6. The EntityManager API

Contents

[1. Introducing the EntityManager API 1](#_Toc459)

[2. Demo: Application-managed EntityManager 3](#_Toc28947)

[3. Demo: Container-managed EntityManager 9](#_Toc19040)

[4. Finding Entities Using the EntityManager 12](#_Toc27513)

[5. Conclusions 13](#_Toc23011)

# 1. Introducing the EntityManager API

=>slides: Pg. 1

Hello. Welcome to this module about The EntityManager API. We'll demonstrate how to use EntityManagerFactory, EntityManager, and the persistence context. We'll analyze the entity's lifecycle and we'll examine how to work with application‑managed and container‑managed EntityManager objects. So, let's dive into these topics.

=>slides: Pg. 2

This module will examine the main interfaces and classes needed to make the persistence, the main methods of the EntityManager interface, the states of the entity instance and the way transitions between them are made, the commit and flush operations, and how to manage an EntityManager through the application and through the container.

=>slides: Pg. 3

So, we have already used the EntityManager in our work, but what exactly is it? EntityManager is an interface defining methods to interact with the persistence context. The persistence context is a set of entity instances. For any entity identity, there is a unique entity instance. The entity instances and their lifecycles are managed inside the persistence context. A persistence unit is defined as the set of entities managed by a given EntityManager instance.

=>slides: Pg. 4

The persistence class from the javax.persistence package provides static utility methods to create EntityManagerFactory instances in a vendor neutral way. The EntityManagerFactory class from the javax.persistence package is the factory for EntityManagers. The EntityManager class is the primary JPA interface used by an application. Each EntityManager manages a set of persistent objects and has APIs to insert, update, or delete objects. When used outside containers, there is a 1:1 relationship between an EntityManager and an EntityTransaction. Entities are persistent objects that correspond to database records. The query interface will find persistent objects meeting certain criteria. A query instance is obtained from an EntityManager that is also a factory for query instances. An EntityTransaction will execute operations on persistent data as indivisible units that either completely succeed or completely fail. The all‑or‑nothing execution is one of the particularities of transactions. If they completely fail, the database remains in its previous state. Transactional executions are important for data integrity. There is a 1:1 relationship between an EntityManager and an EntityTransaction.

=>slides: Pg. 5

An entity instance can be in one of the following states: new, managed, removed, or detached. When a new entity instance is created, it has no persistent identity and is not associated with any persistence context. A managed entity instance is an instance with the persistent identity associated with the persistence context. A removed entity instance is an instance with a persistent identity associated with the persistence context to be removed from the database when the transaction is committed. A detached entity instance is an instance with the persistent identity but not associated with any persistence context.

=>slides: Pg. 6

The EntityManager interface defines the following methods managing entities. Persist makes an entity instance managed and persistent. Remove removes an entity instance. Merge. Merge is the state of an entity into the current persistence context. Find finds an entity using its primary key. Lock locks an entity instance from the persistence context using a specified lock mode type. Detach removes an entity from the persistence context. The managed entity becomes detached. Possible unflushed changes made to the entity or its removal are not sent to the database. Refresh. Refresh is the state of the instance, reloading it from the database. Any changes made to it are overridden.

=>slides: Pg. 7

Let's analyze the effects of applying different methods on an entity depending on the entity current state. Applying the persist method on a new entity, this one becomes managed. The entity is saved into the database when the transaction is committed or as a result of the flush operation. Applying the persist method on a managed entity, it is ignored by the persist operation. However, the persist operation is cascaded to entities referenced by the managed entity. Applying persist on a detached entity, an exception may be thrown when the persist operation is invoked, or an exception may be thrown when commit or flush is executed. Applying persist on a removed entity, this one becomes managed. Executing the merge method, we propagate the state from detached entities into persistent entities managed by the EntityManager.

=>slides: Pg. 8

Applying the merge method on a new entity, a new managed copy of the existing entity is created and the state of the first entity is copied into the one of the new instance. Applying the merge method to a managed entity, it is ignored by the merge operation. However, the merge operation is cascaded to entities referenced by the managed entity. Applying merge on a detached entity, the existing entity is loaded and the state of the merge entity is copied into it. Applying merge on a removed entity instance, IllegalArgumentException will be thrown.

=>slides: Pg. 9

We analyze the effects of applying the remove method on an entity depending on the entity current state. Applying the remove method on a new entity, the operation is ignored. Applying the remove method on a managed entity, the entity is removed. The remove operation is cascaded to entities referenced by the managed entity. Applying the remove method on a detached entity, an IllegalArgumentException will be thrown. Applying the remove method on a removed entity, the operation is ignored. A removed entity will be removed from the database when the transaction is committed or as a result of the flush operation.

=>slides: Pg. 10

We analyzed the effects of applying the refresh method on an entity depending on the entity current state. Applying the refresh method on a managed entity, the state of the entity is refreshed from the database. Possible previous changes to the entity will be overridden. The refresh operation is cascaded to entities referenced by the managed entity. Applying the refresh method on a new, detached, or removed entity an IllegalArgumentException will be thrown.

=>slides: Pg. 11

The state of persistent entities is synchronized to the database in two ways, through committing the transaction or through the flush operation. Calling commit will flush the session and will also end the unit of work. The transaction cannot be rolled back. Calling flush just does a normal synchronization of the session, the changes are not committed.

=>slides: Pg. 12

The EntityManager for a persistence context may be obtained from an EntityManagerFactory. It will be an application‑managed EntityManager, and you will have to programmatically control the whole lifecycle of this object.

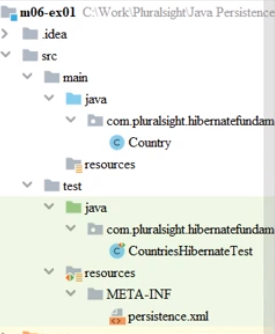
=>slides: Pg. 13

The container‑managed EntityManager is obtained through dependency injection. This means that we'll rely on a container in order to inject the EntityManager that we need for our application. Spring is a framework that has at its heart the dependency injection or the inversion of control. This means that our objects are managed by a container that will be responsible for controlling their lifecycles and managing the dependencies between them. We'll demonstrate the usage of the @PersistenceContext annotation in order to inject a reference to the EntityManager.

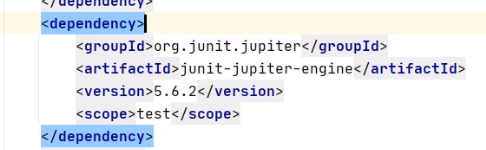
# Demo: Application-managed EntityManager

=>slides: Pg. 14

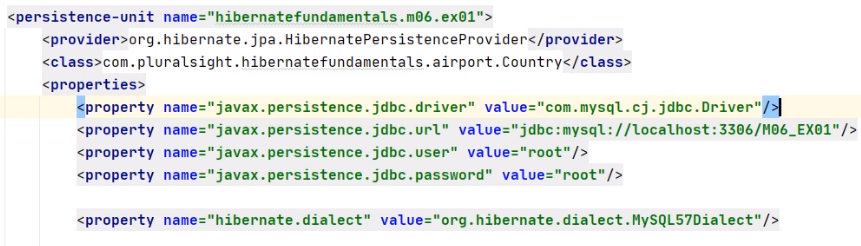
Let's move to the practical demonstration.



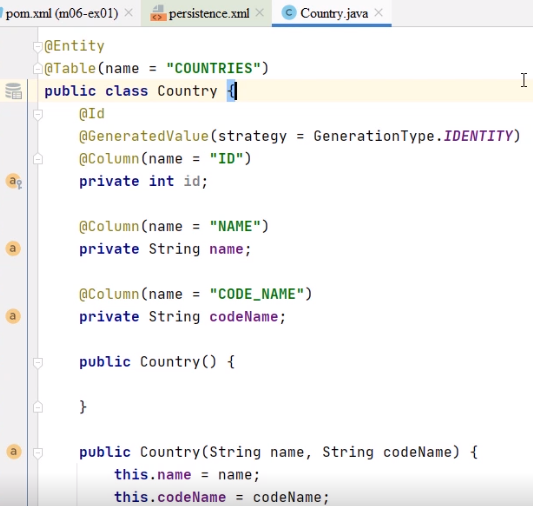
We'll implement an example in which the EntityManager is application‑managed. This will look similarly to our previous examples, but we'll get into some more details. We start with the Maven project, for which the pom.xml is configured similarly to what we presented in the previous demonstrations. We added the hibernate and the postgresSQLdependencies, which were already there in the previously presented projects.



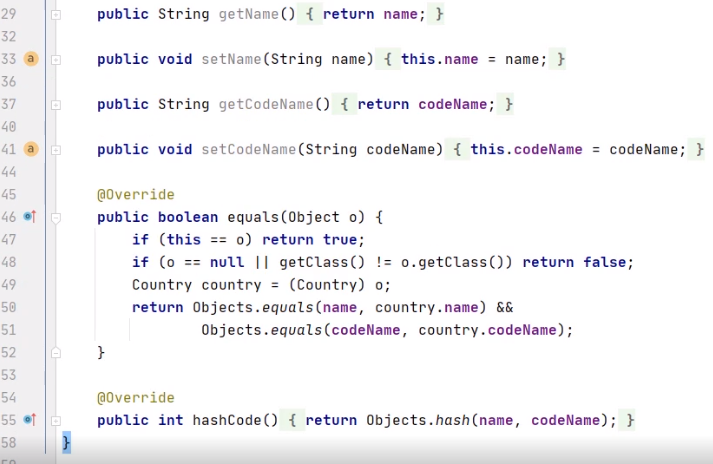
We also used the junit dependency, as we are introducing tests for our code.



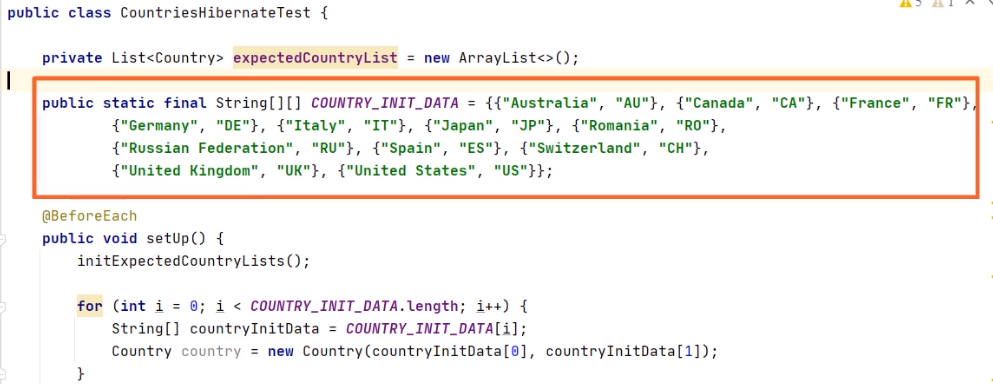
The persistence.xml file is the standard configuration for Hibernate. It is located in the test, resources, META‑INF folder. We have previously encountered it, but now we'll try to explain its full configuration in detail. The persistence‑unit is hibernatefundamentals.m06.ex01. The persistence.xml file must define a persistence unit with a unique name in the currently scoped class loader. We specify the provider, meaning the underlying implementation of the JPA EntityManager. The EntityManager manages a set of persistent objects and has an API to insert new objects and read, update, delete the existing ones. In this case, the EntityManager is hibernate. We define the entity class that is managed by Hibernate as the Country class. We need to do this in addition to the previous demonstrations, as the Country entity needs to be known from within the test and that it is not automatically detected, even if it is annotated with @Entity. The jdbc.driver is postgresSQLbecause this is the database type in use. We specified the URL of the postgresSQLdatabase and the credentials to access the database, a user and a password. We set the SQL dialect for the generated query to MySQL57Dialect and show the generated formatted SQL query on the console. We create the database schema from scratch every time when we execute the test.



We annotated the Country class with @Entity so it can represent objects in a database. The corresponding table in the database is provided by the @Table annotation and is named COUNTRIES. The ID field is marked as the primary key. Its value is automatically generated using the database identity column. The corresponding table column is ID. We also mark the corresponding columns of the name and codeName fields in the class by annotating them with @Column.

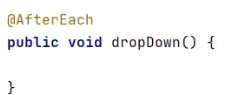


The class contains a default constructor, a constructor with arguments, accessor methods, and the equals and hash code methods. These ones will be needed when verifying the logical equality of the expected objects and of the ones retrieved from the database.



The CountriesHibernateTest class initializes the expectedCountryList to be an empty list and provides the names of the countries in order to build this list of countries.

The initExpectedCountryLists private method will browse the 12 positions of the COUNTRY\_INIT\_DATA array. will create one country at each step, and add it to the expectedCountryList. The setup method executed before each test will initialize the list of the expected countries and will create 12 countries one by one, based on the information from COUNTRY\_INIT\_DATA in order to later persist them in the database. We start the test with a null countryList to be populated later, after retrieving the information from the database. We'll expect a NotNull countryList, and we'll check that each country retreived from the database is equal to the expected one in the expectedCountryList. Remember that we have already overridden the equals and hash code methods in country in order to make a correct logical comparison between two country objects.



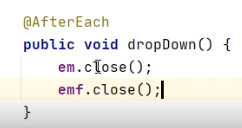
The dropDown method, executed after each test, empty for now, will make sure that we'll close here all objects that we'll use during our tests. We start introducing the new code.



We declare the private EntityManagerFactory and the private EntityManager object.



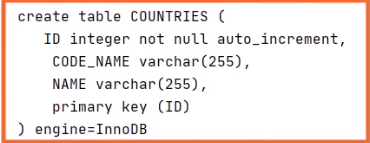
In the setUp method, we initialize the EntityManagerFactory object as we previously used to do in our other demonstrations. We call the Persistence.createEntityManagerFactory starting method. The argument of this method will be the persistence‑unit name as declared in the persistence.xml file. So, we copy it from here and paste it in the code. We create the EntityManager object with the help of the createEntityManager method. We start the transaction by calling em.getTransaction().begin(), and we commit it calling em.getTransaction().commit(). After each country is created, we persist it within the database.



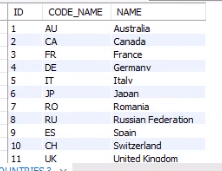
In the dropDown method annotated with @AfterEach, we close the EntityManager and the EntityManagerFactory. We still need, in the test, the list of countries from the database.



We do this by creating and executing a JPQL query, select c from Country c. Java Persistence Query Language, or JPQL, is a platform‑independent, object‑oriented query language that is a part of the JPA specification. We move on the side of the database, and here we have already created the M06\_EX01 database and commute to it with the USE command. And now we may run the Java test. We'll create the needed table, COUNTRIES, we'll insert, then we'll retrieve that information and compare it with what is expected. The test succeeds.



The COUNTRIES table contains the ID, CODE\_NAME, and NAME columns. We also have a series of insert commands for the countries that we persist in the database. We execute SELECT \* FROM COUNTRIES.



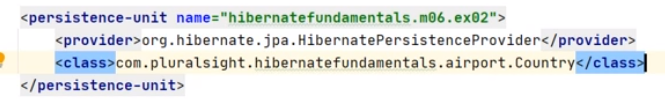
We expect to have here the rows that we have previously introduced. And yes, we have a series of rows. Let's enlarge this window and see that all 12 countries are here. So, this concludes our demonstration. We proved in detail how we can configure and use the EntityManagerFactory and EntityManager object and how we can persist and retrieve a list of countries to and from the database.

# Demo: Container-managed EntityManager

=>slides: Pg. 15

Let's move to the next practical demonstration.

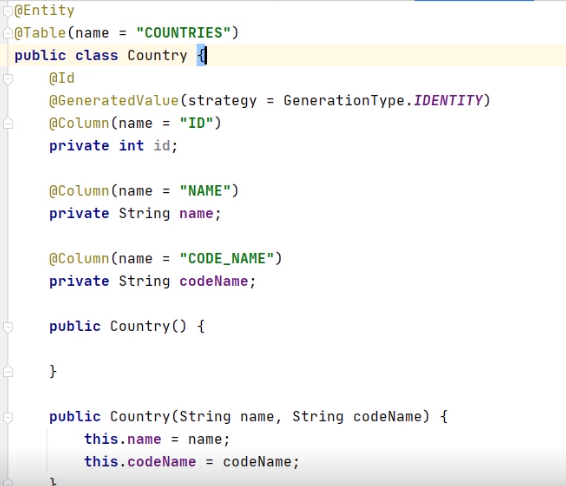


We'll implement an example in which the EntityManager is container‑managed. We'll examine working with Hibernate and Spring, and using the persistence context annotation. We start with a Maven project for which the pom.xml is configured similarly to what we presented in the previous demonstration. We added the Spring dependencies, as we are going to use Spring in order to inject the object that we need. 

The persistence.xml file is the standard configuration for Hibernate. It contains now much less information than in the previous demonstration. It declares Hibernate as Persistence provider and the entity class that is managed by Hibernate. The rest of the configuration has been moved to applicationContext.xml.



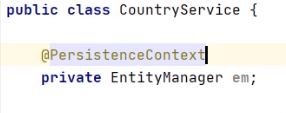
ApplicationContext.xml provides the configuration information to the Spring application. Tx:annotation‑driven tells us that the transactions will be now annotation driven, we will not have to explicitly start them using getTransaction.begin, and we will not have to commit them explicitly using getTransaction.commit. We create a bean data source. This is an object under the management of Spring, and we configure it. The JDBC driver is MySQL, because this is a database type in use. We specify the URL of the postgresSQLdatabase and the credentials to access the database, a user and a password. We declare another bean under the management of Spring, EntityManagerFactory, and we configure it. We specify the persistenceUnitName. We reference the previously declared dataSource bean. We set the SQL dialect for the generated query to MySQL57Dialect, and show the generated formatted SQL query on the console. We create the database schema from scratch every time when we execute the test. We also added two more beans under the management of Spring. TxManager is the bean used for the execution of transactions, and we add one more bean belonging to the CountryService class that we will introduce.



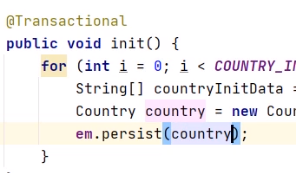
The Country class has exactly the same structure as in the previous demonstration. The same annotations, the same fields, the same constructors, the same accessor methods, the overridden equals and hashCode methods.



The newly introduced Country service class has taken a part of the logic from within the test. We declare the 12 positions COUNTRY\_INIT\_DATA array. The method init will create 12 countries one‑by‑one based on the information from COUNTRY\_INIT\_DATA in order to later persist them in the database.



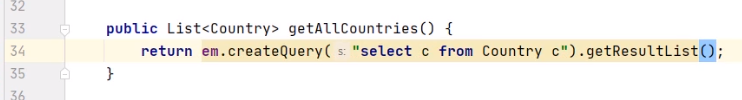
We declare and inject here the EntityManager object, and we annotate it with Persistence context. The EntityManager itself is created by the container using the information in the persistence.xml. Remember, there is an EntityManagerFactory bean we declared in the applicationContext.xml. In order to use the EntityManager, we have to inject it through the PersistenceContext annotation.



We'll no longer begin and commit the transaction explicitly, but the transactional annotation will make sure this will happen when we persist the country.



We also have a clear method annotation with transactional. Here, we'll remove the countries from the database using the delete from Country c JPQL command.

 The getAllCountries method will retrieve the list of countries from the database using the select c from Country c JPQL command.



The CountriesHibernateTest class has a similar structure as in the previous demonstration. We'll emphasize now the differences after moving to the Spring approach.



We'll extend this test using the @ExtendWith(SpringExtension.class) annotation. This will allow us to use the Spring framework features in our tests, for example, the Autowired annotation. We'll use the ContextConfiguration of classpath:application‑context.xml annotation to indicate to Spring where to look for the application ContextConfiguration, in the previously created xml file, in this case. We'll use the Autowired annotation to inject the CountryService bean. Remember, we declared a bean belonging to this class in the applicationContext.xml configuration file. The setUp method, annotated with BeforeEach, will be executed before each test. It will initialize the countryService, meaning that it will create and persist the countries in the database. The test itself will query the database to retrieve the list of countries. It will check the retrieved list is not null and that each country in list is the expected one. At the end of the test, the dropDown method, annotated with AfterEach, is executed. It will delete all records from the countries table. We move on the side of the database, and here we have already created the M06\_EX02 database and commuted to it with the USE command, and now we may run the Java test. We'll create the needed table, Countries, we'll insert the information from our objects, then we'll retrieve that information and compare it with what is expected. At the end, we'll delete the rows from the table. The test succeeds. The Countries table contains the ID, code name, and name columns. We also have a series of insert commands for the countries that we persist in the database, and the delete command. We execute SELECT \* FROM COUNTRIES. Unlike our previous demonstrations, we expect to have here no rows.



Remember that we deleted the countries at the end of the test. And yes, the table exists, it has the expected structure, but there are no rows. Our test created the countries, retrieved them from the database, compared them to the expected ones, then deleted them from the table. So, this concludes our demonstration. We proved in detail how we can configure and use an EntityManager with the help of the Spring dependency injection.

# Finding Entities Using the EntityManager

=>slides: Pg. 16

**m03-ex04**

em.getTransaction().commit();

// Using EntityManager.find()

Ticket foundTicket = em.find(Ticket.**class**, 1);

System.***out***.println("The ticket with Id 1 is found by find() is : " + foundTicket);

// Using JPQL to find entity

foundTicket = em.createQuery("Select t From Ticket t where t.id=1",

Ticket.**class**).getSingleResult();

System.***out***.println("The ticket with Id 1 is found with JPQL is : " + foundTicket);

emf.close();

SELECT c FROM Country c

An equivalent query can be built using the JPA criteria API as follows:

[CriteriaBuilder](https://www.objectdb.com/api/java/jpa/criteria/CriteriaBuilder) cb = em.[getCriteriaBuilder](https://www.objectdb.com/api/java/jpa/EntityManager/getCriteriaBuilder)();

[CriteriaQuery](https://www.objectdb.com/api/java/jpa/criteria/CriteriaQuery)<Country> q = cb.[createQuery](https://www.objectdb.com/api/java/jpa/criteria/CriteriaBuilder/createQuery_Class_)(Country.class);

[Root](https://www.objectdb.com/api/java/jpa/criteria/Root)<Country> c = q.[from](https://www.objectdb.com/api/java/jpa/criteria/AbstractQuery/from_Class_)(Country.class);

q.[select](https://www.objectdb.com/api/java/jpa/criteria/CriteriaQuery/select_Selection_)(c);

[TypedQuery](https://www.objectdb.com/api/java/jpa/TypedQuery)<Country> query = em.[createQuery](https://www.objectdb.com/api/java/jpa/EntityManager/createQuery_CriteriaQuery_)(q);

List<Country> results = query.[getResultList](https://www.objectdb.com/api/java/jpa/Query/getResultList)();

# Conclusions

=>slides: Pg. 17

At the end of this module, let's have a look at what we found out here. We distinguished the main interfaces and classes needed to make the persistence; we examined the methods of the EntityManager interface, persist, remove, merge, find, lock, detach, refresh; we analyzed the states of the entity instance, new, managed, detached, removed, and the way transitions between them are made; we contrasted the commit and flush operations; and we made demonstrations to differentiate between the way we work with an application‑managed EntityManager and a container‑managed EntityManager using Spring as dependency injection framework.

=>slides: Pg. 18

For the end of our journey, let's try some conclusions and have a final look at possible other related courses. This course covered the most important concepts of Java persistence with Hibernate and provided comprehensive demonstrations. First, we introduced ORM and its problems; we analyzed the object‑relational impedance mismatch; we introduced ORM and JPA and demonstrated their benefits and drawbacks; we demonstrated working with entities and mapping the tables to POJOs; we analyzed the entity's access types, mapped database objects, and defined the entity's primary keys and their identities; we demonstrated working with entity relationships; we analyzed the relationships types, on‑to‑one, one‑to‑many, many‑to‑one, many‑to‑many; we compared the unidirectional and bidirectional relationships and demonstrated the creation of relationships; and we demonstrated working with entity inheritance; we examined entity inheritance, hierarchy definitions, and mapping strategies; and we demonstrated the mapping of entity hierarchies using different mapping strategies. We ended up by examining the EntityManager API and its capabilities. We analyzed the entities lifecycle and worked with application‑managed and container‑managed EntityManager objects.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*