Integrating with JPA

Added in 5.3

Tapestry provides a built-in integration with the Java Persistence API (JPA) through the **Tapestry-jpa** module. This module supersedes the 3rd-party Tynamo JPA module.

Contents

Downloading

The **Tapestry-jpa** module is not automatically included in Tapestry applications because of the additional dependencies it requires. If you're using Maven, just add the `tapestry-jpa` dependency to your application's pom.xml file, something like this:

pom.xml (partial)

```
<dependency>
    <groupId>org.apache.tapestry</groupId>
    <artifactId>tapestry-jpa</artifactId>
    <version>${tapestry-version}</version>
</dependency>
```

If you aren't using Maven (or Gradle, Ivy, etc), you'll have to download the jar and its dependencies yourself.

Selecting a JPA Implementation

The Tapestry-jpa module includes a dependency on a JPA specification (API) from Geronimo but not an implementation. You'll have to chose a JPA implementation, such as EclipseLink or Hibernate. The Tapestry-jpa module assumes you'll use Eclipselink. You just have to add the EclipseLink dependency:

pom.xml (partial) for EclipseLink

```
<dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>eclipselink</artifactId>
    <version>${eclipselink-version}</version>
</dependency>
```

Or, if you'd rather use Hibernate as your JPA implementation, you'll want to exclude either the Gernonimo or Hibernate JPA specification JAR:

pom.xml (partial) for Hibernate

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>${hibernate-version}</version>
    <exclusions>
        <exclusion>
            <!-- omit Geronimo JPA spec to avoid conflict with Hibernate JPA spec -->
            <groupId>org.apache.geronimo.specs</groupId>
            <artifactId>geronimo-jpa_2.0_spec</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Configuring JPA

The **persistence.xml** file is the standard configuration file in JPA used to define the persistence units. Tapestry reads this file to create the EntityManagerFactory. The following example demonstrates a persistence.xml file.

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">
   <persistence-unit name="DemoUnit" transaction-type="RESOURCE_LOCAL">
       <properties>
          <property name="javax.persistence.jdbc.driver" value="org.h2.Driver" />
          <property name="javax.persistence.jdbc.url"    value="jdbc:h2:mem:test" />
          <property name="javax.persistence.jdbc.user"   value="sa" />
          <property name="eclipselink.ddl-generation"    value="create-tables"/>
          <property name="eclipselink.logging.level"     value="fine"/>
      </properties>
   </persistence-unit>

</persistence>
```

By default, the persistence descriptor is named *persistence.xml* and is expected to be located on the classpath in the *META-INF* directory. If you want to place the *persistence.xml* file in another directory or name it differently, you can make a contribution to the *SymbolProvider* service, as shown in the following example. This is a quite useful feature if you want to use a different persistence descriptor for tests.

```
public class AppModule {

    @Contribute(SymbolProvider.class)
    @FactoryDefaults
    public static void provideFactoryDefaults(final MappedConfiguration<String, String> configuration) {
        configuration.add(JpaSymbols.PERSISTENCE_DESCRIPTOR, "/org/example/persistence.xml");
    }

}
```

XML-less JPA configuration

With Tapestry, configuring JPA can be much simpler than described by the JPA specification. Tapestry allows you to configure the EntityManagerFactory programmatically, without writing any XML. For example, imagine that you want to use JDBC connections managed by the container and provided through JNDI. The resulting persistence descriptor might look like this:

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             version="2.0">
   <persistence-unit name="JTAUnit" transaction-type="RESOURCE_LOCAL">
      <non-jta-data-source>jdbc/JPATest</non-jta-data-source>
      <properties>
         <property name="eclipselink.ddl-generation" value="create-tables"/>
         <property name="eclipselink.logging.level" value="fine"/>
      </properties>
   </persistence-unit>
</persistence>
```

Now let's see how to provide the same configuration *without* XML. The following demonstrates an equivalent JPA configuration.

AppModule.java (partial)

```
public class AppModule {

   @Contribute(EntityManagerSource.class)
   public static void configurePersistenceUnitInfos(MappedConfiguration<String,PersistenceUnitConfigurer> cfg) {

      PersistenceUnitConfigurer configurer = new PersistenceUnitConfigurer() {
         public void configure(TapestryPersistenceUnitInfo unitInfo) {
            unitInfo.nonJtaDataSource("jdbc/JPATest")
               .addProperty("eclipselink.ddl-generation", "create-tables")
               .addProperty("eclipselink.logging.level", "fine");
         }
     };
     cfg.add("JTAUnit", configurer);
   }
}
```

In the example above you can see a contribution to the *EntityManagerSource* service. This service is responsible for creating the EntityManagerFactory to be used to create EntityManager. When the service is initialized, it parses the *persistence.xml* file, if available. For any persistence unit defined in the XML descriptor a *TapestryPersistenceUnitInfo* object is created. The *TapestryPersistenceUnitInfo* interface is a mutable extension of the PersistenceUnitInfo interface (defined in the JPA specification) that allows you to configure a persistence unit programmatically.

After parsing the persistence descriptor, the EntityManagerSource service applies its configuration to create further persistence units and/or update the existing ones. The service's configuration is a map in which persistence unit names are associated with *PersistenceUnitConfigurer* instances. A PersistenceUnitConfigurer is used to configure a persistence unit programmatically that has been associated with it. In the example above you can see a contribution providing a PersistenceUnitConfigurer for the unit named *JTAUnit*.

Note that the TapestryPersistenceUnitInfo instance passed to the PersistenceUnitConfigurer is either empty or my contain the persistence unit metadata read from the persistence.xml file. What happens if you contribute a PersistenceUnitConfigurer for a persistence unit that has not been defined in the persistence.xml file? In this case Tapestry assumes that you want to configure the persistence unit programmatically and just creates a fresh *TapestryPersistenceUnitInfo* object and passes it to the *PersistenceUnitConfigurer*.

Automatically adding managed classes

If only a single persistence unit is defined, Tapestry scans the *application-root-package.entities* package. The classes in that package are automatically added as managed classes to the defined persistence unit.

If you have additional packages containing entities, you may contribute them to the *JpaEntityPackageManager* service configuration.

AppModule.java (partial)

```
public class AppModule {

   @Contribute(JpaEntityPackageManager.class)
   public static void providePackages(Configuration<String> configuration) {

      configuration.add("org.example.myapp.domain");
      configuration.add("com.acme.model");
   }
}
```

As you can see, you may add as many packages as you wish.

Configuration Settings

Several aspects of Tapestry-jpa can be customized in your application module (usually AppModule.java), just like other Tapestry configuration symbols.

Symbol	Default	Description
JpaSymbols.PROVIDE_ENTITY_VALUE_ENCODERS	true	Whether entity value encoders will be provided automatically. See Using Select with a List.
JpaSymbols.EARLY_START_UP	true	Whether JPA will be started up at application launch, rather than lazily.
JpaSymbols.ENTITY_SESSION_STATE_PERSISTENCE_STRATEGY_ENABLED	true	Whether the "entity" persistence strategy is used to store JPA entities as `Session State Objects`.
JpaSymbols.PERSISTENCE_DESCRIPTOR	/META-INF/persistence.xml	The location of the persistence configuration file, located on the classpath

Injecting the EntityManager

The created entity managers can be injected into page, component and other services.

Injecting the EntityManager into page and component classes

Depending on whether more than one persistence unit has been defined, the way to inject EntityManager varies slightly. Let's start with a simple scenario, where only a single persistence unit is defined. In this case, an EntityManager can be injected using the @PersistenceContext annotation.

CreateAddress.java

```
public class CreateAddress {

   @PersistenceContext
   private EntityManager entityManager;

   @Property
   private Address address;

   @CommitAfter
   void onSuccess() {
      entityManager.persist(address);
   }
}
```

Alternatively, you can use the @Inject annotation to get the EntityManager injected into a page or component, as shown in the following example.

CreateAddress.java

```
public class CreateAddress {

   @Inject
   private EntityManager entityManager;

   @Property
   private Address address;

   @CommitAfter
   void onSuccess() {
      entityManager.persist(address);
   }
}
```

However, if you have multiple instances of persistence-unit defined in the same application, you need to explicitly tell Tapestry which persistence unit you want to get injected. This is what the @PersistenceContext annotation's *name* attribute is used for? The following example demonstrates how to inject the persistence unit named *DemoUnit*.

CreateAddress.java

```
public class CreateAddress {

   @PersistenceContext(unitName = "DemoUnit")
   private EntityManager entityManager;

   @Property
   private Address address;

   @CommitAfter
   @PersistenceContext(unitName = "DemoUnit")
   void onSuccess() {
      entityManager.persist(address);
   }
}
```

Injecting EntityManager into services

While component injection occurs only on fields, the injection in the IoC layer may be triggered by a field or a constructor. The following example demonstrates field injection, when a single persistence unit is defined in the persistence descriptor.

UserDaoImpl.java

```
public class UserDaoImpl implements UserDao {
   @Inject
   private EntityManager entityManager;

   ...
}
```

The constructor injection is demonstrated in the following example.

UserDaoImpl

```
public class UserDaoImpl implements UserDao {

   private EntityManager entityManager;

   public UserDaoImpl(EntityManager entityManager) {
      this.entityManager = entityManager;
   }

   ...
}
```

If multiple persistence units are defined in the same application, you need to disambiguate the unit to inject. This is done with the @PersistenceContext annotation, as shown in the following example. Because @PersistenceContext must not be placed on constructor parameters, you can't use constructor injection and must switch to field injection.

UserDaoImpl

```
public class UserDaoImpl implements UserDao {
   @Inject
   @PersistenceContext(unitName = "DemoUnit")
   private EntityManager entityManager;

   ...
}
```

Value Encoders

The Tapestry-jpa module automatically provides *value encoders* to make it easy to work with entities (especially lists of entities) in your Tapestry pages and components. This is modeled on the similar functionality from the Tapestry-hibernate-core module. See the Hibernate User Guide for all the details.

Transaction Management

As you may already know from the Hibernate integration library, Tapestry automatically manages transactions for you. The JPA integration library defines the @CommitAfter annotation, which acts as the correspondent annotation from the Hibernate integration library. Let's explore the *UserDao* interface to see the annotation in action.

UserDao.java

```
public interface UserDao {

   @CommitAfter
   @PersistenceContext(unitName = "DemoUnit")
   void add(User user);

   List<User> findAll();

   @CommitAfter
   @PersistenceContext(unitName = "DemoUnit")
   void delete(User... users);
}
```

As you can see, the annotation may be placed on service method in order to mark that method as transactional. Any method marked with the @CommitAfter annotation will have a transaction started before, and committed after, it is called. Runtime exceptions thrown by by a transactional method will abort the transaction. Checked exceptions are ignored and the transaction will be committed anyway.

Note that EntityTransaction interface does not support two phase commits. Committing transactions of multiple EntityManagers in the same request might result in data consistency issues. That's why @CommitAfter annotation must be accompanied by the @PersistenceContext annotation if multiple persistence unit are defined in an application. This way you can only commit the transaction of a single persistence unit. You should be very carefully, if you are committing multiple transactions manually in the same request.

After placing the @CommitAfter annotation on methods, you need to tell Tapestry to advise those methods. This is accomplished by adding the transaction advice, as shown in the following example.

AppModule.java (partial)

```
public class AppModule {

   @Match("*Dao")
   public static void adviseTransactionally(
         JpaTransactionAdvisor advisor,
         MethodAdviceReceiver receiver) {

      advisor.addTransactionCommitAdvice(receiver);
   }
}
```