MicroProfile Context Propagation

1.1, October 21, 2020

Table of Contents

Microprofile Context Propagation

MicroProfile Context Propagation Specification

Introduction

The MicroProfile Context Propagation specification introduces APIs for propagating contexts across units of work that are thread-agnostic. It makes it possible to propagate context that was traditionally associated to the current thread across various types of units of work such as `CompletionStage`, `CompletableFuture`, `Function`, `Runnable` regardless of which particular thread ends up executing them.

Motivation

When using a reactive model, execution is cut into small units of work that are chained together to assemble a reactive pipeline. The context under which each unit of work executes is often unpredictable and depends on the particular reactive engine used. Some units might run with the context of a thread that awaits completion, or the context of a previous unit that completed and triggered the dependent unit, or with no/undefined context at all. Existing solutions for transferring thread context, such as the EE Concurrency Utilities `ContextService`, are tied to a specific asynchrony model, promotes usage of thread pools, is difficult to use and require a lot of boilerplate code. This specification makes it possible for thread context propagation to easily be done in a type-safe way, keeping boilerplate code to a minimum, as well as allowing for thread context propagation to be done automatically for many types of reactive models.

We distinguish two main use-cases for propagating contexts to reactive pipelines:

- Splitting units of work into a sequential pipeline where each unit will be executed after the other. Turning an existing blocking request into an async request would produce such pipelines.
- Fanning out units of work to be executed in parallel on a managed thread pool. Launching an asynchronous job from a request without waiting for its termination would produce such pipelines.

Goals

- Pluggable context propagation to the most common unit of work types.
- Mechanism for thread context propagation to `CompletableFuture` and `CompletionStage` units of work that reduces the need for boilerplate code.
- Full compatibility with EE Concurrency spec, such that proposed interfaces can seamlessly work alongside EE Concurrency, without depending on it.

Solution

This specification introduces two interfaces that contain methods that can work alongside EE Concurrency, if available.

The interface, `org.eclipse.microprofile.context.ManagedExecutor`, provides methods for obtaining managed instances of `CompletableFuture` which are backed by the managed executor as the default asynchronous execution facility and the default mechanism of defining thread context propagation. Similar to EE Concurrency's `ManagedExecutorService`, the MicroProfile `ManagedExecutor` also implements the Java SE `java.util.concurrent.ExecutorService` interface, using managed threads when asynchronous invocation is required. It is possible for a single implementation to be capable of simultaneously implementing both `ManagedExecutor` and `ManagedExecutorService` interfaces.

A second interface, `org.eclipse.microprofile.context.ThreadContext`, provides methods for individually contextualizing units of work such as `CompletionStage`, `CompletionFuture`, `Runnable`, `Function`, `Supplier` and more, without tying them to a particular thread execution model. This gives the user finer-grained control over the capture and propagation of thread context by remaining thread execution agnostic. It is possible for a single implementation to be capable of simultaneously implementing both `ThreadContext` and `ContextService` interfaces.

Builders

Instances of `ManagedExecutor` and `ThreadContext` can be constructed via builders with fluent API, for example,

```
    ManagedExecutor executor = ManagedExecutor.builder()
        .propagated(ThreadContext.APPLICATION)
        .cleared(ThreadContext.ALL_REMAINING)
        .maxAsync(5)
        .build();

    ThreadContext threadContext = ThreadContext.builder()
        .propagated(ThreadContext.APPLICATION, ThreadContext.CDI)
        .cleared(ThreadContext.ALL_REMAINING)
        .build();
```

Applications should shut down instances of `ManagedExecutor` that they build after they are no longer needed. The shutdown request serves as a signal notifying the container that resources can be safely cleaned up.

Example usage

For managed executor,

```
    CompletableFuture<Long> stage = executor.newIncompleteFuture()
        .thenApply(function)
        .thenAccept(consumer);
    stage.completeAsync(supplier);
```

Or similarly for thread context,

```
    threadContext.withContextCapture(unmanagedCompletionStage)
        .thenApply(function)
        .thenAccept(consumer);
```

Sharing Instances

The definition of CDI producers at application scope, combined with injection, is a convenient mechanism for sharing instances across an application.

```
    @Produces @ApplicationScoped @SecAndAppContextQualifier
    ManagedExecutor executor1 = ManagedExecutor.builder()
        .propagated(ThreadContext.SECURITY, ThreadContext.APPLICATION)
        .build();

    ... // in some other bean
    @Inject
    void setCompletableFuture(@SecAndAppContextQualifier ManagedExecutor executor2) {
        completableFuture = executor2.newIncompleteFuture();
    }

    ... // in yet another bean
    @Inject @SecAndAppContextQualifier
    ManagedExecutor executor3;

    // example qualifier annotation
    @Qualifier
    @Retention(RetentionPolicy.RUNTIME)
    @Target({ ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER })
    public @interface SecAndAppContextQualifier {}
```

Specifying Defaults via MicroProfile Config

MicroProfile Config properties can be used to specify defaults for configuration attributes that are not otherwise configured on `ManagedExecutor` and `ThreadContext` instances. Here is an example that includes all of the available attributes that can be defaulted:

```
mp.context.ManagedExecutor.propagated=Security,Application
mp.context.ManagedExecutor.cleared=Remaining
mp.context.ManagedExecutor.maxAsync=10
mp.context.ManagedExecutor.maxQueued=-1

mp.context.ThreadContext.propagated=None
mp.context.ThreadContext.cleared=Security,Transaction
mp.context.ThreadContext.unchanged=Remaining
```

Builders for ManagedExecutor and ThreadContext

The MicroProfile Context Propagation spec defines a fluent builder API to programmatically obtain instances of `ManagedExecutor` and `ThreadContext`. Builder instances are obtained via static `builder()` methods on `ManagedExecutor` and `ThreadContext`.

Example ManagedExecutor Builder Usage

```
import java.util.concurrent.CompletionStage;
import java.util.concurrent.CompletableFuture;
import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.eclipse.microprofile.context.ManagedExecutor;
import org.eclipse.microprofile.context.ThreadContext;

public class ExampleServlet extends HttpServlet {
    ManagedExecutor executor;

    public void init(ServletConfig config) {
        executor = ManagedExecutor.builder()
                              .propagated(ThreadContext.APPLICATION)
                              .cleared(ThreadContext.ALL_REMAINING)
                              .maxAsync(5)
                              .build();
    }

    public void doGet(HttpServletRequest req, HttpServletResponse res) {
       completionStage = executor.runAsync(task1)
                            .thenRunAsync(task2)
                            ...
    }

    public void destroy() {
        executor.shutdown();
    }
}
```

Applications are encouraged to cache and reuse `ManagedExecutor` instances. It is the responsibility of the application to shut down `ManagedExecutor` instances that are no longer needed, so as to allow the container to efficiently free up resources.

Example ThreadContext Builder Usage

```java
import java.util.concurrent.CompletableFuture;
import java.util.function.Function;
import java.util.function.Supplier;
import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.eclipse.microprofile.context.ThreadContext;

public class ExampleServlet extends HttpServlet {
    ThreadContext threadContext;

    public void init(ServletConfig config) {
        threadContext = ThreadContext.builder()
                            .propagated(ThreadContext.APPLICATION, ThreadContext
.SECURITY)
                            .unchanged()
                            .cleared(ThreadContext.ALL_REMAINING)
                            .build();
    }

    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        Function<Long, Long> contextFn = threadContext.contextualFunction(x -> {
            ... operation that requires security & application context
            return result;
        });

        // By using java.util.concurrent.CompletableFuture.supplyAsync rather
        // than a managed executor, context propagation is unpredictable,
        // except for the contextFn action that we pre-contextualized using
        // ThreadContext above.
        stage = CompletableFuture.supplyAsync(supplier)
                                 .thenApplyAsync(function1)
                                 .thenApply(contextFn)
                                 ...
    }
}
```

Reuse of Builders

Instances of `ManagedExecutor.Builder` and `ThreadContext.Builder` retain their configuration after the build method is invoked and can be reused. Subsequent invocations of the build() method create new instances of `ManagedExecutor` and `ThreadContext` that operate independently of previously built instances.

CDI Injection

In order to use `ManagedExecutor` and `ThreadContext` as CDI beans, define producer for them as `@ApplicationScoped` so that instances are shared and reused. In most cases, more granular and shorter-lived scopes are undesirable. For instance, having a new `ManagedExecutor` instance created per HTTP request typically does not make sense. In the event that a more granular scope is desired, the application must take care to supply a disposer to ensure that the executor is shut down once it is no longer needed. When using application scope, it is optional to supply a disposer because the specification requires the container to automatically shut down `ManagedExecutor` instances when the application stops.

Example Producer for `ManagedExecutor`

Example qualifier,

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.inject.Qualifier;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER })
public @interface SecurityAndCDIContext {}
```

Example producer and disposer,

```
import org.eclipse.microprofile.context.ManagedExecutor;
import org.eclipse.microprofile.context.ThreadContext;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Produces;

@ApplicationScoped
public class MyFirstBean {
    @Produces @ApplicationScoped @SecurityAndCDIContext
    ManagedExecutor executor = ManagedExecutor.builder()
        .propagated(ThreadContext.SECURITY, ThreadContext.CDI)
        .build();

    void disposeExecutor(@Disposes @SecurityAndCDIContext ManagedExecutor exec) {
        exec.shutdownNow();
    }
}
```

Example injection point,

```
import org.eclipse.microprofile.context.ManagedExecutor;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

@ApplicationScoped
public class MySecondBean {
    @Inject @SecurityAndCDIContext
    ManagedExecutor sameExecutor;
    ...
}
```

Example Producer for `ThreadContext`

Example qualifier,

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.inject.Qualifier;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER })
public @interface AppContext {}
```

Example producer method,

```
import org.eclipse.microprofile.context.ThreadContext;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Produces;

@ApplicationScoped
public class MyFirstBean {
    @Produces @ApplicationScoped @AppContext
    createAppContextPropagator() {
        return ThreadContext.builder()
               .propagated(ThreadContext.APPLICATION)
               .cleared(ThreadContext.SECURITY, ThreadContext.TRANSACTION)
               .unchanged(ThreadContext.ALL_REMAINING)
               .build();
  }
}
```

Example injection point,

```
import org.eclipse.microprofile.context.ThreadContext;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;
...

@ApplicationScoped
public class MySecondBean {
    Function<Integer, Item> findItem;

    @Inject
    protected void setFindItem(@AppContext ThreadContext appContext) {
        findItem = appContext.contextualFunction(i -> {
            try (Connection con =
                 ((DataSource) InitialContext.doLookup("java:comp/env/ds1"))
.getConnection();
                 PreparedStatement stmt = con.prepareStatement(sql)) {
                stmt.setInt(1, i);
                return toItem(stmt.executeQuery());
            } catch (Exception x) {
                throw new CompletionException(x);
            }
        });
    }
    ...
}
```

Establishing default values with MicroProfile Config

If a MicroProfile Config implementation is available, MicroProfile Config can be used to establish default values for configuration attributes of ManagedExecutor and ThreadContext. This allows the application to bypass configuration of one or more attributes when using the builders to create instances.

For example, you could specify MicroProfile Config properties as follows to establish a set of defaults for ManagedExecutor,

```
mp.context.ManagedExecutor.propagated=Remaining
mp.context.ManagedExecutor.cleared=Transaction
mp.context.ManagedExecutor.maxAsync=10
mp.context.ManagedExecutor.maxQueued=-1
```

With these defaults in place, the application can create a ManagedExecutor instance without specifying some of the configuration attributes,

```
executor = ManagedExecutor.builder().maxAsync(5).build();
```

In the code above, the application specifies only the maxAsync attribute, limiting actions and tasks requested to run async to at most 5 running at any given time. The other configuration attributes are defaulted as specified in MicroProfile config, with no upper bound on queued tasks, Transaction context cleared, and all other context types propagated.

As another example, the following MicroProfile Config properties establish defaults for ThreadContext,

```
mp.context.ThreadContext.propagated=None
mp.context.ThreadContext.cleared=Security,Transaction
mp.context.ThreadContext.unchanged=Remaining
```

With these defaults in place, the application can create a ThreadContext instance without specifying some of the configuration attributes,

```
cdiContextPropagator = ThreadContext.builder()
                                    .propagated(ThreadContext.CDI)
                                    .build();
```

In the code above, the application specifies only the propagated attribute, indicating that only CDI context is propagated. The other configuration attributes inherit the defaults, which includes clearing Security and Transaction context and leaving all other thread context types unchanged.

Specifying Defaults for Array Properties in MicroProfile Config

When using MicroProfile Config to define defaults for array type properties (`propagated`, `cleared`, and `unchanged`), the following rules apply for config property values:

- The value can be a single array element, multiple elements (delimited by `,`), or the value `None`, which is interpreted as an empty array of context types.
- Array elements can be any value returned by a `ThreadContextProvider`'s `getThreadContextType()` method.
- Array elements can be any thread context type constant value from ThreadContext (such as `Security`, `Application`, or `Remaining`).
- The usual rules from the MicroProfile Config specification apply, such as escaping special characters.

As of MicroProfile Config 2.0 and above, the value of `None` must be used to indicate an empty array. Prior to MicroProfile Config 2.0, an empty string value could be used to specify an empty array. In order to guarantee that empty string config values are interpreted properly, the MicroProfile Context Propagation implementation must interpret both of the following as indicating empty:

- empty array
- array containing the empty String as its singular element

This is necessary due to a lack of clarity in the first several versions of the MicroProfile Config specification about how the empty string value is to be interpreted for arrays of String. MicroProfile Config 2.0 removed the ability to configure empty arrays and lists altogether.

Overriding values with MicroProfile Config

MicroProfile Config can also be used in the standard way to enable configuration attributes of the `ManagedExecutor` and `ThreadContext` builders to be overridden. For example,

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER })
public @interface SecurityAndAppContext {}

@Produces @ApplicationScoped @SecurityAndAppContext
ManagedExecutor createExecutor(
    @ConfigProperty(name="exec1.maxAsync", defaultValue="5") Integer a,
    @ConfigProperty(name="exec1.maxQueued", defaultValue="20") Integer q) {
    return ManagedExecutor.builder()
                          .maxAsync(a)
                          .maxQueued(q)
                          .propagated(ThreadContext.SECURITY, ThreadContext
.APPLICATION)
                          .cleared(ThreadContext.ALL_REMAINING)
                          .build();
}
```

MicroProfile Config can be used to override configuration attributes from the above example as follows,

```
exec1.maxAsync=10
exec1.maxQueued=15
```

Transaction Context

Implementations of MicroProfile Context Propagation are allowed to provide varying degrees of support for transaction context.

This varies from not supporting transactions at all, to supporting the clearing of transaction context only, to supporting propagation of transactions for serial, or even parallel use. The `ThreadContextProvider` for transaction context raises exceptions that are defined by the specification to indicate lack of support for the various optional aspects of transaction context propagation.

No Support for Transactions

The `ManagedExecutor` and `ThreadContext` builders are allowed to raise `IllegalStateException` from the `build` method when a builder is configured to propagate transaction context but transactions are not supported (no provider of transaction context is available). This follows the general pattern defined by the `build` method JavaDoc concerning unavailable context types.

```
executor = ManagedExecutor.builder()
                          .propagated(ThreadContext.TRANSACTION)
                          .cleared(ThreadContext.ALL_REMAINING)
                          .build(); // <-- raises IllegalStateException
```

Propagation of the Absence of a Transaction, but not of Active Transactions

It can be useful to propagate that a completion stage action does not run under a transaction in order to guarantee deterministic behavior and allow the action to manage its own transactional work. This is important in being able to write applications that reliably access completion stage results from within a transaction without risking that the action might run as part of the transaction. For example,

```
executor = ManagedExecutor.builder()
           .propagated(ThreadContext.TRANSACTION, ThreadContext.APPLICATION)
           .cleared(ThreadContext.ALL_REMAINING)
           .build();

// propagates the absence of a transaction,
// allowing the action to start its own transaction
stage1 = executor.supplyAsync(supplier);
stage2 = stage1.thenApply(u -> {
    try {
        DataSource ds = InitialContext.doLookup("java:comp/env/ds1");
        UserTransaction tx = InitialContext.doLookup("java:comp/UserTransaction");
        tx.begin();
        try (Connection con = ds.getConnection()) {
            return u + con.createStatement().executeUpdate(sql);
        } finally {
            tx.commit();
        }
    } catch (Exception x) {
        throw new CompletionException(x);
    }
});

tx.begin();
... do transactional work here

updateCount = stage2.join(); // <-- stage2's action is guaranteed to never
                             //     run under this transaction because absence
                             //     of a transaction is propagated to it

... more transactional work
```

It should be noted that cleared, rather than propagated, transaction context can accomplish the same.

A `ThreadContextProvider` that supports propagation of the absence of a transaction, but not propagation of an active transaction is allowed to raise `IllegalStateException` from its `currentContext` method. The exception flows back to the application on operations such as `managedExecutor.supplyAsync(supplier)`, `threadContext.withContextCapture`, or `threadContext.contextualFunction`, indicating the restriction against propagating active transactions. The `IllegalStateException` should have a meaningful message making it clear to the user that lack of support for the propagation of active transactions is the cause of the error.

For example, the application can expect to see `IllegalStateException` here if the optional behavior of propagating active transactions to other threads is not supported,

```
tx.begin();
stage = executor.runAsync(action); // <-- raises IllegalStateException
...
```

Propagation of Active Transactions for Serial Use, but not Parallel

Some transaction managers and transactional resources may allow for propagation of an active transaction to multiple threads serially, with the limitation that the transaction is active on at most one thread at any given time.

For example,

```
executor = ManagedExecutor.builder()
           .propagated(ThreadContext.TRANSACTION)
           .build();

// Allowed because it limits the transaction to serial use
stage1 = executor.newIncompleteFuture();
tx.begin();
try {
    stage3 = stage1.thenApply(updateCount -> {
        try (Connection con = dataSource.getConnection()) {
            return updateCount + con.createStatement().executeUpdate(sql2);
        } catch (SQLException x) {
            throw new CompletionException(x);
        }
    }).thenApply(updateCount -> {
        try (Connection con = dataSource.getConnection()) {
            return updateCount + con.createStatement().executeUpdate(sql3);
        } catch (SQLException x) {
            throw new CompletionException(x);
        }
    }).whenComplete((result, failure) -> {
        try {
            if (failure == null && tx.getStatus() == Status.STATUS_ACTIVE)
                tx.commit();
            else
                tx.rollback();
        } catch (Exception x) {
            if (failure == null)
                throw new CompletionException(x);
        }
    });
} finally {
    // vendor-specific means required to obtain TransactionManager instance
    transactionManager.suspend();
}

// ... possibly on another thread
stage1.complete(initialCount);
```

A 'ThreadContextProvider' that supports serial use of a propagated transaction, but not parallel use, is allowed to raise `IllegalStateException` upon attempts to associate a JTA transaction to a second thread when the JTA transaction is already active on another thread. The transaction context provider raises `IllegalStateException` upon `ThreadContextSnapshot.begin`, which exceptionally completes the action or task without running it. When rejecting parallel use of a transaction, the transaction context provider should also mark the transaction for rollback. The `IllegalStateException` should have a meaningful message making it clear to the user that lack of support for the propagation of active transactions for parallel use across multiple threads is the cause of the error.

The application sees the error when requesting the result of the corresponding stage. For example,

```
tx.begin();
try {
    stage = executor.supplyAsync(() -> {
        try (Connection con = dataSource.getConnection()) {
            return con.createStatement().executeUpdate(sql1);
        } catch (SQLException) {
            throw new CompletionException(x);
        }
    });

    try (Connection con = dataSource.getConnection()) {
        con.createStatement().executeUpdate(sql2);
    });

    stage.join(); // <-- raises CompletionException with a chained
                  //     IllegalStateException indicating lack of support
                  //     for propagating an active transaction to multiple
                  //     threads

    tx.commit();
} catch (Exception x) {
    tx.rollback();
    ...
```

Propagation of Active Transactions for Parallel Use

An implementation that supports the optional behavior of propagating active transactions for use on multiple threads in parallel may choose whether or not to support commit and rollback operations from dependent stage actions. If unsupported, these operations raise `SystemException` when invoked from a separate completion stage action. As always, the application is responsible for following best practices to ensure transactions are properly resolved and transactional resources are properly cleaned up under all possible outcomes.

Here is an example of committing the transaction in a dependent stage action,

```
tx.begin();
try {
    stage1 = executor.runAsync(action1);
    stage2 = executor.runAsync(action2);
    stage3 = stage1.runAfterBoth(stage2, (u,v) -> action3)
                   .whenComplete((result, failure) -> {
        try {
            if (failure == null && tx.getStatus() == Status.STATUS_ACTIVE)
                tx.commit();   // <-- raises SystemException if unsupported within
dependent stage
            else
                tx.rollback(); // <-- raises SystemException if unsupported within
dependent stage
        } catch (Exception x) {
            if (failure == null)
                throw new CompletionException(x);
        }
    });
} finally {
    // vendor-specific means required to obtain TransactionManager instance
    transactionManager.suspend();
}
```

Here is an example of committing the transaction from the main thread,

```
tx.begin();
try {
    stage1 = executor.runAsync(action1);
    stage2 = executor.runAsync(action2);
    stage3 = CompletableFuture.allOf(stage1, stage2);
    stage3.join();
} finally {
    if (tx.getStatus() == Status.STATUS_ACTIVE && !stage3.isCompletedExceptionally())
        tx.commit();
    else
        tx.rollback();
}
```

MicroProfile Context Propagation Examples

This section includes some additional examples of spec usage.

Contextualize a new CompletableFuture and all dependent stages

```
    executor = ManagedExecutor.builder()
                   .cleared(ThreadContext.TRANSACTION, ThreadContext.SECURITY)
                   .propagated(ThreadContext.ALL_REMAINING)
                   .build();

    CompletableFuture<Long> stage1 = executor.newIncompleteFuture();
    stage1.thenApply(function1)      // runs with captured context
          .thenApply(function2);     // runs with captured context
    stage1.completeAsync(supplier1); // runs with captured context
```

Apply thread context to a CompletionStage and all its dependent stages

```
    threadContext = ThreadContext.builder()
                        .propagated(ThreadContext.SECURITY)
                        .unchanged()
                        .cleared(ThreadContext.ALL_REMAINING)
                        .build();

    stage = threadContext.withContextCapture
(invokeSomeMethodThatReturnsUnmanagedCompletionStage());
    stage.thenApply(function1)  // runs with captured context
         .thenAccept(consumer); // runs with captured context
```

Apply thread context to a single CompletionStage action

```
    threadContext = ThreadContext.builder()
                        .propagated(ThreadContext.SECURITY)
                        .unchanged()
                        .cleared(ThreadContext.ALL_REMAINING)
                        .build();

    Consumer<String> contextualConsumer = threadContext.contextualConsumer(s -> {
            ... do something that requires context
        });

    stage = invokeSomeMethodThatReturnsUnmanagedCompletionStage();
    stage.thenApply(function1)            // context is unpredictable
         .thenAccept(contextualConsumer); // runs with captured context
```

Reusable Context Snapshot

```
    threadContext = ThreadContext.builder()
                                 .cleared(ThreadContext.TRANSACTION)
                                 .unchanged(ThreadContext.SECURITY)
                                 .propagated(ThreadContext.ALL_REMAINING)
                                 .build();
    contextSnapshot = threadContext.currentContextExecutor();

    ... on some other thread,
    contextSnapshot.execute(() -> {
        ... do something that requires the previously captured context
    });
```

Run under the transaction of the executing thread

If you do not want to either propagate or clear a context, you need to explicitly mark it as `unchanged`. In this example we want to capture and propagate only the application context, but we don't want to clear the transaction context because we're going to manually set it up for the new thread where we're going to use the captured application context:

```
threadContext = ThreadContext.builder()
                    .propagated(ThreadContext.APPLICATION)
                    .unchanged(ThreadContext.TRANSACTION)
                    .cleared(ThreadContext.ALL_REMAINING)
                    .build();

Callable<Integer> updateDatabase = threadContext.contextualCallable(() -> {
    DataSource ds = InitialContext.doLookup("java:comp/env/ds1");
    try (Connection con = ds.getConnection()) {
        return con.createStatement().executeUpdate(sql);
    }
}));

... on some other thread,

tx.begin();
... do transactional work
// runs as part of the transaction, but with the captured application scope
updateDatabase.call();
... more transactional work
tx.commit();
```

Thread Context Providers

The initial release of EE Concurrency assumed a single monolithic implementation of the full set of EE specifications that could thus rely on vendor-specific internals to achieve context propagation. However, in practice, open source implementations of various specs are often pieced together into a comprehensive solution.

The thread context provider SPI is defined to bridge the gap, allowing any provider of thread context to publish and make available the type of thread context it supports, following a standard and predictable pattern that can be relied upon by a MicroProfile Context Propagation implementation, enabling it to facilitate the inclusion of any generic thread context alongside the spec-defined thread context types that it captures and propagates.

With this model, the provider of thread context implements the `org.eclipse.microprofile.context.spi.ThreadContextProvider` interface and packages it in a way that makes it available to the `ServiceLoader`. `ThreadContextProvider` identifies the thread context type and provides a way to capture snapshots of thread context as well as for applying empty/cleared context to threads.

Example

The following is a working example of a thread context provider and related interfaces. The example context type that it propagates is the priority of a thread. This is chosen, not because it is useful in any way, but because the concept of thread priority is simple, well understood, and already built into Java, allowing the reader to focus on the mechanisms of thread context capture/propagate/restore rather than the details of the context type itself.

ThreadContextProvider

The interface, `org.eclipse.microprofile.context.spi.ThreadContextProvider`, is the first point of interaction between the MicroProfile Context Propagation implementation and a thread context provider. This interface is the means by which the MicroProfile Context Propagation implementation requests the capturing of a particular context type from the current thread. It also provides a way to obtain a snapshot of empty/cleared context of this type and identifies the name by which the user refers to this context type when configuring a `ManagedExecutor` or `ThreadContext`.

```java
package org.eclipse.microprofile.example.context.priority;

import java.util.Map;
import org.eclipse.microprofile.context.spi.ThreadContextProvider;
import org.eclipse.microprofile.context.spi.ThreadContextSnapshot;

public class ThreadPriorityContextProvider implements ThreadContextProvider {
    public String getThreadContextType() {
        return "ThreadPriority";
    }

    public ThreadContextSnapshot currentContext(Map<String, String> props) {
        return new ThreadPrioritySnapshot(Thread.currentThread().getPriority());
    }

    public ThreadContextSnapshot clearedContext(Map<String, String> props) {
        return new ThreadPrioritySnapshot(Thread.NORM_PRIORITY);
    }
}
```

ThreadContextSnapshot

The interface, `org.eclipse.microprofile.context.spi.ThreadContextSnapshot`, represents a snapshot of thread context. The MicroProfile Context Propagation implementation can request the context represented by this snapshot to be applied to any number of threads by invoking the `begin` method. An instance of `org.eclipse.microprofile.context.spi.ThreadContextController`, which is returned by the `begin` method, stores the previous context of the thread. The `ThreadContextController` instance provided for one-time use by the MicroProfile Context Propagation implementation to restore the previous context after the context represented by the snapshot is no longer needed on the thread.

```
package org.eclipse.microprofile.example.context.priority;

import java.util.concurrent.atomic.AtomicBoolean;
import org.eclipse.microprofile.context.spi.ThreadContextController;
import org.eclipse.microprofile.context.spi.ThreadContextSnapshot;

public class ThreadPrioritySnapshot implements ThreadContextSnapshot {
    private final int priority;

    ThreadPrioritySnapshot(int priority) {
        this.priority = priority;
    }

    public ThreadContextController begin() {
        Thread thread = Thread.currentThread();
        int priorityToRestore = thread.getPriority();
        AtomicBoolean restored = new AtomicBoolean();

        ThreadContextController contextRestorer = () -> {
            if (restored.compareAndSet(false, true))
                thread.setPriority(priorityToRestore);
            else
                throw new IllegalStateException();
        };

        thread.setPriority(priority);

        return contextRestorer;
    }
}
```

ServiceLoader entry

To make the `ThreadContextProvider` implementation available to the `ServiceLoader`, the provider JAR includes a file of the following name and location,

```
META-INF/services/org.eclipse.microprofile.context.spi.ThreadContextProvider
```

The content of the aforementioned file must be one or more lines, each specifying the fully qualified name of a `ThreadContextProvider` implementation that is provided within the JAR file. For our example context provider, this file consists of the following line:

```
org.eclipse.microprofile.example.context.priority.ThreadPriorityContextProvider
```

Usage from Application

The following example shows application code that uses a `ManagedExecutor` that propagates the example context type. If the provider is implemented correctly and made available on the application's thread context class loader, the async `Runnable` should report that it is running with a priority of 3.

```
ManagedExecutor executor = ManagedExecutor.builder()
                                          .propagated("ThreadPriority")
                                          .cleared(ThreadContext.ALL_REMAINING)
                                          .build();
Thread.currentThread().setPriority(3);

executor.runAsync(() -> {
    System.out.println("Running with priority of " +
        Thread.currentThread().getPriority());
});
```

Context Manager Provider

A MicroProfile Context Propagation implementation provides an implementation of the `org.eclipse.microprofile.context.spi.ContextManagerProvider` interface via either of the following mechanisms:

- By manually registering the implementation via the static `register(ContextManagerProvider)` method. This register returns a `ContextManagerProviderRegistration` instance which can be used to unregister.
- Alternately, via the `ServiceLoader`, by including a file of the following name and location: `META-INF/services/org.eclipse.microprofile.context.spi.ContextManagerProvider`. The content of the aforementioned file must a single line specifying the fully qualified name of the `ContextManagerProvider` implementation that is provided within the JAR file.

The `ContextManagerProvider` implementation has one main purpose, which is to supply and maintain instances of `ContextManager` per class loader. This is done via the `getContextManager(ClassLoader)` method.

In the case where the `ContextManagerProvider` is fully integrated with the container, all other methods of `ContextManagerProvider` are optional, with their default implementations being sufficient.

In the case where the `ContextManagerProvider` implementation is distinct from the container, several other methods are made available to allow the container to build new instances of `ContextManager` (via `getContextManagerBuilder`), register these instances with the `ContextManagerProvider` per class loader (`registerContextManager`), and unregister these instances when the class loader is no longer valid (`releaseContextManager`).

Context Manager

`ContextManager’s purpose is to provide builders for `ManagedExecutor` and `ThreadContext`. The builders create instances of `ManagedExecutor` and `ThreadContext` where thread context management is based on the `ThreadContextProvider’s that are accessible to the `ServiceLoader` from the class loader that is associated with the `ContextManager` instance.

Context Manager Builder

The builder for `ContextManager` is optional if the `ContextManagerProvider` is inseparable from the container, in which case there is no need to provide an implementation.

This builder enables the container to create customized instances of `ContextManager` for a particular class loader. The container can choose to have thread context providers loaded from the class loader (`addDiscoveredThreadContextProviders`) or manually supply its own (`withThreadContextProviders`). Similarly, the container can choose to have extensions loaded from the class loader (`addDiscoveredContextManagerExtensions`) or provide its own (`withContextManagerExtensions`). The container is responsible for managing registration and unregistration of all `ContextManager` instances that it builds.

Context Manager Extension

`ContextManagerExtension` is an optional plugin point that allows you to receive notification upon creation of each `ContextManager`. This serves as a convenient invocation point for enabling system wide context propagator hooks. After creating each `ContextManager`, the MicroProfile Context Propagation implementation queries the `ServiceLoader` for implementations of `org.eclipse.microprofile.context.spi.ContextManagerExtension` and invokes the setup method of each.

To register a `ContextManagerExtension`, a JAR file that is accessible from the class loader associated with the `ContextManager` must include a file of the following name and location,

```
META-INF/services/org.eclipse.microprofile.context.spi.ContextManagerExtension
```

The content of the aforementioned file must be one or more lines, each specifying the fully qualified name of a `ContextManagerExtension` implementation that is provided within the JAR file.

Release Notes for MicroProfile Context Propagation 1.1

MicroProfile Context Propagation Spec PDF MicroProfile Context Propagation Spec HTML MicroProfile Context Propagation Spec Javadocs

Key features:

- CompletableFuture/CompletionStage implementations with predictable thread context and using managed threads for async actions
- Ability to contextualize only specific actions/tasks
- Compatibility with EE Concurrency
- CDI injection as well as builder pattern
- Configurable via MicroProfile Config

To get started, add this dependency to your project:

```
<dependency>
    <groupId>org.eclipse.microprofile.context-propagation</groupId>
    <artifactId>microprofile-context-propagation-api</artifactId>
    <version>1.1</version>
    <scope>provided</scope>
</dependency>
```

Use the fluent builder API to construct a ManagedExecutor:

```
    ManagedExecutor executor = ManagedExecutor.builder()
                       .propagated(ThreadContext.APPLICATION, ThreadContext.CDI)
                       .cleared(ThreadContext.ALL_REMAINING)
                       .maxAsync(5)
                       .build();
```

Then obtain a CompletableFuture or CompletionStage from the ManagedExecutor, and from there use it the same as Java SE:

```
    CompletableFuture<Integer> cf1 = executor.supplyAsync(supplier1)
                                             .thenApplyAsync(function1)
                                             .thenApply(function2);
```

Take care to shut down managed executors once the application no longer needs them:

```
    executor.shutdownNow();
```

Similarly, you can construct `ThreadContext` instances and use them to more granularly control thread propagation to individual stages:

```
    ThreadContext secContext = ManagedExecutor.builder()
                       .propagated(ThreadContext.SECURITY)
                       .cleared(ThreadContext.TRANSACTION)
                       .unchanged(ThreadContext.ALL_REMAINING)
                       .build();
    ...
    CompletableFuture<Void> stage2 = stage1.thenAccept(secContext.contextualConsumer
(consumer1));
```