MicroProfile LRA

The MicroProfile community and its contributors

2.0-RC1, December 21, 2021: Draft

Table of Contents

```
Specification: MicroProfile LRA
Version: 2.0-RC1
Status: Draft
Release: December 21, 2021
```

Copyright

Eclipse Foundation Specification License

Disclaimers

Introduction

The proposal introduces annotations and APIs for services to coordinate long running activities whilst still maintaining loose coupling and doing so in such a way as to guarantee a globally consistent outcome without the need to take locks on data.

Motivation

In a loosely coupled service based environment there is sometimes a need for different services to provide consistency guarantees. Typical examples include:

- order processing involving three services (take order, bill customer, ship product). If the shipping service finds that it is out of stock then the customer will have been billed with no prospect of receiving the item.
- an airline overbooks a flight which means that the booking count and the flight capacity are inconsistent.

There are various ways that systems overcome such inconsistency but it would be advantageous to provide a generic solution which handles failure conditions, maintains state for those flows that span long periods of time and ensures that remedial activities are called correctly.

Traditional techniques for guaranteeing consistency in distributed environments has focused on XA transactions where locks may be held for long periods thereby introducing strong coupling between services and decreasing concurrency to unacceptable levels. Additionally, if such a transaction aborts then valuable work which may be valid will be rolled back. In view of these issues an alternative approach is desirable.

Goals

- support long running actions
- no strong coupling between services
- allow actions to finish early
- allow some parts of a computation to be cancelled without affecting other parts of the computation (nested activities)
- guarantee execution of compensating actions if a business activity is cancelled

The solution

We propose a compensation based approach in which participants make changes visible but register a compensatory action which is performed if something goes wrong. We call the model LRA (short for Long Running Action) and is based on work done within the OASIS Web Services Composite Application Framework Technical Committee, namely Long Running Action transaction model, but updated to be more suited for use in microservice based architectures.

In the LRA model, an activity reflects business interactions: all work performed within the scope of an activity is required to be compensatable and the protocol ensures that when the activity terminates then either all work will be accepted or will be compensated, i.e. the activity's work is either performed successfully or undone. For example, when a user reserves a seat on a flight, the airline reservation centre may take an optimistic approach and actually book the seat and debit the user's account, relying on the fact that most of their customers who reserve seats later book them; the compensation action for this activity would be to un-book the seat and credit the user's account. How services perform their work and ensure it can be undone if compensation is required are implementation choices and is not exposed to the LRA model which simply defines the triggers for compensation actions and the conditions under which those triggers are executed. In other words, the LRA protocol is concerned only with ensuring participants obey the protocol necessary to make an activity compensatable and is intended to better model interactions between microservices. Issues such as isolation of services between potentially conflicting activities and durability of service work are assumed to be implementation decisions. Although this may result in non-atomic behaviour for the overall business activity, other activities may be started by the service to attempt to compensate in some other manner.

The Model

The model concerns participants (aka compensators), a logical coordinator and a client. A client starts a new LRA via an annotation which results in the creation of a new LRA. When a business service does work that may have to be later compensated it enlists a participant with the LRA which, when the client later closes or cancels the LRA, operates on behalf of the service to undo the work the service performed within the scope of an LRA or to compensate for the fact that the original work could not be completed. The following diagram shows the sequence of events that this model implies:

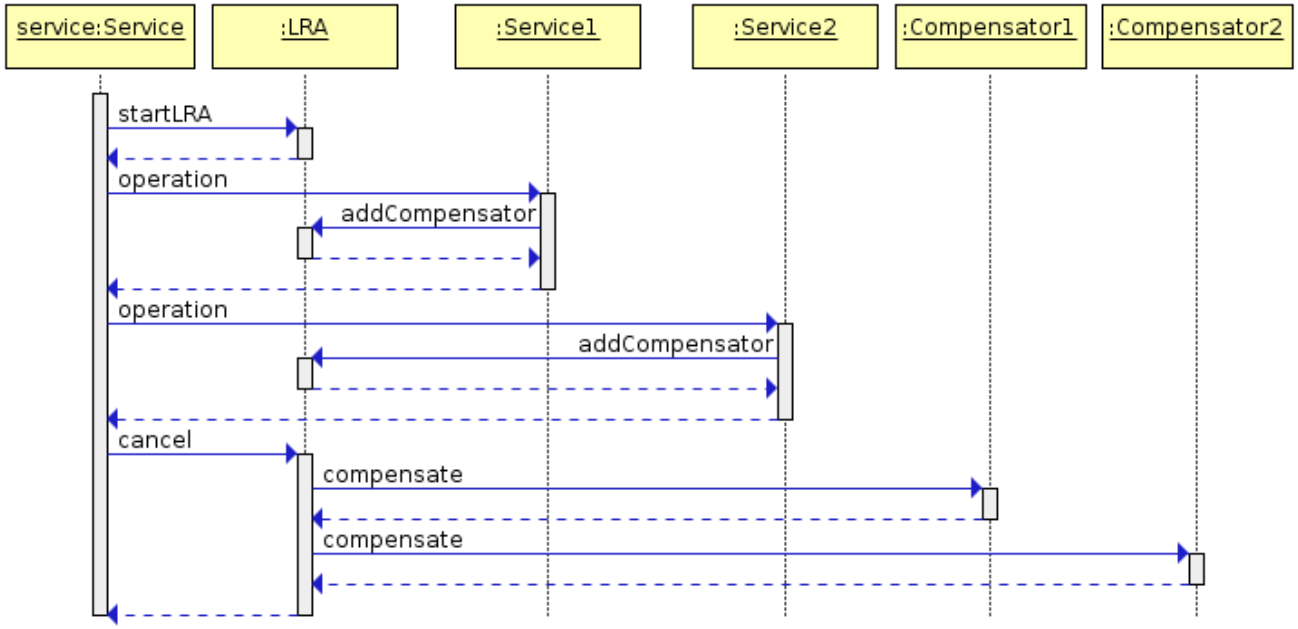

Figure 1. LRA Protocol Sequence

The participants will be invoked in the following way when the LRA terminates:

- **Success**: the client has closed the LRA and the activity has completed successfully. All participants (including ones enlisted with any LRAs nested under the top level LRA) that are associated with the LRA are informed that the activity has terminated and they can perform any necessary cleanup.
- **Fail**: the client has cancelled the LRA and the activity has completed unsuccessfully. All participants that are registered with the LRA will be invoked to perform compensation in the reverse order [1]. The LRA forgets about all participants that indicated they operated correctly. Otherwise, compensation may be attempted again (possibly after a period of time) or alternatively a compensation violation has occurred and must be logged. Each service is required to log sufficient information in order to ensure (with the best effort) that compensation is possible even in the event of failures. Note that compensation violations include the case where a participant completes when asked to compensate.

Similar remarks apply to completion violations.

Interposition (nesting) and reliably storing the state of participants allows the system to drive a consistent view of the outcome and to take recovery actions in the event of failure, but allowing always the possibility that recovery isn't possible which must be logged or flagged for the administrator and manual intervention may be necessary to restore an application's consistency.

A LRA and a participant both follow similar state models (with slightly different wording to indicate that the states refer to the LRA as a whole rather than to individual participants):

- `Active`: the LRA exists and has not been asked to close or cancel yet
- `Cancelling`: the LRA is currently being cancelled
- `Cancelled`: the LRA has successfully cancelled
- `FailedToCancel`: at least one participant was not able to compensate
- `Closing`: the LRA is currently being closed

- `Closed`: the LRA has closed
- `FailedToClose`: at least one participant was not able to complete

Note that the model also allows parties to be reliably notified when LRAs reach one of the final states. Such parties are referred to as LRA listeners (in contrast to participants).

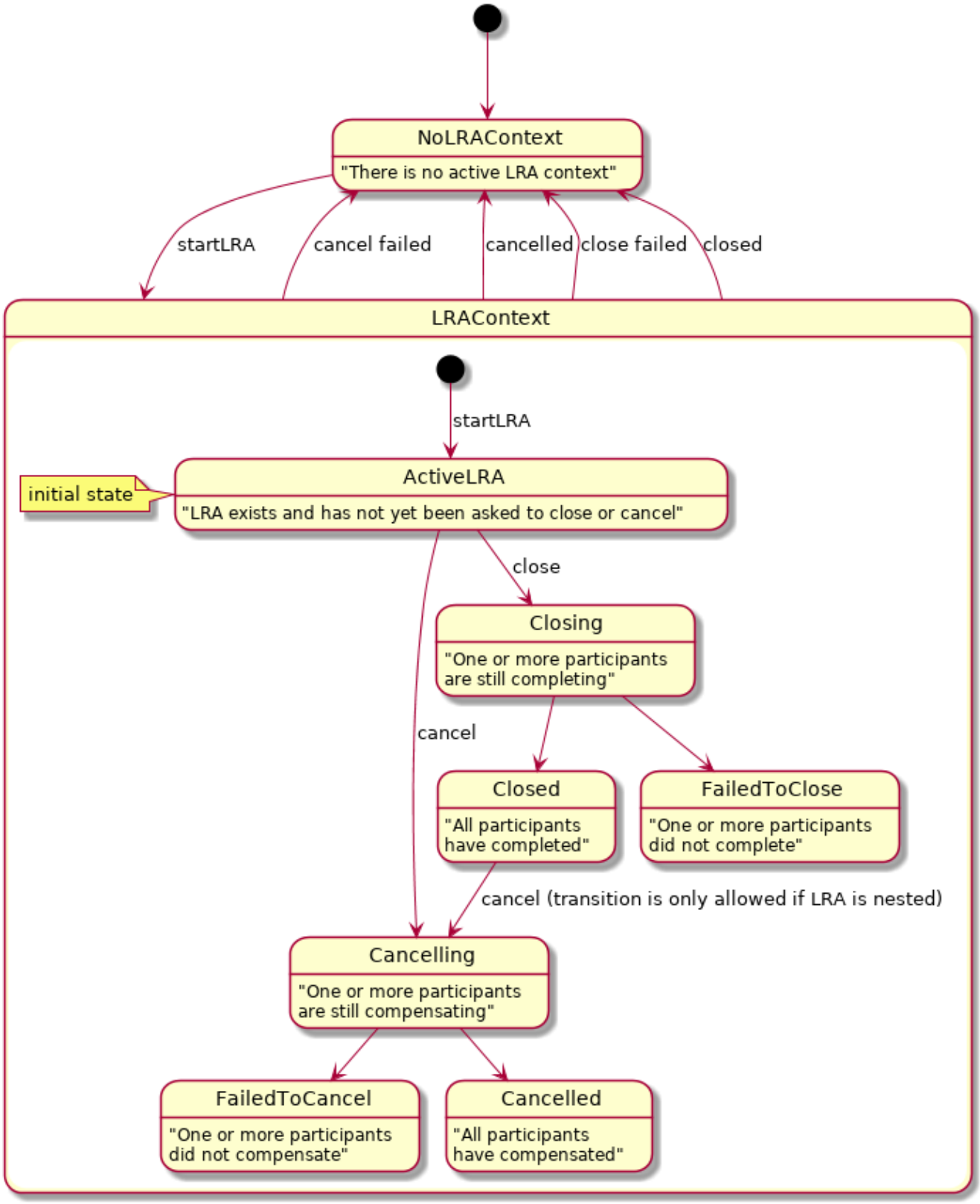

Figure 2. LRA State Model

And the participants state model has the following states:

- `Active`: the participant exists and has not been asked to compensate or complete yet
- `Compensating`: a participant is currently compensating for the work that it performed during the LRA.
- `Compensated`: a participant has successfully compensated for the work that it performed during the LRA.
- `FailedToCompensate`: the participant was not able to compensate for the work it did during the LRA.
- `Completing`: the participant is tidying up after being told to complete.
- `Completed`: the participant has confirmed that it has finished tidying up.
- `FailedToComplete`: the participant was unable to tidy-up some resources it allocated during the LRA.

When a participant joins an LRA it is said to be enlisted and enters the state model in the `Active` state.

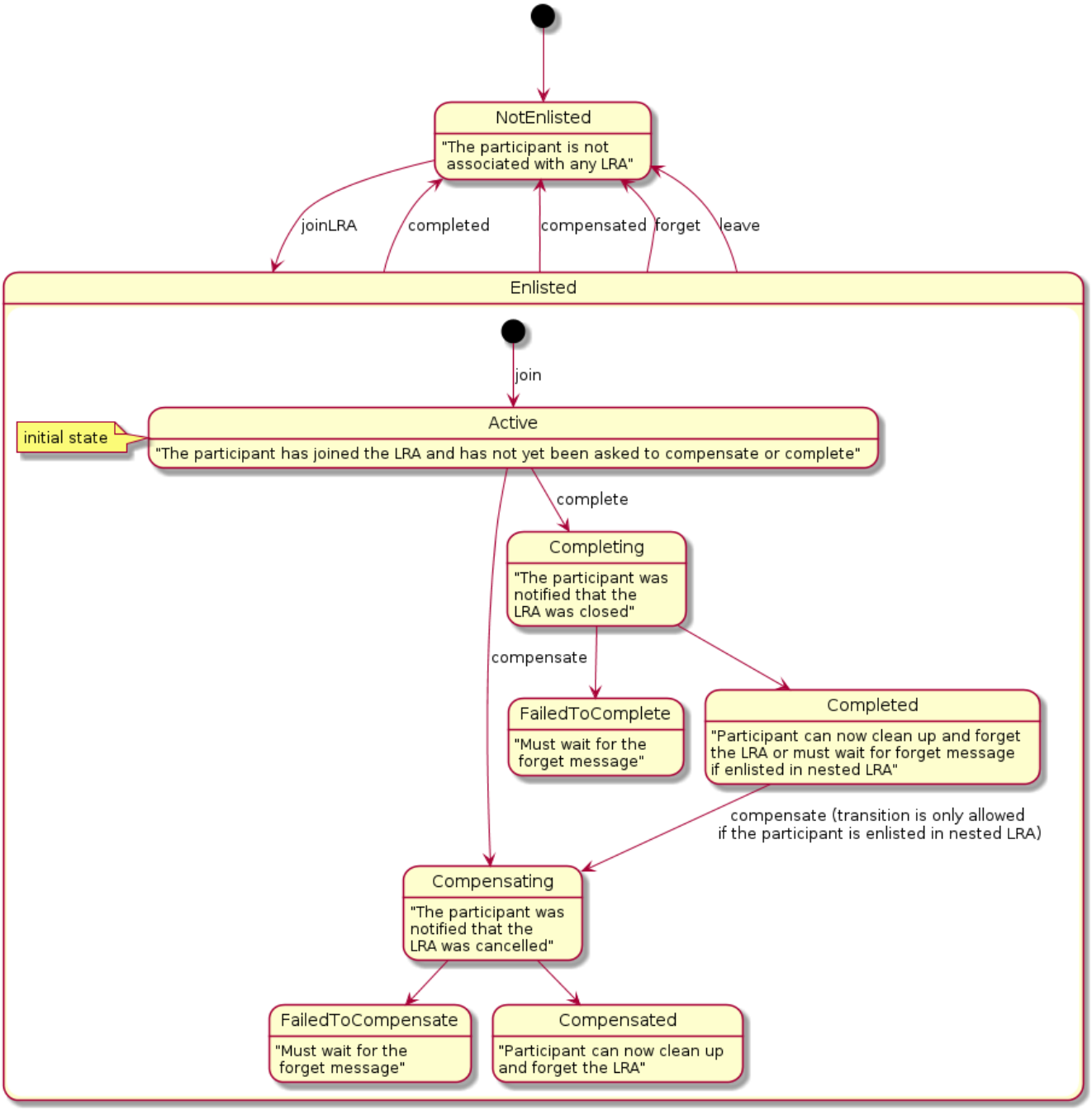

Figure 3. Participant State Model

Note that when the LRA has been asked to cancel it enters the state `Cancelling` and will eventually ask all registered participants to enter the state `Compensating`. A similar remark applies to the LRA state `Closing` and participant state `Completing`.

The participant can be enlisted only once per LRA instance. When participant is about to be enlisted multiple times (e.g., client calls the same `@LRA` method several times) the follow-up enlistments of such participant MUST be ignored.

Different usage patterns for LRAs are possible, for example LRAs may be used sequentially and/or concurrently, where the termination of one LRA signals the start of some other unit of work within an application. Additionally, speculative execution of work can be supported by nesting LRAs, some of which can be cancelled independently of the parent LRA whilst others are closed based on the outcome of other LRAs. LRAs are units of compensatable work and an application may have as many such units of work operating simultaneously as it needs to accomplish its tasks. Furthermore,

the outcome of work within LRAs may determine how other LRAs are terminated. An application can be structured so that LRAs are used to assemble units of compensatable work and then held in the active state while the application performs other work in the scope of different (concurrent or sequential) LRAs. Only when the right subset of work (LRAs) is arrived at by the application will that subset be confirmed; all other LRAs will be told to cancel (complete in a failure state).

In the rest of this proposal we specify an API for controlling the life cycle of and participation in LRAs:

Java Annotations for LRAs

Support for the proposal in MicroProfile is primarily based upon the use of Java annotations for controlling the life cycle of LRAs and of participants, i.e., the service developer annotates JAX-RS resources to specify how LRAs should be controlled and when to *enlist a class* as a participant.

Quick overview of annotations

The definitive documentation for how the annotations affect the behaviour of a running program is the javadoc and this specification should be read in conjunction with it. The annotations are defined in the org.eclipse.microprofile.lra.annotation package:

Annotation	Description
@LRA	Controls the life cycle of an LRA.
@Compensate	Indicates that the method should be invoked if the LRA is cancelled.
@Complete	Indicates that the method should be invoked if the LRA is closed.
@Forget	Indicates that the method may release any resources that were allocated for this LRA.
@Leave	Indicates that this class is no longer interested in this LRA.
@Status	When the annotated method is invoked it should report the status.
@AfterLRA	When an LRA has reached a final state the annotated method is invoked.

Briefly, these annotations are used as follows:

The application annotates some JAX-RS resource method with `@LRA` which determines whether the method will run in the context of an LRA. Generally, if a method starts a new LRA, it will be closed when the method finishes execution (but elements of the annotation facilitate full control over the LRA life cycle).

If execution is to run with an active LRA and the associated class contains other methods annotated with `@Compensate` and `@Complete` then these methods will be associated with the active LRA. They will be invoked when the LRA is subsequently cancelled or closed. If the participant successfully compensates or completes then it may forget about the LRA. Otherwise, it should remember that it is still associated with the LRA and it MUST report the status of the association using values in the `ParticipantStatus` enum according to the participant state model defined earlier. It can report the status directly from the `@Compensate` or `@Complete` methods if they are idempotent, otherwise it MUST provide a method annotated with `@Status`.

If the participant is no longer associated with the LRA (because it has successfully compensated or completed or because it has left the LRA) it MAY return the `410 Gone` HTTP status code from any of these methods.

If the participant knows it will never be able to compensate or complete then it MUST remember that it could not perform the requested action until explicitly told that it can clean up by providing a method annotated with `@Forget` (the requirement is marked MUST because message delivery is not guaranteed in a distributed system).

If multiple methods are annotated with the same annotation an arbitrary one will be chosen.

Configuration parameters

This specification has 1 configuration parameter which is retrieved using the MicroProfile Config principles.

```
mp.lra.propagation.active=true
```

The allowed values for the configuration parameter are defined by the MicroProfile Config converter for boolean: `values for true (case insensitive) "true", "1", "YES", "Y" "ON". Any other value will be interpreted as false.

When a JAX-RS endpoint, or the containing class, is not annotated with @LRA, but it is called on a MicroProfile LRA compliant runtime, the system will propagate the LRA related HTTP headers when this parameter resolves to true. The behaviour is similar to the `LRA.Type SUPPORTS` (when true) and `NOT_SUPPORTED` (when false) values but only defines the propagation aspect. In other words the class does not have to be a participant in order for the LRA context to propagate, i.e., such propagation of the header does not imply that the LRA is in any particular state, and in fact the LRA may not even correspond to a valid LRA.

Also, there is no validation performed in any form of the LRA related HTTP headers. They are just propagated on an outgoing request in the case the configuration value resolves to `true`. All values for the headers defined in `org.eclipse.microprofile.lra.annotation.ws.rs.LRA.LRA_HTTP_CONTEXT_HEADER` and `org.eclipse.microprofile.lra.annotation.ws.rs.LRA.LRA_HTTP_PARENT_CONTEXT_HEADER` needs to be propagated.

Because of this last statement, an implementation does not need to guard against "spoofing of LRA ids": security is specifically not defined in the specification and it is up to implementers to define the security elements of their approach.

The LRA Context

When a method (that is annotated with any of the annotations defined in this specification) is invoked in the context of an LRA, its identifier (of type `java.net.URI`) MUST be made available to the business logic. This LRA identifier is referred to as the `active context`.

For JAX-RS resource methods the LRA identifier is made available to the business method (via

standard JAX-RS mechanisms, i.e., a JAX-RS `@Context` annotation or by injecting a JAX-RS header param with the name specified by the Java constant `LRA_HTTP_CONTEXT_HEADER` as defined in the LRA annotation class. This header is referred to as the `context header`.

When using non-JAX-RS based `@Complete` and `@Compensate` methods (see Non-JAX-RS participant methods) this identifier is passed as a method parameter.

The implementation MUST propagate the `active context` on outgoing JAX-RS requests.

If an LRA is propagated to a resource that is not annotated with `@LRA` then the implementation looks at the configuration parameter `mp.lra.propagation.active` to decide if the LRA context information in the header must be propagated with the outgoing requests (value evaluates to `true`, the default) or not (value evaluates to `false`).

The user is allowed to manually assign the `context header` on JAX-RS requests and responses:

Setting the context on outgoing JAX-RS requests

If the context is set on an outgoing JAX-RS request then it becomes the `active context` for that invocation. When the invocation returns the context that was active before the invocation once again becomes the active context.

Setting the context on JAX-RS responses

A resource is allowed to manually set the `context header` on JAX-RS responses. Users should be careful when using this capability since this context will overwrite whatever the implementation has set.

Starting and Ending LRAs

The life cycle of an LRA is controlled via the LRA annotation (@LRA).

The annotation MUST be applied to JAX-RS annotated methods, classes, superclasses or interface methods otherwise it has no effect. It determines whether or not the JAX-RS method will run in the context of an LRA and controls whether or not:

- any incoming context should be suspended and if so if a new one should be started.
- to start a new LRA.
- to end any LRA context when the method ends.
- to throw an exception if there should be an LRA present on method entry.
- to end an LRA if the method returns particular HTTP status codes.
- to enlist the class as a participant of the LRA.

If the incoming context has already ended or does not exist the method will immediately return 410 Gone HTTP status code.

Similarly, if any of the preconditions listed in the Type element (see LRA.Type) of the LRA annotation are not met then the method will immediately return 412 Precondition Failed HTTP

status code.

When the @LRA annotation is defined multiple times, this is the order of precedence

- On the JAX-RS method itself.
- On the class declaration containing the JAX-RS method.
- On the method declaration of a superclass.
- On the method declaration of an interface.

More specifically, when the JAX-RS method and the class containing the JAX-RS method both have the `@LRA` annotation, the one from the JAX-RS method should be used.

If the method is to run in the context of an LRA then the annotated class MUST also contain a method annotated with one or both of `@Compensate` and `@AfterLRA`. If the `@Compensate` annotation is present then the class (aka participant or compensator) will be enlisted with the LRA. If the `@AfterLRA` annotation is present then the class will be notified when the LRA reaches one of the final LRA states. In the case that the class contains a `@Compensate` annotated method, this means that before executing the `@LRA` annotated method, the implementation must be able to guarantee that the participant has been or will eventually be registered with the LRA. This implies that the LRA is effectively validated as being active since it is not possible to enlist with an inactive LRA. There is an exception to this statement if the LRA is nested and in the `Closed` state (since the transition `Closed` → `Cancelling` is permitted). In this case, the method invocation will be allowed to proceed provided that the participant is already registered with the LRA (please refer to the javadoc for the `NESTED` value of the `LRA#value()` element for more information about this situation, in particular if the method invocation causes the LRA to close then the `@Complete` annotated method, if present, **is not** called again). And if the participant is not registered with the LRA then the caller of the LRA method will receive a `412` HTTP status response as described in the javadoc.

In the case that the class contains a `@AfterLRA` annotated method, this means that before executing the `@LRA` annotated method, the implementation must be able to guarantee that the listener will be notified about the outcome of the LRA. The practical consequence of these requirements is that the implementation must durably record that the participant/listener is associated with the LRA before allowing the business invocation to proceed.

Enlisting with an LRA means that the instance will be notified when the current LRA is subsequently cancelled or closed (if the class also contains a method annotated with `@Complete`). In addition, the implementation MUST generate a URI for this participant and make it available to the business method (on which the LRA annotation is defined) via a JAX-RS header param with the name defined by the Java constant `LRA_HTTP_RECOVERY_HEADER` as defined in the LRA annotation (github link). The header MUST also be made available to the application whenever any of the participant callbacks are invoked. The application is free to ignore this header. Enlisting in an LRA is explained in more detail in a later section of this document.

It is the `LRA.Type` element of the LRA annotation that provides a fine-grained control over the LRA for the duration of the execution of the annotated method and the reader should consult the javadoc for details.

Bear in mind that the LRA annotation is expected to be used with JAX-RS which expects the `Response`

being the method return type. If a `200 OK` response is to be returned then the method can return any data type that it desires. The specification does not discuss what happens when an uncaught exception is thrown from the `@LRA` annotated method. It's up to the framework to transform the thrown exception to the JAX-RS `Response` error status code. It's a usual practice the `RuntimeException` to be transformed to the `5xx Response` status code.

The LRA annotation can be used to control whether or not the LRA context should be closed when the method finishes (see the javadoc for the `LRA#end()` element for details).

The LRA annotation can be used to control whether or not the LRA context should be cancelled when the method finishes (see the javadoc for the `LRA#cancelOn()` and `cancelOnFamily` elements for details).

When an LRA is present, its identifier MUST be made available to the business logic via request and response headers as described earlier (using a header name specified in the Java constant `LRA_HTTP_CONTEXT_HEADER`).

The following is a simple example of how to start an LRA and how to receive a notification when the LRA is later cancelled (`@Compensate` is invoked) or closed (`@Complete` is invoked):

```
@Path("/")
@ApplicationScoped
public class SimpleLRAParticipant
{
    @LRA(LRA.Type.REQUIRES_NEW)
    @Path("/performInLRA")
    @PUT
    public void doInTransaction(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId)
    {
        /*
         * Perform business actions in the context of the LRA identified by the
         * value in the injected JAX-RS header. This LRA was started just before
         * the method was entered (REQUIRES_NEW) and will be closed when the
         * method finishes at which point the completeWork method below will be
         * invoked.
         */
    }

    @Complete
    @Path("/complete")
    @PUT
    public Response completeWork(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId)
    {
        /*
         * Free up resources allocated in the context of the LRA identified by the
         * value in the injected JAX-RS header.
         *
         * Since there is no @Status method in this class, completeWork MUST be
         * idempotent and MUST return the status.
         */
         return Response.ok(ParticipantStatus.Completed.name()).build();
    }

    @Compensate
    @Path("/compensate")
    @PUT
    public Response compensateWork(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId)
    {
        /*
         * The LRA identified by the value in the injected JAX-RS header was
         * cancelled so the business logic should compensate for any actions
         * that have been performed while running in its context.
         *
         * Since there is no @Status method in this class, compensateWork MUST be
         * idempotent and MUST return the status
         */
         return Response.ok(ParticipantStatus.Compensated.name()).build();
    }
}
```

The example also shows that when an LRA is present its identifier can be obtained by reading the request headers.

The next example demonstrates how to execute business logic in the context of a new or existing LRA (the `bookTrip` method) and to close it in a different method (the `confirmTrip` method) using the LRA `end` element. [Note that this specification supports nested contexts so if there was already a context present on method entry it is possible that it is nested - if the business logic needs to do any special actions for nested contexts, it should inject the `LRA_HTTP_PARENT_CONTEXT_HEADER` header and check whether the value is null].

The example also shows how to configure the LRA to be automatically cancelled if the business method returns the particular HTTP status codes identified in the `cancelOn` and `cancelOnFamily` elements:

```
  @LRA(value = LRA.Type.REQUIRED, // if there is no incoming context a new one is
created
       cancelOn = {
           Response.Status.INTERNAL_SERVER_ERROR // cancel on a 500 code
       },
       cancelOnFamily = {
           Response.Status.Family.CLIENT_ERROR // cancel on any 4xx code
       },
       end = false) // the LRA will continue to run when the method finishes
  @Path("/book")
  @POST
  public Response bookTrip(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
                           @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI parentLRA)
{
      if (parentLRA != null) { // is the context nested
          // code which is sensitive to executing with a nested context goes here
      }
      ...
  }

  @LRA(value = LRA.Type.MANDATORY, // requires an active context before method can be
executed
       end = true) // end the received LRA when the method finishes
  @Path("/confirm")
  @PUT
  public Booking confirmTrip(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
                           @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI parentLRA,
                           Booking booking) throws BookingException {
      if (parentLRA != null) { // is the context nested
          // code which is sensitive to executing with a nested context goes here
      }
  }
```

The `end = false` element on the bookTrip method forces the LRA to continue running when the method finishes and the `end = true` element on the confirmTrip method forces the LRA (started by

the bookTrip method) to close the LRA.

Discovering the Outcome of an LRA

As remarked in the previous section, a JAX-RS resource method runs with an active context depending upon the value specified in the `@LRA` annotation. The final state of this LRA can be discovered by marking one of the other methods in the class with the `@AfterLRA` annotation. When the LRA enters a terminal state, the method will be passed the id of the LRA together with the `LRAStatus` (see the javadoc for the `@AfterLRA` annotation for more information). Note that the final states of an LRA are defined by the LRA state model and it would be a specification violation if the implementation calls the method when the LRA is not in a final state. Further information about method signatures and JAX-RS response codes is given below in section JAX-RS methods.

Note that the resource does not need to be a participant in order to receive this notification. Therefore, in the following resource definition, although no method is annotated with `@Compensate`, if the method called `activityWithLRA` is invoked then the method `notifyLRAFinished` will be called when the LRA finishes:

```
@Path("/")
@ApplicationScoped
public class BusinessResource {
    @PUT
    @Path("/work")
    @LRA(value = LRA.Type.REQUIRES_NEW)
    public Response activityWithLRA(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId) {
        // perform a business action in the context of lraId
        return Response.ok().build();
    }

    @PUT
    @Path("/after")
    @AfterLRA // method is called when the LRA associated with the method
activityWithLRA finishes
    public Response notifyLRAFinished(@HeaderParam(LRA_HTTP_ENDED_CONTEXT_HEADER) URI
lraId, // we use the LRA_HTTP_ENDED_CONTEXT_HEADER because the LRA_HTTP_CONTEXT_HEADER
may contain a newly started LRA
                                      @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI
parentLRA,
                                      LRAStatus status) {

        if (parentLRA != null) { // is the context nested
            // code which is sensitive to executing with a nested context goes here
        }

        switch (status) {
            case Closed:
                // the LRA was successfully closed
            ...
        }
    }
}
```

Compensating Activities

As remarked elsewhere, the LRA specification attempts to enforce some of the traditional guarantees provided by transactional systems such as atomicity (all or nothing) whilst relaxing others, such as isolation of work amongst participants. The characteristic of strong consistency of data (the system can only be observed to transition between consistent states) is also relaxed in favour of what is referred to as `eventual consistency`. The LRA specification ensures atomicity and eventual consistency by placing certain requirements on the entities that participate in the protocol which we now discuss (further details of these responsibilities can be found in the javadoc for the participant annotations).

The application developer indicates which method to use for a compensating activity by marking it with the `@Compensate` annotation. Whenever the associated resource is invoked in the context of an LRA, the method corresponding to this `@Compensate` method MUST be enlisted with the LRA: enlistment means that if the LRA is subsequently cancelled then the compensation method MUST

be invoked.

The specification does not mandate when this method is invoked but it does guarantee that it will eventually be called (this is the precise meaning of the term `eventual consistency` as used in this specification). Under failure conditions the system will keep retrying until it is certain that the participant has been successfully notified.

The LRA model supports both synchronous and asynchronous programming models. For example, if a compensating activity is brief then the synchronous model may be appropriate as the following code snippet shows. The snippet also shows how the compensating code can check for a nested context (by reading the injected `LRA_HTTP_PARENT_CONTEXT_HEADER` header):

```
    @Compensate // marks this method as the callback to use if the LRA cancels
    @Path("/compensate")
    @PUT // the implementation will only invoke the callback using HTTP PUT
    public Response compensateForAPreviousAction(
               @HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
               @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI parentLRA) {

        if (parentLRA != null)
            // A nested context is being cancelled.
            //
            // Code which is sensitive to executing with a nested context goes here.
            // Nested contexts are explained in detail in a later section.
            //
            // Generally only the @Complete handler will be sensitive to executing
            // with a nested context (because Compensated is a final state in the
            // LRA participant model whereas the Completed state is contingent upon
            // the parent completing).
        }

        // perform business logic which compensates for any previous work performed
        // in the context of the LRA
        getActivity(lraId).setCompensated();

        // If the compensation was successful then the participant can safely clean
        // up. But if the participant failed to compensate is must remember that it
        // failed until explicitly told it may forget about the LRA (via the forget
        // method below).
        if (compensationWasSuccessful) {
            return Response.ok().build();
        } else {
            // return one of the other status codes
            ...
        }
    }

    @Complete // marks this method as the callback to use if the LRA closes
    @Path("/complete")
    @PUT
```

```
    public Response cleanupAfterAPreviousAction(
                @HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
                @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI parentLRA) {

        // business logic for handling the closing LRA (provided by the business
developer)
        if (parentLRA != null)
            // A nested context is being closed.
            //
            // Code which is sensitive to executing with a nested context goes here.
            //
            // Note that if a nested context is closed but the parent context is later
            // cancelled then any work performed when the nested context was active
            // must still be capable of being compensated for. Therefore,
            // even though no more work will be executed with the nested context, the
            // business logic must remember how to compensate for any work it did do
            // (until it is explicitly told to forget about the nested context via the
            // forget method below).
            getActivity(lraId).setProvisionallyCompleted();
        } else {
            getActivity(lraId).setFullyCompleted();
        }

        if (completionWasSuccessful) {
            return Response.ok().build();
        } else {
            // return one of the other status codes
            ...
        }
    }

    @Forget
    @Path("/forget")
    @DELETE
    public Response forget(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,) {
                           @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI parentLRA)
{

        if (parentLRA != null) {
            // Code which is sensitive to executing with a nested context goes here.
            //
            // As mentioned in the comment in the @Complete method
(cleanupAfterAPreviousAction)
            // the participant may have saved some information about this LRA and this
            // forget callback is the opportunity for the participant to finally clean
up.
            // Note that if instead the participant had compensated (in the method
            // compensateForAPreviousAction) then it should already have cleaned up
after it
            // successfully compensated (for the actions it performed in the context
of the LRA).
```

```
            // mark the activity as fully complete (note that if the participant
compensated
            // then this forget method would not be invoked).
            if (provisionallyComplete) {
                getActivity(lraId).setFullyCompleted();
                ...
            }
            ...
        }
        ...
    }
```

On the other hand, the compensating logic may involve concerted activities, perhaps even compensating in the context of another LRA. In this case, the protocol accommodates a more decoupled mode of operation - the following example shows how a compensating activity can be started in the background:

```
    @Compensate
    @Path("/compensate")
    @PUT
    public Response compensatePreviousAction(
            @HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
            @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI parentLRA) {

        // business logic for handling the LRA (provided by the business developer)
        if (parentLRA != null) { // is the context nested
            // code which is sensitive to executing with a nested context goes here
        }

        ActivityClient client = getActivity(lraId).getResourceForCompensation();
        String backgroundActivity = client.compensate(lraId);
        ...
        return Response.accepted().build();
    }
```

Here the business logic reports that the compensation is in progress by returning the `202 Accepted` HTTP status code. Of course the system must still guarantee atomic outcomes, so the participant is responsible for reporting when it has finished compensating: it may do this by allowing the compensation method to be called multiple times in the context of the same LRA until the final state is known. But if the method `compensatePreviousAction` should not be called a second time (i.e., it is not idempotent) then the participant has the option of reporting its progress using the `@Status` annotation:

```
    @Status
    @Path("/status")
    @GET
    public Response status(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
                           @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI parentLRA)
{
        if (parentLRA != null) { // is the context nested
            // code which is sensitive to executing with a nested context goes here
        }

        if (isFinished(lraId))  // Business logic (provided by the business developer)
            return Response.ok().entity(ParticipantStatus.Compensated).build();
        else
            return Response.ok().entity(ParticipantStatus.Compensating).build();
    }
```

Notice that in this code example the participant is reporting progress using the appropriate `ParticipantStatus` enum according to the the participant state model.

But what if the business logic is unable to compensate for a previous action? In this case, the participant must remember that it was unable to compensate by reporting `FailedToCompensate` either via the compensate method using the `409 Conflict` status code, for example:

```
    @Compensate
    @Path("/compensate")
    @PUT
    public Response compensatePreviousAction(
                        @HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
                        @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI parentLRA) {
        if (parentLRA != null) { // is the context nested
            // code which is sensitive to executing with a nested context goes here
        }

        if (isFailed(lraId)) // Business logic (provided by the business developer)
              return Response.status(Response.Status.CONFLICT).build();
        ...
    }
```

or it can report the failure via the `@Status` method:

```
    @Status
    @Path("/status")
    @GET
    public Response status(
                        @HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
                        @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI parentLRA) {
        if (parentLRA != null) { // is the context nested
            // code which is sensitive to executing with a nested context goes here
        }
        if (isFailed(lraId)) // Business logic (provided by the business developer)
            return Response.ok().entity(ParticipantStatus.FailedToCompensate).build();
        ...
    }
```

In the successful case the participant SHOULD clean up any resources it allocated in the context of the LRA (but note that if the LRA is nested the participant may need to delay fully cleaning up until it knows what the parent decided - please refer to the section which describes nested contexts for more detail). Any requests (including the current one) made in the context of the same LRA MAY return a `410 Gone` status code.

In the failure case, the participant is responsible for remembering that it failed until it is explicitly told that it can clean up via the `@Forget` method:

```
    @Forget
    @Path("/forget")
    @DELETE
    public Response forget(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
                           @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI parentLRA)
{
        if (parentLRA != null) { // is the context nested
            // code which is sensitive to executing with a nested context goes here
        }
        // Business logic (provided by the business developer)
        if (isFailed(lraId)) {
            cleanupResources(lraId);
        }

        return Response.ok().build();
    }
```

The resource class MAY also contain a method marked with the `@Complete` annotation. If such a method is present then the method MUST be invoked when the associated LRA is closed (the participant can determine whether or not the closing context is nested by injecting the `LRA_HTTP_PARENT_CONTEXT_HEADER` header as described in the section about nested contexts). Again, the specification does not MANDATE when the method is called, just that it will eventually be invoked. Typically, the resource would use this call to perform any clean up actions. The method is optional since such clean up actions may not be necessary. For example, consider a system that just tracks hotel reservations and has operations for booking a room or cancelling the reservation

(`@Compensate`). Since this system is passive, once a room is booked, it does not make any difference if the LRA is completed or not: the room will be unavailable for others. If it receives a call to `@Compensate` then it will free the room. But it won't do anything on `@Complete`.

In the case of a top level LRA, if the participant successfully compensates or completes then it may forget about the LRA. In the case of a nested LRA, the participant may or may not need to wait (depending upon the needs of the business logic) to see what the outcome of the top level LRA is. If the participant cannot compensate it should remember that it is still associated with the LRA in which case, if it is a JAX-RS method, it MUST report the status of the association using the appropriate JAX-RS response code and if it is a non JAX-RS based participant then it MUST report the status of the association using one of the values in the `ParticipantStatus` enum according to the the participant state model.

If the compensation and completion methods are not idempotent then there MUST be a method annotated with `@Status` which reports the status. Otherwise, the compensation and completion methods should return the status. If the participant is no longer associated with the LRA (because it has successfully compensated/completed) it MAY return the `410 Gone` HTTP status code from any of these methods. If it knows it will never be able to compensate or complete then it MUST remember the fact until explicitly told that it can clean up by providing a method annotated with `@Forget` (the requirement is marked MUST because message delivery is not guaranteed in a distributed system).

If there is no `@Status` then the `@Compensate` or `@Complete` methods will continue to be invoked until the implementation knows it has the final status.

If the `@Compensate` or `@Complete` annotation is present on multiple methods then an arbitrary one is chosen.

The javadoc for the Compensate annotation provides more details about this annotation.

Similarly, the javadoc for the Complete annotation provides details about the `@Complete` annotation.

Participant marker annotations method signatures

The participant marker annotations are annotations that allow users to mark a method for the execution by the LRA implementation according to the the participant state model. These annotations are:

- `@Compensate` — a method to be executed when the LRA is cancelled
- `@Complete` — a method to be executed when the LRA is closed
- `@Status` — a method that allow user to state status of the participant with regards to a particular LRA
- `@Forget` — a method to be executed when the LRA allows participant to clear all associated information

There is also a listener marker annotation:

- `@AfterLRA` — a method that will be reliably invoked when the LRA enters one of the final states

This specification differentiates two types of participant method definitions — methods associated

with the JAX-RS resource method or the methods which are not bound to JAX-RS.

JAX-RS methods

The following table presents expectations that are placed on individual annotations when associated with JAX-RS resource methods:

Annotation	Required HTTP method	Expected status codes	Response
`@Compensate`	PUT	200, 202, 409, 410	see javadoc
`@Complete`	PUT	200, 202, 409, 410	see javadoc
`@Status`	GET	200, 202, 410	ParticipantStatus
`@Forget`	DELETE	200, 410	no expectations
`@AfterLRA`	PUT	200	no expectations

Please refer to the javadoc for each annotation for the description of the conditions under which the various JAX-RS response codes are returned.

Returning the status code 410, when appropriate - see the above table, has the same effect as status 200. The participant indicates with this response that it is no longer aware of the LRA identification but the implementation MUST assume that all required actions are performed, which is equivalent with return status 200, and that the participant already forgot about it (participant is allowed to forget about a LRA identification when completely handled)

If the method annotated with `@AfterLRA` returns an unexpected HTTP status or never reaches the caller then the implementation MUST invoke the same method again.

If the annotated method returns an unexpected HTTP status code, the implementation MAY invoke the same method again with the following caveat: if there is no `@Status` method and the implementation receives an unexpected response code from either of the `@Compensate` or `@Complete` invocations then it MUST reinvoke the method. [Note that this caveat applies to the situation where the response is lost since the caller will not see the correct code].

Users are allowed to reuse existing JAX-RS endpoints for participant method definitions. In this case, the LRA implementation MUST ensure that invoking these methods outside of the implementation of the LRA specification will not influence any running LRA.

Specifically, developers should NEVER call any JAX-RS endpoint for participant callback methods (`@Compensate`, `@Complete`, `@Status`, `@Forget`, and `@AfterLRA`) where they add a header defined by the Java constant `LRA_HTTP_RECOVERY_HEADER`. This way, a developer can distinguish if the call is made by an end-user or the implementation and make sure that it will not influence the participant of the LRA when called directly.

Non-JAX-RS participant methods

When the participant annotations are applied to the non-JAX-RS resource methods they MUST adhere to these predefined signatures:

- **Return type**:
  - `void`: successful execution is mapped to `Compensated` or `Completed` participant statuses, error execution is handled by exceptions thrown in the participant method
    - not applicable for `@Status` participant methods
  - ParticipantStatus
  - `jakarta.ws.rs.core.Response`: handled similarly as for JAX-RS participant methods
  - `java.util.concurrent.CompletionStage`: with the parameter of any of the previously defined types
- **Arguments**: up to 2 arguments of types in this order:
  - `java.net.URI`: representing current LRA context identification
  - `java.net.URI`: representing potential parent LRA context identification

Declaring more than two arguments, different types of arguments or different return type for any non-JAX-RS method annotated with the participant marker annotation MUST result in the prohibition of the successful application startup (e.g., through the startup time runtime exception).

Please note that both arguments are optional but the order is required. This means that if only one argument is provided this argument will contain the value of the current active LRA context (not the parent LRA context in case of nested LRA).

Examples of valid signatures:

```
@Compensate
public void compensate(URI lraId, URI parentId)

@Complete
public Response complete(URI lraId)

@Status
public CompletionStage<ParticipantStatus> status(URI lraId)
```

Examples of invalid signatures:

```
@Compensate
public void compensate(String lraId, String parentId) // invalid types of arguments

@Compensate
public String compensate(URI lraId) // invalid return type

@Forget
public void forget(URI lraId, URI parentId, String additional) // too many arguments
```

If any of the described methods throws an exception, we distinguish two cases depending on the exception type:

- `WebApplicationException` — the exception is mapped to the HTTP response it carries and then handled as defined in the section JAX-RS participant methods
- any other exception
  - @Compensate and @Complete - results into `FailedToCompensate` or `FailedToComplete` participant states
  - @Status and @Forget - as the participant may have already compensated (or completed) or may in the process of compensation (completion) the exception in these methods should result into failure condition (in JAX-RS this condition is represented by 500 return HTTP status code) which individual interpretation is left further unspecified.

In case the implementation of this specification exposes non-JAX-RS participant methods to be able to call them externally (e.g., the HTTP proxy) then it MUST protect every exposed method from unauthorized access. The specific security details are not specified.

Non-JAX-RS afterLRA method

A method annotated with `@AfterLRA` that is not a JAX-RS resource method MUST accept two arguments of type `URI` and `LRAStatus`, in that order. The first parameter holds an LRA context and the second parameter holds the final status of the LRA. If the signature does not conform to this requirement then the implementation MUST prohibit the successful startup of the application (e.g., through the startup time runtime exception).

An example of a valid signature is:

```
@AfterLRA
public void onLRAEnd(URI lraId, LRAStatus status)
```

Eventual compensations

If a resource cannot perform a compensation activity immediately the `@Compensate` method MUST report that the activity is still in progress using one of the following options:

- Return a `CompletionStage` or mark the method as asynchronous (using the `jakarta.ws.rs.container.Suspended` annotation). The future must report the final status when the stage completes (if it delivers an intermediate state then the implementation MUST use the `@Status` method if it exists, and if there is no such method it will reinvoke the `@Compensate` method). Please refer to the section about reactive support for more details.
- A JAX-RS method can return a `202 Accepted` HTTP status code. If there is no `@Status` method then the response MAY provide a status URL in the `HTTP Location` header field so that the implementation can discover the final outcome. This URL, if present, MUST obey the requirements specified in the javadoc for the Status annotation. If the developer has not provided an `@Status` method nor a status URL then the implementation MUST reinvoke the `@Compensate` method (i.e., it MUST be idempotent).
- A non JAX-RS method can return `ParticipantStatus.Compensating`.

The `@Status` method, if present, MUST report the progress of the compensation.

Similarly, if the resource cannot perform a completion activity immediately.

Nesting LRAs

An activity can be scoped within an existing LRA using the `@LRA.Type.NESTED` annotation element value. Invoking a method marked with this annotation will start a new LRA whose behaviour is as follows:

1. A nested LRA can close or cancel independently of its parent.
2. A nested LRA which has closed must retain the ability to cancel the effects if the parent cancels. This requirement must be enforced by participants. If the participant has a Forget method then it MUST be invoked if the parent LRA is closed. The Forget method is described in the section Forgetting an LRA.
3. If a nested LRA cancels then all of its children must cancel (even if they closed - see 2).
4. If a nested LRA closes then it, and all of its children, must close (but retain the ability to later compensate - see 2).

The javadoc for the LRA annotation discusses this element in much more detail (look for the javadoc for the `NESTED` enum value of the `LRA.Type` element).

The concept of nested transaction contexts is not new and is present in a number of industry standards and research initiatives. Some notable examples include OASIS Web Services [2], extensions to the concept of Sagas [3] and the OMG's Additional Structuring Mechanisms footnote:[Additional Structuring Mechanisms for the OTS Specification, September 2000, https://www.omg.org/spec/OTS/About-OTS/

In the following example, the `bookFlight` method supports the presence of an incoming LRA context and if there is one present then (because of the `NESTED` annotation element) it books a flight in a new LRA nested inside the incoming context. Since the context is nested it can be cancelled independently of the parent LRA but can only be provisionally closed (contingent upon the parent closing).

On the other hand, if there is no incoming context present then (because of the NESTED annotation element) a new top level LRA is started.

In either case a new LRA is started which will be terminated when the bookFlight method returns.

```
    @Inject
    private FlightService service;

    @LRA(value = LRA.Type.NESTED, end = true)
    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Booking bookFlight(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
                              @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI
parentLRA,
                              @QueryParam("flightNumber") String flightNumber) {
        if (parentLRA == null) {
            // code that should execute in the context of a top level LRA
        } else {
            // code which should execute in the context of a nested LRA
        }

        // business logic for handling the LRA (provided by the business developer)
        return service.book(lraId, flightNumber);
    }
```

Note that the mechanics of cancelling nested and top level LRAs is the same.

Timing out LRAs

The business logic may wish to control how long an LRA should remain active before it becomes eligible for automatic cancellation by providing values for the `timeLimit` and `timeUnit` element of the `@LRA` annotation. For example, to indicate that an LRA should automatically cancel after 100 milliseconds:

```
  @LRA(value = LRA.Type.REQUIRED, timeLimit = 100, timeUnit = ChronoUnit.MILLIS)
  @Path("/doitASAP")
  @GET
  @Produces(MediaType.APPLICATION_JSON)
  public Response theClockIsTicking(
      @HeaderParam(LRA.LRA_HTTP_CONTEXT_HEADER) URI lraId) {...}
```

Furthermore, the ability to compensate or complete may be transient capabilities of a service so participants can also be timed out. When the timeout is reached, the LRA moves to the state `Cancelling` and all registered participants receive a `Compensate` call. The same actions are performed as in the case where the LRA is cancelled.

When a participant joins an existing transaction using the `@LRA(value = ⋯)` annotation, the timeout of the already active Long Running Action can be influenced by the `timeLimit` and `timeUnit` defined on this annotation. The implementation should calculate the moment in time the LRAs would become eligible for cancellation based on the time the request enters the JAX-RS method and the timeout information found in the `@LRA` annotation. When this moment is earlier than the moment calculated for the LRA at that time, this new moment becomes the timeout moment for the LRA. So when multiple participants define a timeout period, the earliest one will trigger the cancellation of

the LRA.

Since the specification targets microservices running in different JVM's there are potential issues around how timing constraints are handled:

A timeout is a duration which results in some calculated end time. If different systems use different timezones or clock granularity then there is potential for different parts of the system to have different views of the current time. Therefore, implementations must be careful to avoid using absolute times and be aware that absolute synchronisation between clocks is not possible, and therefore, where possible, rely on a duration instead. If an absolute time is required then the recommendation is to use UTC which is a universal time standard to which all timezones can be canonically converted. This will provide some degree of alignment between clocks (where alignment means that clocks are only approximately globally synchronised).

Potential reasons for requiring absolute time values, as opposed to a duration, include:

1. When a failed server is restarted the implementation [of this specification] needs to calculate how long is left for timed LRAs to become eligible for cancellation based on how much time elapsed before the server failed and how much time elapsed before the failed server was restarted. Such calculations can only be made if the absolute time of expiry of the LRA is known.
2. In a centralised coordinator based system, if another server takes control of the LRA then the absolute time would be required.
3. One could envisage a non centralised implementation that relies on absolute time values to determine when LRAs may be cancelled. In such an approach it is even more important that different clocks are aligned. Failure to align clocks in such a system would result in higher failure rates for LRAs.

If there are edge cases around timing of actions or if local clocks cannot be relied upon then there are risks that different parts of the system may perform conflicting actions. The implementation must strive to minimise these risks but if they do occur then it MUST detect the conflict and force the LRA into one of the failure states and it MUST report the conflict (the actual mechanism by which it does so is unspecified).

Also note that time limits are never absolute with respect to when LRAs are actually cancelled: they are hints to the system that if the time limit is breached then compensation may not be possible and an implementation should move the state of the LRA to cancelling. It is also possible for an implementation to miss a deadline (for whatever reason) and some other part of the system cancels the LRA before it notices. In this scenario there would be a potential conflict since one of the participants is no longer able to compensate. But as mentioned in the introduction to this specification, failure states are permissible but "they must be logged or flagged for the administrator and manual intervention may be necessary to restore an application's consistency".

Leaving an LRA

If a resource method annotated with `@Leave` is invoked in the context of an LRA and if the bean class has registered a participant with the active LRA, it will be removed from the LRA just before the bean method is entered (and will not be asked to complete or compensate when the LRA is

subsequently ended). Even though the method runs without an LRA context, the implementation MUST still make the context available via a JAX-RS header.

Please refer to the javadoc for the Leave annotation for greater detail. Notice that only the `LRA_HTTP_CONTEXT_HEADER` is required to be present when the resource method is invoked, i.e., there is no requirement for the `LRA_HTTP_PARENT_CONTEXT_HEADER` to be injected.

An example of this annotation is shown next:

```
@Leave
@Path("/leave")
@PUT
public Response leaveWork(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId) {
  // clean up since this participant is no longer associated with the LRA
}
```

Reporting the status of a participant

This specification supports distributed communications amongst services and due to the unreliable nature of networks messages/requests can be lost, delayed, duplicated, etc. and the implementation component responsible for invoking participant completion/compensation logic may lose track of the status of a participant. In this case, ideally it would just resend the completion or compensation notification but if the participant does not support idempotency then it MUST be able to report its status by annotating one of the methods with the `@Status` annotation which should report the status according to the participant state model by returning one of the ParticipantStatus enum values (github link).

If the participant has already responded successfully to an `@Compensate` or `@Complete` method invocation then it MAY report `410 Gone` HTTP status code or in the case of non-JAX-RS method returning ParticipantStatus `null`.

Let's provide some intuition on this matter with the following example.

Let's have an LRA participant containing methods annotated with `@Compensate` and `@Status`. The LRA is cancelled and the LRA implementation tries to invoke the participant's `@Compensate` method. At that time a network failure happens. The LRA implementation receives a network error as the result of the call. Now it cannot be sure if the `@Compensate` call passed through and participant has already compensated the work and only the response was lost or if the `@Compensate` call has not reached the participant at all. As the participant defines the `@Status` method the LRA implementation MUST invoke it to find the participant's true status (invocation can be processed several times until it succeeds). The participant's `@Status` method is expected to indicate whether the `@Compensate` was successfully processed.

- If the participant processed the `@Compensate` call it may has already forgotten about the LRA identification. Then for JAX-RS participant, the response is `410 Gone`. For non-JAX-RS participant, the method returns `null`. The implementation MUST assume that all required actions are performed and the implementation MUST NOT repeat the invocation.
- If the participant compensated successfully but it has not erased the notion of the LRA identity

yet then it may return ParticipantStatus of `Compensated`. For a JAX-RS participant, it is a body of the response with the return code `200`. For the non-JAX-RS participant case, consult the section Non-JAX-RS participant methods (e.g., one option is to provide `Compensated` as the method return value). The implementation MUST NOT repeat the invocation of the `@Compensate` method.

- If the `@Compensate` method was invoked initially but the compensation logic failed, the participant status is `FailedToCompensate`. The LRA implementation then has to log sufficient information in such situation. For the JAX-RS participant, the response is `200 OK` with participant status sent in the response body. For the non-JAX-RS participant case, consult section Non-JAX-RS participant methods about return values. The implementation MUST NOT repeat the invocation of the `@Compensate` method.
- If the participant has not yet compensated (i.e., the initial invocation of `@Compensate` did not go through) then the `ParticipantStatus` will indicate the participant status with `Active`. The implementation MUST repeat the invocation of the `@Compensate` method.

Forgetting an LRA

If a participant is unable to complete or compensate immediately (i.e., it has indicated that the request has been accepted and is in progress) or because of a failure (i.e., will never be able to finish) then it must remember the fact (by reporting its status via the `@Status` method) until explicitly told that it can clean up using this `@Forget` annotation.

This requirement ensures that the implementation will be able to guarantee the expectations of the LRA protocol under various failure conditions. Only when the implementation is certain that participant has finished will it tell it that it is okay to release any resources it associated with the LRA.

Additionally, if a participant is enlisted in a nested LRA, then it can ask to be notified when the parent LRA closes using this `@Forget` annotation. This feature is useful since a nested LRA can be closed independently of its parent but it must retain the ability to compensate until the parent has finished. Typically, a participant would perform clean up actions in this method.

Reactive Support

Implementations are expected to operate correctly when services use the asynchronous and reactive features provided by JAX-RS. In particular, the implementation has no control over which thread the service logic uses to do its work, therefore asynchronous operations may complete on any of:

- the caller thread;
- a managed thread;
- an unmanaged thread.

Furthermore, both the service writer and implementation should be aware that the actual thread used to perform the operation may be used by several requests running concurrently.

It has already been noted that participant completion and compensation callbacks can execute asynchronously but the same must also be true for the business methods that execute in the context of an LRA. It is the responsibility of the implementation to ensure that JAX-RS asynchronous

features continue to behave in the presence of these LRAs. The following example shows a resource invocation that runs in the context of a long running action and uses a Java 8 completion stage to return an asynchronous response:

```
    @LRA(value = LRA.Type.REQUIRED,  // the method must run with an LRA
            end = true, // the LRA must end when the method completes
            cancelOnFamily = Response.Status.Family.SERVER_ERROR, // cancel LRA on any
5xx code
            cancelOn = NOT_FOUND) // cancel LRA on 404
    @Path("async-path")
    @POST
    public CompletionStage<Response> asyncInvocationWithLRA(
               @HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
               @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI parentLRA) {
        return CompletableFuture.supplyAsync(
                () -> {
                    try {
                        // a long running operation with lraId
                        if (parentLRA != null) { // is the context nested
                            // code which is sensitive to executing with a nested
context goes here
                        }
                        ...
                        return Response.ok().entity(lraId).build(); // close the LRA
                    } catch (BusinessException ex) {
                        return Response.status(NOT_FOUND).entity(lraId).build(); //
cancel the LRA
                    }
                },
                getExcecutorService()
        );
    }
```

With completion stages it is also possible to complete exceptionally. The following example should run business logic asynchronously in the context of an LRA but the LRA should be cancelled: forcing any registered participant compensation handlers to run:

```
    @LRA(value = LRA.Type.REQUIRED, // the method must run with an LRA
         end = true, // the LRA must end when the method completes
         cancelOn = {Response.Status.NOT_FOUND}) // cancel on a 404 code
    @Path("completion-stage-exceptionally-lra")
    @POST
    public CompletionStage<Response> asyncInvocationWithException(
                @HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
                @HeaderParam(LRA_HTTP_PARENT_CONTEXT_HEADER) URI parentLRA) {

        final CompletableFuture<Response> response = new CompletableFuture<>();

        executorService.submit(() -> {
            if (parentLRA != null) { // is the context nested
                // code which is sensitive to executing with a nested context goes
here
            }
            // execute long running business activity finishing with a NOT_FOUND error
            // which causes the LRA to cancel
            response.completeExceptionally(
                    new WebApplicationException(
                        Response.status(Response.Status.NOT_FOUND).entity(lraId).
build()));
        });

        return response;
    }
```

In addition to the use of completion stages, a resource method may also produce asynchronous responses by injecting a JAX-RS `AsyncResponse` parameter using the JAX-RS `@Suspended` annotation:

```
    @LRA(value = LRA.Type.REQUIRES_NEW, // the method must run with an LRA
         end = true) // the LRA must end when the method completes
    @Path("asyncresponse-lra")
    @POST
    public void asyncResponseLRA(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId,
                         final @Suspended AsyncResponse ar) {
        executorService.submit(() -> {
            // Notice that since a new LRA was created for running this
            // method (`LRA.Type.REQUIRES_NEW`) the context cannot be
            // nested (in contrast to the previous example which did have
            // a nested context).
            //
            // execute long running business activity and resume when done
            ar.resume(Response.ok().entity(lraId).build());
        });
    }
```

The previous use cases required a Java executor service, but it is also possible to use other

asynchronous API's. The next snippet shows an LRA consuming an async API using an AWS S3 client:

```
    <dependency>
        <groupId>software.amazon.awssdk</groupId>
        <artifactId>s3</artifactId>
        <version>2.0.0-preview-5</version>
    </dependency>
```

```
    S3AsyncClient client = S3AsyncClient.create();

    @LRA(value = LRA.Type.REQUIRES_NEW, end = true)
    @Path("completion-stage-lra")
    @POST
    public CompletionStage<PutObjectResponse> asyncInvocationWithLRA(
            @HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId) {

        return client.putObject(
                PutObjectRequest.builder()
                        .bucket("aws-bucket")
                        .key("keyfile.in")
                        .build(),
                AsyncRequestProvider.fromFile(Paths.get("myfile.in"))
        ).whenComplete((r, e) -> {
            if (e == null) {
                Response.ok().entity(lraId).build();
            } else {
                Response.status(INTERNAL_SERVER_ERROR).entity(lraId).build());
            }
        });
    }
```

Finally, here is an example of how to run a non JAX-RS compensation asynchronously:

```
    @Compensate
    public CompletionStage<ParticipantStatus> compensate(URI lraId, URI parentId) {
        // the compensation includes two long running operations:
        CompletableFuture<Void> memUpdate = CompletableFuture.runAsync((() -> {/* ...
*/}));
        CompletableFuture<Void> dbUpdate = CompletableFuture.runAsync((() -> {/* ...
*/}));

        CompletableFuture<Boolean> stage1 = memUpdate.handle((s, e) -> e == null);
        CompletableFuture<Boolean> stage2 = dbUpdate.handle((s, e) -> e == null);

        return stage1.thenCombine(stage2, (b1, b2) -> {
            if (b1 && b2) {
                // the memory and db updates finished successfully so report success
                return Compensated;
            }

            // otherwise report that there was a compensation failure
            return FailedToCompensate;
        });
    }
```

Recovery Requirements

This LRA specification provides guarantees of Atomicity, Consistency and Durability of work which places responsibilities on both spec implementers and application writers. Failure points include loss of contact with components managing the life cycle of LRAs and of participants. Application writers need to know how to associate work with an LRA context so that the correct work can be compensated for even after JVM or node crashes. Likewise, infrastructure components may become unavailable and state must survive system failures. The specification is not prescriptive about how an implementation achieves resiliency provided that it obeys the requirements of the specification as laid out in this document.

[1] Note that calling participants in reverse order does not guarantee that the compensation actions will be performed in strict sequential order since participants are allowed to indicate that the compensation is in progress and will complete at some future time. Furthermore a participant can indicate that it failed to compensate, or could be unavailable in which case it will be periodically retried (out of order).

[2] OASIS LRA: https://www.oasis-open.org/committees/document.php?document_id=12794

[3] Hector Garcia-Molina, Modeling long-running activities as nested sagas https://dl.acm.org/doi/10.5555/108798.108801

Release Notes for MicroProfile LRA 1.0

MicroProfile LRA Spec PDF MicroProfile LRA Spec HTML MicroProfile Propagation LRA Javadocs

Key features:

A transaction model which isn't full ACID:

- an activity reflects business interactions
- all scoped work must be compensatable
- activities are visible to other services
- when an activity ends all work is either accepted or all work is compensated
- the LRA model defines the triggers for when and where compensation actions are executed
- defines annotations for the safe/transactional execution of activities supporting long running activities involving loosely coupled processes

Supports:

- relaxion of atomicity (using nested transactions);
- locking is optional (⇒ loss of isolation);
- forward progress by allowing work to finish early, to provisionally perform subsets of work (nesting), time bounds, composition of activities

Provides CDI annotations:

Table 1. LRA Annotations

Annotation	Description	JAX-RS
@LRA	Controls the life cycle of an LRA	Yes
@AfterLRA	Notification that an LRA has finished	Yes/Optional

Table 2. Participant Annotations

Annotation	Description	JAX-RS
@Compensate	Indicates that the method should be invoked if the LRA is cancelled.	Optional
@Complete	Indicates that the method should be invoked if the LRA is closed.	Optional
@Leave	Indicates that this class is no longer interested in this LRA.	Yes

Annotation	Description	JAX-RS
@Status	When the annotated method is invoked it should report the status.	Optional
@Forget	The method may release any resources associated with the LRA	Optional

- reactive support:
  - CompletionStage
  - @Suspended AsyncResponse
  - HTTP 202 Accepted response code

To get started, add this dependency to your project:

```
<dependency>
    <groupId>org.eclipse.microprofile.lra</groupId>
    <artifactId>microprofile-lra-api</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
</dependency>
```

Create a JAX-RS business resource and annotate the methods that you would like to be included in a long running action using the @LRA annotation. Minimally you should define which business method should be run if the LRA is cancelled using the @Compensate annotation.

```
@Path("resource")
public class SimpleParticipant {
    @PUT
    @Path("action")
    @LRA(value = LRA.Type.REQUIRED)
    public Response businessOp(@HeaderParam(LRA_HTTP_RECOVERY_HEADER) URI recoveryId,
                               @HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId) {
        // perform some business action in the context of the LRA with id lraId
        return Response.ok().build();
    }

    @PUT
    @Path("compensate")
    @Compensate
    public Response compensateWork(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId) {
        // compensate for any actions that were performed in the context of the LRA
with id lraId

        return Response.ok().build();
    }
}
```

Appendix 1: Selected Javadoc API Descriptions

LRA Annotation

```
/*
 ******************************************************************************
 * Copyright (c) 2018-2021 Contributors to the Eclipse Foundation
 *
 * See the NOTICE file(s) distributed with this work for additional
 * information regarding copyright ownership.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * You may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 ******************************************************************************/

package org.eclipse.microprofile.lra.annotation.ws.rs;

import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.time.temporal.ChronoUnit;

import org.eclipse.microprofile.lra.annotation.AfterLRA;
import org.eclipse.microprofile.lra.annotation.Compensate;
import org.eclipse.microprofile.lra.annotation.Complete;
import org.eclipse.microprofile.lra.annotation.Forget;
import org.eclipse.microprofile.lra.annotation.LRAStatus;
import org.eclipse.microprofile.lra.annotation.Status;

import jakarta.ws.rs.HeaderParam;
import jakarta.ws.rs.core.Response;

/**
 * <p>
 * An annotation for controlling the lifecycle of Long Running Actions (LRAs).
 * </p>
```

```
 *
 * <p>
 * This annotation MUST be combined with either the {@link Compensate} or the {@link
AfterLRA} annotated methods in the
 * same bean class otherwise the deployment will be rejected.
 * </p>
 *
 * <p>
 * The annotation <b>SHOULD</b> be applied to JAX-RS annotated methods otherwise it
<b>MAY</b> have no effect. The
 * annotation determines whether or not the annotated method will run in the context
of an LRA and controls whether or
 * not:
 * </p>
 *
 * <ul>
 * <li>any incoming context should be suspended and if so if a new one should be
started</li>
 * <li>to start a new LRA</li>
 * <li>to end any LRA context when the method ends</li>
 * <li>to return an error status code without running the annotated method if there
should have been an LRA context
 * present on the method entry</li>
 * <li>to cancel the LRA context when the method returns particular HTTP status
codes</li>
 * </ul>
 *
 * Some of the above defined LRA operations are performed before the execution
proceeds to the LRA annotated business
 * method. If any of these LRA operations cannot complete the implementation MUST
return one of the defined HTTP status
 * codes which means that the business method is not invoked. For the definition of
the allowed status codes and
 * condition in which they are returned please see the specification document.
 *
 * <p>
 * Newly created LRAs are uniquely identified and the id is referred to as the LRA
context. The context is passed around
 * using a JAX-RS request/response header called {@value #LRA_HTTP_CONTEXT_HEADER}.
The implementation (of the LRA
 * specification) is expected to manage this context and the application developer is
expected to declaratively control
 * the creation, propagation and destruction of LRAs using this {@link LRA}
annotation. When a JAX-RS resource method is
 * invoked in the context of an LRA, any JAX-RS client requests that it performs will
carry the same header so that the
 * receiving resource knows that it is inside an LRA context. The behaviour may be
overridden by manually setting the
 * context header.
 * </p>
 *
```

```
 * <p>
 * If an LRA is propagated to a resource that is not annotated with any particular LRA
behavior, then the MicroProfile
 * Config value <code>mp.lra.propagation.active</code> determines how the LRA is
propagated. The default of this
 * parameter is <code>true</code> which means that the LRA will be propagated to
outgoing requests. For example, suppose
 * resource <code>A</code> starts an LRA and then performs a JAX-RS request to
resource <code>B</code> which does not
 * contain any LRA annotations. If resource <code>B</code> then performs a JAX-RS
request to a third service,
 * <code>C</code> say, which does contain LRA annotations then the LRA context started
at <code>A</code> must be
 * propagated to <code>C</code> (for example, if <code>C</code> uses an annotation to
join the LRA, then <code>C</code>
 * must be enlisted in the LRA that was started at <code>A</code>).
 * </p>
 *
 * <p>
 * Resource methods can access the LRA context id by inspecting the request headers
using standard JAX-RS mechanisms, ie
 * <code>&#64;Context</code> or by injecting it via the JAX-RS {@link HeaderParam}
annotation with value
 * {@value #LRA_HTTP_CONTEXT_HEADER}. This may be useful, for example, for associating
business work with an LRA.
 * </p>
 */
@Inherited
@Retention(value = RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface LRA {
    /**
     * When a JAX-RS invocation is made with an active LRA, it is made available via
an HTTP header field with the
     * following name. The value contains the LRA id associated with the HTTP
request/response and it is of type
     * {@link java.net.URI}.
     */
    String LRA_HTTP_CONTEXT_HEADER = "Long-Running-Action";

    /**
     * The header name holding the LRA context of an LRA that has finished - to be
used in conjunction with the
     * {@link AfterLRA} annotation.
     */
    String LRA_HTTP_ENDED_CONTEXT_HEADER = "Long-Running-Action-Ended";

    /**
     * When an implementation of the LRA specification invokes any of the participant
callbacks (namely
     * {@link Compensate}, {@link Complete}, {@link Status}, {@link Forget}, and
```

```
{@link AfterLRA}) in the context of a
     * nested LRA it must ensure that the parent LRA is made available via an HTTP
header field with the following name.
     * The value contains the parent LRA context associated with the HTTP
request/response and it is of type
     * {@link java.net.URI}.
     */
    String LRA_HTTP_PARENT_CONTEXT_HEADER = "Long-Running-Action-Parent";

    /**
     * the name of the HTTP header field that contains a recovery URI corresponding to
a participant enlistment in an
     * LRA. The value is of type {@link java.net.URI}.
     */
    String LRA_HTTP_RECOVERY_HEADER = "Long-Running-Action-Recovery";

    /**
     * <p>
     * The {@link Type} element of the LRA annotation indicates whether a resource
method is to be executed within the
     * context of an LRA.
     * </p>
     *
     * <p>
     * If the method is to run in the context of an LRA and the annotated class also
contains a method annotated with
     * {@link Compensate} then the resource will be enlisted with the LRA. Enlisting
with an LRA means that the resource
     * MUST be notified when the current LRA is later cancelled. The resource can also
receive notifications when the
     * LRA is closed if it additionally defines a method annotated with {@link
Complete}. The specification does not
     * mandate when these notifications are issued but it does guarantee that they
will eventually be sent. Under
     * failure conditions the system will keep retrying until it is certain that all
participants have been successfully
     * notified.
     * </p>
     *
     * <p>
     * If the method is to run in the context of an LRA and the annotated class also
contains a method annotated with
     * {@link AfterLRA} then the resource will also be notified of the final state of
the LRA.
     * </p>
     *
     * <p>
     * The element values {@link LRA.Type#REQUIRED} and {@link LRA.Type#REQUIRES_NEW}
can start new LRAs.
     * </p>
     *
```

```
     * <p>
     * If the method runs in the context of an LRA then the application can control
whether or not it is closed when the
     * method finishes using the {@link LRA#end()} element.
     * </p>
     *
     * <p>
     * When an LRA is present, its identifier is made available to the business logic
in the JAX-RS request and response
     * headers with the name {@value #LRA_HTTP_CONTEXT_HEADER} of type {@link
java.net.URI}.
     * </p>
     *
     * @return the type of behaviour expected when the annotated method is executed.
     */
    Type value() default Type.REQUIRED;

    enum Type {
        /**
         * <p>
         * If called outside an LRA context the invoked method will run with a new
context.
         * </p>
         *
         * <p>
         * If called inside an LRA context the invoked method will run with the same
context.
         * </p>
         */
        REQUIRED,

        /**
         * <p>
         * If called outside an LRA context the invoked method will run with a new
context.
         * </p>
         *
         * <p>
         * If called inside an LRA context the invoked method will run with a new
context. The original context is
         * ignored.
         * </p>
         */
        REQUIRES_NEW,

        /**
         * <p>
         * If called outside an LRA context the method is not executed and a <code>412
Precondition Failed</code> HTTP
         * status code is returned to the caller.
         * </p>
```

```
         *
         * <p>
         * If called inside a transaction context the resource method execution will
then continue within that context.
         * </p>
         */
        MANDATORY,

        /**
         * <p>
         * If called outside an LRA context the resource method execution must then
continue outside an LRA context.
         * </p>
         *
         * <p>
         * If called inside an LRA context the resource method execution must then
continue with the same LRA context.
         * </p>
         */
        SUPPORTS,

        /**
         * <p>
         * The resource method is executed without an LRA context.
         * </p>
         */
        NOT_SUPPORTED,

        /**
         * <p>
         * If called outside an LRA context, i.e., {@link #LRA_HTTP_CONTEXT_HEADER} is
not present, the resource method
         * execution must then continue outside an LRA context.
         * </p>
         *
         * <p>
         * If called inside an LRA context, i.e., {@link #LRA_HTTP_CONTEXT_HEADER} is
present referring to an active,
         * inactive or non-existent LRA, the method is not executed and a <code>412
Precondition Failed</code> HTTP
         * status code is returned to the caller.
         * </p>
         */
        NEVER,

        /**
         * <p>
         * An LRA (called the child) can be scoped within an existing LRA (called the
parent) using the NESTED element
         * value. A new LRA will be created even if there is already one present when
the method is invoked, i.e., these
```

```
         * LRAs will then either be top-level or nested automatically depending upon
the context within which they are
         * created. If invoked without a context, the new LRA will be top level. If
invoked with an LRA present, a new
         * nested LRA is started whose outcome depends upon whether or not the
enclosing LRA is closed or cancelled. The
         * id of the parent LRA MUST be present in the header with the name
         * {@value
org.eclipse.microprofile.lra.annotation.ws.rs.LRA#LRA_HTTP_PARENT_CONTEXT_HEADER} and
the value is of
         * type {@link java.net.URI}.
         * </p>
         *
         * <p>
         * A nested LRA is treated just like any other LRA with respect to participant
enlistment. When an invocation
         * results in the creation of a nested LRA that LRA becomes the "current
context" and any further operations
         * performed by the method will be executed with that context. The semantics
of nested LRAs follows previous
         * transactions models:
         * </p>
         * <ol>
         * <li>A nested LRA can close or cancel independently of its parent.
         * <li>A nested LRA which has closed must retain the ability to cancel the
effects if the the parent cancels.
         * This requirement must be enforced by participants.
         * <li>If a nested LRA cancels then all of its children must effectively
cancel (even if they have previously
         * been asked to close - see 2)
         * <li>If a nested LRA closes then it, and all of its children, must close
(but retain the ability to later
         * cancel - see 2).
         * </ol>
         * <p>
         * Downstream LRAs will only be part of this nesting hierarchy if the
downstream methods carry the NESTED
         * element, otherwise they are independent of the current nested LRA.
         * </p>
         *
         * <p>
         * The reason why the model does not allow a cancelled nested LRA to be closed
is because the business activity
         * has already been compensated for which means there is no longer any
outstanding work in need of completion.
         * </p>
         * <p>
         * On the other hand it does make sense to cancel the effects of a closed
nested LRA since the work has been
         * done and there is something that can be compensated for. In this case the
LRA method invocation is allowed to
```

```
         * proceed only if the participant is already enlisted with the LRA, otherwise
the method invocation is rejected
         * using a <code>412 Precondition Failed</code> HTTP status code. If the
method invocation causes the LRA to
         * close then the {@link Complete} annotated method, if present, <i>is not</i>
called again. If the LRA method
         * causes the LRA to cancel then the nested LRA is moved to the {@link
LRAStatus#Cancelling} state and the
         * {@link Compensate} callback will be invoked.
         * </p>
         * <p>
         * Therefore, as a consequence of requirement 2, any activities performed in
the context of a closed nested LRA
         * must remain compensatable until the top level parent LRA finishes. So if
the nested LRA is closed the
         * participants registered with it will be asked to complete, but if the top
level parent LRA is then told to
         * cancel the nested participants will be told to compensate. This implies
that the nested participants must be
         * aware that they are nested and the JAX-RS header with the name
         * {@value
org.eclipse.microprofile.lra.annotation.ws.rs.LRA#LRA_HTTP_PARENT_CONTEXT_HEADER} is
guaranteed to
         * hold the parent context whenever a nested LRA is being propagated.
         * </p>
         *
         * <p>
         * A participant which has completed can determine when the top level parent
has closed by providing a
         * {@link Forget} callback handler. When this method is called there is no
longer a requirement to maintain the
         * ability to reverse the effects of a closed nested LRA. This can be used by
the participant to clean up any
         * resources it used to implement this guarantee.
         * </p>
         *
         * <p>
         * Note that it is possible for the same resource to be registered with both
the parent and the child LRAs and
         * in this case it will be asked to complete or compensate twice, once with
the nested context and a second time
         * with the parent context. The order in which the two callbacks are invoked
is undefined.
         * </p>
         *
         * <p>
         * Note that the elements of the LRA annotation always apply to the LRA
context used to execute the annotated
         * method. Thus elements such as {@link #timeLimit()}, {@link #timeUnit()},
{@link #cancelOn()},
         * {@link #cancelOnFamily()} and {@link #end()} will always be applied to the
```

```
nested or top level LRA.
         */
        NESTED
    }

    /**
     * <p>
     * If the annotated method runs with an LRA context then this element determines
the period for which the LRA will
     * remain valid. When this period has elapsed the LRA becomes eligible for
cancellation. The units are specified in
     * the {@link LRA#timeUnit()} element. A value of zero indicates that the LRA will
always remain valid.
     * </p>
     *
     * <p>
     * Methods running with an active LRA context must be resilient to it being
cancelled while the method is still
     * executing.
     * </p>
     *
     * @return the period for which the LRA is guaranteed to run for before becoming
eligible for cancellation.
     */
    long timeLimit() default 0;

    /**
     * @return the unit of time that the {@link LRA#timeLimit()} element is measured
in.
     */
    ChronoUnit timeUnit() default ChronoUnit.SECONDS;

    /**
     * <p>
     * If the annotated method runs with an LRA context then this element determines
whether or not it is closed when
     * the method returns. If the element has the value {@literal true} then the LRA
will be ended and all participants
     * that have the {@link Complete} annotation MUST eventually be asked to complete.
If the element has the value
     * {@literal false} then the LRA will not be ended when the method finishes.
     * </p>
     *
     * <p>
     * If the <code>end</code> value is set to {@literal false} but the annotated
method finishes with a status code
     * that matches any of the values specified in the {@link #cancelOn()} or {@link
#cancelOnFamily()} elements then
     * the LRA will be cancelled. In other words, the {@link #cancelOn()} and {@link
#cancelOnFamily()} elements take
     * precedence over the <code>end</code> element.
```

```
     * </p>
     *
     * @return true if an LRA that was active when the method ran should be closed
when the method execution finishes.
     */
    boolean end() default true;

    /**
     * <p>
     * The cancelOnFamily element can be set to indicate which families of HTTP
response codes will cause the current
     * LRA to cancel. If the LRA has already been closed when the annotated method
returns then this element is silently
     * ignored, Cancelling an LRA means that all participants will eventually be asked
to compensate (by having their
     * {@link Compensate} annotated method invoked).
     * </p>
     *
     * <p>
     * If a JAX-RS method is annotated with this element and the method returns a
response code which matches any of the
     * specified families then the LRA will be cancelled. The method can return status
codes in a {@link Response} or
     * via a JAX-RS exception mappper.
     * </p>
     *
     * @return the {@link jakarta.ws.rs.core.Response.Status.Family} status families
that will cause cancellation of the
     *         LRA
     */
    Response.Status.Family[] cancelOnFamily() default {
            Response.Status.Family.CLIENT_ERROR, Response.Status.Family.SERVER_ERROR
    };

    /**
     * <p>
     * The cancelOn element can be set to indicate which HTTP response codes will
cause the current LRA to cancel. If
     * the LRA has already been closed when the annotated method returns then this
element is silently ignored,
     * Cancelling an LRA means that all participants will eventually be asked to
compensate (by having their
     * {@link Compensate} annotated method invoked).
     * </p>
     *
     * <p>
     * If a JAX-RS method is annotated with this element and the method returns a
response code which matches any of the
     * specified status codes then the LRA will be cancelled. The method can return
status codes in a {@link Response}
     * or via a JAX-RS exception mappper.
```

```
     * </p>
     *
     * @return the {@link jakarta.ws.rs.core.Response.Status} HTTP status codes that
will cause cancellation of the LRA
     */
    Response.Status[] cancelOn() default {};
}
```

Leave Annotation

```
/*
 ******************************************************************************
 * Copyright (c) 2018-2021 Contributors to the Eclipse Foundation
 *
 * See the NOTICE file(s) distributed with this work for additional
 * information regarding copyright ownership.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * You may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 ******************************************************************************/

package org.eclipse.microprofile.lra.annotation.ws.rs;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * <p>
 * If a resource method is annotated with <code>&#64;Leave</code> and is invoked in
the context of an LRA and if the
 * bean class has registered a participant with that LRA then it will be removed from
the LRA just before the bean
 * method is entered. The participant can forget about this LRA, in particular it will
not be asked to complete or
 * compensate when the LRA is subsequently ended. Even though the method runs without
an LRA context, the implementation
 * MUST still make the context available via a JAX-RS header and any outgoing JAX-RS
invocations performed by the method
 * will still carry the context that the participant has just left. Therefore the
business logic must be careful about
 * any JAX-RS invocations it makes in the body of the annotated method which may
result in other resources being
 * enlisted with the LRA.
 * </p>
 *
 * <p>
 * If the resource method (or class) is also annotated with <code>&#64;LRA</code> the
```

```
method will execute with the
 * context dictated by the <code>&#64;LRA</code> annotation. If this
<code>&#64;LRA</code> annotation results in the
 * creation of a new LRA then the participant will still be removed from the incoming
context and will be enlisted with
 * the new context (and the method will execute with this new context). Note that in
this case the context exposed in
 * the <code>&#64;LRA_HTTP_CONTEXT_HEADER</code> JAX-RS header will be set to the new
LRA (and not the original one),
 * i.e., the original context will not be available to the business logic.
 * </p>
 *
 * <p>
 * Also note that it is not possible to join or leave an LRA that has already been
asked to cancel or close since that
 * would conflict with the the participant state model as defined in the LRA
specification.
 * </p>
 *
 * <p>
 * Leaving a particular LRA has no effect on any other LRA - i.e., the same resource
can be enlisted with many different
 * LRAs and leaving one particular LRA will not affect its participation in any of the
other LRAs it has joined.
 * </p>
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Leave {
}
```

Compensate Annotation

```
/*
 ******************************************************************************
 * Copyright (c) 2018-2021 Contributors to the Eclipse Foundation
 *
 * See the NOTICE file(s) distributed with this work for additional
 * information regarding copyright ownership.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * You may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 ******************************************************************************/

package org.eclipse.microprofile.lra.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.eclipse.microprofile.lra.annotation.ws.rs.LRA;

/**
 * <p>
 * If a resource method executes in the context of an LRA and if the containing class
has a method annotated with
 * <code>&#64;Compensate</code> then this method will be invoked if the LRA is
cancelled. The resource should attempt to
 * compensate for any actions it performed in the context of the LRA. If the
annotation is present on more than one
 * method then an arbitrary one will be chosen. The LRA specification makes no
guarantees about when Compensate method
 * will be invoked, just that it will eventually be called.
 * </p>
 *
 * <p>
 * In the case where the ability to compensate the Long Running Action is time
bounded, you can limit the lifespan of
 * the Long Running action by providing values for the {@link LRA#timeLimit()} and
{@link LRA#timeUnit()} attributes.
 * When the time limit is reached the LRA becomes eligible for automatic cancellation.
```

```
 * </p>
 *
 * <p>
 * If the annotation is applied to a JAX-RS resource method then the request method
MUST be {@link jakarta.ws.rs.PUT}.
 * The LRA context of the currently running LRA can be obtained by inspecting the
incoming JAX-RS headers. If this LRA
 * is nested then the parent LRA MUST be present in the header with the name {@link
LRA#LRA_HTTP_PARENT_CONTEXT_HEADER}
 * and the header value will be of type {@link java.net.URI}.
 * </p>
 *
 * <p>
 * If the annotated method is not a JAX-RS resource method then the LRA context of the
currently running LRA and its
 * parent LRA (if it is nested) can be obtained by adhering to predefined method
signatures as defined in the LRA
 * specification document. For example,
 * </p>
 *
 * <pre>
 *     <code>
 *        &#64;Compensate
 *        public void compensate(URI lraId, URI parentId) { ...}
 *     </code>
 * </pre>
 *
 * <p>
 * would be a valid compensation method declaration. If an invalid signature is
detected, the implementation of this
 * specification MUST prohibit successful startup of the application (e.g. with a
runtime exception).
 * </p>
 *
 * <p>
 * If the participant cannot compensate immediately then it must report that the
compensation request was received and
 * that the compensation is in progress by either returning a future (such as
 * {@link java.util.concurrent.CompletionStage}) which will eventually report one of
the final states, or a
 * <code>202 Accepted</code> JAX-RS response code or, in the case of non JAX-RS
resource methods, by returning
 * {@link ParticipantStatus#Compensating} (see the specification document for more
details).
 * </p>
 *
 * <p>
 * Note that according to the state model defined by {@link LRAStatus}, it is not
possible to receive compensation
 * notifications after an LRA has been asked to cancel. Therefore combining this
annotation with an
```

```
 * <code>&#64;LRA</code> annotation that does not start a new LRA will result in a
<code>412 PreCondition Failed</code>
 * JAX-RS response code. On the other hand, combining it with an <code>&#64;LRA</code>
annotation that begins a new LRA
 * can in certain use cases make sense, but in this case, the LRA that this method is
being asked to compensate for will
 * be unavailable.
 * </p>
 *
 * <p>
 * If the method is a JAX-RS resource method (or is a non JAX-RS method annotated with
<code>&#64;Compensate</code> with
 * return type <code>jakarta.ws.rs.core.Response</code>) then the following are the
only valid response codes:
 * </p>
 *
 * <table border="0" cellpadding="3" cellspacing="0" summary="Valid JAX-RS
compensation response codes">
 * <caption> <span>JAX-RS Compensation Response Codes</span> <span> </span>
</caption>
 * <tr>
 * <th scope="col">Code</th>
 * <th scope="col">Response Body</th>
 * <th scope="col">Meaning</th>
 * </tr>
 * <tr>
 * <td scope="row">200</td>
 * <td scope="row">Empty</td>
 * <td scope="row">The resource has successfully compensated</td>
 * </tr>
 * <tr>
 * <td scope="row">202</td>
 * <td scope="row">Empty</td>
 * <td scope="row">The resource is still attempting compensation</td>
 * </tr>
 * <tr>
 * <td scope="row">409</td>
 * <td scope="row">{@link ParticipantStatus} enum value</td>
 * <td scope="row">
 * <p>
 * The resource has failed to compensate. The payload contains the reason for the
failure. A participant MUST remember
 * this state until its {@link Forget} method is called.
 * </p>
 * <p>
 * The actual value is not important but it MUST correspond to a valid {@link
ParticipantStatus} enum value. For
 * example, if compensation was not possible because the resource already completed
(without being asked to) then a
 * value such as {@link ParticipantStatus#Completed} would be appropriate or if it was
due to a generic failure then
```

```
 * {@link ParticipantStatus#FailedToCompensate} would be valid.
 * </p>
 * <p>
 * Note that the actual state as reported by the {@link Status} method MUST be
 * {@link ParticipantStatus#FailedToCompensate}
 * </p>
 * </td>
 * </tr>
 * <tr>
 * <td scope="row">410</td>
 * <td scope="row">Empty</td>
 * <td scope="row">The resource does not know about the LRA</td>
 * </tr>
 * </table>
 *
 * <p>
 * The implementation will handle the return code 410 in the same way as the return
code 200. Specifically, when the
 * implementation calls the Compensate method as a result of the LRA being cancelled
and the participant returns the
 * code 410, the implementation assumes that the action is compensated and the
participant returned a 410 since
 * participant is allowed to forget about an action which is completely handled by the
participant.
 * </p>
 *
 * <p>
 * If any other code is returned (or, in the 409 case, the body does not correspond to
a valid state) then the
 * implementation SHOULD either keep retrying or attempt to discover the status by
calling the {@link Status} method if
 * present or a combination of both. If the implementation stops retrying then it
SHOULD log a warning. An example
 * scenario where the implementation might attempt to invoke the compensate method
twice and the status method is as
 * follows:
 * </p>
 *
 * <ol>
 * <li>The implementation invokes the compensate method via JAX-RS.</li>
 * <li>The JAX-RS server returns a 500 code (i.e., the notification does not reach the
participant).</li>
 * <li>If there is a status method then the implementation uses that to get the
current state of the participant. If the
 * status is Active then the implementation may infer that the original request never
reached the participant so it is
 * safe to reinvoke the compensate method.</li>
 * </ol>
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
```

```
public @interface Compensate {

}
```

Complete Annotation

```
/*
 ********************************************************************************
 * Copyright (c) 2018-2021 Contributors to the Eclipse Foundation
 *
 * See the NOTICE file(s) distributed with this work for additional
 * information regarding copyright ownership.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * You may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 ********************************************************************************/

package org.eclipse.microprofile.lra.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.eclipse.microprofile.lra.annotation.ws.rs.LRA;

/**
 * <p>
 * If a resource method executes in the context of an LRA and if the containing class
has a method annotated with
 * <code>&#64;Complete</code> (as well as method annotated with
<code>&#64;Compensate</code>) then this Complete method
 * will be invoked if the LRA is closed. The resource should attempt to perform any
clean up activities relating to any
 * actions it performed in the context of the LRA. If the annotation is present on
more than one method then an
 * arbitrary one will be chosen. The LRA specification makes no guarantees about when
the Complete method will be
 * invoked, just that it will eventually be called.
 * </p>
 *
 * <p>
 * In the case where the ability to complete the Long Running Action is time bounded,
you can limit the lifespan of the
 * Long Running action by providing values for the {@link LRA#timeLimit()} and {@link
```

```
LRA#timeUnit()} timeUnit}
 * attributes. When the time limit is reached the LRA becomes eligible for automatic
cancellation.
 * </p>
 *
 * <p>
 * If the annotation is applied to a JAX-RS resource method then the request method
MUST be {@link jakarta.ws.rs.PUT}.
 * The id of the currently running LRA can be obtained by inspecting the incoming JAX-
RS headers. If this LRA is nested
 * then the parent LRA MUST be present in the header with the name {@link
LRA#LRA_HTTP_PARENT_CONTEXT_HEADER} and the
 * header value will be of type {@link java.net.URI}.
 * </p>
 *
 * <p>
 * If the annotated method is not a JAX-RS resource method then the id of the
currently running LRA and its parent LRA
 * (if it is nested) can be obtained by adhering to predefined method signatures as
defined in the LRA specification
 * document. For example,
 * </p>
 *
 * <pre>
 *     <code>
 *        &#64;Complete
 *        public void complete(URI lraId, URI parentId) { ...}
 *     </code>
 * </pre>
 *
 * <p>
 * would be a valid completion method declaration. If an invalid signature is detected
the implementation of this
 * specification MUST prohibit successful startup of the application (e.g. with a
runtime exception).
 * </p>
 *
 * <p>
 * If the participant cannot complete immediately then it must report that completion
is in progress by either returning
 * a future (such as {@link java.util.concurrent.CompletionStage}) which will
eventually report one of the final states,
 * or a <code>202 Accepted</code> JAX-RS response code or, in the case of non JAX-RS
resource methods, by returning
 * {@link ParticipantStatus#Completing} (see the specification document for more
details).
 * </p>
 *
 * <p>
 * Note that according to the state model defined by {@link LRAStatus}, it is not
possible to receive completion
```

```
 * notifications after an LRA has been asked to close. Therefore combining this
annotation with an <code>&#64;LRA</code>
 * annotation that does not start a new LRA will result in a <code>412 PreCondition
Failed</code> JAX-RS response code.
 * On the other hand, combining it with an <code>&#64;LRA</code> annotation that
begins a new LRA can in certain use
 * cases make sense, but in this case the LRA that this method is being asked to
complete for will be unavailable.
 * </p>
 *
 * <p>
 * If the method is a JAX-RS resource method (or is a non JAX-RS method annotated with
<code>&#64;Complete</code> with
 * return type <code>jakarta.ws.rs.core.Response</code>) then the following are the
only valid response codes:
 * </p>
 *
 * <table border="0" cellpadding="3" cellspacing="0" summary="Valid JAX-RS completion
response codes">
 * <caption><span>JAX-RS Completion Response Codes</span><span> </span></caption>
 * <tr>
 * <th scope="col">Code</th>
 * <th scope="col">Response Body</th>
 * <th scope="col">Meaning</th>
 * </tr>
 * <tr>
 * <td scope="row">200</td>
 * <td scope="row">Empty</td>
 * <td scope="row">The resource has successfully completed</td>
 * </tr>
 * <tr>
 * <td scope="row">202</td>
 * <td scope="row">Empty</td>
 * <td scope="row">The resource is still attempting completion</td>
 * </tr>
 * <tr>
 * <td scope="row">409</td>
 * <td scope="row">{@link ParticipantStatus} enum value</td>
 * <td scope="row">
 * <p>
 * The resource has failed to complete. The payload contains the reason for the
failure. A participant MUST remember
 * this state until its {@link Forget} method is called.
 * </p>
 * <p>
 * The actual value is not important but it MUST correspond to a valid {@link
ParticipantStatus} enum value. For
 * example, if completion was not possible because the resource already compensated
(without being asked to) then a
 * value such as {@link ParticipantStatus#Compensated} would be appropriate or if it
was due to a generic failure then
```

```
 * {@link ParticipantStatus#FailedToComplete} would be valid. If the response body
does not contain a valid status then
 * the implementation MUST either reinvoke the method or discover the status using the
{@link Status} annotation if
 * present.
 * </p>
 * <p>
 * Note that the actual state as reported by the {@link Status} method MUST be
 * {@link ParticipantStatus#FailedToComplete}
 * </p>
 * </td>
 * </tr>
 * <tr>
 * <td scope="row">410</td>
 * <td scope="row">Empty</td>
 * <td scope="row">The resource does not know about the LRA</td>
 * </tr>
 * </table>
 *
 * <p>
 * The implementation will handle the return code 410 in the same way as the return
code 200. Specifically, when the
 * implementation calls the Complete method as a result of the LRA being closed and
the participant returns the code
 * 410, the implementation assumes that the action is completed and participant
returns a 410 since participant is
 * allowed to forget about an action which is completely handled by the participant.
 * </p>
 *
 * <p>
 * If any other code is returned (or, in the 409 case, the body does not correspond to
a valid state) then the
 * implementation SHOULD either keep retrying or attempt to discover the status by
calling the {@link Status} method if
 * present or a combination of both. If the implementation stops retrying then it
SHOULD log a warning. An example
 * scenario where the implementation might attempt to invoke the complete method twice
and the status method is as
 * follows:
 * </p>
 *
 * <ol>
 * <li>The implementation invokes the complete method via JAX-RS.</li>
 * <li>The JAX-RS server returns a 500 code (i.e., the notification does not reach the
participant).</li>
 * <li>If there is a status method then the implementation uses that to get the
current state of the participant. If the
 * status is Active then the implementation may infer that the original request never
reached the participant so it is
 * safe to reinvoke the complete method.</li>
 * </ol>
```

```
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Complete {

}
```

Status Annotation

```
/*
 ******************************************************************************
 * Copyright (c) 2018-2021 Contributors to the Eclipse Foundation
 *
 * See the NOTICE file(s) distributed with this work for additional
 * information regarding copyright ownership.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * You may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 ******************************************************************************/

package org.eclipse.microprofile.lra.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.eclipse.microprofile.lra.annotation.ws.rs.LRA;

import jakarta.ws.rs.core.Response;

/**
 * <p>
 * The LRA specification supports distributed communications amongst software
components and due to the unreliable
 * nature of networks, messages/requests can be lost, delayed, duplicated, etc., so
the implementation component
 * responsible for invoking {@link Compensate} and {@link Complete} annotated methods
may lose track of the status of a
 * participant. In this case, ideally it would just resend the completion or
compensation notification but if the
 * participant (the class that contains the Compensate and Complete annotations) does
not support idempotency then it
 * must be able to report its status by by annotating one of the methods with this
<code>&#64;Status</code> annotation.
 * The annotated method should report the status according to one of the {@link
ParticipantStatus} enum values.
 * </p>
```

```
 *
 * <p>
 * If the annotation is applied to a JAX-RS resource method then the request method
MUST be {@link jakarta.ws.rs.GET}.
 * The context of the currently running LRA can be obtained by inspecting the incoming
JAX-RS headers. If this LRA is
 * nested then the parent LRA MUST be present in the header with the name {@link
LRA#LRA_HTTP_PARENT_CONTEXT_HEADER} and
 * value is of type {@link java.net.URI}.
 * </p>
 *
 * <p>
 * If the annotated method is not a JAX-RS resource method, the context of the
currently running LRA can be obtained by
 * adhering to a predefined method signature as defined in the LRA specification
document. Similarly the method may
 * determine whether or not it runs with a nested LRA by providing a parameter to hold
the parent context. For example,
 * </p>
 *
 * <pre>
 *     <code>
 *          &#64;Status
 *          public void status(URI lraId, URI parentId) { ...}
 *     </code>
 * </pre>
 *
 * <p>
 * would be a valid status method declaration. If an invalid signature is detected the
implementation of this
 * specification MUST prohibit successful startup of the application (e.g., with a
runtime exception).
 * </p>
 *
 * <p>
 * If the participant has already responded successfully to an invocation of the
{@link Compensate} or {@link Complete}
 * method then it may report <code>410 Gone</code> HTTP status code or in case of non-
JAX-RS method returning
 * {@link ParticipantStatus} to return <code>null</code>.
 * </p>
 *
 * <p>
 * Since the participant generally needs to know the id of the LRA in order to report
its status there is generally no
 * benefit to combining this annotation with the <code>&#64;LRA</code> annotation
(though it is not prohibited).
 * </p>
 *
 * <p>
 * If the method is a JAX-RS resource method (or is a non JAX-RS method annotated with
```

```
<code>&#64;Status</code> with
 * return type {@link Response}) then the following are the only valid response codes:
 * </p>
 *
 * <table border="0" cellpadding="3" cellspacing="0" summary="Valid JAX-RS response
codes for Status methods">
 * <caption><span>JAX-RS Response Codes For Status
Methods</span><span> </span></caption>
 * <tr>
 * <th scope="col">Code</th>
 * <th scope="col">Response Body</th>
 * <th scope="col">Meaning</th>
 * </tr>
 * <tr>
 * <td scope="row">200</td>
 * <td scope="row">{@link ParticipantStatus} enum value</td>
 * <td scope="row">The current status of the participant</td>
 * </tr>
 * <tr>
 * <td scope="row">202</td>
 * <td scope="row">Empty</td>
 * <td scope="row">The resource is attempting to determine the status and the caller
should retry later</td>
 * </tr>
 * <tr>
 * <td scope="row">410</td>
 * <td scope="row">Empty</td>
 * <td scope="row">The method does not know about the LRA</td>
 * </tr>
 * </table>
 *
 * <p>
 * The implementation will handle the return code 410 in the same way as the return
code 200. Specifically, when the
 * implementation calls the Status method after it has called the Complete or
Compensated method and received a response
 * which indicates that the process is in progress (with a return code 202, for
example). The response code 410 which is
 * received when calling this Status annotated method, MUST be interpreted by the
implementation that the process is
 * successfully completed and the participant already forgot about the LRA.
 * </p>
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Status {
}
```

ParticipantStatus

```
/*
 ******************************************************************************
 * Copyright (c) 2018-2021 Contributors to the Eclipse Foundation
 *
 * See the NOTICE file(s) distributed with this work for additional
 * information regarding copyright ownership.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * You may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 ******************************************************************************/

package org.eclipse.microprofile.lra.annotation;

import org.eclipse.microprofile.lra.annotation.ws.rs.Leave;

/**
 * A representation of the status of a participant according to a participant state
model:
 *
 * The initial state {@link #Active} is entered when a participant is first associated
with a Long Running Action.
 *
 * The state {@link #Compensating} is entered when a compensate notification is
received (which indicates that the
 * associated LRA was cancelled). The transition to end state {@link #Compensated}
should occur when the participant has
 * compensated for any actions it performed when the LRA was executing. If
compensation is not, and will never be,
 * possible then the final state of {@link #FailedToCompensate} is entered and the
participant cannot leave this state
 * until it receives a {@link Forget} notification.
 *
 * The state {@link #Completing} is entered when a complete notification is received
(which indicates that the
 * associated LRA was closed). This state is followed by the {@link #Completed} or
{@link #FailedToComplete} state
 * depending upon whether the participant was or was not able to tidy up.
 *
 * Note that a particant can leave this state model via the {@link Leave} annotation
```

```
provided that the associated LRA is
 * in the state {@link LRAStatus#Active}.
 *
 * The name value of the enum should be returned by participant methods marked with
the {@link Status},
 * {@link Compensate} and {@link Complete} annotations.
 */
public enum ParticipantStatus {
    /**
     * The participant has not yet been asked to Complete or Compensate
     */
    Active,
    /**
     * The participant is currently compensating for any work it performed
     */
    Compensating,
    /**
     * The participant has successfully compensated for any work it performed
     */
    Compensated,
    /**
     * The participant was not able to compensate for the work it performed (and it
must remember it could not
     * compensate until such time that it receives a forget message ({@link Forget})
     */
    FailedToCompensate,
    /**
     * The participant is tidying up after being told to complete
     */
    Completing,
    /**
     * The participant has confirmed that is has completed any tidy-up actions
     */
    Completed,
    /**
     * The participant was unable to tidy-up
     */
    FailedToComplete,
}
```