Java Technologies

Enterprise Java Beans (EJB)

The Context

- We are in the context of developing **large**, **distributed** applications.
- What components should contain the code that fulfills the purpose of the application: where should we write the **business logic**?
  - complex algorithms, database access objects, etc.
- How to ensure the **scalabilty** of the application?
- How to control more easily:
  - transactions, concurrency, security?
- How to **reuse** such business logic components?

Server Setups for Applications

- One Server for Everything
  - database, Web (UI), Application (Bussiness)
- Separate Database Server
  - Single, Master-Slave Replication
- Separate Web Server(s)
  - Load Balancer, HTTP Accelerators (Cache)
- Separate Application Server(s)
  - Load Balancer, Bussiness Modules

Enterprise Beans

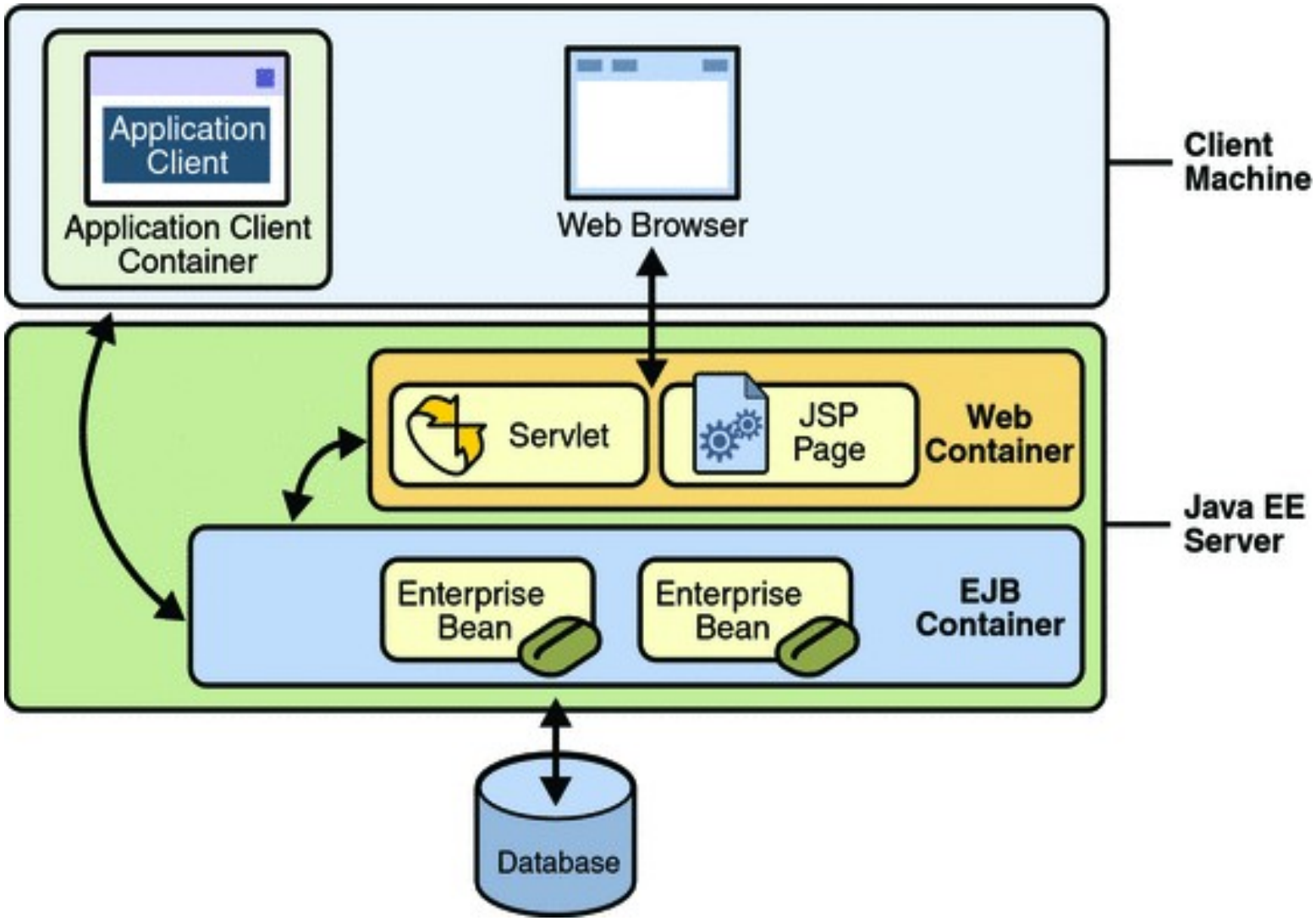
- **Server-side, managed components** that *encapsulate the business logic* of an application in a **standard manner** (locating, invoking)
- **Integration** with the:
  - Persistence services (JPA)
  - Messaging services (JMS)
  - Web services
  - Security services, etc.
- Managed by **EJB Containers**

Simplified development

Portability

Sharing and reusing logic

EJB Container



Enterprise Applications

- **Web Application**
  - Components: Servlets, JSP, HTML, CSS, Images, etc.
  - Purpose: Creating the *User Interface Layer*.
  - Needed: Web Container
  - Deployment: **WAR** archive
- **EJB Application**
  - Components: Enterprise Java Beans.
  - Purpose: Creating the *Bussines Logic Layer*.
  - Needed: EJB Container
  - Deployment: **JAR** archive
- **Enterprise Application**
  - Web Applications + EJB Applications (called **Modules**)
  - Deployment: **EAR** archive
  - Needed: EE Application Server (Glassfish, WildFly, etc.)

Example

person.xhtml *uses* **JSF PersonBean**
needs ***database access, CNP validation, etc.***

```
@Stateless
public class PersonService {
  @PersistenceContext
  private EntityManager entityManager;

  public void create(Person person) {
    entityManager.persist(person);
  }
}
```

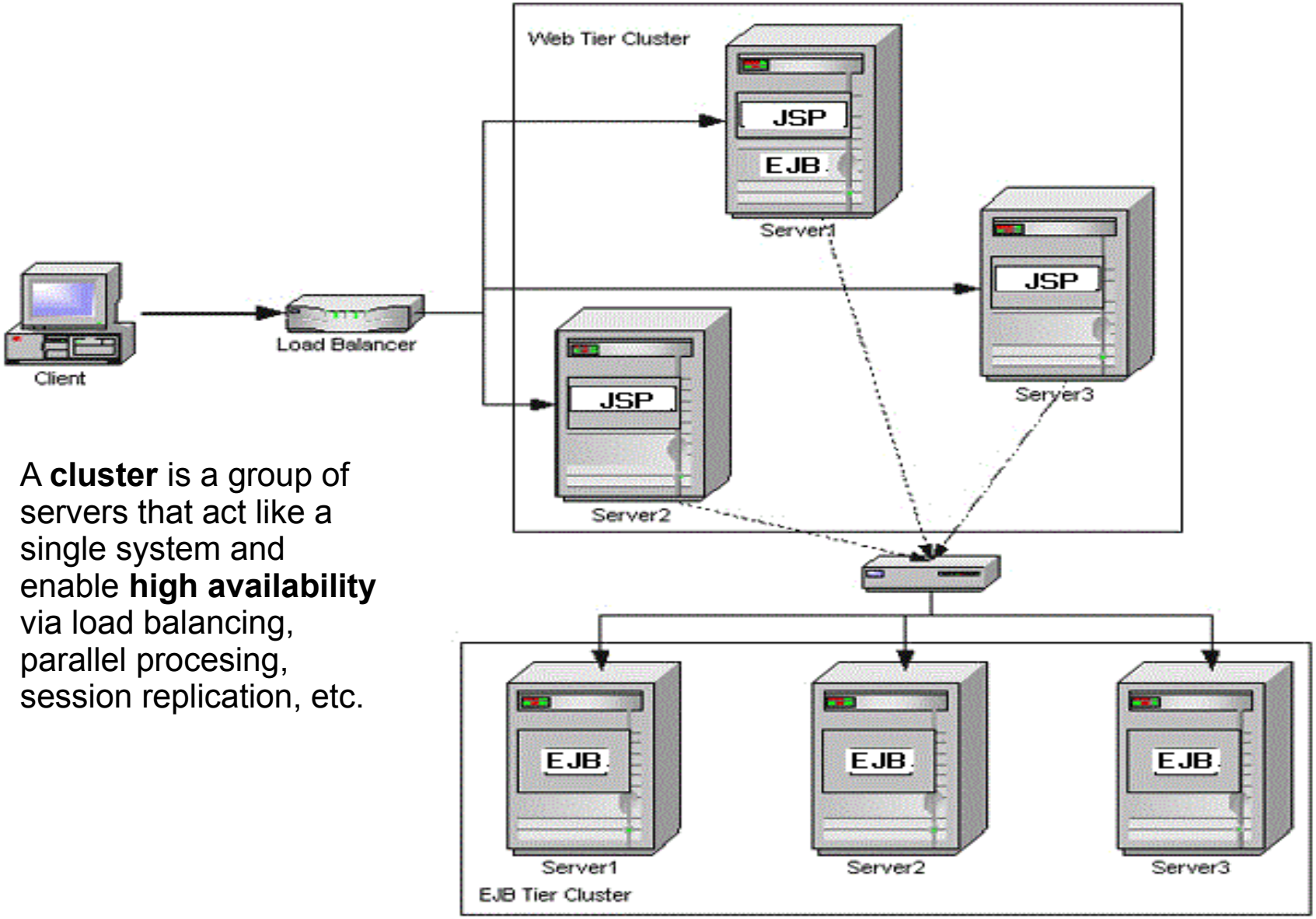
EJB Component

```
@Named
@SessionScoped
public class PersonBean {
   @EJB
   private PersonService personService;

   public String addPerson() {
      Person person = new Person(...);
      personService.create(person);
      return "success";
   }
}
```

JSF Managed Bean using an EJB

Clustering and Scalability



A **cluster** is a group of servers that act like a single system and enable **high availability** via load balancing, parallel procesing, session replication, etc.

Remote / Local

- **Remote Beans** - can be accessed from *anywhere* on the network.
  Client ↔ Stub ↔ (marshalling) ↔ Skeleton ↔ EJB
  Implemented using **RMI**.
  Passing parameters involves serializing them.

- **Local Beans** - for "internal use", called by other components in the same JVM.
  Client ↔ EJB
  Passing parameters is done using references.

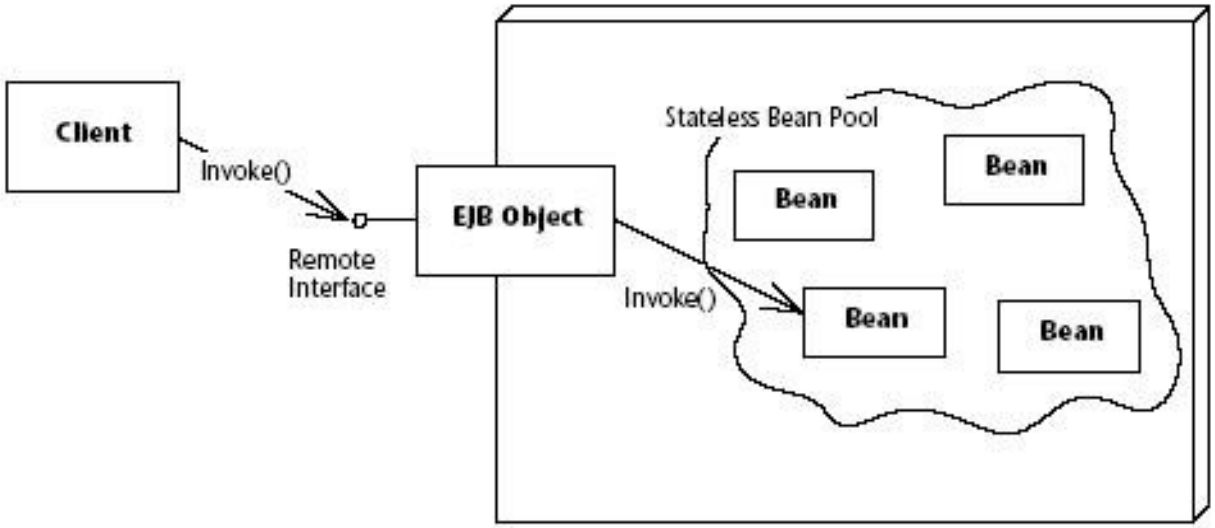
Location Transparency: the precise location of a bean executing a specific request *may not be known* to the caller. The caller only knows the *JNDI name* of the bean.

Types of Enterprise Beans

- **Session Beans**: *performs a specific action* for a client, shielding it from complexity by executing business tasks inside the server.
  - **Stateful, Stateless, Singleton**
  - Optionally, may implement a web service.
- **Message-Driven Beans (MDB)**: acts as a *listener* for a particular messaging type.
- ~~Entity-Beans~~ (deprecated)

Stateless Session Beans

- **Does not maintain a conversational state** with the client.
- **Not shared**: Each client gets his own instance.
- **Not persistent**: Its state is not saved at the end of the action performed by the bean.
- Offer better scalability for applications that require large numbers of clients.



Creating a Stateless Bean

```
@Stateless
@LocalBean
public class HelloBean {
  public String sayHello(String name) {
    return "Hello " + name;
  }
}
```

A **no-interface view** of an enterprise bean exposes the public methods of the enterprise bean implementation class to clients.

Clients that run within a Java EE server-managed environment, JavaServer Faces web applications, JAX-RS web services, other enterprise beans, or Java EE application clients support **dependency injection using the javax.ejb.EJB annotation.**

```
public class HelloServlet extends HttpServlet {
  @EJB
  private HelloBean hello;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response) {
  ...
  out.println(hello.sayHello("World!");
  ...
}
```

To the local client, the location of the enterprise bean it accesses **is not transparent**.

Defining the Bean as Remote

- Define the *Remote* interface

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

- Create the implementation

```
@Stateless
//@Remote(Hello.class)
public class HelloBean implements Hello {
    @Override
    public String sayHello(String name) {
        return "Remote Hello " + name;
    }
}
```

- Use the EJB

```
@EJB
private Hello hello;
```

Accessing an EJB from “outside”

Applications that run outside a Java EE server-managed environment, such as Java SE applications, must perform an explicit lookup. JNDI supports a global syntax for identifying Java EE components to simplify this explicit lookup.

```
public class Main {

  public static void main(String[] args) throws Exception {

    InitialContext context = new InitialContext();

    Hello hello = (Hello) context.lookup(
      "java:global/MyEEApp/MyEjbModule/HelloBean");

    System.out.println(hello.sayHello("World!"));
  }
}
```

The java:global JNDI namespace is the portable way of finding remote enterprise beans using JNDI lookups.

To a remote client, the location of the enterprise bean **is transparent.**

“Sharing” an EJB

- Several applications might use the same EJBs
- Create a **Library** containing the *interfaces*

```
public interface Hello { ... }
```

- Create an **EJB Module** containing the implementations of the interfaces and deploy it on the server (use the library)

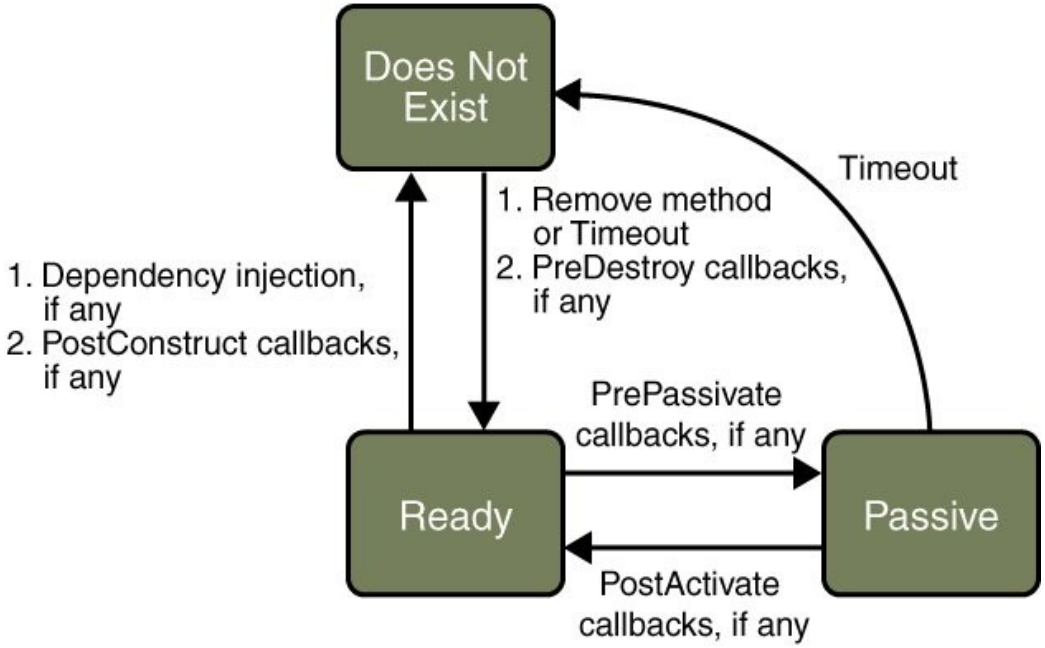
```
@Remote(Hello.class)
@Stateless
public class HelloBean implements Hello, Serializable { ... }
```

- In all other applications deployed on the server you can use the EJB (use the library).

```
@EJB(lookup="java:global/EJBModule/HelloBean")
private Hello hello;
```

Stateful Session Beans

- A session bean is similar to an *interactive session*: the instance variables of the bean represent the state of a unique client/bean session → **conversational state**.
- A session bean is also **not shared**: each client gets his own instance of the bean.



Creating a Stateful Bean

```
@Stateful
@LocalBean
public class ShoppingCartBean {

  List<String> contents;

  @PostConstruct
  public void init() {
    contents = new ArrayList<>();
  }

  public void addItem(String item) {
    contents.add(item);
  }
  public List<String> getContents() {
    return contents;
  }

  @Remove
  public void save() {
    System.out.println("Saving ... \n" + contents);
  }
}
```

The bean needs to hold information about the client across method invocations.

Indicates to the container that the stateful session bean is to be removed by the container after completion of the method.

Using the Stateful Bean

Inside an enterprise application client

Application clients directly access enterprise beans running in the business tier, and may, as appropriate, communicate via HTTP with servlets running in the Web tier. An application client is typically downloaded from the server, but can be installed on a client machine.

```
public class Main {

  @EJB
  private static ShoppingCartBean cart;

  public static void main(String[] args) {

    // The cart intance was already created via dependency injection
    // The PostConstruct method was already invoked

    //Invoke business methods
    cart.addItem("Christmas Tree");
    cart.addItem("Jingle Bells");

    //The state of the bean is maintained
    System.out.println(cart.getContents());

    //The conversation ends here due to the Remove annotation
    cart.save();
  }
}
```

Singleton Session Beans

- **Instantiated once per application** and exists for the lifecycle of the application:
  - as opposed to a pool of stateless session beans, any of which may respond to a client request.
- Designed for circumstances in which a single enterprise bean instance is **shared** across and *concurrently accessed* by clients.
- Singleton session beans **maintain their state between client invocations** but are not required to maintain their state across server crashes or shutdowns.

Creating a Singleton Bean

```
@Singleton
@ConcurrencyManagement(BEAN)
public class CounterBean {
  private int hitCount;


  //Data access synchronization
  public synchronized int incrementAndGetHitCount() {
    return hitCount++;
  }
}
```

Bean Managed Concurrency (BMC)

ConcurrencyManagementType
BEAN: Bean developer is responsible for managing concurrent access to the bean.
CONTAINER: Container is responsible for managing concurrent access to the bean.

Container Managed Concurrency

```
@Singleton
@ConcurrencyManagement(CONTAINER)
public class ExampleSingletonBean {

  private String state;

  @Lock(READ)
  public String getState() {
    return state;
  }

  @Lock(WRITE)
  public void setState(String newState) {
    state = newState;
  }

  @Lock(WRITE)
  @AccessTimeout(value=60,
                 timeUnit=SECONDS)
  public void doTediousOperation {
    ...
  }
}
```

@Lock
Declares a concurrency lock for a singleton session bean with container managed concurrency.

@Lock(READ)
Allows simultaneous access to methods designated as READ, as long as no WRITE lock is held.

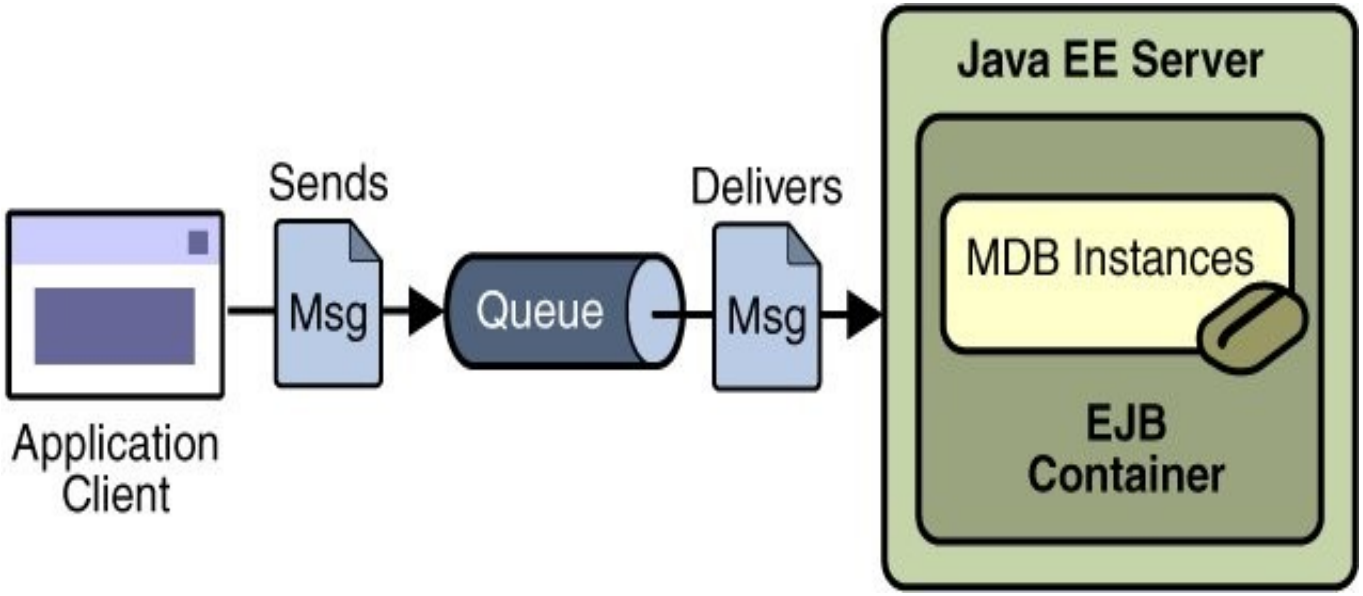
@Lock(WRITE) → *default*
A WRITE lock can only be acquired when no other method with either a READ or WRITE lock is currently held.

Singletons for Startup / Shutdown

- If the singleton is annotated with the **@Startup** the singleton instance is created upon *application deployment*.
- The method annotated **@PostConstruct** can be used for application startup initialization.
- The method annotated **@PreDestroy** is invoked only at application shutdown.
- Like a stateless session bean, a singleton session bean is never passivated.

Message-Driven Beans

- Process messages **asynchronously**.
- Similar to an event listener, implements *javax.jms.MessageListener* interface.



Creating a MDB

```
@MessageDriven(mappedName="jms/Queue")
public class SimpleMessageBean implements MessageListener {
  public void onMessage(Message inMessage) {
    TextMessage msg = null;
    try {
      if (inMessage instanceof TextMessage) {
        msg = (TextMessage) inMessage;
      } else {
        ...
      }
    } catch (JMSException e) {
      e.printStackTrace();
    }
  }
}
```

Sending a Message

```
@Resource(mappedName="jms/ConnectionFactory")
private ConnectionFactory connectionFactory;

@Resource(mappedName="jms/Queue")
private static Queue queue;

JMSContext context = connectionFactory.createContext();

producer = context.createProducer(queue);

Message message = context.createTextMessage();
message.setText("Hello");

producer.send(message);
```

the destination

Asynchronous Methods

- **Session beans can also implement asynchronous methods:** long-running operations, processor-intensive tasks, background tasks, or whenever the invocation result isn't required immediately.
- Declaring an asynchronous method:

```
@Asynchronous
public Future<String> longRunningOperation() {
    //Execute heavy stuff
    ...

    //At some point of the execution (or from time to time)
    //we may check to see if the request was not cancelled
    if (SessionContext.wasCancelled()) ...

    String response = ...;
    return new AsyncResult<>(response);
```

Who gets the response?

Invoking an asynchronous method

```
Future<String> result = asyncService.longRunningOperation();
//The invocation of this method returns immediately
...
//At some point of the excution, we may need the response.
try {
  if (result.isDone()) {
    //Handle successful invocation
    System.out.println("success: " + result.get());

  } else {
    System.out.println("not yet done, let's cancel it...");
    result.cancel();
  }
} catch (InterruptedException | ExecutionException ex) {
    System.err.println(ex);
}
```

Transactions

- A **transaction** is an indivisible unit of work.

```
begin transaction
    add value to account1
    subtract value from account2
commit transaction
```

- Transactions end in a **commit** or a **rollback**.
- The Java Transaction API (JTA) allows applications to access transactions in a manner that is independent of specific implementations.
  - Container-Managed Transactions
  - Bean-Managed Transactions

    → **`javax.transaction.UserTransaction`**

Container-Managed Transactions

The EJB container sets by default the boundaries of the transactions → **Implicit Middleware**.

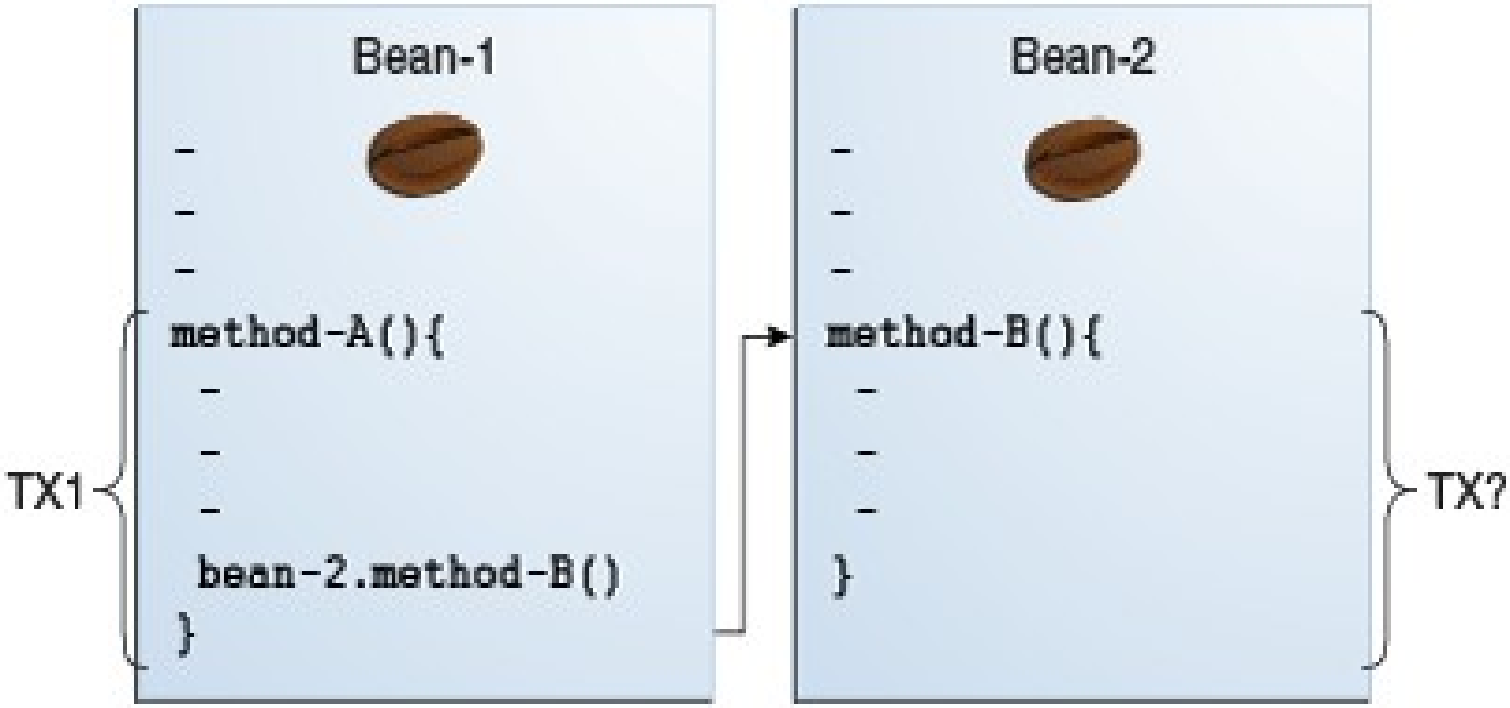
```
@Stateful
public class ShoppingCartBean  {
  @PersistenceContext(name="MyPU")
  private EntityManager em;

  public void save(ShoppingCart shoppingCart) throws Exception {
    em.persist(shoppingCart);
    shoppingCart.setDiscount(100);
  }
}
```

each business method
has an active transaction

Typically, the container begins a transaction immediately before an enterprise bean method starts and commits the transaction just before the method exits.
Each method can be associated with a single transaction.
Nested or multiple transactions are not allowed within a method.

Transaction Scope



method-A begins a transaction and then invokes method-B of Bean-2.
When method-B executes, does it run within the scope of the transaction started by method-A, or does it execute with a new transaction?

Transaction Attributes

- **REQUIRED**: If the client is running within a transaction and invokes the enterprise bean's method, *the method executes within the client's transaction*. If the client is not associated with a transaction, the container *starts a new transaction* before running the method. **(default)**
- **REQUIRES_NEW**: The container will suspends the client's transaction, start a new transaction, delegate the call to the method and then resume the client's transaction after the method completes.
- **MANDATORY**: The method executes within the client's transaction. If the client is not associated with a transaction, the container throws a *TransactionRequiredException*.
- **NOT_SUPPORTED**: The container suspends the client's transaction before invoking the method. After the method has completed, the container resumes the client's transaction.
- **SUPPORTS**: The method executes within the client's transaction. If the client is not associated with a transaction, the container does not start a new transaction before running the method.
- **NEVER**: If the client is running within a transaction and invokes the enterprise bean's method, the container throws a *RemoteException*.

Setting Transaction Attributes

```
@Stateless public class MyFirstEjbBean  {
  @EJB private MySecondEjb ejb2;
  @PersistenceContext(name="MyPU")
  private EntityManager em;

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void createPerson(String name, String deptName){
    Person person = new Person(name);
    em.persist(person);
    Departament dept = ejb2.findDepartament(deptName);
    dept.getPersonList().add(person);
  }
}
```

```
@Stateless public class MySecondEjbBean  {
  @PersistenceContext(name="MyPU")
  private EntityManager em;
  @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
  public Departament findDepartament(String name){
    Query q = Query.createQuery("...").setParameter("name", name);
    try{
      return (Departament) q.getSingleResult();
    } catch(NoResultException nre){ return null; }
  }
}
```

Rolling Back a Container-Managed Transaction

- Runtime exception is thrown → the container will **automatically roll back** the transaction.

```
@Stateless
public class MyFirstEjbBean {
  @PersistenceContext(name="MyPU")
  private EntityManager em;
  public void boom(){
    throw new IllegalStateException("Boom!");
  }
  public void bang(){
    try{
        throw new IllegalStateException("Bang!!");
    } catch(Throwable t){
        ErrorLog log = new ErrorLog(t.getMessage());
        em.persist(log); //???
    }
  }
}
```

- Invoking the *setRollbackOnly* method of the *EJBContext* interface.

Example: DataRepository

```
public abstract class DataRepository<T, ID extends Serializable>
                                implements Serializable {

    protected Class<T> entityClass;

    @Inject
    private EntityManager em;

    protected DataRepository(Class<T> entityClass) {
        this.entityClass = entityClass;
    }

    @PostConstruct
    protected void init() { ... } //why this instead of the constructor?

    public T newInstance() {
        try {
            return entityClass.newInstance();
        } catch (InstantiationException | IllegalAccessException e) {
            //...should throw a custom runtime exception
            return null;
        }
    }
    ...
}
```

Why use @*Inject* instead of @*PersistenceContext?*

DataRepository (continued)

```
public void persist(T entity) {
    em.persist(entity);
}

public void update(T entity) {
    em.merge(entity);
}

public void remove(T entity) {
    if (!em.contains(entity)) {
        entity = em.merge(entity);
    }
    em.remove(entity);
}

public T refresh(T entity) {
    if (!em.contains(entity)) {
        entity = em.merge(entity);
    }
    em.refresh(entity);
    return entity;
}
```

DataRepository (continued)

```
public T findById(ID id) {
    if (id == null) {
        return null;
    }
    return em.find(entityClass, id);
}

public List<T> findAll() {
    String qlString =
       "select e from " + entityClass.getSimpleName() + " e";
    return em.createQuery(qlString).getResultList();
}

public void clearCache() {
    em.getEntityManagerFactory().getCache().evictAll();
}

...
```

DataRepository (usage)

- Creating an actual implementation

```
@Stateless
public class PersonRepository extends DataRepository<Person, Integer> {
    @Inject
    private EntityManager em;

    public PersonRepository() {
        super(Person.class);
    }
    ...
}
```

- Using it in a *managed* component

```
@Named
@SessionScoped
public class PersoaneView extends DataView<Person, Integer> {

    @EJB
    private PersonRepository personRepo;
    ...
}
```

Creating Threads in EJBs

- “The enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread, or to change a thread’s priority or name. The enterprise bean must not attempt to manage thread groups.” [EJB 3.1 spec]
- “An enterprise bean must not use thread synchronization primitives to synchronize execution of multiple instances.”
- Why?

TimerService

```
@Stateless
public class TimerSessionBean {
  @Resource
  TimerService timerService;

  public void createTimer(long milliseconds) {
    Date timeout = new Date(new Date().getTime() + milliseconds);
    timerService.createTimer(timeout, "Hello World!");
  }
  @Timeout
  public void timeoutHandler(Timer timer) {
    logger.info("Timer event: " + timer.getInfo());
  }
  //or simply
  @Schedule(second="*/1", minute="*",hour="*", persistent=false)
  public void doWork(){
    System.out.println("Working hard...");
  }
}
```

The EJB Timer Service allows stateless session beans, singleton session beans, message-driven beans to be registered for timer callback events at a specified time, after a specified elapsed time, after a specified interval, or according to a calendar-based schedule.

EJB Interceptors

```
@Stateful
public class ShoppingCartBean {
  // All the bean's methods will be intercepted by "log()"
  @AroundInvoke
  public Object log (InvocationContext ctx) throws Exception {
    String className = ctx.getTarget().getClass().getName();
    String methodName = ctx.getMethod().getName();
    String target = className + "." + methodName + "()";
    long t1 = System.currentTimeMillis();
    try {
      return ctx.proceed();
    } catch(Exception e) {
      throw e;
    } finally {
      long t2 = System.currentTimeMillis();
      System.out.println(target + " took " +
        (t2-t1) + "ms to execute");
    }
  }
}
```

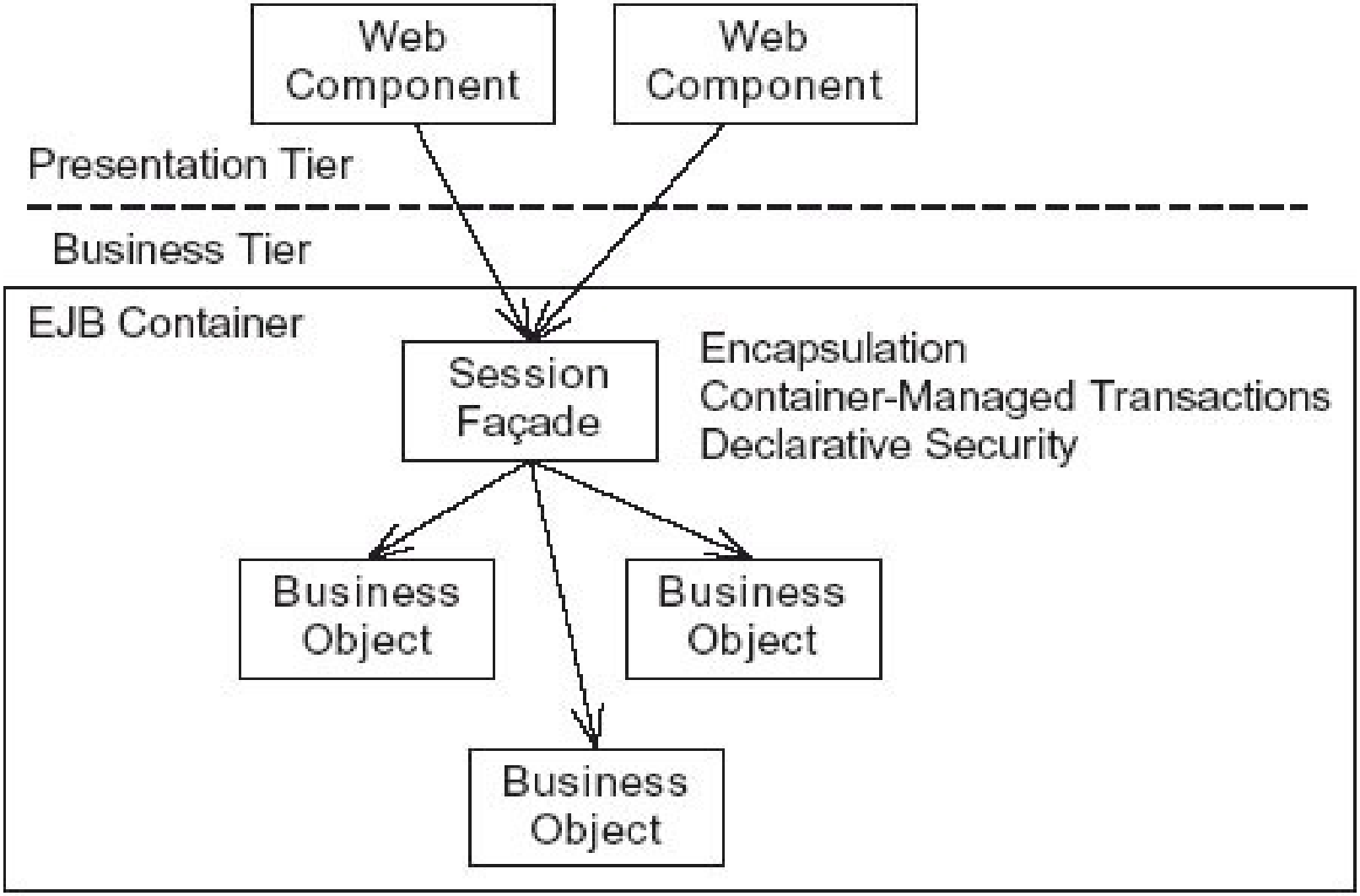
Security Services (to be continued...)

```
@Stateless
@SecurityDomain("users")
public class ShoppingCartBean {
  @RolesAllowed({"AdminUser"})
  public void addProduct (Product product, int quantity) {
    // ... ...
  }

  @RolesAllowed({"RegularUser"})
  public float getDiscount (Product product) {
    // ... ...
  }

  @PermitAll
  public List<Product> getProducts () {
    // ... ...
  }
}
```

EJB Session Facade



When to Use EJBs?

- When the application is **distributed** across multiple servers.
- When the application is **performance-centric**, EJBs and the application server provide high performance and very good scalability.
- When **transaction management is required** to ensure data integrity.
- When you need to **declaratively manage security.**
- When you want a **pure separation between presentation and business logic.**