The most basic persistence.xml configuration

You can use JPA with a very short, basic configuration. You only need a persistence element as the root element and a *persistence-unit* element with a name attribute. The attribute is used to identify the persistence unit, and you can use it during the bootstrapping process to instantiate a specific *EntityManagerFactory*.

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.2"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
        <!-- Define persistence unit -->
        <persistence-unit name="my-persistence-unit">
        </persistence-unit>
</persistence>
```

When you use this configuration, you configure a persistence unit with the name "*my-persistence-unit*" without defining a dependency to a specific JPA implementation. You also rely on a list of defaults defined by the specification:

- Your persistence provider scans the root of your persistence unit and adds all annotated managed persistence classes to the persistence unit.
- If your *META-INF* directory contains a file called orm.xml, it gets treated as a mapping file and all included mapping information get used.
- The used transaction type depends on the environment in which you deploy your application. In a Jakarta EE environment, JPA expects that the container provides a JTA-compliant

connection provider. In a Java SE environment, it uses a *RESOURCE_LOCAL* transaction instead.

- You don't configure any database connection. JPA, therefore, expects that you provide a datasource at runtime.
- All JPA implementations support a set of proprietary configuration parameters. Examples for that are the logging configuration in EclipseLink JPA or Hibernate's database dialect. As you don't define any of them in this basic configuration, you also rely on all provider-specific defaults.

Optional configuration elements you should know

You can use the following XML elements to customize the basic configuration:

<description>This is a short text describing my persistence unit.</description>	Provide a short *description* of your persistence unit.
<class>org.thoughts.on.java.jpa.beginners.Professor</class>	By adding one or more *class* elements to your persistence-unit configuration, you can add classes to your persistence unit which are not in the root of the persistence unit.
<jar-file>my-entities.jar</jar-file>	You can use one or more *jar-file* elements to add all managed classes contained in these jar files.
<exclude-unlisted-classes>true</exclude-unlisted-classes>	The *exclude-unlisted-classes* element enables you to exclude all classes from the persistence-unit which were not explicitly included.
<mapping-file>file:\\\C:\dev\wrk\XmlMapping\XmlMappings\myMappings.xml</mapping-file>	You can use external, XML-based mapping files. By default, your persistence provider checks if the META-INF directory contains a file called orm.xml and includes its mapping information. If you want to use multiple

	mapping files or if the name of your file doesn't match the default naming pattern, you can reference them in one or more *mapping-file* elements.
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>	If you use any proprietary features of your persistence provider, you should specify a dependency to it.
<jta-data-source>java:app/jdbc/MyDataSource</jta-data-source> <non-jta-data-source>java:app/jdbc/MyDataSource</non-jta-data-source>	These elements are mostly used in Jakarta EE environments. They enable you to reference the JNDI name of a datasource that is or is not compliant with the Java Transaction API.
<shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>	You can activate the 2nd level cache and specify its mode with the *shared-cache-mode element*. You can choose between 4 different options: 1. ALL - To cache all entities 2. NONE - To cache none of your entities (default) 3. ENABLE_SELECTIVE - To cache only the entities annotated with @Cacheable or @Cacheable(true) 4. DISABLE_SELECTIVE - To cache all entities not annotated with @Cacheable(false)
<validation-mode>CALLBACK</validation-mode>	The 3 different values supported by the *validation-mode* element enable you to activate or deactivate the validation: 1. AUTO - Perform the validation if a bean validation implementation is available (default) 2. CALLBACK- Activate the validation and throw an

	exception if no bean validation implementation is available 3. NONE - Do not perform any validation

<properties>
    <property name = "javax.persistence.lock.timeout" value="100"/>
    <property name = "javax.persistence.query.timeout" value="100"/>
</properties>
	You can use the properties javax.persistence.lock.timeout and javax.persistence.query.timeout to define the pessimistic lock timeout and the query timeout in milliseconds.

<properties>
    <property name = "javax.persistence.validation.group.pre-persist" value = "javax.validation.groups.MyPersistValidation"/>
    <property name = "javax.persistence.validation.group.pre-update" value = "javax.validation.groups.MyUpdateValidation"/>
    <property name = "javax.persistence.validation.group.pre-remove" value = "javax.validation.groups.MyRemovetValidation"/>
</properties>
	You can configure one or more groups that get validated for each lifecycle state change by using these property.

<properties>
    <property name = "javax.persistence.jdbc.driver" value = "org.postgresql.Driver" />
    <property name = "javax.persistence.jdbc.url" value = "jdbc:postgresql://localhost:5432/jpaForBeginners" />
	In a Java SE environment, you might not have a datasource that you can reference to define the database connection. In these situations, you can use this set of properties to specify the JDBC driver class, the connection URL and the login information that your persistence provider shall use to connect to the database.

`<property name = "javax.persistence.jdbc.user" value="postgres" />` `<property name = "javax.persistence.jdbc.password" value = "postgres" />` `</properties>`	
`<properties>` `<property name = "javax.persistence.schema-generation.database.action" value = "drop-and-create" />` `<property name = "javax.persistence.schema-generation.create-script-source" value = "create-db.sql" />` `<property name = "javax.persistence.schema-generation.drop-script-source" value = "drop-db.sql" />` `<property name = "javax.persistence.sql-load-script-source" value = "data.sql" />` `</properties>`	Since version 2.1, JPA can create a new database at startup and initialize it with a predefined dataset. You can activate and configure this feature by adding these properties to your configuration.
`<properties>` `<property name = "javax.persistence.schema-generation.scripts.action" value = "drop-and-create"/>` `<property name = "javax.persistence.schema-generation.scripts.create-target" value = "./create.sql"/>` `<property name = "javax.persistence.schema-generation.scripts.drop-target" value = "./drop.sql"/>` `</properties>`	You can tell your persistence provider to generate your database scripts by configuring these properties. Please be aware that these scripts often need to be adapted and optimized before you can use them in production.