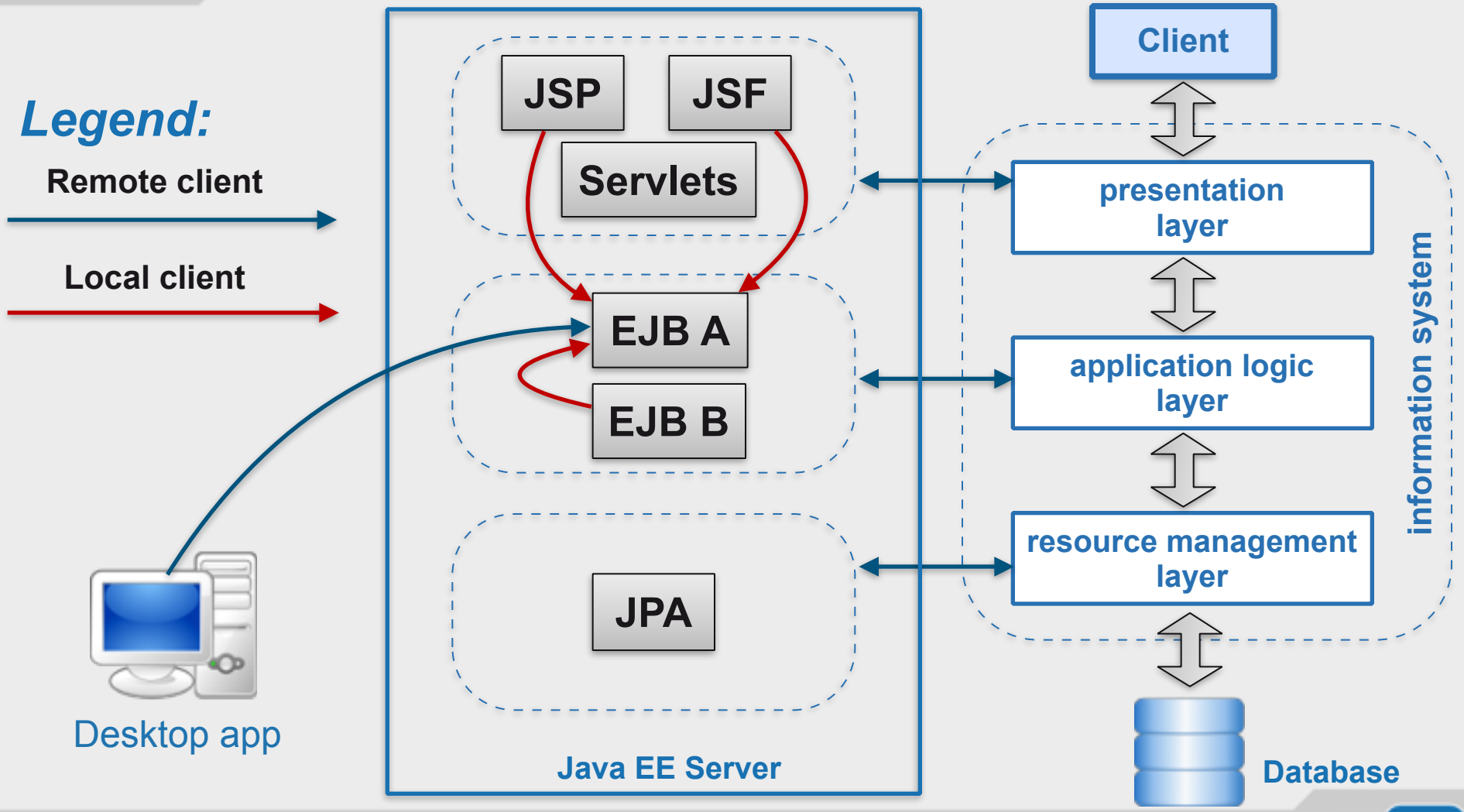
Category Related Annotations

Bean Type	Annotation
Session Bean	@Stateless @Stateful @Singleton
Message-driven Bean	@MessageDriven
JPA Entities	@Entity* @EntityManager*

Contents of EJB

- **Enterprise bean class:**
  - business methods of the enterprise bean
  - any lifecycle callback methods.
- **Business interfaces:**
  - define the business methods
  - not required if the enterprise bean exposes a local, no-interface view
- **Helper classes:**
  - other classes needed by the enterprise bean class, such as exception and utility classes.

Remote & Local Access



Access Mode Annotations

Bean Type	Annotation
Session Bean	@Local @Remote @LocalBean* @WebService
Message-driven Bean	@WebService
JPA Entities	N/A

Remote & Local Access

- Whether to allow local or remote access depends on the following factors.
  - Tight or loose couple of related beans
  - Type of client: web components on the same server or application clients, etc.
  - Component Distribution: A scalable server on multiple servers
  - Performance: Remote calls may be slower than local calls. Distributed computing on different servers for performance.

EJB Business Interfaces

- **Local Interface**

```
@Local
public interface MyStatelessBeanLocal {
    String sayHello(String name);
}
```

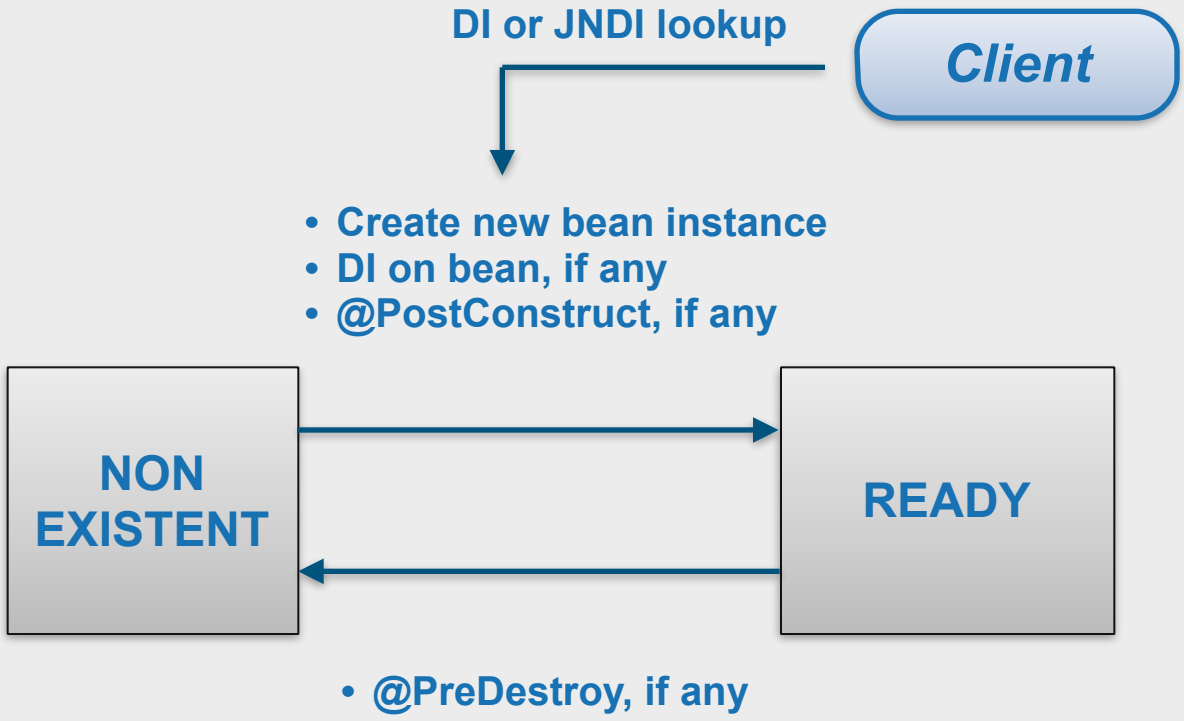
- **Remote Interface**

```
@Remote
public interface MyStatelessBeanRemote {
    String sayHello(String name);
}
```

Stateless Session Beans

- Each invocation of a stateless business method is independent from previous invocations
- Because stateless session beans are "stateless" they tend to process requests faster and use less resources
- All instances are equivalent – the EJB container can assign a pooled stateless bean instance to any client, improving scalability

Stateless session bean lifecycle



Instances are removed automatically from the pool by the container

Example Hello Bean (Stateless)

```
@Stateless
@Local(MyStatelessBeanLocal.class)
@Remote(MyStatelessBeanRemote.class)
public class MyStatelessBean implements MyStatelessBeanRemote,
        MyStatelessBeanLocal {
    public MyStatelessBean() {
        // TODO Auto-generated constructor stub
    }

    @Override
    public String sayHello(String name) {
        return "hello, " + name;
    }

}
```


JNDI Lookup

- Clients find the bean via JNDI
  - Client Java code doesn't even know the machine on which the bean resides
- Clients use the bean like a normal POJO
  - But arguments and return values are sent across network
    So, custom classes should be Serializable

```
InitialContext context = new InitialContext();
InterfaceName bean =(InterfaceName)context.lookup("JNDI- Name");
```

- jndi.properties
  - Text file in classpath; gives remote URL and other info

Accessing Local EJB

- **Dependency Injection**

```
@EJB
MyStatelessBeanLocal myDIBeanLocal;
```

- **JNDI Lookup**

```
Hashtable jndiProperties = new Hashtable();
jndiProperties.put(Context.URL_PKG_PREFIXES,
    "org.jboss.ejb.client.naming");
Context context = new InitialContext(jndiProperties);
MyStatelessBeanLocal myJNDIBeanLocal = (MyStatelessBeanLocal)
context.lookup("java:module/MyStatelessBean!
com.ece.jee.MyStatelessBeanLocal");
```

- **Clients do not use the new operator to obtain a new instance**

Accessing Remote EJB

- **Dependency Injection**

```
@EJB
MyStatelessBeanRemote myDIBeanRemote;
```

- **JNDI Lookup**

```
Hashtable jndiProperties = new Hashtable();
jndiProperties.put(Context.URL_PKG_PREFIXES,
    "org.jboss.ejb.client.naming");
Context context = new InitialContext(jndiProperties);
MyStatelessBeanLocal myJNDIBeanLocal = (MyStatelessBeanLocal)
context.lookup("java:module/MyStatelessBean!
com.ece.jee.MyStatelessBeanLocal");
```

- **Clients do not use the new operator to obtain a new instance**

Portable JNDI Syntax

- java:global
  - JNDI namespace for remote EJBs

    java:global[/application name]/module name /enterprise bean name[/ interface name]

- java:module
  - JNDI namespace for local EJBs within the same module.

    java:module/enterprise bean name/[interface name]

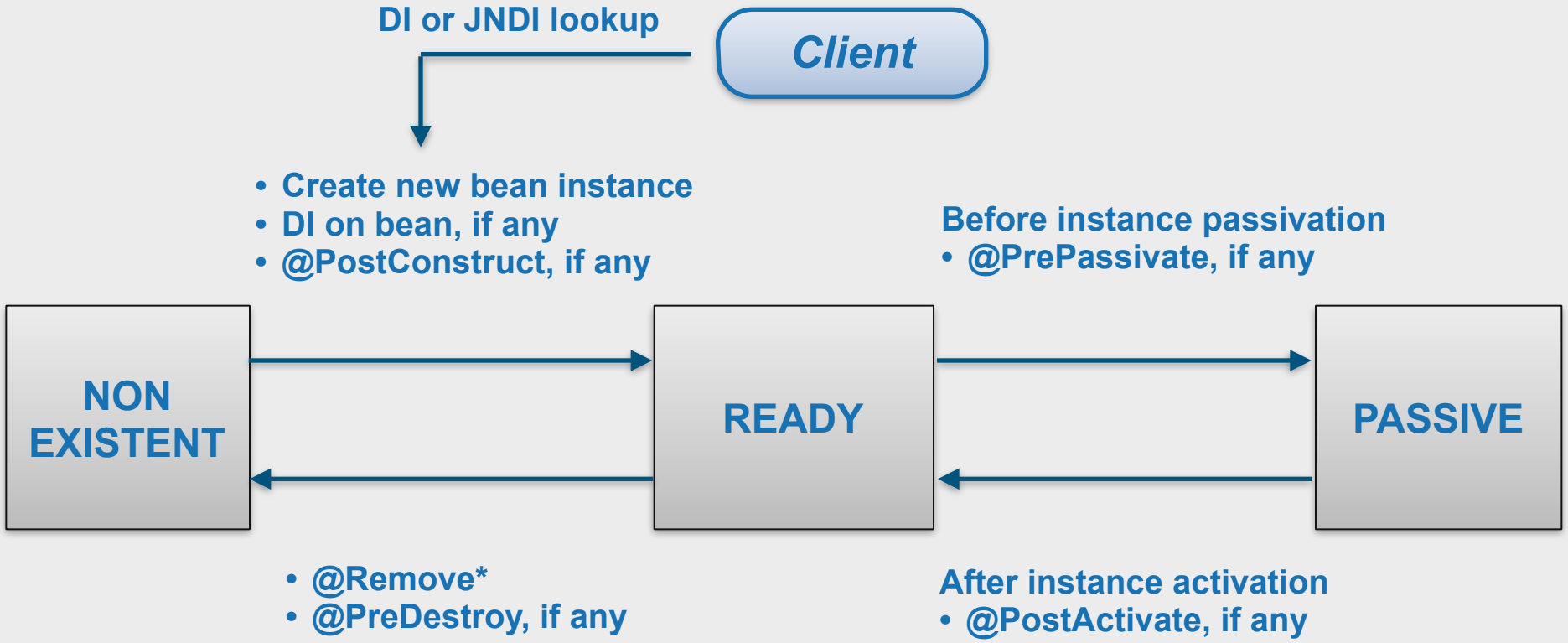
- java:app
  - JNDI namespace is used to look up local EJBs packaged within the same application.

    java:app[/module name]/enterprise bean name [/interface name]

Stateful Session Beans

- POJOs
  - Instance of the Bean relates to a specific client (in memory while he/she is connected)
  - Expires in case of inactivity (similar to session in Servlet/Jsp)
  - Ordinary Java classes; no special interfaces or parent classes.
- Local or remote access
  - Can be accessed either on local app server or remote app server
- Session Expiry
  - The Session expires after the method annotated with @Remove is executed
  - Session can also expire in case of a time-out

Stateful session bean lifecycle



* Method called by the client code, other methods are called by container

Callback method annotations

@PostConstruct

public void initialize() { ... at Bean's initialization ... }

@PreDestroy

- public void destroy() { ... destruction of Bean ... }

@PrePassivate //only for Stateful beans

public void beforeSwap() { ... to do before Bean is passivated ... }

@PostActivate //only for Stateful beans

public void afterSwap() { ... to do after Bean is activated ... }

@Remove

- Container does not manage a pool for Stateful EJBs
- If the instance is not removed it stays in the memory
- A timeout from the server destroys the bean instance from READ or PASSIVE state
- A method with @Remove annotation is used to manage the destruction of instance

```
@Remove
    public void destroy(){

    }
```


Singleton Session Beans

- Only one instance is created per bean
- All clients use the same instance
- @Startup loads it, when server starts
- Concurrency can be managed by two ways
  - Container Managed Concurrency
  - Bean Managed Concurrency

- Bean Concurrency

```
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class MySingleton {

Use synchronised, volatile, etc.
```


Container Managed Concurrency

- Uses READ and WRITE locks

- WRITE lock: No other READ or WRITE method can be executed concurrently
- READ lock: Only READ lock methods can be executed concurrently

- Default concurrency management is Container
- Default lock for all methods is WRITE lock

Container Managed Concurrency

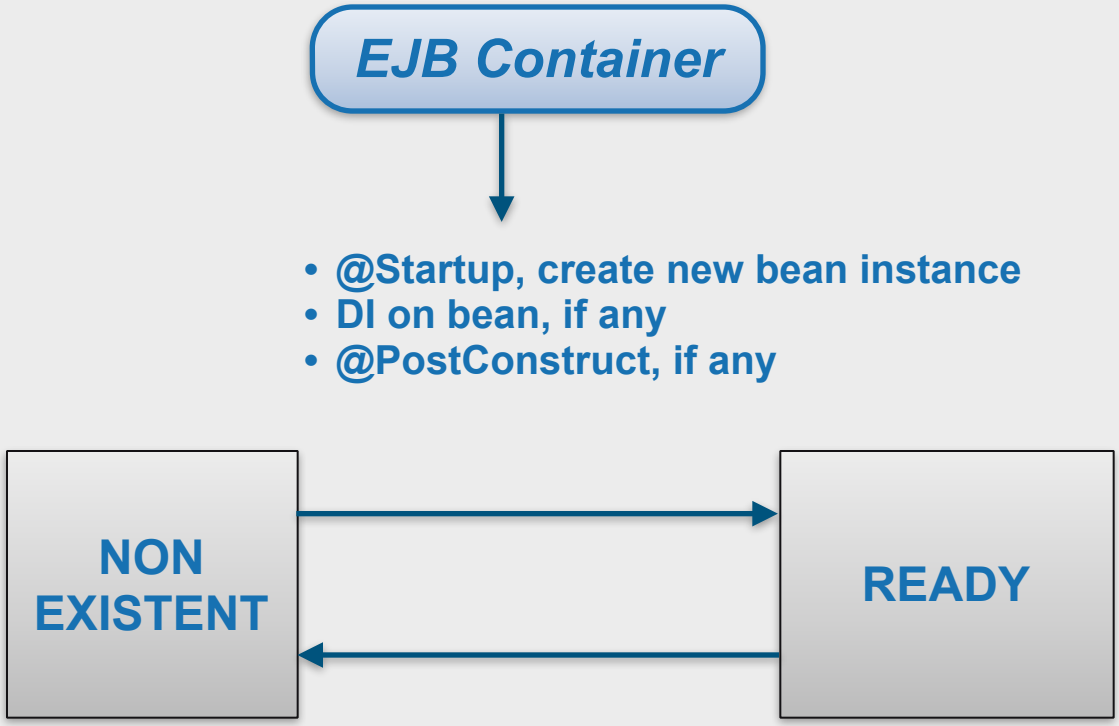
```
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
public class MySingleton {

@Lock(LockType.WRITE)
public void myMethod1() {}

@Lock(LockType.READ)
    public void myMethod2() {}

}
```

Singleton session bean lifecycle



EJB and Web Services

- A client can access a JavaEE application through
  - JAX-WS web service
  - Business methods of EJB

```
@Stateless
@WebService
public class HelloServiceBean {
    private final String message = "Hello, ";
    public void HelloServiceBean() {
    }
    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```


Message-Driven Vs Stateless Beans

- Resemblances
  - Retain no data or conversational state for a specific client.
  - All instances are equivalent. EJB container can
    - assign a message to any message-driven bean instance
    - pool these instances to allow streams of messages to be processed concurrently
  - A single message-driven bean can process messages from multiple clients.

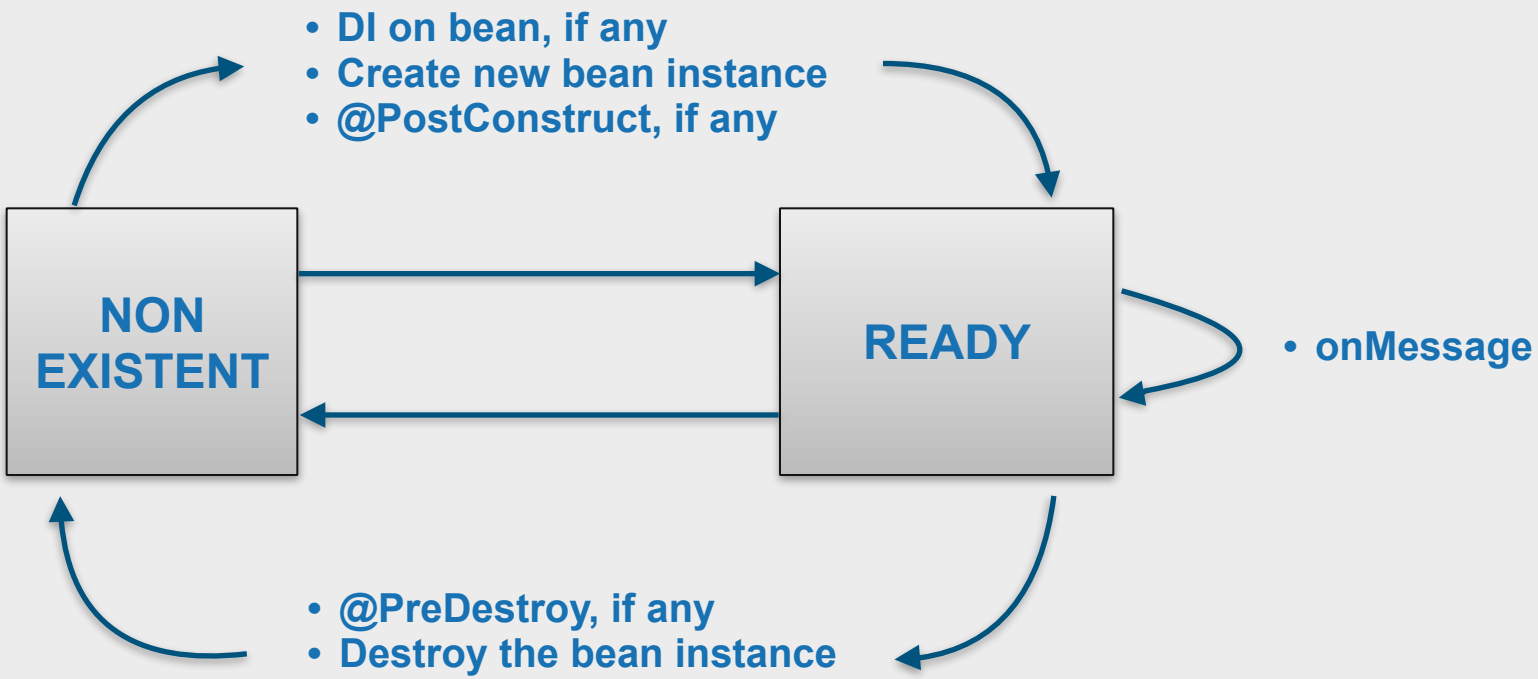
Message-Driven Vs Stateless Beans

- Differences
  - clients do not access message-driven beans through interfaces
  - contain some state across the handling of client messages,
    - JMS API connection, an open database connection, etc.
  - client access through JMS
    - by sending messages to the message destination for which the message-driven bean class is the MessageListener.

Message Driven Bean

- They have the following characteristics
  - They execute upon receipt of a single client message.
  - They are invoked asynchronously.
  - They are relatively short-lived.
  - They do not represent directly shared data in the database, but they can access and update this data.
  - They can be transaction-aware.
  - They are stateless.

Message-driven bean lifecycle



Message-driven bean

- JMS (java.sun.com/jms)
- Two mothers of communication
  - Queue : Thread of discussion (one consumer)
  - Topic : Topic of discussion (diffusion)
- ConnectionFactory : Factory of connections towards queue/topic
- Connection : connection towards queue/topic
- Session :
  - Creation of an sender and of a receiver
  - Can be transactional

Message Driven Bean

```
@MessageDriven(activationConfig ={
  @ActivationConfigProperty( propertyName = "destination",
propertyValue = "topic_ece"),
@ActivationConfigProperty( propertyName = "destinationType",
propertyValue = "javax.jms.Topic")})

public class Mdb implements MessageListener {
      public void onMessage(Message inMessage) {
      System.out.println(((TextMessage)msg).getText());
      }
}
```

Message Driven Bean (Sender)

```
@Resource(name="rigolo", mappedName="topic_rigolo")
Topic topic;
@Resource(name="factory", mappedName="JTCF")
TopicConnectionFactory factory;
TopicSession session;
TopicPublisher sender;
public void publish(String value) {
TopicConnection tc = factory.createTopicConnection();
session = tc.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
sender = session.createPublisher(topic);
TextMessage msg = session.createTextMessage();
msg.setText("MDB: " + value);
sender.publish(msg);
}
```

Transaction Management

- Business Transaction
  - Interaction in real world
  - Usually between enterprise & person or between enterprises
- Information processing that is divided into individual, indivisible operations, called transactions
- Performs function on (shared) database

The ACID Properties

- A set of properties that guarantee that transactions are processed reliably
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Atomicity

- All (commit) or nothing (abort)
  - "all or nothing": if one part of the transaction fails, the entire transaction fails
  - Example: transfer money between two bank accounts
- Must handle situations including power failures, errors, and crashes

Consistency

- Each transaction takes valid states to valid states:
  - Satisfy integrity constraints, triggers
- Sometimes the only notion of “valid” state is “a state that could have been produced by executing a sequence of transactions

Isolation

- Each transaction behaves as if it were executed in isolation at some instant in time
- AKA Serializability
  - Ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially
- Consistency + Isolation implies the data remains consistent even when multiple transaction programs execute concurrently

Durability

- The effect of a committed transaction will not be lost
  - Even in the event of power loss, crashes, or errors
- So data must be on stable storage before commit
- Usually done with a log (or journal) that must be forced before commit and used in case of crash recovery

Transactions

- There are two types of transactions
- Local Transactions
  - They span only on a single resource
- Global Transactions (JTA Transactions)
  - They span on multiple resources
  - Default transaction in Java EE
  - Two management styles
    - Container managed transactions (default)
    - Bean managed transactions

Management Mode Annotations

Bean Type	Annotation
Transactions	@TransactionManagement (CONTAINER) @TransactionManagement (BEAN)
Security	@RunAs() @RolesAllowed

Bean Managed Transactions

- **Manually manage the transaction**

```
@TransactionManagement(TransactionManagementType.BEAN)
public class MyBean {
@Resource
private UserTransaction tx;

public void myMethod() {
        try {
            tx.begin();
            methodcall1();
            methodcall2();
            methodcall3();
            tx.commit();
        } catch (Exception e) {
            e.printStackTrace();}
    }
}
```


Contrainer Managed Transactions

- **The start and stop of transactions is managed by container**

```
@TransactionManagement(TransactionManagementType.CONTAINER)
public class MyTester {

    @TransactionAttribute(TransactionAttributeType.MANDATORY)
    public void myMethod() {
        methodcall1();
        methodcall2();
        methodcall3();
    }
}
```


Transaction Attributes

- **REQUIRED**
  - Client in transaction: uses transaction
  - Client without transaction: creates new transaction
- **REQUIRES_NEW**
  - Client in transaction: creates new transaction
  - Client without transaction: creates new transaction
- **MANDATORY**
  - Client in transaction: uses transaction
  - Client without transaction: throws exception

Transaction Attributes

- NEVER
  - Client in transaction: throws exception
  - Client without transaction: without a transaction
- SUPPORTS
  - Client in transaction: uses transaction
  - Client without transaction: without a transaction
- NOT_SUPPORTED
  - Client in transaction: without a transaction
  - Client without transaction: without a transaction