Advanced Software Tools Summary

Chapter 1:

- An enterprise application is a **software application** typically used in **large** business organizations.
    - Often provide the following features:
        - Support for concurrent users and external systems.
        - Support for scalability.
        - Distributed platform to ensure high availability.
        - Support for highly secure access.
        - Ability to integrate with back-end systems and web services.
- Java Enterprise Edition (Java EE) is a specification for developing **enterprise applications** using **Java** that is developed under the guidance of the **Java Community Process** (JCP).
    - Benefits of using Java EE:
        - Applications can be developed and will run on many different types of operating systems.
        - Applications are portable across Java EE compliant application servers due to the Java EE standard.
        - The Java EE specification provides many APIs typically used by enterprise applications such as web services, asynchronous messaging, transactions, database connectivity, thread pools, batching utilities, and security. There is no need to develop these components manually, so reducing development time.
- Java SE is generally used to develop **stand-alone programs**, tools, and utilities that are mainly run from the command line, GUI programs, and server processes that need to run as daemons.
- It is important to note that unlike Java SE, Java EE is mainly a set of standard specifications for an API, and runtime environments that implement these APIs are generally called as application servers.
- **Java Archive (JAR) files:** Individual modules of an application and Enterprise Java Beans (EJBs) can be deployed as separate JAR files.
- **Web Archive (WAR) files:** If your Java EE application has a web-based front end.
- **Enterprise Archive (EAR) files:** It is essentially a compressed file with one or more WAR or JAR files and some XML deployment descriptors inside it.
- To run a stand-alone application that uses only the Java SE API, you can use the **java -jar** command.

- If an older version of the WAR file is already deployed, the old version is undeployed and the new version is deployed without restarting the application server. Such a process is called **hot deployment** and is used extensively during development, testing, as well as in production rollouts.
- The Java Community Process (JCP) is an open, participatory process to develop, maintain, and revise Java technology specifications. It manages the specification of many APIs using Java Specification Requests (JSR).
- Any individual or organization can join the JCP and participate in the standardization process.
- Expert Groups (EG) manage the development and evolution of JSRs under the leadership of a Specification Lead (SL).
- Java EE applications are designed with a multi-tier architecture in mind. The application is split into components, each serving a specific purpose.
- Each component is arranged logically in a tier, some of the tiers run on separate physical machines or servers.
- The advantage of using tiered architectures is that as the application scales to handle more and more end users.
- In a classic web-based Java EE application architecture, there are four tiers:
    - **Client Tier:** This is usually a browser for rendering the user interface on the end-user machines.
    - **Web Tier:** The web tier components run inside an application server and generate HTML or other markup that can be rendered or consumed by components in the client tier.
    - **Business Logic Tier:** The components in the business logic tier contain the core business logic for the application.
    - **Enterprise Information Systems (EIS) Tier:** Many enterprise applications store and manipulate persistent data that is consumed by multiple systems and applications within an organization.
- Web-centric architecture:
    - This type of architecture is for simple applications with a browser-based front end and a simple back end powered by Servlets, Java Server Pages (JSP), or Java Server Faces (JSF).
- Combined web and business logic component-based architecture:
    - A browser in the client tier interfaces with a web tier consisting of Servlets, JSPs, or JSF pages, which are responsible for rendering the user interface, controlling page flow, and security.

- Business-to-Business architecture (B2B):
    - The front end is usually not an interactive graphical user interface (GUI) that is accessed by end users, but an internal or external system that integrates with the application and exchanges data using a mutually understood standard protocol.
- Web service application architecture:
    - The application provides an API that is accessed over an HTTP-based protocol such as SOAP or REST via a set of services (endpoints) corresponding to the business function of the application.

Chapter 2:

- An application server is a software component that provides the necessary runtime environment and infrastructure to host and manage Java EE enterprise applications.
- The application server provides many features.
- In a Java SE application, these features must be implemented manually by the developer, which is time consuming and difficult to implement correctly.
- An important concept of the EAP architecture is the concept of a module. A module provides code (Java Classes) to be used by EAP services or by applications.
- A container is a logical component within an application server that provides a runtime context for applications deployed on the application server.
- A container acts as an interface between the application components and the low-level infrastructure services provided by the application server.
- There are two main types of containers within a Java EE application server:
    - Web containers
    - EJB containers
- Packaging and Deploying Java EE Applications:
    - **JAR files:** When deployed into an application server, depending on the type of components inside the JAR file, the application server looks for XML deployment descriptors, or code-level annotations, and deploys each component accordingly.
    - **WAR files:** A WAR file is used for packaging web applications.
    - **EAR files:** An EAR file contains multiple JAR and WAR files, as well as XML deployment descriptors.

Chapter 3:

- An Enterprise Java Bean (EJB) is a Java EE component typically used to encapsulate **business logic** in an enterprise application.
- Unlike simple Java beans in Java SE, where concepts such as multi-threading, concurrency, transactions, and security must be explicitly implemented by the developer.
- In an EJB, the application server provides these features at runtime and enables the developer to focus on writing the business logic for the application.
- The Java EE specification defines two different types of EJBs:
    - **Session:** Performs an operation when called from a client.
    - **Message Driven Bean (MDB):** Used for asynchronous communication between components in a Java EE application.
- A bean in EJB is a component offering one or more business interfaces to potential clients of that component.
- A Session Bean provides an interface to clients and encapsulates business logic methods that can be invoked by multiple clients either locally or remotely over different protocols.
- There are **three** different types of session beans, depending on the application use case, that can be deployed on a Java EE compatible application server:
    - **Stateless Session Beans (SLSB):**
        - A stateless session bean does not maintain state with clients between calls.
        - When clients interact with the stateless session bean and invoke methods on it, the application server allocates an instance from a pool of stateless session beans, which are pre-instantiated.
        - Once a client completes the invocation and disconnects, the bean instance is either released back into the pool or destroyed.
        - A stateless session bean is useful in scenarios where the application must serve many clients concurrently accessing the bean's business methods.
        - Each bean can be accessed or invoked my multiple clients.
        - Example: “search” in a shopping website without login.

- **Stateful Session Beans (SFSB):**
    - Stateful session beans maintain state with clients across multiple calls.
    - There is a one-to-one relationship between the number of Stateful bean instances and the number of clients.
    - When a client completes the interaction with the bean and disconnects, the bean instance is destroyed.
    - A new client results in a new stateful bean with its own unique state.
    - The application server ensures that each client receives the same instance of a stateful session bean for each method call.
    - Example: “add to cart” in a shopping website.
- **Singleton Session Beans:**
    - Singleton session beans are instantiated once per application and exists for the lifecycle of the application.
    - Every client request for a singleton bean goes to the same instance.
    - Singleton session beans are used in scenarios where a single enterprise bean instance is shared across multiple clients.
    - For example, database connections, JNDI lookups, JMS remote connection factory creation, and many more.

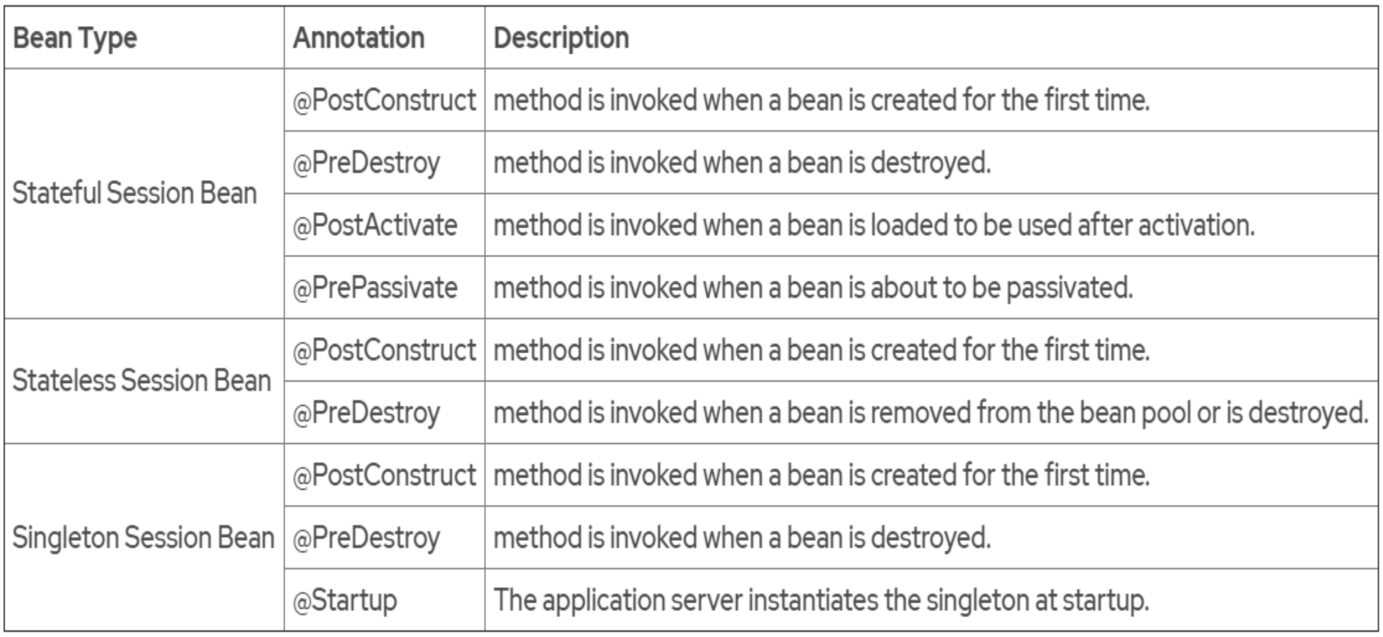
- Message Driven Beans:
    - A Message Driven Bean (MDB) enables Java EE applications to process *messages* **asynchronously** (does not happen at the same time).
    - Once deployed on an application server, it listens for JMS messages and for each message received, it performs an action (the **onMessage()** method of the MDB is invoked).
    - MDBs provide an **event driven** model for application development.
    - The MDB is **stateless** and does not maintain any state with clients.
    - The application server maintains a pool of MDBs and manages their lifecycle by assigning and returning instances from and to the pool.
    - For example: two systems can communicate by passing messages in an asynchronous manner, which ensures that the two systems can independently evolve without impacting each other.
- EJBs are instantiated by the EJB container, so they **cannot use a constructor that relies on arguments**.
- An EJB is a portable component containing business logic running on an application server.
- If the client and EJB are **local**, that is, they are running within the same JVM process, the client can invoke all public methods in the EJB. In cases where the EJB is **remote**, a Remote Interface, which is a simple Java interface that exposes the business methods of an EJB, must be provided.

- Locally: Clients can invoke methods on the EJB by injecting the EJB directly into code using the **@EJB** annotation.
- Remotely:
  - Declare an interface listing the business methods of the EJB.
  - Let the EJB implement and override these methods.
  - Use JNDI to lookup the EJB.
- Looking Up Remote EJBs Using JNDI:
  - Java EE specified a standard JNDI lookup scheme for clients to look up EJBs.

```
/<application-name>/<module-name>/<bean-name>!<fully-qualified-interface-name>
```

  - **application-name:** The application name is the name of the EAR that the EJB is deployed in. If the EJB JAR is not deployed in an EAR, then this is blank.
  - **module-name:** By default, the module name is the name of the EJB JAR file.
  - **bean-name:** The name of the EJB to be invoked (the implementation class).
  - **fully-qualified-interface-name:** The fully qualified class name of the remote interface. Include the full package name.
- Describing the Stateful Session Bean Life Cycle:
  - **Does Not Exist:** The stateful EJB is not created and does not exist in application server memory.
  - **Ready:** The stateful EJB (object) is created in application server memory either by JNDI call or CDI injection and is ready to have its business methods invoked by clients.
  - **Passivated:** Since a stateful EJB has object state that is persisted across multiple client calls, the application server may decide to passivate (deactivate) the EJB to secondary storage to optimize memory consumption.
- Describing the Stateless Session Bean Life Cycle:
  Typically, the application server creates and maintains a pool of stateless session EJB instances in memory as a performance optimization.
  - **Does Not Exist:** The stateless EJB is not created and does not exist in application server memory.
  - **Ready:** The stateless EJB (object) is created in application server memory either by JNDI call or CDI injection and is ready to have its business methods invoked by clients.
- Describing the Singleton Session Bean Life Cycle:
  - **Does Not Exist:** The singleton is not created and does not exist in application server memory.
  - **Ready:** The singleton EJB (a single object) is created in application server memory at startup, or by CDI injection and is ready to have its business methods invoked by clients.

- <u>The Bean Life Cycle Annotations:</u>



Bean Type	Annotation	Description
Stateful Session Bean	@PostConstruct	method is invoked when a bean is created for the first time.
	@PreDestroy	method is invoked when a bean is destroyed.
	@PostActivate	method is invoked when a bean is loaded to be used after activation.
	@PrePassivate	method is invoked when a bean is about to be passivated.
Stateless Session Bean	@PostConstruct	method is invoked when a bean is created for the first time.
	@PreDestroy	method is invoked when a bean is removed from the bean pool or is destroyed.
Singleton Session Bean	@PostConstruct	method is invoked when a bean is created for the first time.
	@PreDestroy	method is invoked when a bean is destroyed.
	@Startup	The application server instantiates the singleton at startup.

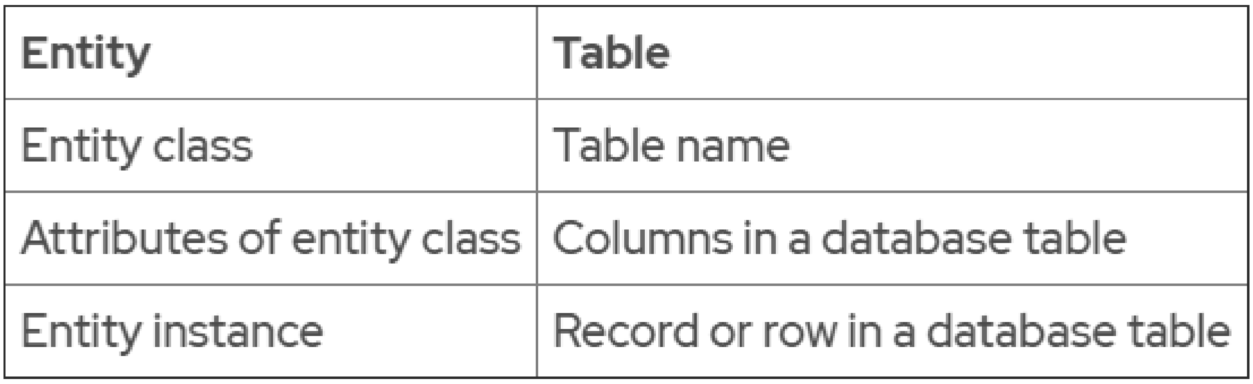
- <u>Understanding Transactions in Java EE Applications:</u>
  - A transaction is a series of actions that must be executed as a single atomic unit, it follows an "all or nothing" approach.
  - Transactions ensure that data integrity is maintained by controlling concurrent access to the data.
  - Transactions ensure that a failed business transaction does not leave the system in an inconsistent or invalid state.
  - If all the methods in the transaction are executed without any errors or failures, the transaction is **Committed**.
  - There are two different ways to manage transactions in Java EE:
    - **Implicit or Container Managed Transaction (CMT):** The application server manages the transaction boundary and automatically commits and rolls back transactions without the developer writing code for managing transactions. This is the default option unless explicitly overridden by the developer.
    - **Explicit or Bean Managed Transaction (BMT):** The transactions are managed by the developer in code at the bean level (in EJBs). The developer is responsible for controlling the transaction scope and boundary explicitly.

- Setting Transaction Attributes (CMT):
    - **@TransactionAttribute(TransactionAttributeType.REQUIRED):**
        - If there is a transaction, then the function executes within the same transaction.
        - If there is no transaction when the function is called, then the application server starts a new transaction before executing this function.
    - **@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW):**
        - If there is a function running within a transaction, the application server suspends this transaction and starts a new transaction before executing the new function. When the new function finishes executing, control moves back to the old function and the suspended transaction resumes.
        - If there is no transaction when the function is called, then the application server starts a new transaction before executing this function.
    - **@TransactionAttribute(TransactionAttributeType.MANDATORY):**
        - If there is a transaction, then the function executes within the same transaction.
        - If there is no transaction when the function is called, then the application server throws a TransactionRequiredException.
    - **@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED):**
        - If there is a function running within a transaction, the application server suspends this transaction and runs the other function without any transaction context.
        - If there is no transaction when the function is called, then the application server does not start a new transaction.
    - **@TransactionAttribute(TransactionAttributeType.SUPPORTS):**
        - If there is a transaction, then the function executes within the same transaction.
        - If there is no transaction when the function is called, then the application server does not start a new transaction.
    - **@TransactionAttribute(TransactionAttributeType.NEVER):**
        - The application server throws a RemoteException.
        - If there is no transaction when the function is called, then the application server does not start a new transaction.

- Understanding Bean Managed Transaction (BMT) Semantics:
    - You have fine-grained control over when a transaction begins and ends, and control when to commit and rollback.
    - You can use the **begin()**, **commit()**, and **rollback()** methods from the **javax.transaction.UserTransaction** interface to control the transaction boundaries and scope explicitly.

Chapter 4:

- Persistence:
    - When an application stores data in a permanent store like a flat file, XML file, or a database for durability, it is known as **persistence**.
    - Business data in a Java EE enterprise application is defined as **Java objects**.
    - These objects are preserved in corresponding database tables.
    - Java objects and database tables use different data types, such as a **String** in Java and **Varchar** in a database, to store business data.
        - This is called as **impedance mismatch.**
- The technique to automate bridging the impedance mismatch is known as **Object Relational Mapping (ORM).**
- ORM software uses metadata to describe mapping between the classes defined in an application and the schema of a database table.



Entity	Table
Entity class	Table name
Attributes of entity class	Columns in a database table
Entity instance	Record or row in a database table

- An **entity** is a lightweight domain object that is persist-able. An entity class is mapped to a table in a relational database.
- Each instance of an entity class has a primary key field.
- The primary key field is used to map an entity instance to a row in a database table.
- In Java, an entity is a Plain Old Java Object (POJO) class that is annotated with **@Entity** annotation.
- **@Entity:** specifies that a class is an entity.
- **@Table(name="x"):** when the name of an entity class is different from the name of a table in the database.
- **@Column(name="x"):** to map a field or property to a column in the database if their names are different.
- **@Temporal(TemoralType.Date):** used with a **Date** type of attribute.
- **@Transient:** to specify a non-persistent field.
- **@Id:** to specify the primary key.
- **@EmbeddedId** or **@IdClass:** to specify the composite primary key.
- **@Id** annotation is used to specify a simple primary key. **@GeneratedValue** annotation is applied to the primary key field or property to specify the primary key generation strategy.

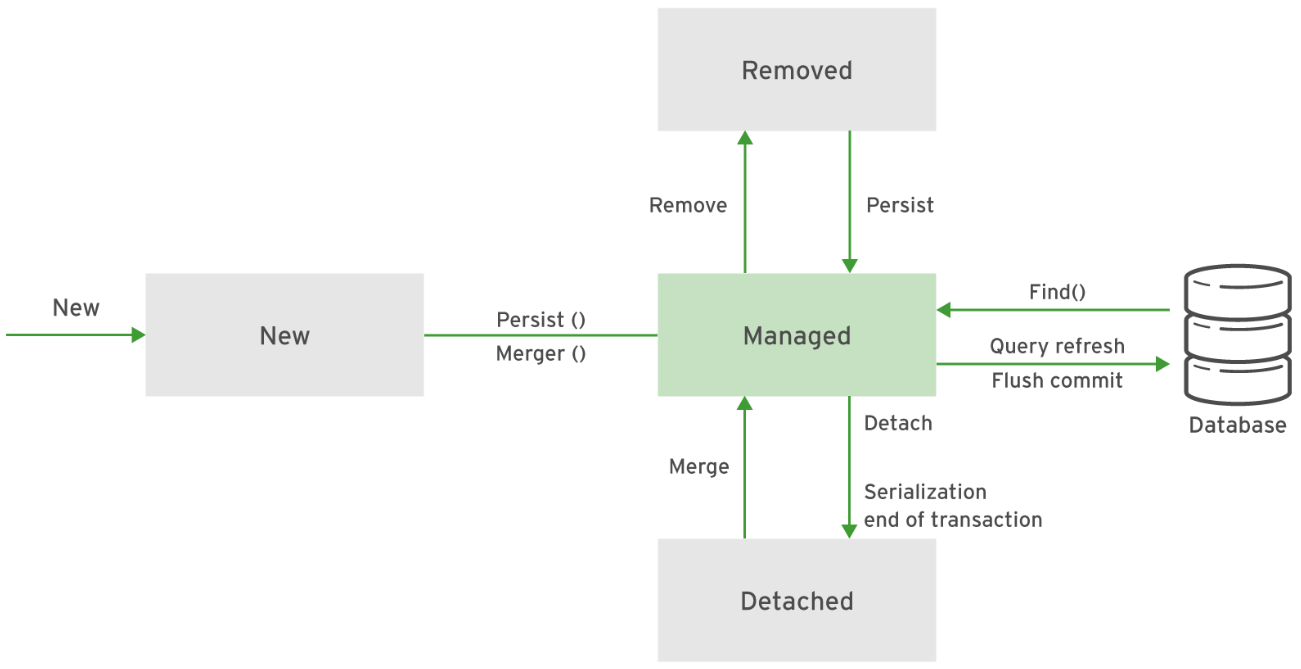
- **GenerationType.AUTO:**
    - The JPA provider uses any strategy of its choice to generate the primary key.
- **GenerationType.SEQUENCE:**
    - The JPA provider uses the database sequence to generate the primary key.
- **GenerationType.IDENTITY:**
    - The JPA provider uses the database identity column to generate the primary key.
- **GenerationType.TABLE:**
    - The JPA provider uses database ID generation table.
- The **EntityManager** API is defined to perform persistence operations.
- An entity manager works within a set of managed entity instances. These managed entity instances are known as the entity manager's **persistence context**.
- An entity manager obtains the reference to an entity and performs the actual **CRUD** (Create, Read, Update, and Delete) operations on the database.
- An **EntityManager** instance can be obtained from an **EntityManagerFactory** object.
- The **persistence.xml** is a configuration file that contains information about the entity classes, data source, transaction type, and other configuration information.

```
@Stateless
public class ItemService {
  //ItemPU is the name of the persistence unit
    EntityManagerFactory emFactory =
 Persistence.createEntityManagerFactory("ItemPU");
    EntityManager em = emFactory.createEntityManager();
    ....
}
```

```
 public class EMProducer {
      @Produces
      @PersistenceContext(unitName= "ItemPU")
      private EntityManager em;
 }
```

- A Persistence unit is configured in a persistence.xml file in the application's META-INF directory. A Persistence unit contains information about the persistence unit name, data source, and transactions type.
- Database provider is not defined in the persistence.xml file.
- persistence-unit name, transaction type, jta-data-source and properties element are defined in the persistence.xml file.

- EntityManager Interface and Key Methods:
  - persist() throws **PersistenceException** if persist operation fails.
    - **entityManager.persist(customer);**
  - find() method searches an entity of a specific class by its primary key and returns a managed entity instance. If the object is not found, it returns a null.
    - customer = **entityManager.find(Customer.class,custId);**
  - contains() – returns true or false.
    - **return entityManager.contains(customer);**
  - merge() – update
    - **entityManager.merge(customer);**
  - remove() – remove.
    - **entityManager.remove(customer);**
  - clear() – detach data from memory.
    - **entityManager.clear();**
  - refresh() – to see the update.
    - **entityManager.refresh(customer);**



- **Bean validation** is a model for validating data in Java objects by using built-in and custom annotations that can apply predefined constraints.
- **Validation constraints** are the rules that are applied to validate data.
- These constraints are applied in the form of **annotations** to attributes, methods, properties, or constructors.
- The **javax.validation.constraints** package contains several built-in constraints.

Annotation	Description	Example

`@NotNull`	`@NotNull` annotation verifies that the value in the field or property is not null.	`@NotNull`
		
		`private String itemName;`

`@Null`	`@Null` annotation verifies that the value in the field or property is null.	`@Null`
		
		`private String comments;`

`@Size`	`@Size` annotation verifies that the size of the field is between the `min` and `max` including the boundary values.	`@Size (min=3, max=40)`
		
		`private String name;`

`@Min`	`@Min` annotation verifies that the value in the field or property is greater than or equal to the value defined in the `Min`. The value is a required element of the `long` type.	`@Min(100)`
		
		`private int min Stock;`

`@Max`	`@Max` annotation verifies that the value in the field or property is lesser than or equal to the value defined in the `Max`. The value is a required element of the `long` type.	`@Max(1000)`
		
		`private int max Stock;`

`@Digits`	`@Digits` annotation verifies the precision and scale of the field. The field must be a number within the specified range. The range is defined by the `integer` and the `fraction` elements. An integer and the fraction are required elements of an `int` type.	`@Digits (integer=7, fraction=2)`
		
		`private double monthlySale;`

`@DecimalMin`	`@DecimalMin` annotation verifies if the value in the field or property is a decimal value greater than or equal to the value defined in the `DecimalMin`. The value is a required element of the `String` type.	`@DecimalMin ("8.5")`
		
		`private double minTax;`

`@DecimalMax`	`@DecimalMax` annotation verifies if the value in the field or property is a decimal value lesser than or equal to the value defined in the `DecimalMax`. The value is a required element of the `String` type.	`@DecimalMax ("19.5")`
		
		`private double maxTax;`

Annotation	Description	Example

`@Future`	`@Future` annotation verifies if the value in the field or property is a date in the future.	`@Future`
		`private Date promotionDate;`

`@Past`	`@Past` annotation verifies if the value in the field or property is a date in the past.	`@Past`
		`private Date startDate;`

`@Pattern`	`@Pattern` annotation verifies if the value in the field or property matches the `regexp` expression. The regular expression is a required element of the `String` type.	`@Pattern (regexp="\\(\\d{3}\\)\\d{3}-\\d{4}")`
		`private String phoneNumber;`

`@AssertFalse`	`@AssertFalse` annotation verifies if the value in the field or property is false.	`@AssertFalse`
		`private boolean isAvailable;`

`@AssertTrue`	`@AssertTrue` annotation verifies if the value in the field or property is true.	`@AssertTrue`
		`private boolean isOrdered;`

- Creating Queries:
  - Java Persistence Query Language (**JPQL**) is a platform-independent query language defined as a part of the JPA specification to perform queries on entities in an object-oriented manner.
  - JPQL is like SQL in syntax, but JPQL queries are expressed in terms of Java entities rather than database tables and columns.
  - JPQL supports the **SELECT**, **UPDATE**, and **DELETE** statements.
  - The JPQL query to retrieve records of all employees from the database is as follows:
    - SELECT e FROM Employee e;

- Dynamic queries are created at runtime by an application using the following process:
  - Create a string containing a JPQL query.
  - Pass the string to the entity manager's createQuery method and store the returned Query object.
  - Use the query's getResultList() method to execute the query and return the selected rows from the database.

```
String simpleQuery="SELECT e from Employee e";

Query query=entityManager.createQuery(simpleQuery);

List<Employee> persons = query.getResultList();
```

- Database functions such as LOWER, UPPER, LENGTH, as well as arithmetic functions, can also be applied to JPQL queries:

```
•  Query query=entityManager.createQuery("SELECT UPPER(e.empName) from Employee e");
```

- JPA also supports the **TypedQuery<?>** class, which allows static typing of queries to avoid any issues with casting results.
- To create a **TypedQuery**, pass the class that should match the type of the query results into the **createQuery** method.

```
TypedQuery<Employee> query=entityManager.createQuery("SELECT e from Employee e
 where e.salary >?1 or e.empName=?2", Employee.class);
```

- The **WHERE** clause is used to define the conditions on the data that the query returns.
- **<, =, >, <=, >=, <>** are used to compare the arithmetic values.
- **IN** and **NOT IN** are used for all types. **IN** operator is used to determine whether the data in a field is one of the values provided in a list of values.
- **LIKE** and **NOT LIKE** are used for string values. It is used to determine whether data in a field matches a sequence of characters provided in the string.
- **BETWEEN** and **NOT BETWEEN** are used for arithmetic, date, time, and string values. It is used to determine whether data in a field lies in a certain range of values.
- **MEMBER OF**, **NOT MEMBER OF**, **IS EMPTY**, and **IS NOT EMPTY** are used for **Collection** types.

- Named parameters in queries:

```
public List<Employee> getEmployeesWithGreaterSalary(double salary) {
 Query query=entityManager.createQuery("SELECT e from Employee e where e.salary
 >:sal");
 query.setParameter("sal", salary);
 List<Employee> persons = query.getResultList();

 return persons;
}
```

- Positional parameters in queries:

```
public List<Employee> getAllPersonsWithPositionParam(double salary) {
 Query query=entityManager.createQuery("SELECT e from Employee e where e.salary >?
1");
 query.setParameter(1, salary);
 return query.getResultList();
}
```

- Named queries:

```
@Entity
@NamedQuery(
 name="getAllEmployees",
 query="select e from Employee e where e.salary > :sal")
public class Employee implements Serializable{

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
 private String empName;
 private double salary;
```

```
  public List<Employee> getPersonsWithNamedQuery(double salary) {
 Query query=entityManager.createNamedQuery("getAllEmployees")
 query.setParameter("sal", salary);
 retrun query.getResultList();
}
```

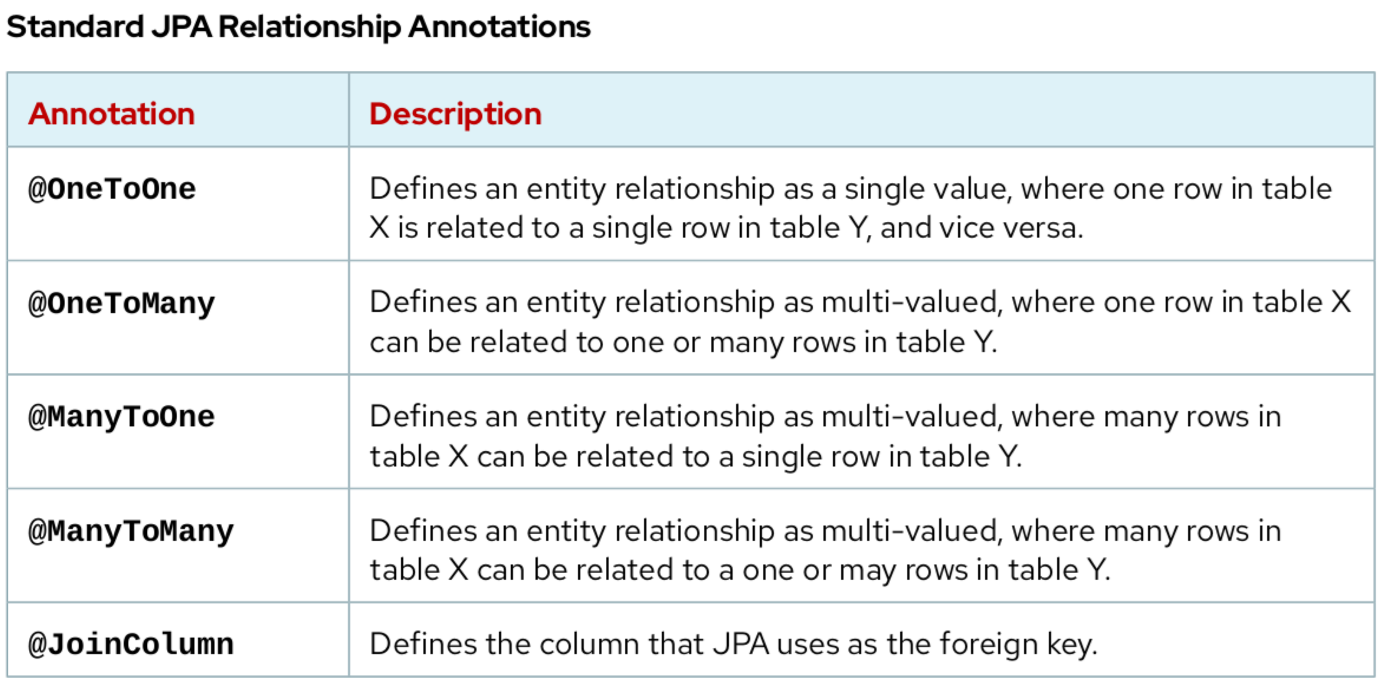
```
@Entity
@NamedQueries({
 @NamedQuery(name="getAllEmployees",
  query="select e from Employee e where e.salary > :sal"),
 @NamedQuery(name="getEmployeesWithSalaryOrName",
  query="select e from Employee e where e.salary > :sal or e.empName=:name")
})
public class Employee implements Serializable{

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
```

```
public List<Employee> getAllPersonsWithNamedQueries(double salary, String ename) {

 Query query=entityManager.createNamedQuery("getEmployeesWithSalaryOrName");
 query.setParameter("sal", salary);
 query.setParameter("name", ename);
 List<Employee> persons = query.getResultList();

      return persons;
}
```

*Note: any dependencies needed should be identified in the pom.xml file, to let th mvn download the neeeded jar files.

Chapter 5:

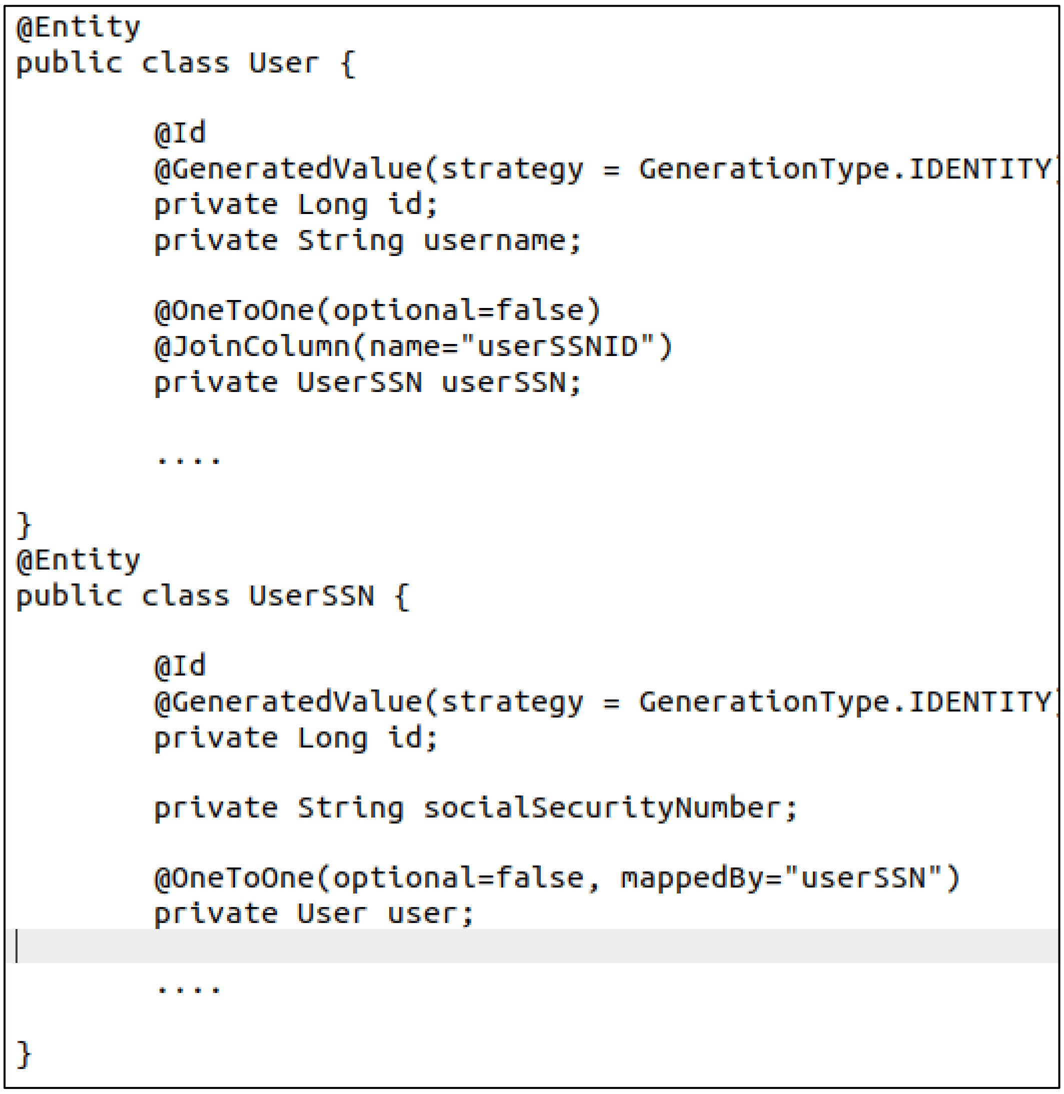
- When building enterprise applications, developers use relational databases to store business data created and updated using the application. Application data typically spans **multiple database tables**, so it is common for data in one table to need to reference data in another.
- A **foreign key** is a column where the value of the data in the column is a reference to the ID or primary key of a row in another table.
- When representing database tables in Java EE, developers use **entity beans**, one for each table. To create a relationship between two entities, class-level variables are used to represent an instance of one entity as an attribute of another entity.



Standard JPA Relationship Annotations

Annotation	Description
`@OneToOne`	Defines an entity relationship as a single value, where one row in table X is related to a single row in table Y, and vice versa.
`@OneToMany`	Defines an entity relationship as multi-valued, where one row in table X can be related to one or many rows in table Y.
`@ManyToOne`	Defines an entity relationship as multi-valued, where many rows in table X can be related to a single row in table Y.
`@ManyToMany`	Defines an entity relationship as multi-valued, where many rows in table X can be related to a one or may rows in table Y.
`@JoinColumn`	Defines the column that JPA uses as the foreign key.

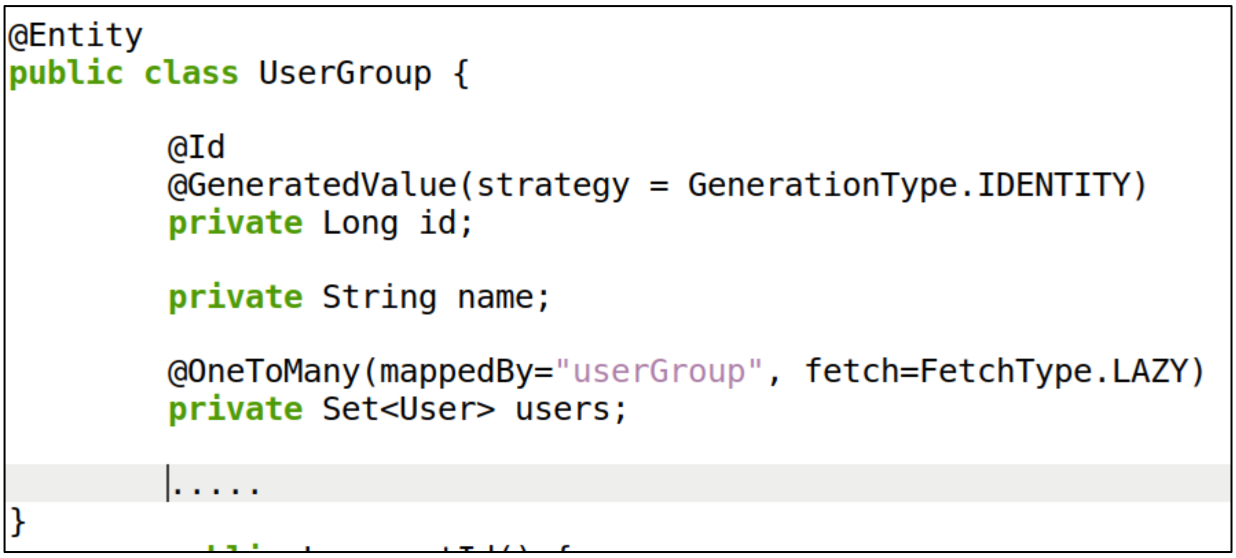
- Use the **@OneToOne** annotation when two entities relate to each other such that an instance of one entity only relates to a single instance of the other entity.



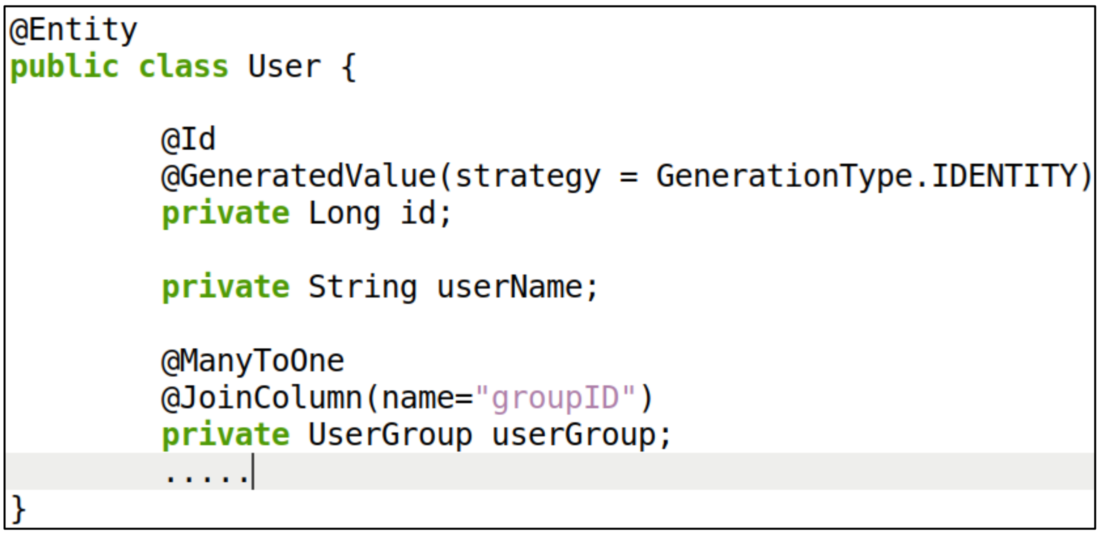
```
@Entity
public class User {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY
        private Long id;
        private String username;

        @OneToOne(optional=false)
        @JoinColumn(name="userSSNID")
        private UserSSN userSSN;

        ....

}
@Entity
public class UserSSN {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY
        private Long id;

        private String socialSecurityNumber;

        @OneToOne(optional=false, mappedBy="userSSN")
        private User user;

        ....

}
```

- The **@OneToOne** annotation tells JPA that a single instance of **UserSSN** is related to each **User** instance. The **optional option** tells JPA that this field is required, and an error is thrown if the database contains a row without a value for this column.

- Use the **@OneToMany** and **@ManyToOne** annotations to map a JPA relationship anytime two entities relate to each other such that one instance of one entity relates to potentially multiple instances of the other entity.



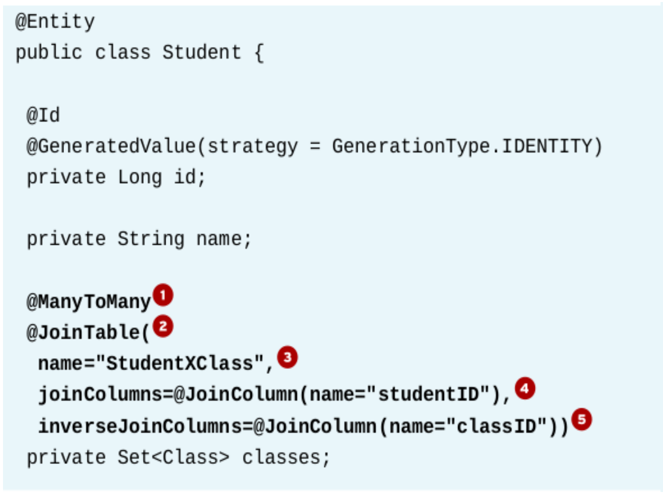
```
@Entity
public class UserGroup {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        private String name;

        @OneToMany(mappedBy="userGroup", fetch=FetchType.LAZY)
        private Set<User> users;

        .....
}
```



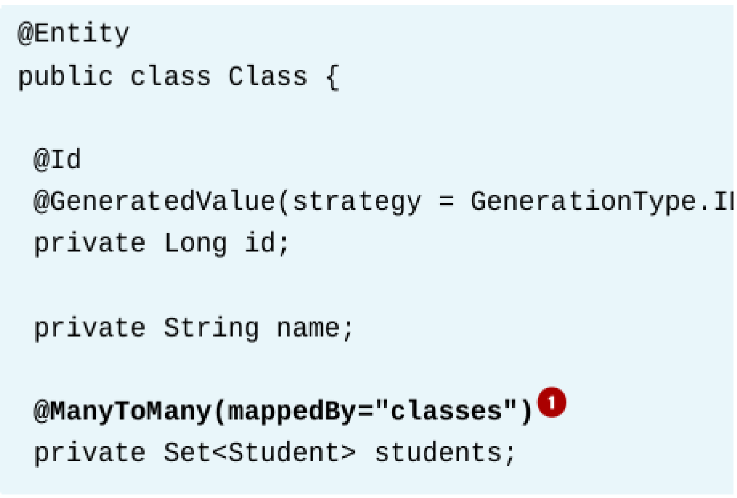
```
@Entity
public class User {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        private String userName;

        @ManyToOne
        @JoinColumn(name="groupID")
        private UserGroup userGroup;
        .....
}
```

- Nesting relationships or complex database schemas with lots of relationships can be especially harmful to performance, as sometimes multiple queries must be executed to retrieve necessary data, or complex joins must be used. It is also very easy to use JPA to retrieve more data than intended, especially if every relationship is mapped bidirectionally on both entities.
- Even if one entity has a relationship mapped to another entity with JPA annotations, it is not always necessary to retrieve that related entity when loading the data for the first entity from the database. This depends on the business logic that processes the instance of the entity being loaded. For example, different screens in an application might require different amounts of information required to display or have different performance requirements.
- If ***fetch=FetchType.LAZY*** is used, JPA does not attempt to load all its instances.
- If ***fetch=FetchType.EAGER*** is used, JPA will load all its instances.

- A many-to-many relationship occurs when each row in one table has multiple related rows in another table, and vice versa.
- Instead of using a single **@JoinColumn** JPA, use a **@JoinTable** annotation and two **@JoinColumn** annotations to map a many-to-many relationship. The **@JoinTable** annotation is the bridge that allows JPA to use the join-table to populate the relationships between two objects, and the two **@JoinColumn** annotations are used to define how to join the join-table back to the entity table, as well as to the related entity table.
- **name**
    - The name of the join-table to be used by JPA when populating the relationship.
- **joinColumns**
    - The column of the join-table to use to join back to the entity class. This attribute accepts an instance of @JoinColumn, which is defined with a name attribute that maps to the column name on the join-table.
- **inverseJoinColumns**
    - The column of the join-table to use to join from the join-table to the related entity class. This attribute accepts an instance of @JoinColumn, defined with a name attribute that maps to the column name on the join-table.



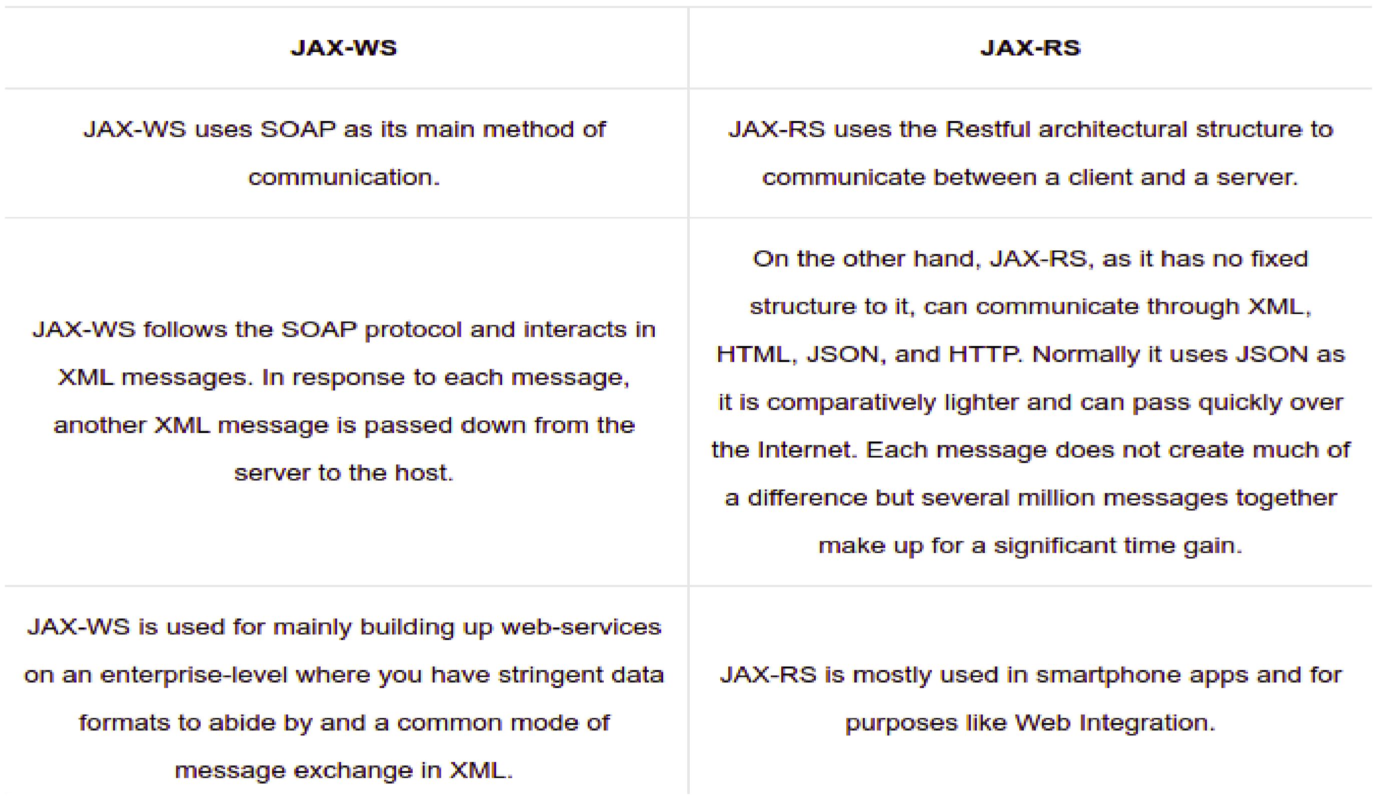
```
@Entity
public class Student {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @ManyToMany❶
 @JoinTable(❷
  name="StudentXClass",❸
  joinColumns=@JoinColumn(name="studentID"),❹
  inverseJoinColumns=@JoinColumn(name="classID"))❺
 private Set<Class> classes;
```



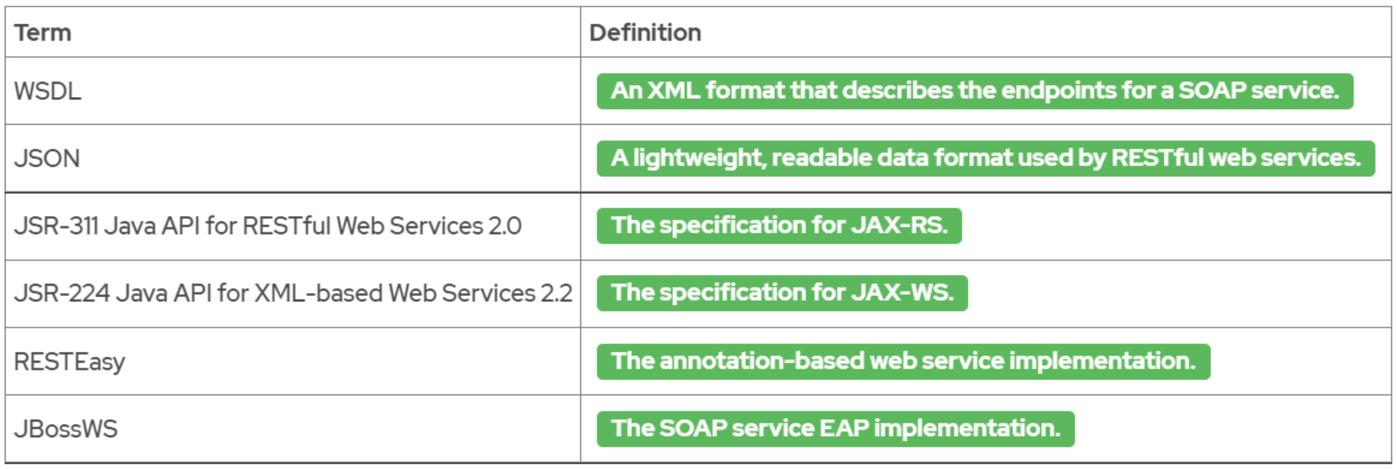
```
@Entity
public class Class {

 @Id
 @GeneratedValue(strategy = GenerationType.I
 private Long id;

 private String name;

 @ManyToMany(mappedBy="classes")❶
 private Set<Student> students;
```

Chapter 6:

- Web services expose standardized communication for interoperability between application components over HTTP. By abstracting applications into individual components that communicate across web services, each system becomes loosely coupled to each other. This separation provides a greater ability to modify applications or integrate new systems into the application. Using a standard format for data transfer, such as JSON or XML, allows applications that consume web services to require only the ability to make an HTTP request to the service and to process the service's response.
- Types of web services:



JAX-WS	JAX-RS
JAX-WS uses SOAP as its main method of communication.	JAX-RS uses the Restful architectural structure to communicate between a client and a server.
JAX-WS follows the SOAP protocol and interacts in XML messages. In response to each message, another XML message is passed down from the server to the host.	On the other hand, JAX-RS, as it has no fixed structure to it, can communicate through XML, HTML, JSON, and HTTP. Normally it uses JSON as it is comparatively lighter and can pass quickly over the Internet. Each message does not create much of a difference but several million messages together make up for a significant time gain.
JAX-WS is used for mainly building up web-services on an enterprise-level where you have stringent data formats to abide by and a common mode of message exchange in XML.	JAX-RS is mostly used in smartphone apps and for purposes like Web Integration.



Term	Definition
WSDL	An XML format that describes the endpoints for a SOAP service.
JSON	A lightweight, readable data format used by RESTful web services.
JSR-311 Java API for RESTful Web Services 2.0	The specification for JAX-RS.
JSR-224 Java API for XML-based Web Services 2.2	The specification for JAX-WS.
RESTEasy	The annotation-based web service implementation.
JBossWS	The SOAP service EAP implementation.

- Creating RESTful Web Services:
    - The first step to create the web service is to create a class that extends the class javax.ws.rs.core.Application.
    - In addition to declaring the RESTful web service, the new subclass is used to also define the base URI for the web service with the @ApplicationPath annotation.

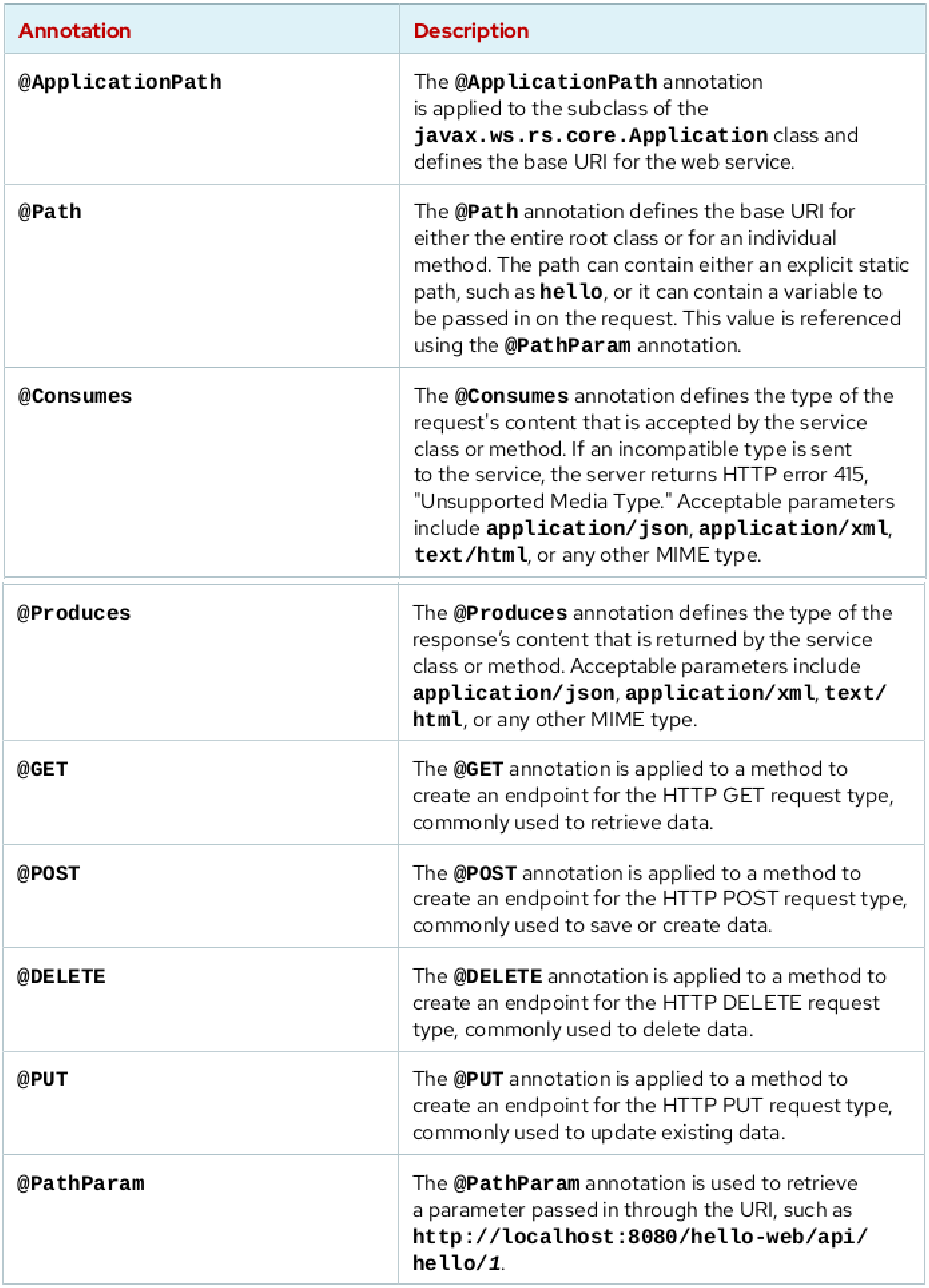
```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/api")
public class Service extends Application {
 //Can be left empty
}
```

- Using the available JAX-RS annotations is an easy way to create a RESTful web service from an existing POJO class.

```
public class HelloWorld {

  public String hello() {
     return "Hello World!";
  }
}
```

```
@Stateless
@Path("hello")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class HelloWorld {

  @GET
  public String hello() {
     return "Hello World!";
  }
}
```

HTTP Status Code	Description
200	"OK." The request is successful.
400	"Bad Request." The request is malformed or it is pointing to the wrong endpoint.
403	"Forbidden." That client did not provide the correct credentials.
404	"Not Found." The path or endpoint is not found or the resource does not exist.
405	"Method Not Allowed." The client attempted to use an HTTP method on an endpoint that does not support it.
409	"Conflict." The requested object cannot be created because it already exists.
500	"Internal Server Error." The server failed to process the request. Contact the owner of the REST service to investigate the reason.



Annotation	Description
`@ApplicationPath`	The **`@ApplicationPath`** annotation is applied to the subclass of the **`javax.ws.rs.core.Application`** class and defines the base URI for the web service.
`@Path`	The **`@Path`** annotation defines the base URI for either the entire root class or for an individual method. The path can contain either an explicit static path, such as **`hello`**, or it can contain a variable to be passed in on the request. This value is referenced using the **`@PathParam`** annotation.
`@Consumes`	The **`@Consumes`** annotation defines the type of the request's content that is accepted by the service class or method. If an incompatible type is sent to the service, the server returns HTTP error 415, "Unsupported Media Type." Acceptable parameters include **`application/json`**, **`application/xml`**, **`text/html`**, or any other MIME type.
`@Produces`	The **`@Produces`** annotation defines the type of the response's content that is returned by the service class or method. Acceptable parameters include **`application/json`**, **`application/xml`**, **`text/html`**, or any other MIME type.
`@GET`	The **`@GET`** annotation is applied to a method to create an endpoint for the HTTP GET request type, commonly used to retrieve data.
`@POST`	The **`@POST`** annotation is applied to a method to create an endpoint for the HTTP POST request type, commonly used to save or create data.
`@DELETE`	The **`@DELETE`** annotation is applied to a method to create an endpoint for the HTTP DELETE request type, commonly used to delete data.
`@PUT`	The **`@PUT`** annotation is applied to a method to create an endpoint for the HTTP PUT request type, commonly used to update existing data.
`@PathParam`	The **`@PathParam`** annotation is used to retrieve a parameter passed in through the URI, such as **`http://localhost:8080/hello-web/api/hello/1`**.

Chapter 9:

- Defining which users have access to an application is known as **authentication**, while defining the permissions within the application for those users is known as **authorization**.
- Java Authentication and Authorization Service (JAAS) is a security API that is used to implement user authentication and authorization in Java applications.
- Declarative security separates security concerns from application code by using the container to manage security. The container provides an authorization system based on annotations and XML descriptors within the application code that secures resources.
- **Declarative Security:**
  - Deployment descriptors are needed for managing authentication and authorization.
  - Developers use the **web.xml or jboss-web.xml** file to define which resources in an application should be secured how they are secured, and what data is used to validate the credentials.

```
<security-constraint>
  <web-resource-collection>
       <web-resource-name>All resources</web-resource-name>
       <url-pattern>/*</url-pattern>❶
   </web-resource-collection>
   <auth-constraint>
          <role-name>*</role-name>❷
   </auth-constraint>
 </security-constraint>
 <login-config>
    <auth-method>BASIC</auth-method>❸
    <realm-name>basicRealm</realm-name>❹
 </login-config>
```

1) The resources that the security constraint applies to. The "/*" indicates that all resources are secured.

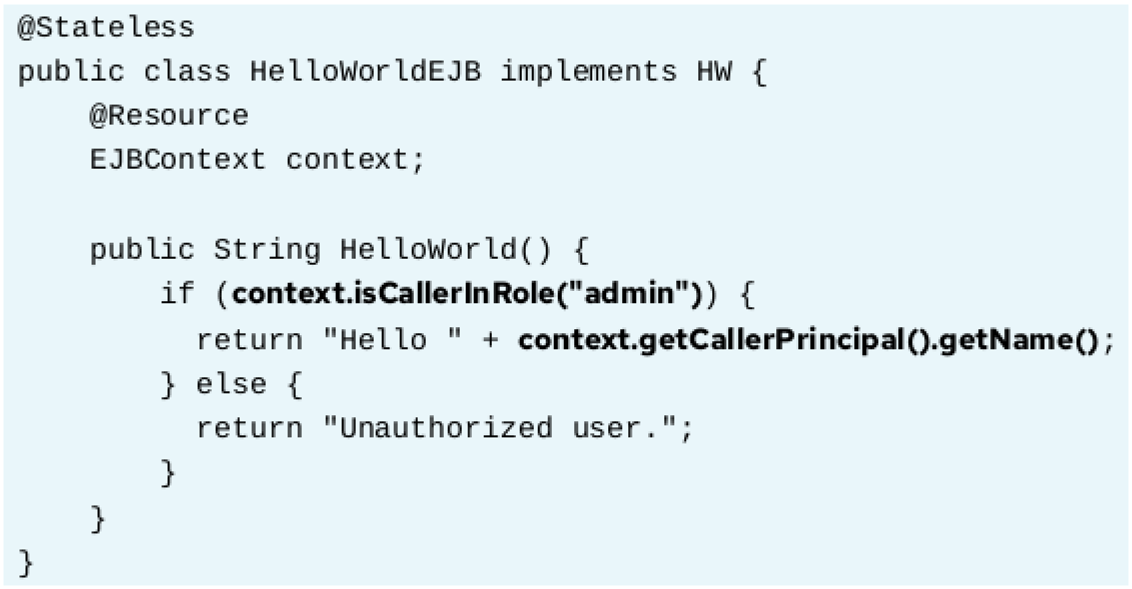
2) The roles that are authorized to access the resources. In this instance, all roles may access the application.

3) The method the application uses to access the credentials of a user. BASIC prompts the user in a pop-up window as soon as the application is accessed.

4) The name of the realm that stores the user credential information. This topic is discussed further in the next section.

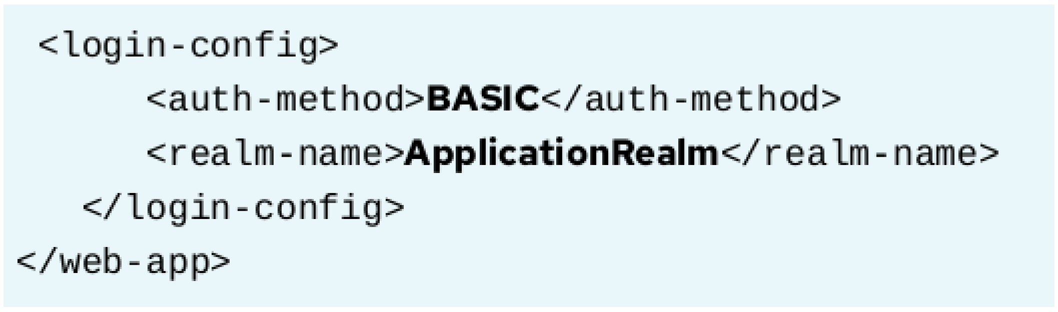
- **@SecurityDomain:** Located at the <u>beginning</u> of the class, this annotation defines the security domain by name to use for the EJB. [Jboss-web.xml file]
- **@DeclareRoles:** Located at the <u>beginning</u> of the class, this annotation defines the roles that are tested for **permissions** in the class.
- **@RolesAllowed:** Located either at the <u>beginning</u> of the class or <u>before</u> a method header, this annotation defines a list of one or more roles allowed to **access** a method. If placed before the class header, methods in the class without an annotation default to this annotation.
- **@PermitAll:** Located either at the <u>beginning</u> of the class or <u>before</u> a method header, this annotation specifies that all roles are allowed to access a method.
- **@DenyAll:** Located either at the <u>beginning</u> of the class or <u>before</u> a method header, this annotation specifies that no roles are allowed to access a method.
- **@RunAs:** Located either at the <u>beginning</u> of the class or <u>before</u> a method header, this annotation specifies the role used when running a method. This annotation is useful when an EJB is calling another EJB and needs to take on a new role for security restrictions in another EJB.

- **Programmatic Security:**
  - Developers may prefer also using programmatic security to have more control over **authentication and authorization decisions within the application.**
  - **EJBContext** object contains information about the current security context.
  - **isCallerInRole(String role)**: Returns a boolean indicating whether a user belong to a given role.
  - **getCallerPrincipal()**: Returns the currently authenticated user.



```
@Stateless
public class HelloWorldEJB implements HW {
    @Resource
    EJBContext context;

    public String HelloWorld() {
        if (context.isCallerInRole("admin")) {
          return "Hello " + context.getCallerPrincipal().getName();
        } else {
          return "Unauthorized user.";
        }
    }
}
```

- EAP and other application servers provide utilities and predefined default configurations that help manage authentication and authorization. EAP manages the user security information in security realms. By default, EAP defines the ApplicationRealm, which uses the following files to store the users and their roles, respectively:
    - **application-users.properties:** This file stores usernames and passwords as a key-value pair, for example: <username>=<password>
    - **application-roles.properties:** This file stores users and roles as key-value pairs using the following syntax: <role>=<user1>,<user2>
- EAP includes **several built-in login modules** that developers can use for **authentication** within a security domain. These login modules include the ability to **read user information** from a **relational database, an LDAP server, or flat files.** It is also possible to build a **custom module** depending on the security requirements of the application.



```
 <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>ApplicationRealm</realm-name>
   </login-config>
</web-app>
```

- In jboss-web.xml file we use the new security domain named userroles:
    - <security-domain>userroles</security-domain>
- In web.xml file we put the security constraints, auth constraints and login config.

- **Security with RESTEasy:**
  - To enable role-based security for REST APIs, update the web.xml to contain the following **<context-param>**

```
<context-param>
  <param-name>resteasy.role.based.security</param-name>
  <param-value>true</param-value>
</context-param>
```

  - After setting the **resteasy.role.based.security**, update the **web.xml** file with a security **constraint set to the path** of the RESTEasy service.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>RESTEasy</web-resource-name>
    <url-pattern>/todo/api/*</url-pattern> ❶
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>❷
  </auth-constraint>
</security-constraint>
```

  - Each role listed in the security constraint also needs to be defined in the web.xml as a **<security-role>**

```
<security-role>
  <role-name>admin</role-name>
</security-role>
```

  - Complete the web.xml file by defining the login-config.

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>ApplicationRealm</realm-name>
</login-config>
```

  - RESTEasy Annotations:
    - @RolesAllowed: Defines the role or roles that can access the method.
    - @PermitAll: All roles defined in the web.xml can access the method.
    - @DenyAll: Denies access to all roles to the method.