RFC 221 Transaction Control

Draft

51 Pages

Abstract

10 point Arial Centered.

The OSGi Alliance Enterprise Expert Group has developed adaptations of the JDBC, JTA, and JPA Java EE specifications. These resulted in a disparate set of services, that is they provide factory services (DataSourceFactory and EntityManagerFactoryBuilder) and a JTA Transaction service. However, the native OSGi model is to be able to use configured instance services so they can directly be injected through DS and used together. This RFC proposes a client transaction management specification that fully leverages the OSGi service model and Java 8, whilst also making it easy to use resources as part of the transaction.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the “Distribution”) in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

0.4 Table of Contents

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 1.

`Source code is shown in this typeface.`

0.6 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	Feb 09 2016	Initial version
0.1	Feb 15.2016	Minor Updates after the EG call

1 Introduction

The EEG has developed adaptations of the JDBC and JPA Java specifications. These resulted in a non-managed model, that is they provided factory services (Data Source Factory and Entity Manager Factory Builder) and a JTA Transaction service, but no easy way to combine these services into a transactional programming model.

This is at odds with the typical OSGi model where configured instances of services can directly be injected through DS and used together in a simple way. This RFC proposes a client API for transaction management that leverages the OSGi service model and Java 8. This API will make it simple for client applications to use transactions with JDBC, JPA or other technologies, without the need for adapters or additional boilerplate.

This RFC originates from OSGi enRoute work, and from RFP-170. In the enRout project a number of services were identified, designed and implemented based on their needs for web based applications. These use cases help to define the problem that need to be solved.

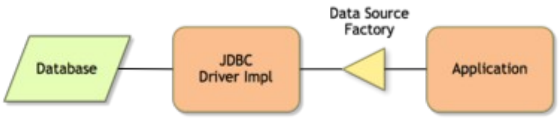
2 Application Domain

Persistence is one of the most important aspects of modern applications. The current mainstream standard for Java is the Java Persistence Architecture (JPA), an Object Relational Mapping (ORM) framework that allows developers to work with objects and builds the corresponding SQL to persist those objects in a relational database.

JPA collaborates with the Java DataBase Connectivity (JDBC) and the Java Transaction Architecture (JTA) specifications. In the first OSGi Enterprise Release these Java EE specifications were adapted to OSGi to make them service based. Together these are called the OSGi *persistence services*. The OSGi persistence services are very much *factory* oriented instead of *instance* based. Instance based services are ready to use, they are usually configured by Configuration Admin, and have their dependencies resolved.

2.1.1 Data Source Factory

The JDBC Data Source Factory specification describes how a *database driver* can register a Data Source Factory service.



The Data Source Factory Service provides a number of methods to create a *Data Source*. A Data Source configures the underlying database and then provides a way to get the database *connections*. Since connections are expensive objects, implementations of a Data Source are often required to *pool* these connections. Pooling reuses connections for other requests after they are closed. Many libraries have been developed to optimize this pooling; these libraries often act as an intermediate between the actual Data Source or Driver and the application. They often use dynamically generated proxies since the JDBC API has gone through several non-backward compatible changes. This is so common that the API supports methods to unwrap these proxies.

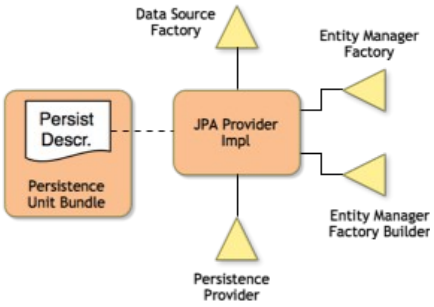
There is also an XA Data Source, which provides access to the XA 'resource' protocol. The XA protocol is used by databases and other transaction aware resources to participate in transactions.

Since the underlying connections can be pooled, it is crucial that operations are properly *scoped*. That is, any obtained connections must be closed to allow them to returned to the pool. Connection Pooling is typically achieved using a library which “wraps” the DataSource. BoneCP, Apache DBCP and C3P0 are examples of libraries which offer this functionality.

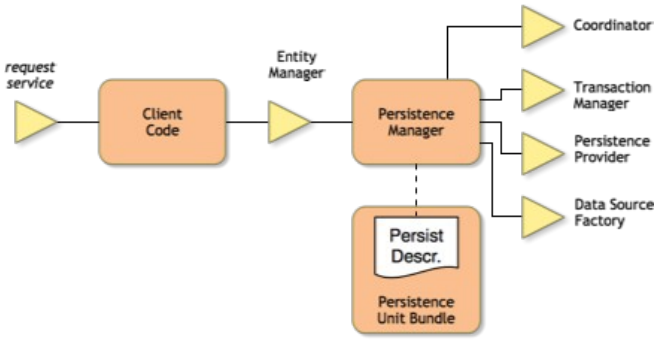
In the OSGi enRoute project the Data Source Factory model was extended with a component with a well known PID and set of configuration properties to automatically register Data Source services.

2.2 JPA

The JPA specification defines how a JPA *provider* can discover a *persistence descriptor* in a bundle. After this discovery, the provider registers an Entity Manager Factory when the persistence descriptor has sufficient information and can be associated with a Data Source Factory. It also registers a Entity Manager Factory Builder that can be used by the application to provide additional properties for configuration and that can create an Entity Manager Factory. Applications that created an Entity Manager from this factory whenever they had to execute a request. Entity Managers are not thread safe.

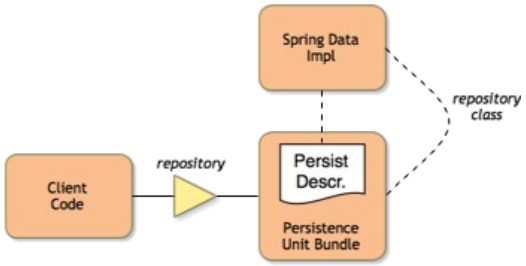


In OSGi enRoute, a component was defined with a well known PID and properties schema that registered an Entity Manager if an appropriate Data Source was available. This required the Entity Manager to proxy an actual Entity Manager since now the life cycle could be managed per thread. All requests would start a Transaction for the request thread. This allowed the Entity Manager proxy to detect the first request and it would then join the Transaction Synchronization Registry. At the end of the transaction, the resources used by the Entity Manager liked pooled connections could then be cleaned up automatically. A similar model is used in Apache Aries



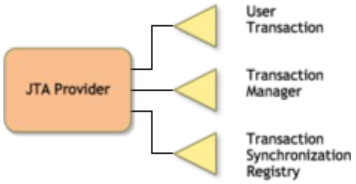
2.3 Spring Data

Christian Baranowski [4]. was inspired by OSGi enRoute and Spring Data. Spring Data provides a model where the methods on an interface are used to specify the query. That is, `findBlogByTitle("hello")` would translate to a request to the database to retrieve the blogs that match the title "hello". He provides an implementation that gets the class name for the repository, which must extend a provided base class, and then registers the repository service for that persistent unit.



2.4 Transactions

The OSGi JTA specification only provides access to the different Transaction objects via services. It does not provide any other features than that are available from the existing JTA specification.



2.4.1 Client-side Transaction management

A Transaction is started by a client by beginning a transaction. It then executes the request code inside a block. If the requests is successful, the transaction is committed, otherwise it is rolled back. In general the catch block and finally block are used to ensure proper termination of the transaction. For example:

```
Transaction transaction = tm.getTransaction();

try {
```

```
   … do work

   transaction.commit();

} catch( Throwable t) {

   transaction.rollback();

   throw t;

}
```

In general, applications must minimize any code inside a transaction since transactions lock database tables. In applications build from different parts, this creates the Transaction Composability problem, see [3]. When a method gets called it can be called inside a transaction or outside a transaction. However, it can require that no transaction is active, that a transaction should be active, or that it requires a new transaction. Handling this inline increases the boiler plate code significantly. For this reason, Spring provides transaction annotations.

```
@Transactonal(readOnly = false, propagation = Propagation.REQUIRES_NEW)

public void doWork() {...}
```

These annotations require a proxy that sets up a transaction before the method is executed. This implies that the method cannot call methods on the `this` object since this bypasses the proxy.

In containers like Java EE the transaction manager is often not directly used. The container provides an Entity Manager that is already associated with a transaction manager. Since the container knows when a request is finished, it can perform any required cleanup. This called *managed transactions.*

2.4.2 Infrastructure integration with Transactions

For work to be included in a transaction It is necessary, but not sufficient, for a client to run code within a transaction. The other critical point is that any resource access that occurs within the transaction must be registered with the transaction manager. Typically this work is not performed by the client, but by a middleware component. In a Java EE application server this role is usually performed a JCA Resource Adapter.

The integration point is responsible for registering the connection with the transaction. If a two-phase commit (XA) transaction is needed then the XAResource associated with the XAConnection must be registered with the Transaction. This enables the Transaction manager to log the details of the resource (to enable recovery) and to prepare/commit/rollback any work. If no XA transaction is required then the resource adapter must register a synchronization with the transaction. This synchronization is responsible for rolling back or committing the work performed with that connection.

2.5 Versions and Migration

Code has an actual dependency on the layout of the database but this tight version dependency is rarely managed. New bundles often require a *migration* of the database but do not have standard support. New columns and tables must be added and sometimes data must be converted. This is often a manual error prone operation. An interesting development is Liquibase [6]. and Flyaway [7]. These projects allow refactoring of databases.

Terminology + Abbreviations

3 Problem Description

The Java EE persistence model with its statics and factories is not natural for OSGi and therefore offers a number of challenges to use. Additionally, the primary providers of JPA Hibernate, OpenJPA, and Eclipselink have varying levels of OSGi adoption/toleralnce, which makes it hard to find a complete solution. The Apache Aries project provides most parts but the Gemini project has all but died. However, even with a full set of implementations the model is non-trivial to use since it still requires the client code to handle a lot of configuration details.

Overall, JPA persistence has its problems in Java EE due to portability problems but in OSGi using JPA is really hard and clearly does not provide the plug and play as well as the collaborative component model that OSGi promotes. The current specifications are not very well matched to the OSGi service model that is configured instance based and not factory based (let alone statics).

Additionally, there are a number of promising developments on the horizon.

- Java 8 lambdas will make it easier to use transactions in a way that is as easy as annotations without the corresponding drawbacks
- The Spring Data JPA Repository looks very interesting for simple database models. Though large enterprise applications might have no use for this, it would lower the threshold for OSGi if it was easier to get started with small models. Obviously there should be a migration path to go to the Entity Manager and Data Source.
- Migration of databases and dependency management on the installed db version is becoming increasingly important. Obviously, the OSGi require-capability and extender model make excellent mechanisms to provide this kind of support.

This RFC therefore seeks a comprehensive service based model for transaction management persistence in OSGi leveraging Java 8 features.

4 Requirements

4.1 General

- G0010– The service solutions must be able to work with JDBC 4 and JPA 2.1
- G0020 – Provide a simplified way to handle the transaction composition that does not require boiler plate code nor suffers from the problem that methods on the 'this' pointer are different..

- G0030 – The solution must not require that configured Data Access resources are publicly available. This prevents other bundles having access to database credentials.

4.2 Data Source Service

- D0010 – Provide a configuration model for a Data Source service
- D0014 – Must be able to transparently register connections with an ongoing transaction.
- D0017 – Must be able to support both two-phase (XA) resources, and non XA resources
- D0020 – Must be able to transparently handle connection pooling of “plain” DataSources
- D0030 – Must support all OSGi JDBC Factory defined possibilities and properties
-

4.3 Entity Manager Service

- E0010 – Provide a configuration model for an Entity Manager service
- E0020 – Must be able to use the Data Source service from 4.2
- E0030 – Must be able to have managed transactions. That is, the client uses an injected Entit Manager.
-
- E0050 – Must provide access to the underlying Data Source

4.4 Database Versioning

- V0010 – Provide a require-capability model for handling the version of the database
- V0020 – Provide a model so that bundles can be used to migrate and rollback a database.

5 Technical Solution

OSGi components and services are designed to be modular, lightweight and easily deployable across a variety of environments. Whilst this model has many benefits, it means that transaction management is difficult when compared to centralised container solutions. This solution therefore needs to offer a simple, way to manage transactions in OSGi components.

5.1 Client Transaction Management

In Java EE and Spring application containers transactions are typically managed declaratively via annotations.

```
@Transactonal(readOnly = false, propagation = Propagation.REQUIRES_NEW)

public void doWork() {...}
```

The problem with this model is that it requires a third-party to proxy the doWork() method, adding the necessary transaction management semantics. Furthermore, the doWork() method must be public and if the doWork() method is called from within the object, or called without using the proxy (for example in Test code), then no transaction management will occur.

Due to the limitations of proxy-based approaches, the undesirabilty of universal proxying in an OSGi container, and the lambda expression support added in Java 8, this RFC proposes a programmatic transaction management model using callback functions. These callback functions are known as “work”.

Another issue with transaction management is that both the Java EE and Spring transaction models have a poorly defined behaviour when no transaction is active. This RFC therefore proposes the following three states:

- Unscoped: if code execution is unscoped then it means that the TransactionControl service has no context associated with it (i.e. it is not being run as a piece of work). No callbacks, data storage, or transaction management are available to unscoped executions
- No Transaction Scope: if a piece of work has “No Transaction” scope then the Transaction Control service has not created a transaction, but has created a coordination to represent the scope of the piece of work. The coordination will complete when the piece of work has completed. Clients may use scoped variables and register for a final callback at the end of the piece of work, but they may not register resources to participate in a transaction.
- Active Transaction Scope: if a transaction is active this means that the TransactionControl service has created a transaction *and* a coordination around a piece of work. Resources used during the transaction will be committed or rolled back based on the outcome of the piece of work.

5.1.1 Beginning a transaction

Client management of transactions occurs through the `TransactionControl` service. This service allows for simple management of transaction lifecycles using four methods:

Method	Behaviour
`required(Callable)`	Begins a new Active Transaction scope around the piece of work if there is no Active Transaction, otherwise the existing transaction scope continues.
`requiresNew(Callable)`	Begins a new Active Transaction scope around the piece of work, Suspending the the existing transaction if it exists. Afterwards the suspended transaction is resumed
`supports(Callable)`	Continues an Active Transaction or No Transaction scope if it exists, otherwise a new No Transaction scope is begun around the piece of work
`notSupported(Callable)`	Continues an existing No Transaction scope if it exists, otherwise a new No Transaction scope is begun around the piece of work, suspending an ongoing Active Transaction if necessary. Afterwards the suspended transaction is resumed

A client can therefore begin a transaction as follows:

```
txControl.required(() -> {
```

```
                // Do some work in here

                return result;

        });
```

5.1.2 Querying the current Transaction Context

The TransactionControl Service provides methods that can be used to query the current transaction context.

Method	Behaviour
`activeTransaction()`	Returns true if there is an Active Transaction Scope
`activeScope()`	Returns true if there is an Active Transaction or a No Transaction scope
`getCurrentContext()`	Gets the current transaction context, or null if none exists

These methods can be used to verify that particular a particular transaction scope is active, or to interact with the transaction lifecycle.

5.1.3 Controlling Transaction Rollback

By default a transaction will commit automatically when the piece of work completes normally. If this is not desired (for example the work's business logic determines that the transaction should not complete) then the work may trigger a rollback in one of two ways:

1. Throw an exception from the piece of work. By default all exceptions will cause the transaction to be rolled back. Note that this is different from Java EE behaviour, where checked exceptions do not trigger rollback. This is a deliberate difference as many applications get the wrong behaviour based on this default. For example SQLException is a commonly thrown Exception in JDBC, but is rarely, if ever, a “normal return”. Forgetting to override this means that production code will fail to enforce the correct transaction boundaries.

2. Call setRollbackOnly() on the Transaction Control object. This will mark the transaction for rollback so that it will never commit, even if the method completes normally. This is a one-way operation, and the rollback state can be queried using getRollbackOnly().

5.1.3.1 Avoiding Rollback

Sometimes it is preferrable for a piece of work to throw an exception, but for that exception not to trigger a rollback of the transaction. For example some business exceptions may be considered “normal”, or it may be the case that the work performed so far must be persisted for audit reasons.

There are two ways to prevent a transaction from rolling back when a particular exception occurs:

1. The TransactionControl service provides a TransactionBuilder. This builder can be used to define sets of Exception types that should, or should not, trigger rollback. The most specific match will be used to determine whether the transaction should roll back or not.

2. The Transaction Control service provides an ignoreException() method. This can be used from within an Active Transaction to declare a specific Exception object that should not trigger rollback.

5.2 Resource Providers

Whilst it is important that clients can easily control the transaction boundaries within their application it is equally important that the resources that the clients use participate in these transactions. In a Java EE Application server this is achieved by having the central application container create and manage all of the resources. In the Spring framework the Application context is responsible for ensuring that the resources are linked to a Transaction Manager.

In existing OSGi solutions this is the weakest part of the transactional persistence story. Typically libraries provide some level of transaction enlistment by shadowing partially configured services, and the way in which the resources are linked to the active transaction is poorly defined.

This RFC defines the concept of a Resource Provider as a generic way to ensure that a particular resource will be enlisted with the correct transaction context.

5.2.1 Generic Resource Providers

The purpose of a ResourceProvider is to provide the client with a configured resource which will automatically integrate with the correct transaction context at runtime.

```
public <T> T getResource(TransactionControl txControl);
```

The returned resource is a proxy to an underlying resource factory. When the resource is accessed then then the proxy will check the current transaction scope. If this is the first time the resource has been accessed in this scope then the factory returns a new copy of the resource. If the scope is an Active Transaction then the resource must also be enlisted into the transaction at this point. Subsequent resource access within the same scope must use the same backing resource.

5.2.1.1 Resource Lifecycle

When a scope finishes any resources associated with the scope must be cleaned up without action required by the client. This rule applies to both the Active Transaction scope and the No Transaction scope, meaning that a client can safely write code using TransactionControl#supports(Callable) without being concerned about resource leaks.

5.2.1.2 Unscoped Resource Access

If a resource is accessed by unscoped code then it may throw a TransactionException to indicate that it cannot be used.

5.2.1.3 Closing Resources

If a client attempts to close a scoped resource then this operation should be silently ignored. The resource will be automatically cleaned up when the current scope completes. If the resource were prematurely closed then it may prevent other services from accessing the resource within this scope. Also in an Active Transaction the resource cannot be committed or rolled back until the commit operation occurs.

5.2.2 JDBC

One of the most common resources to use in a transaction is a JDBC Connection. This RFC defines a specialised resource provider for obtaining JDBC Connections called a JDBCConnectionProvider. The purpose of this type is simply to reifiy the generic type of the ResourceProvider.

5.2.2.1 JDBC Active Transaction behaviours

When enlisted in an Active Transaction a JDBC connection will have autocommit set to false. Also the following methods must not be called by the client and will trigger a TransactionException.

- commit()
- rollback() and rollback(Savepoint)
- setAutoCommit()
- setSavepoint(), setSavepoint(String) and releaseSavepoint()

If the Active Transaction commits the JDBC Connection must commit any work performed in the transaction. Similarly if the Active Transaction rolls back then the JDBC Connection must roll back any work performed in the transaction. After the transaction completes the JDBC connection must be cleaned up in an appropriate way, for example by closing it or returning it to a connection pool.

5.2.2.2 JDBC No Transaction behaviours

When accessed with from the No Transaction scope the JDBC connection may have autocommit set to true or false depending on its underlying configuration. This value may be changed by the client within the scope of the resource access, but will be reset after the end of the scope.

In the No Transaction context the JDBC connection will not be committed or rolled back, it is therefore the client's responisibility to call commit() or rollback() as appropriate. Savepoints may be used for partial rollback if desired.

After the end of the scope the JDBC connection must be automatically cleaned up in an appropriate way, for example by closing it or returning it to a connection pool.

5.2.2.3 JDBCConnectionProviderFactory

The JDBCConnectionProvider may be provided directly in the OSGi service registry, however this may not be acceptable in all use cases. JDBC Connections are often authenticated using a username and password. If the username and password relate to a specific bundle then it may not be appropriate to have the configured connections available in the Service Registry. In this case the JDBCConnectionProviderFactory offers several factory methods that can programatically create a JDBCConnectionProvider.

Provider Configuration

Each factory method here supplies set of properties which are used to configure the ResourceProvder, including the connection pooling behaviour, and whether the ResourceProvider can be enlisted with XA and/or Local transactions.

By default the JDBCConnectionProvider will have a pool of 10 connections with a connection timeout of 30 seconds, an idle timeout of 10 minutes and a maximum connection lifetime of 3 hours. The JDBCConnectionProvider will also, by default, work with Local and XA transactions.

If the JDBCConnectionProvider is configured to enable XA then the DataSourceFactory or DataSource must support the creation of an XADataSource

Creating a JDBCConnectionProvider Using a DataSourceFactory

```
JDBCConnectionProvider getProviderFor(DataSourceFactory dsf, Properties
jdbcProperties, Map<String,Object> resourceProviderProperties);
```

In this case the client provides the DataSourceFactory that should be used, along with the properties that should be used to create the DataSource/XADataSource. If XA transactions are enabled then the factory must create an XADataSource, otherwise the “osgi.use.driver” property can be used to force the creation of a Driver instance rather than a DataSource.

Creating a JDBCConnectionProvider Using a DataSource

```
JDBCConnectionProvider getProviderFor(DataSource ds, Map<String,Object>
resourceProviderProperties);
```

In this case the client provides a preconfigured DataSource that should be used. If XA transactions are enabled then the datasource must either implement XADataSource or be “unwrappable” to XADataSource.

Creating a JDBCConnectionProvider Using an XADataSource

```
JDBCConnectionProvider getProviderFor(XADataSource xaDs,
Map<String,Object> resourceProviderProperties);
```

In this case the client provides a preconfigured XADataSource that should be used.

Creating a JDBCConnectionProvider Using a Driver

```
JDBCConnectionProvider getProviderFor(Driver driver, Properties
jdbcProperties, Map<String,Object> resourceProviderProperties);
```

In this case the client provides the pre-loaded driver class that should be used, along with the properties that should be used to create the JDBC connection. XA transactions may not be enabled when using a Driver instance.

5.2.3 JPA

JPA is a popular Object Relational Mapping (ORM) framework used to absract away the low-level database access from business code. As an alternative means of accessing a database it is just as important for JPA resources to participate in transactions as it is for JDBC resources. This RFC therefore defines the JPAEntityManagerProvider interface as a specialised resource provider for JPA.

5.2.3.1 JPA Active Transaction behaviours

When enlisted in an Active Transaction a JPA EntityManager will automatically track the state of persisted entity types and update the database as necessary. When participating in a transaction it is forbidden to call getTransaction on the EntityManager as manual transaction management is disabled.

If the Active Transaction commits then the JPA EntityManager must commit any work performed in the transaction. Similarly if the Active Transaction rolls back then the JJPA EntityManager must roll back any work performed in the transaction. After the transaction completes the JDBC connection must be cleaned up in an appropriate way, for example by closing it or returning it to a connection pool.

5.2.3.2 JPA No Transaction behaviours

When accessed with from the No Transaction scope the JPA EntityManager will not be committed or rolled back, it is therefore the client's responisibility to set up an EntityTransaction and to call commit() or rollback() as appropriate.

After the end of the scope the EntityManager must be automatically cleaned up in an appropriate way, for example by closing it or returning it to a pool.

5.2.3.3 RESOURCE_LOCAL and JTA EntityManagerFactory instances

When defining a JPA Persistence Unit the author must declare whether the EntityManagerFactory integrates with JTA transactions, or is suitable for resource local usage. The JPAEntityManagerProvider must take this into account when creating the transactional resource.

JTA scoped EntityManager instances may not manage their own transactions and must throw a JPA TransactionRequiredException if the client attempts to use the EntityTransaction interface. In effect the EntityManager behaves as a Synchronized, Transaction-Scoped, Managed Persistence Context as per the JPA 2.1 Specification. It is important to ensure that the Database connections used in a JTA Persistence Unit are integrated with the ongoing transaction.

RESOURCE_LOCAL scoped EntityManager instances may not participate in XA transactions, but otherwise behave in much the same way as JTA EntityManager instances. The one significant difference is that RESOURCE_LOCAL EntityManager instances may obtain an EntityTransaction when running in the No Transaction context.

5.2.3.4 JPAEntityManagerProviderFactory

The JPAEntityManagerProvider may be provided directly in the OSGi service registry, however this may not be acceptable in all use cases. Database Connections are often authenticated using a username and password. If the username and password relate to a specific bundle then it may not be appropriate to have the configured connections available in the Service Registry. In this case the JPAEntityManagerProviderFactory offers several factory methods that can programatically create a JPAEntityManagerProvider.

Creating a JPAEntityManagerProvider Using an EntityManagerFactoryBuilder

```
JPAEntityManagerProvider getProviderFor(EntityManagerFactoryBuilder emfb,
Map<String,Object> jpaProperties, Map<String,Object>
resourceProviderProperties);
```

In this case the client provides the EntityManagerFactoryBuilder that should be used, along with the properties that should be used to create the EntityManagerFactory. The “osgi.jdbc.provider” property can be passed to the resource provider defining the JDBCConnectionProvider that should be converted into a DataSource and passed to the EntityManageFactoryBuilder using the javax.persistence.jtaDataSource property.

Creating a JPAEntityManagerProvider Using an EntityManagerFactory

```
JPAEntityManagerProvider getProviderFor(EntityManagerFactory emf,
Map<String,Object> jpaProperties, Map<String,Object>
resourceProviderProperties);
```

In this case the client provides the configured EntityManagerFactory that should be used, along with the properties that should be used to create the EntityManager.

5.2.4 Connection Pooling

Database connections are usually heavyweight objects that require significant time to create. They may also consume physical resources such as memory or network ports. Creating a new database connection for every request is therefore wasteful, and adds unnecessary load to both the application and the database. Caching of database connections is therefore a useful way of improving performance. On the other hand applications must be careful not to create too many database connections. If one thousand requests arrive simultaneously then creating one thousand database connections is likely to crash the database server. These two requirements make database connections an excellent candidate for pooling. A small number of connections are made available and recycled after use. This saves the cost of recreating the connection and limits the overall load on the database.

In fact pooling is an excellent solution for many transactional resources, including JMS and EIS access.

5.2.4.1 Connection Pooling in OSGi

Pooling has traditionally been difficult in OSGi because most connection pooling libraries use reflective access to load the underlying resource connector. This obviously fails unless the pooling library creates a static wiring to the connector, or has dynamic package imports. Both of these “solutions” are bad practices which create brittle dependencies.

The correct way to obtain Database connections in OSGi is to use a DataSourceFactory, however this offers no Connection Pooling. There is no real equivalent of a DataSourceFactory for JMS ConnectionFactory instances, but they also require manual decoration to enable connection pooling.

As pooling is such a core requirement for transactional resource access it is required for JDBCConnectionProviderFactory instances to offer connection pooling. The resource provider properties can be used to override the connection pooling configuration defaults (or to disable connection pooling entirely).

Property	Default	Purpose
osgi.connection.pooling.enabled	true	Whether connection pooling is enabled for this ResourceProvider
osgi.connection.timeout	30~~,000~~ seconds	The maximum time that a client will wait for a connection
osgi.idle.timeout	~~180,000~~ 3 minutes	The time that a connection will remain idle before being closed
osgi.connection.lifetime	3 hours	The maximum time that a connection will remain open
osgi.connection.min	10	The minimum number of connections that will be kept alive
osgi.connection.max	10	The maximum number of connections that will exist in the pool

5.3 The TransactionContext

When a client uses the TransactionControl service to scope a piece of work, the scope gains an associated Transaction Context. The current transaction context is not normally needed by clients, but is an important integration point for ResourceProviders, and for clients that wish to register transaction completion callbacks.

The current TransactionContext is available using the getCurrentContext() method of the TransactionControl service. The context will be null if the currently executing code is unscoped. If the current work has a No Transaction scope then a TransactionContext will be returned, however it will report its status as NO_TRANSACTION.

5.3.1 Scoped Variables

A Transaction context may be used to store scoped variables. These variables are attached to the TransactionContext, and will be de-referenced once the Context finishes.

Variables may be added to the scope using putScopedValue() and retrieved using getScopedValue(). These methods are valid both for Active Transactions and the No Transaction scope.

5.3.2 The Transaction Key

Every Active Transaction has an associated key, which will be unique within the lifetime of the TransactionControl service's registration i.e. a registered TransactionControl will never reuse a key. The key object is opaque, but is guaranteed to be suitable for use as a key in a HashMap. Note that the Transaction Key is not globally unique, but only unique to the registered TransactionControl service. In particular, two concurrently registered TransactionControl services may simultaneously use the same key, and/or a TransactionControl implementation may reuse keys if it unregisters and then re-registers its service with a different service id.

TransactionContexts for the NoTransaction scope have a null key

5.3.3 Local Transactions

A LocalTransaction is not persistent, and therefore not recoverable. It also may not be atomic or consistent if multiple resources are involved (although it is if only a single resource is used). Local transactions do, however, provide isolation and durability, even when mulitple resources are involved.

A Local Transaction is therefore a very good choice when a single resource is involved as it is extremely lightweight and provides ACID behaviour. Local Transactions do provide benefits when multiple resources are involved, however it is important to realise that Local Transactions may end up in a state where some commits have succeeded and others failed.

5.3.3.1 The Local Transaction Lifecycle

The transaction context for a local transaction begins in the ACTIVE state, and may enter the MARKED_ROLLBACK state if the client calls setRollbackOnly().

A local transaction will return true from the supportsLocal() method, indicating that LocalResource participants may be registered using the registerLocalResource(LocalResource) method.

Once the transactional work has completed the transaction will be proceed as follows:

Work state	Action
ACTIVE TransactionContext, successful return	Set the Transaction Status to COMMITTING Call commit on each LocalResource. • If the first LocalResource fails to commit then set the Transaction status to ROLLING_BACK , rollback all subsequent resources and throw a TransactionRolledBackException with its cause set to the first failure. Subsequent failures must be added as suppressed Exceptions • • If the first LocalResource commits and then one or more LocalResource instances fail to commit then a TransactionException must be thrown with the cause set to the first failure. Subsequent failures must be added as suppressed Exceptions Set the Transaction Status to COMMITTED or ROLLED_BACK as appropriate

Work state	Action
	The return value is returned by the Transaction
ACTIVE TransactionContext, noRollbackFor Exception	As for a successful return, but rethrow the exception afterwards rather than returning a value
ACTIVE TransactionContext, rollbackFor Exception	Set the Transaction Status to ROLLING_BACK Call rollback on each LocalResources gathering any exceptions that occur. Set the Transaction Status to ROLLED_BACK as appropriate Throw a TransactionRolledBackException with the cause set to the original Exception and suppressed exceptions for any rollback failures
MARKED_ROLLBACK TransactionContext, successful return	Set the Transaction Status to ROLLING_BACK Call rollback on each LocalResourcer gathering any exceptions that occur. Set the Transaction Status to ROLLED_BACK as appropriate If no rollback failures occurred then return the successful return value, otherwise throw a TransactionException with the cause set to the first rollback exception and any subsequent failures as suppressed exceptions
MARKED_ROLLBACK TransactionContext, noRollbackFor Exception	As for a successful return, but rethrow the exception afterwards rather than returning a value
MARKED_ROLLBACK TransactionContext, rollbackFor Exception	Same as ACTIVE TransactionContext, rollbackFor Exception

5.3.3.2 Local Transaction Support Service Properties

A TransactionControl Service which supports local transactions may be identified using the “osgi.local.enabled” property which will be set to Boolean.TRUE.

5.3.4 XA Transactions

An XA transaction is persistent, and therefore can be recoverable. It is also may atomic and consistent even if multiple resources are involved.

An XA Transaction is therefore a very good choice when a multiple resource are involved as it provides ACID behaviour. XA transactions are, however, more heavyweight than local transactions, and should only be used where they are needed.

5.3.4.1 The XA Transaction Lifecycle

The transaction context for anX A transaction begins in the ACTIVE state, and may enter the MARKED_ROLLBACK state if the client calls setRollbackOnly().

An XA transaction will return true from the supportsXA() method, indication that XAResource participants may be registered using the registerXAResource(XAResource) method.

Once the transactional work has completed the transaction will be proceed as follows: (TransactionContext Status changes follow the XA 2PC algorithm)

Work state	Action
ACTIVE TransactionContext, successful return	Begin committing the XA Transaction If the commit fails then throw a TransactionRolledBackException with its cause set to the XA failure. Otherwise the return value is returned by the Transaction
ACTIVE TransactionContext, noRollbackFor Exception	As for a successful return, but rethrow the exception afterwards rather than returning a value
ACTIVE TransactionContext, rollbackFor Exception	Begin rolling back the XA Transaction Throw a TransactionRolledBackException with its cause set to the thrown Exception. If the rollback fails then add the XA failure as a suppressed Exception.
MARKED_ROLLBACK TransactionContext, successful return	Begin rolling back the XA Transaction If the rollback fails then throw a TransactionException with its cause set to the XA failure. Otherwise the return value is returned by the Transaction
MARKED_ROLLBACK TransactionContext, noRollbackFor Exception	As for a successful return, but rethrow the exception afterwards rather than returning a value
MARKED_ROLLBACK TransactionContext, rollbackFor Exception	Same as ACTIVE TransactionContext, rollbackFor Exception

:

5.3.4.2 XA support service properties

A TransactionControl Service which supports local transactions may be identified using the “osgi.xa.enabled” property which will be set to Boolean.TRUE.

5.3.5 Mixed mode transactions

Some TransactionControl Providers will support the use of both XAResources and LocalResources witihin the same transaction. In this case they should register service properties for both resource types. The TransactionControl service may place restrictions on mixed resource registration. For example ~~it may not be possible to register both XA resources and LocalResources in the same Transaction, or~~ it may only be possible to register one LocalResource in addition to a number of XAResources (as per the last resource gambit). If the Transaction cannot support the registration of a resource then it must throw a TransactionException when that resource registration is attempted.

If a TransactionControl implementation supports the use of both XAResources and LocalResources, but not within the same transaction, then the implementation must register separate services (with the relevant service properties) for each type of resource.

In the case where both XAResources and LocalResources are in the same transaction the XAResources should be ~~committed~~prepared first, then the LocalResource(s) committed, then the XAResources committed afterwards. If any of the LocalResources experience a failure then ~~it should be treated as if a virtual “first LocalResource” had~~the overall transaction should roll back. If some LocalResources have committed then the Transaction Control service must log which of these are in an inconsistent state. ~~already completed with the same result as the XAResources~~.

5.3.6 Lifecycle callbacks

In addition to registering Resources with the TransactionContext clients or resources may register callback functions. Callback functions may run either before or after the transaction commits, depending as to whether they are registered using preCompletion or postCompletion to register their callbacks.

Precompletion callbacks are run after the transactional work, but immediately before the commit operation begins. PostCompletion callbacks are run after the commit process. In the case of a No Transaction context there is no commit, so the postcompletion callbacks immediately follow the precompletion callbacks, and are passed a status of NO_TRANSACTION.

5.3.6.1 Exceptions from callbacks

Exceptions generated by precompletion callbacks are gathered, If any of the generated Exceptions would trigger rollback then the transaction is treated as having failed with the first of those exceptions. Any other exceptions are added as suppressed exceptions.

Exceptions generated by postcompletion callbacks are unable to affect the outcome of the transaction, and must therefore be logged, but not acted on further by the TransactionControl service.

6 Data Transfer Objects

RFC 185 defines Data Transfer Objects as a generic means for management solutions to interact with runtime entities in an OSGi Framework. DTOs provides a common, easily serializable representation of the technology.

For all new functionality added to the OSGi Framework the question should be asked: would this feature benefit from a DTO? The expectation is that in most cases it would.

The DTOs for the design in this RFC should be described here and if there are no DTOs being defined an explanation should be given explaining why this is not applicable in this case.

This section is optional and could also be provided in a separate RFC.

7 Javadoc

OSGi Javadoc

2/15/16 4:28 PM

Package Summary		Page
org.osgi.service.transaction.control	Transaction Control Service Package Version 1.0.	23
org.osgi.service.transaction.control.jdbc		40
org.osgi.service.transaction.control.jpa		46

Package org.osgi.service.transaction.control

```
@org.osgi.annotation.versioning.Version(value="1.0.0")
```

Transaction Control Service Package Version 1.0.

See:
Description

Interface Summary		Page
LocalResource	Resources that can integrate with local transactions should do so using this interface	24
ResourceProvider	A resource provider is used to provide a transactional resource to the application	25
TransactionContext	A transaction context defines the current transaction, and allows resources to register information and/or synchronisations	28
TransactionControl	The interface used by clients to control the active transaction context	31
TransactionStarter	Implementations of this interface are able to run a piece of work within a transaction	35

Class Summary		Page
TransactionBuilder	A builder for a piece of transactional work	26

Enum Summary		Page
TransactionStatus	The status of the transaction A transaction may not enter all of the states in this enum, however it will always traverse the enum in ascending order.	37

Exception Summary		Page
TransactionException	An Exception indicating that there was a problem with starting, finishing, suspending or resuming a transaction	33
TransactionRolledBackException	An Exception indicating that the active transaction was rolled back	34

Package org.osgi.service.transaction.control Description

Transaction Control Service Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.transaction.control; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.transaction.control; version="[1.0,1.1)"
```

Interface LocalResource

org.osgi.service.transaction.control

```
public interface LocalResource
```

Resources that can integrate with local transactions should do so using this interface

---

Method Summary		Page
void	**commit**() Commit the resource	24
void	**rollback**() Roll back the resource	24

Method Detail

commit

```
void commit()
     throws TransactionException
```

Commit the resource

Throws:

TransactionException

---

rollback

```
void rollback()
       throws TransactionException
```

Roll back the resource

Throws:

TransactionException

Interface ResourceProvider

org.osgi.service.transaction.control

All Known Subinterfaces:
JDBCConnectionProvider, JPAEntityManagerProvider

```
public interface ResourceProvider
```

A resource provider is used to provide a transactional resource to the application

---

Method Summary		Page
T	**getResource**(TransactionControl txControl) Get a resource which will associate with the current transaction context when used	25

Method Detail

getResource

```
T getResource(TransactionControl txControl)
        throws TransactionException
```

Get a resource which will associate with the current transaction context when used

Returns:
The resource which will participate in the current transaction
Throws:
TransactionException - if the resource cannot be registered with the transaction

Class TransactionBuilder

org.osgi.service.transaction.control

```
java.lang.Object
  └ org.osgi.service.transaction.control.TransactionBuilder
```

All Implemented Interfaces:
TransactionStarter

```
abstract public class TransactionBuilder
extends Object
implements TransactionStarter
```

A builder for a piece of transactional work

Constructor Summary	Page
TransactionBuilder()	26

Method Summary		Page
TransactionBuilder	**noRollbackFor**(Class<? extends Throwable> t, Class<? extends Throwable>... throwables) Declare a list of Exception types (and their subtypes) that *must not* trigger a rollback.	27
TransactionBuilder	**rollbackFor**(Class<? extends Throwable> t, Class<? extends Throwable>... throwables) Declare a list of Exception types (and their subtypes) that *must* trigger a rollback.	26

Methods inherited from interface org.osgi.service.transaction.control.TransactionStarter
notSupported, required, requiresNew, supports

Constructor Detail

TransactionBuilder

```
public TransactionBuilder()
```

Method Detail

rollbackFor

```
@SafeVarargs
public final TransactionBuilder rollbackFor(Class<? extends Throwable> t,
                                            Class<? extends Throwable>... throwables)
```

Declare a list of Exception types (and their subtypes) that *must* trigger a rollback. By default the transaction will rollback for all `Exception`s. If a more specific type is registered using `noRollbackFor(Class, Class...)` then that type will not trigger rollback. If the same type is registered using both `rollbackFor(Class, Class...)` and `noRollbackFor(Class, Class...)` then the transaction *will not* begin and will instead throw a `TransactionException`

Note that the behaviour of this method differs from Java EE and Spring in two ways:

- ! In Java EE and Spring transaction management checked exceptions are considered "normal returns" and do not trigger rollback. Using an Exception as a normal return value is considered a *bad* design practice. In addition this means that checked Exceptions such as

java.sql.SQLException do not trigger rollback by default. This, in turn, leads to implementation mistakes that break the transactional behaviour of applications.

! In Java EE it is legal to specify the same Exception type in `rollbackFor` and `noRollbackFor`. Stating that the same Exception should both trigger *and* not trigger rollback is a logical impossibility, and clearly indicates an API usage error. This API therefore enforces usage by triggering an exception in this invalid case.

Parameters:

`throwables` - The Exception types that should trigger rollback

Returns:

this builder

---

noRollbackFor

```
@SafeVarargs
public final TransactionBuilder noRollbackFor(Class<? extends Throwable> t,
                                              Class<? extends Throwable>... throwables)
```

Declare a list of Exception types (and their subtypes) that *must not* trigger a rollback. By default the transaction will rollback for all `Exception`s. If an Exception type is registered using this method then that type and its subtypes will *not* trigger rollback. If the same type is registered using both `rollbackFor(Class, Class...)` and `noRollbackFor(Class, Class...)` then the transaction *will not* begin and will instead throw a `TransactionException`

Note that the behaviour of this method differs from Java EE and Spring in two ways:

! In Java EE and Spring transaction management checked exceptions are considered "normal returns" and do not trigger rollback. Using an Exception as a normal return value is considered a *bad* design practice. In addition this means that checked Exceptions such as java.sql.SQLException do not trigger rollback by default. This, in turn, leads to implementation mistakes that break the transactional behaviour of applications.

! In Java EE it is legal to specify the same Exception type in `rollbackFor` and `noRollbackFor`. Stating that the same Exception should both trigger *and* not trigger rollback is a logical impossibility, and clearly indicates an API usage error. This API therefore enforces usage by triggering an exception in this invalid case.

Parameters:

`t` - An exception type that should not trigger rollback

`throwables` - further exception types that should not trigger rollback

Returns:

this builder

Interface TransactionContext

org.osgi.service.transaction.control

public interface **TransactionContext**

A transaction context defines the current transaction, and allows resources to register information and/or synchronisations

Method Summary		Page
boolean	**getRollbackOnly**() Is this transaction marked for rollback only	29
Object	**getScopedValue**(Object key) Get a value scoped to this transaction	28
Object	**getTransactionKey**() Get the key associated with the current transaction	28
TransactionStatus	**getTransactionStatus**()	29
void	**postCompletion**(Consumer<TransactionStatus> job) Register a callback that will be made after the decision to commit or rollback	29
void	**preCompletion**(Runnable job) Register a callback that will be made before a call to commit or rollback	29
void	**putScopedValue**(Object key, Object value) Associate a value with this transaction	29
void	**registerLocalResource**(LocalResource resource) Register an XA resource with the current transaction	30
void	**registerXAResource**(XAResource resource) Register an XA resource with the current transaction	30
void	**setRollbackOnly**() Mark this transaction for rollback	29
boolean	**supportsLocal**()	30
boolean	**supportsXA**()	30

Method Detail

getTransactionKey

Object **getTransactionKey**()

Get the key associated with the current transaction

Returns:
the transaction key, or null if there is no transaction

getScopedValue

Object **getScopedValue**(Object key)

Get a value scoped to this transaction

Returns:
The resource, or `null`

putScopedValue

```
void putScopedValue(Object key,
                    Object value)
```

Associate a value with this transaction

getRollbackOnly

```
boolean getRollbackOnly()
                throws IllegalStateException
```

Is this transaction marked for rollback only

Returns:
true if this transaction is rollback only
Throws:
`IllegalStateException` - if no transaction is active

setRollbackOnly

```
void setRollbackOnly()
             throws IllegalStateException
```

Mark this transaction for rollback

Throws:
`IllegalStateException` - if no transaction is active

getTransactionStatus

```
TransactionStatus getTransactionStatus()
```

Returns:
The current transaction status

preCompletion

```
void preCompletion(Runnable job)
            throws IllegalStateException
```

Register a callback that will be made before a call to commit or rollback

Throws:
`IllegalStateException` - if no transaction is active or the transaction has already passed beyond the `TransactionStatus.MARKED_ROLLBACK` state

postCompletion

```
void postCompletion(Consumer<TransactionStatus> job)
            throws IllegalStateException
```

Register a callback that will be made after the decision to commit or rollback

Throws:
`IllegalStateException` - if no transaction is active

---

supportsXA

```
boolean supportsXA()
```

Returns:
true if the current transaction supports XA resources

---

supportsLocal

```
boolean supportsLocal()
```

Returns:
true if the current transaction supports Local resources

---

registerXAResource

```
void registerXAResource(XAResource resource)
                 throws IllegalStateException
```

Register an XA resource with the current transaction

Throws:
`IllegalStateException` - if no transaction is active, or the current transaction is not XA capable

---

registerLocalResource

```
void registerLocalResource(LocalResource resource)
                   throws IllegalStateException
```

Register an XA resource with the current transaction

Throws:
`IllegalStateException` - if no transaction is active, or the current transaction is not XA capable

Interface TransactionControl

org.osgi.service.transaction.control

All Superinterfaces:
TransactionStarter

```
public interface TransactionControl
extends TransactionStarter
```

The interface used by clients to control the active transaction context

---

Method Summary		Page
boolean	**activeScope**()	31
boolean	**activeTransaction**()	31
TransactionBuilder	**build**() Build a transaction context to surround a piece of transactional work	31
TransactionContext	**getCurrentContext**()	32
boolean	**getRollbackOnly**() Gets the rollback status of the active transaction	32
void	**ignoreException**(Throwable t) Marks that the current transaction should not be rolled back if the supplied Exception is thrown by the current transactional work	32
void	**setRollbackOnly**() Marks the current transaction to be rolled back	32

Methods inherited from interface org.osgi.service.transaction.control.TransactionStarter
notSupported, required, requiresNew, supports

Method Detail

build

TransactionBuilder **build**()

Build a transaction context to surround a piece of transactional work

Returns:
A builder to complete the creation of the transaction

---

activeTransaction

boolean **activeTransaction**()

Returns:
true if a transaction is currently active

---

activeScope

boolean **activeScope**()

Returns:
true if a transaction is currently active, or if there is a "no transaction" context active

---

getCurrentContext

```
TransactionContext getCurrentContext()
```

Returns:
The current transaction context, which may be a "no transaction" context, or null if there is no active context

---

getRollbackOnly

```
boolean getRollbackOnly()
                throws IllegalStateException
```

Gets the rollback status of the active transaction

Returns:
true if the transaction is marked for rollback
Throws:
`IllegalStateException` - if no transaction is active

---

setRollbackOnly

```
void setRollbackOnly()
             throws IllegalStateException
```

Marks the current transaction to be rolled back

Throws:
`IllegalStateException` - if no transaction is active

---

ignoreException

```
void ignoreException(Throwable t)
             throws IllegalStateException
```

Marks that the current transaction should not be rolled back if the supplied Exception is thrown by the current transactional work

Parameters:
`t` - The exception to ignore
Throws:
`IllegalStateException` - if no transaction is active

Class TransactionException

org.osgi.service.transaction.control

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ java.lang.RuntimeException
              └ org.osgi.service.transaction.control.TransactionException
```

All Implemented Interfaces:
Serializable

Direct Known Subclasses:
TransactionRolledBackException

```
public class TransactionException
extends RuntimeException
```

An Exception indicating that there was a problem with starting, finishing, suspending or resuming a transaction

---

Constructor Summary	Page
TransactionException(String message) Creates a new TransactionException with the supplied message	33
TransactionException(String message, Throwable cause) Creates a new TransactionException with the supplied message and cause	33

Constructor Detail

TransactionException

```
public TransactionException(String message)
```

Creates a new TransactionException with the supplied message

TransactionException

```
public TransactionException(String message,
                            Throwable cause)
```

Creates a new TransactionException with the supplied message and cause

Class TransactionRolledBackException

org.osgi.service.transaction.control

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ java.lang.RuntimeException
              └ org.osgi.service.transaction.control.TransactionException
                  └ org.osgi.service.transaction.control.TransactionRolledBackException
```

All Implemented Interfaces:
Serializable

```
public class TransactionRolledBackException
extends TransactionException
```

An Exception indicating that the active transaction was rolled back

---

Constructor Summary	Page
TransactionRolledBackException(String message) Create a new TransactionRolledBackException with the supplied message	34
TransactionRolledBackException(String message, Throwable cause) Create a new TransactionRolledBackException with the supplied message	34

Constructor Detail

TransactionRolledBackException

```
public TransactionRolledBackException(String message)
```

Create a new TransactionRolledBackException with the supplied message

---

TransactionRolledBackException

```
public TransactionRolledBackException(String message,
                                      Throwable cause)
```

Create a new TransactionRolledBackException with the supplied message

Interface TransactionStarter

org.osgi.service.transaction.control

All Known Subinterfaces:
TransactionControl

All Known Implementing Classes:
TransactionBuilder

```
public interface TransactionStarter
```

Implementations of this interface are able to run a piece of work within a transaction

---

Method Summary		Page
T	**notSupported**(Callable<T> work) The supplied piece of work must be run outside the context of a transaction.	36
T	**required**(Callable<T> work) A transaction is required to run the supplied piece of work.	35
T	**requiresNew**(Callable<T> work) A new transaction is required to run the supplied piece of work.	35
T	**supports**(Callable<T> work) The supplied piece of work may run inside or outside the context of a transaction.	36

Method Detail

required

```
T required(Callable<T> work)
    throws TransactionException,
           TransactionRolledBackException
```

A transaction is required to run the supplied piece of work. If no transaction is active then it must be started and associated with the work and then completed after the transactional work has finished.

Returns:
The value returned by the work
Throws:
TransactionException - if there is an error starting or completing the transaction
TransactionRolledBackException - if the transaction rolled back due to a failure

---

requiresNew

```
T requiresNew(Callable<T> work)
       throws TransactionException,
              TransactionRolledBackException
```

A new transaction is required to run the supplied piece of work. If an existing transaction is active then it must suspended and a new transaction started and associated with the work. After the work has completed the new transaction must also complete and any suspended transaction be resumed.

Returns:
The value returned by the work
Throws:
TransactionException - if there is an error starting or completing the transaction
TransactionRolledBackException - if the transaction rolled back due to a failure

notSupported

```
T notSupported(Callable<T> work)
        throws TransactionException
```

The supplied piece of work must be run outside the context of a transaction. If an existing transaction is active then it must be suspended and a "no transaction" context associated with the work. After the work has completed any suspended transaction must be resumed.

The "no transaction" context does not support resource enlistment, and will not commit or rollback any changes, however it does provide a post completion callback to any registered functions. This function is suitable for final cleanup, such as closing a connection

Returns:
The value returned by the work
Throws:
TransactionException - if there is an error starting or completing the transaction

supports

```
T supports(Callable<T> work)
    throws TransactionException
```

The supplied piece of work may run inside or outside the context of a transaction. If an existing transaction or "no transaction" context is active then it will continue, otherwise a new "no transaction" context is associated with the work. After the work has completed any created transaction context must be completed.

The "no transaction" context does not support resource enlistment, and will not commit or rollback any changes, however it does provide a post completion callback to any registered functions. This function is suitable for final cleanup, such as closing a connection

Returns:
The value returned by the work
Throws:
TransactionException - if there is an error starting or completing the transaction

Enum TransactionStatus

org.osgi.service.transaction.control

```
java.lang.Object
  └ java.lang.Enum<TransactionStatus>
      └ org.osgi.service.transaction.control.TransactionStatus
```

All Implemented Interfaces:

Comparable<TransactionStatus>, Serializable

```
public enum TransactionStatus
extends Enum<TransactionStatus>
```

The status of the transaction A transaction may not enter all of the states in this enum, however it will always traverse the enum in ascending order. In particular if the TransactionStatus is reported as X then it will never proceed into a state Y where X.compareTo(Y) >= 0;

Enum Constant Summary	Page
ACTIVE A transaction is currently in progress	38
COMMITTED The transaction has committed	38
COMMITTING The transaction is in the process of being committed	38
MARKED_ROLLBACK A transaction is currently in progress and has been marked for rollback	38
NO_TRANSACTION No transaction is currently active	37
PREPARED A two phase commit is occurring and the transaction has been prepared	38
PREPARING A two phase commit is occurring and the transaction is being prepared	38
ROLLED_BACK The transaction has been rolled back	38
ROLLING_BACK The transaction is in the process of rolling back	38

Method Summary		Page
static TransactionStatus	**valueOf**(String name)	39
static TransactionStatus[]	**values**()	39

Enum Constant Detail

NO_TRANSACTION

```
public static final TransactionStatus NO_TRANSACTION
```

No transaction is currently active

ACTIVE

public static final TransactionStatus **ACTIVE**

A transaction is currently in progress

MARKED_ROLLBACK

public static final TransactionStatus **MARKED_ROLLBACK**

A transaction is currently in progress and has been marked for rollback

---

PREPARING

public static final TransactionStatus **PREPARING**

A two phase commit is occurring and the transaction is being prepared

---

PREPARED

public static final TransactionStatus **PREPARED**

A two phase commit is occurring and the transaction has been prepared

---

COMMITTING

public static final TransactionStatus **COMMITTING**

The transaction is in the process of being committed

COMMITTED

public static final TransactionStatus **COMMITTED**

The transaction has committed

ROLLING_BACK

public static final TransactionStatus **ROLLING_BACK**

The transaction is in the process of rolling back

ROLLED_BACK

public static final TransactionStatus **ROLLED_BACK**

The transaction has been rolled back

Method Detail

values

```
public static TransactionStatus[] values()
```

---

valueOf

```
public static TransactionStatus valueOf(String name)
```

Package org.osgi.service.transaction.control.jdbc

Interface Summary		Page
JDBCConnectionProvider	A specialised `ResourceProvider` suitable for obtaining JDBC connections.	*41*
JDBCConnectionProviderFactory	A factory for creating JDBCConnectionProvider instances	*42*

Interface JDBCConnectionProvider

org.osgi.service.transaction.control.jdbc

All Superinterfaces:
ResourceProvider<Connection>

```
public interface JDBCConnectionProvider
extends ResourceProvider<Connection>
```

A specialised `ResourceProvider` suitable for obtaining JDBC connections.

Instances of this interface may be available in the Service Registry, or can be created using a `JDBCConnectionProviderFactory`.

Methods inherited from interface org.osgi.service.transaction.control.ResourceProvider
getResource

Interface JDBCConnectionProviderFactory

org.osgi.service.transaction.control.jdbc

```
public interface JDBCConnectionProviderFactory
```

A factory for creating JDBCConnectionProvider instances

This factory can be used if the JDBCConnectionProvider should not be a public service, for example to protect a username/password.

Field Summary		Page
String	**CONNECTION_LIFETIME** The property used to set the maximum amount of time that connections in the pool should remain open	43
String	**CONNECTION_POOLING_ENABLED** The property used to determine whether connection pooling is enabled for this resource provider	43
String	**CONNECTION_TIMEOUT** The property used to set the maximum amount of time that the pool should wait for a connection	43
String	**IDLE_TIMEOUT** The property used to set the maximum amount of time that connections in the pool should remain idle before being closed	43
String	**LOCAL_ENLISTMENT_ENABLED** The property used to determine whether local enlistment is enabled for this resource provider	43
String	**MAX_CONNECTIONS** The property used to set the maximum number of connections that should be held in the pool	43
String	**MIN_CONNECTIONS** The property used to set the minimum number of connections that should be held in the pool	43
String	**USE_DRIVER** The property used to set the maximum number of connections that should be held in the pool	44
String	**XA_ENLISTMENT_ENABLED** The property used to determine whether XA enlistment is enabled for this resource provider	43

Method Summary		Page
JDBCConnectionProvider	**getProviderFor**(Driver driver, Map<String,Object> resourceProviderProperties) Create a private JDBCConnectionProvider using an existing Driver.	44
JDBCConnectionProvider	**getProviderFor**(DataSource ds, Map<String,Object> resourceProviderProperties) Create a private JDBCConnectionProvider using an existing DataSource.	44
JDBCConnectionProvider	**getProviderFor**(XADataSource ds, Map<String,Object> resourceProviderProperties) Create a private JDBCConnectionProvider using an existing XADataSource.	44
JDBCConnectionProvider	**getProviderFor**(org.osgi.service.jdbc.DataSourceFactory dsf, Properties jdbcProperties, Map<String,Object> resourceProviderProperties) Create a private JDBCConnectionProvider using a DataSourceFactory.	44

Field Detail

XA_ENLISTMENT_ENABLED

```
public static final String XA_ENLISTMENT_ENABLED = "osgi.xa.enabled"
```

The property used to determine whether XA enlistment is enabled for this resource provider

---

LOCAL_ENLISTMENT_ENABLED

```
public static final String LOCAL_ENLISTMENT_ENABLED = "osgi.local.enabled"
```

The property used to determine whether local enlistment is enabled for this resource provider

---

CONNECTION_POOLING_ENABLED

```
public static final String CONNECTION_POOLING_ENABLED = "osgi.connection.pooling.enabled"
```

The property used to determine whether connection pooling is enabled for this resource provider

---

CONNECTION_TIMEOUT

```
public static final String CONNECTION_TIMEOUT = "osgi.connection.timeout"
```

The property used to set the maximum amount of time that the pool should wait for a connection

---

IDLE_TIMEOUT

```
public static final String IDLE_TIMEOUT = "osgi.idle.timeout"
```

The property used to set the maximum amount of time that connections in the pool should remain idle before being closed

---

CONNECTION_LIFETIME

```
public static final String CONNECTION_LIFETIME = "osgi.connection.lifetime"
```

The property used to set the maximum amount of time that connections in the pool should remain open

---

MIN_CONNECTIONS

```
public static final String MIN_CONNECTIONS = "osgi.connection.min"
```

The property used to set the minimum number of connections that should be held in the pool

---

MAX_CONNECTIONS

```
public static final String MAX_CONNECTIONS = "osgi.connection.max"
```

The property used to set the maximum number of connections that should be held in the pool

---

USE_DRIVER

```
public static final String USE_DRIVER = "osgi.use.driver"
```

The property used to set the maximum number of connections that should be held in the pool

Method Detail

getProviderFor

```
JDBCConnectionProvider getProviderFor(org.osgi.service.jdbc.DataSourceFactory dsf,
                                      Properties jdbcProperties,
                                      Map<String,Object> resourceProviderProperties)
```

Create a private JDBCConnectionProvider using a DataSourceFactory.

Parameters:
`jdbcProperties` - The properties to pass to the `org.osgi.service.jdbc.DataSourceFactory` in order to create the underlying `DataSource`
`resourceProviderProperties` - Configuration properties to pass to the JDBC Resource Provider runtime
Returns:
A JDBCConnectionProvider that can be used in transactions

---

getProviderFor

```
JDBCConnectionProvider getProviderFor(DataSource ds,
                                      Map<String,Object> resourceProviderProperties)
```

Create a private JDBCConnectionProvider using an existing `DataSource`.

Parameters:
`resourceProviderProperties` - Configuration properties to pass to the JDBC Resource Provider runtime
Returns:
A JDBCConnectionProvider that can be used in transactions

---

getProviderFor

```
JDBCConnectionProvider getProviderFor(Driver driver,
                                      Map<String,Object> resourceProviderProperties)
```

Create a private JDBCConnectionProvider using an existing `Driver`.

Parameters:
`resourceProviderProperties` - Configuration properties to pass to the JDBC Resource Provider runtime
Returns:
A JDBCConnectionProvider that can be used in transactions

---

getProviderFor

```
JDBCConnectionProvider getProviderFor(XADataSource ds,
                                      Map<String,Object> resourceProviderProperties)
```

Create a private `JDBCConnectionProvider` using an existing `XADataSource`.

Parameters:

`resourceProviderProperties` - Configuration properties to pass to the JDBC Resource Provider runtime

Returns:

A `JDBCConnectionProvider` that can be used in transactions

Package org.osgi.service.transaction.control.jpa

Interface Summary		Page
JPAEntityManagerProvider	A specialised `ResourceProvider` suitable for obtaining JPA `EntityManager` instances.	*47*
JPAEntityManagerProviderFactory	A factory for creating JDBCConnectionProvider instances	*48*

Interface JPAEntityManagerProvider

org.osgi.service.transaction.control.jpa

All Superinterfaces:
ResourceProvider<EntityManager>

```
public interface JPAEntityManagerProvider
extends ResourceProvider<EntityManager>
```

A specialised `ResourceProvider` suitable for obtaining JPA `EntityManager` instances.

Instances of this interface may be available in the Service Registry, or can be created using a `JPAEntityManagerProviderFactory`.

Methods inherited from interface org.osgi.service.transaction.control.ResourceProvider
getResource

Interface JPAEntityManagerProviderFactory

org.osgi.service.transaction.control.jpa

public interface **JPAEntityManagerProviderFactory**

A factory for creating JDBCConnectionProvider instances

This factory can be used if the JDBCConnectionProvider should not be a public service, for example to protect a username/password.

Field Summary		Page
String	**LOCAL_ENLISTMENT_ENABLED** The property used to determine whether local enlistment is enabled for this resource provider	48
String	**TRANSACTIONAL_DB_CONNECTION** The property used to provide a JDBCConnectionProvider to the resource provider.	48
String	**XA_ENLISTMENT_ENABLED** The property used to determine whether XA enlistment is enabled for this resource provider	48

Method Summary		Page
JPAEntityManagerProvider	**getProviderFor**(EntityManagerFactory emf, Map<String,Object> jpaProperties, Map<String,Object> resourceProviderProperties) Create a private JPAEntityManagerProvider using an existing EntityManagerFactory.	49
JPAEntityManagerProvider	**getProviderFor**(org.osgi.service.jpa.EntityManagerFactoryBuilder emfb, Map<String,Object> jpaProperties, Map<String,Object> resourceProviderProperties) Create a private JPAEntityManagerProvider using an org.osgi.service.jpa.EntityManagerFactoryBuilder	49

Field Detail

XA_ENLISTMENT_ENABLED

public static final String **XA_ENLISTMENT_ENABLED** = "osgi.xa.enabled"

The property used to determine whether XA enlistment is enabled for this resource provider

LOCAL_ENLISTMENT_ENABLED

public static final String **LOCAL_ENLISTMENT_ENABLED** = "osgi.local.enabled"

The property used to determine whether local enlistment is enabled for this resource provider

TRANSACTIONAL_DB_CONNECTION

public static final String **TRANSACTIONAL_DB_CONNECTION** = "osgi.jdbc.provider"

The property used to provide a JDBCConnectionProvider to the resource provider. This will be converted into a DataSource by the factory, and passed to the

`org.osgi.service.jpa.EntityManagerFactoryBuilder` using the javax.persistence.jtaDataSource property

Method Detail

getProviderFor

```
JPAEntityManagerProvider getProviderFor(org.osgi.service.jpa.EntityManagerFactoryBuilder emfb,
                                        Map<String,Object> jpaProperties,
                                        Map<String,Object> resourceProviderProperties)
```

Create a private `JPAEntityManagerProvider` using an `org.osgi.service.jpa.EntityManagerFactoryBuilder`

Parameters:

`jpaProperties` - The properties to pass to the `org.osgi.service.jpa.EntityManagerFactoryBuilder` in order to create the underlying `EntityManagerFactory` and `EntityManager` instances

`resourceProviderProperties` - Configuration properties to pass to the JPA Resource Provider runtime

Returns:

A `JPAEntityManagerProvider` that can be used in transactions

---

getProviderFor

```
JPAEntityManagerProvider getProviderFor(EntityManagerFactory emf,
                                        Map<String,Object> jpaProperties,
                                        Map<String,Object> resourceProviderProperties)
```

Create a private `JPAEntityManagerProvider` using an existing `EntityManagerFactory`.

Parameters:

`jpaProperties` - The properties to pass to the `EntityManagerFactory` in order to create the underlying `EntityManager` instances

`resourceProviderProperties` - Configuration properties to pass to the JDBC Resource Provider runtime

Returns:

A `JPAEntityManagerProvider` that can be used in transactions

8 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

9 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

10 Document Support

10.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. http://blog.osgi.org/2013/11/the-transaction-composability-problem.html

[4]. http://www.slideshare.net/tux2323/osgi-and-spring-data-for-simple-web-application-development
[5]. http://projects.spring.io/spring-data/
[6]. http://www.liquibase.org/
[7]. http://flywaydb.org/

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.***

10.2 Author's Address

Name	
Company	
Address	
Voice	
e-mail	

10.3 Acronyms and Abbreviations

10.4 End of Document