# Using Software Transactional Memory in Quarkus

Software Transactional Memory (STM) has been around in research environments since the late 1990's and has relatively recently started to appear in products and various programming languages. We won't go into all of the details behind STM but the interested reader could look at this paper. However, suffice it to say that STM offers an approach to developing transactional applications in a highly concurrent environment with some of the same characteristics of ACID transactions, which you've probably already used through JTA. Importantly though, the Durability property is relaxed (removed) within STM implementations, or at least made optional. This is not the situation with JTA, where state changes are made durable to a relational database which supports the X/Open XA standard.

Note, the STM implementation provided by Quarkus is based on the Narayana STM implementation. This document isn't meant to be a replacement for that project's documentation so you may want to look at that for more detail. However, we will try to focus more on how you can combine some of the key capabilities into Quarkus when developing Kubernetes native applications and microservices.

## Why use STM with Quarkus?

Now you may still be asking yourself "Why STM instead of JTA?" or "What are the benefits to STM that I don't get from JTA?" Let's try to answer those or similar questions, with a particular focus on why we think they're great for Quarkus, microservices and Kubernetes native applications. So in no specific order …

- The goal of STM is to simplify object reads and writes from multiple threads/protect state from concurrent updates. The Quarkus STM implementation will safely manage any conflicts between these threads using whatever isolation model has been chosen to protect that specific state instance (object in the case of Quarkus). In Quarkus STM, there are two isolation implementations, pessimistic (the default), which would cause conflicting threads to be blocked until the original has completed its updates (committed or aborted the transaction); then there's the optimistic approach which allows all of the threads to proceed and checks for conflicts at commit time, where one or more of the threads may be forced to abort if there have been conflicting updates.

- STM objects have state but it doesn't need to be persistent (durable). In fact the default behaviour is for objects managed within transactional memory to be volatile, such that if the service or microservice within which they are being used crashes or is spawned elsewhere, e.g., by a scheduler, all state in memory is lost and the objects start from scratch. But surely you get this and more with JTA (and a suitable transactional datastore) and don't need to worry about restarting your application? Not quite. There's a trade-off here: we're doing away with persistent state and the overhead of reading from and then writing (and sync-ing) to the datastore during each transaction. This makes updates to (volatile) state very fast but you still get the benefits of atomic updates across multiple STM objects (e.g., objects your team wrote then calling objects you

inherited from another team and requiring them to make all-or-nothing updates), as well as consistency and isolation in the presence of concurrent threads/users (common in distributed microservices architectures). Furthermore, not all stateful applications need to be durable - even when JTA transactions are used, it tends to be the exception and not the rule. And as you'll see later, because applications can optionally start and control transactions, it's possible to build microservices which can undo state changes and try alternative paths.

- Another benefit of STM is composability and modularity. You can write concurrent Quarkus objects/services that can be easily composed with any other services built using STM, without exposing the details of how the objects/services are implemented. As we discussed earlier, this ability to compose objects you wrote with those other teams may have written weeks, months or years earlier, and have A, C and I properties can be hugely beneficial. Furthermore, some STM implementations, including the one Quarkus uses, support nested transactions and these allow changes made within the context of a nested (sub) transaction to later be rolled back by the parent transaction.

- Although the default for STM object state is volatile, it is possible to configure the STM implementation such that an object's state is durable. Although it's possible to configure Narayana such that different backend datastores can be used, including relational databases, the default is the local operating system file system, which means you don't need to configure anything else with Quarkus such as a database.

- Many STM implementations allow "plain old language objects" to be made STM-aware with little or no changes to the application code. You can build, test and deploy applications without wanting them to be STM-aware and then later add those capabilities if they become necessary and without much development overhead at all.

# Building STM applications

There is also a fully worked example in the quickstarts which you may access by cloning the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or by downloading an archive. Look for the `software-transactional-memory-quickstart` example. This will help to understand how you can build STM-aware applications with Quarkus. However, before we do so there are a few basic concepts which we need to cover.

Note, as you will see, STM in Quarkus relies on a number of annotations to define behaviours. The lack of these annotations causes sensible defaults to be assumed but it is important for the developer to understand what these may be. Please refer to the Narayana STM manual and the STM annotations guide for more details on all of the annotations Narayana STM provides.

> This technology is considered preview.
>
> In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our mailing list or as issues in our GitHub issue tracker.
>
> For a full list of possible extension statuses, check our FAQ entry.

# Setting it up

To use the extension include it as a dependency in your application pom:

```
<dependencies>
    <!-- STM extension -->
    <dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-narayana-stm</artifactId>
      <version>${quarkus.version}</version>
    </dependency>
</dependencies>
```

# Defining STM-aware classes

In order for the STM subsystem to have knowledge about which classes are to be managed within the context of transactional memory it is necessary to provide a minimal level of instrumentation. This occurs by categorising STM-aware and STM-unaware classes through an interface boundary; specifically all STM-aware objects must be instances of classes which inherit from interfaces that themselves have been annotated to identify them as STM-aware. Any other objects (and their classes) which do not follow this rule will not be managed by the STM subsystem and hence any of their state changes will not be rolled back, for example.

The specific annotation that STM-aware application interfaces must use is `org.jboss.stm.annotations.Transactional`. For example:

```
@Transactional
public interface FlightService {
    int getNumberOfBookings();
    void makeBooking(String details);
}
```

Classes which implement this interface are able to use additional annotations from Narayana to tell the STM subsystem about things such as whether a method will modify the state of the object, or what state variables within the class should be managed transactionally, e.g., some instance variables may not need to be rolled back if a transaction aborts. As mentioned earlier, if those annotations are not present then defaults are chosen to guarantee safety, such as assuming all methods will modify state.

```
public class FlightServiceImpl implements FlightService {
    @ReadLock
    public int getNumberOfBookings() { ... }
    public void makeBooking(String details) {...}

    @NotState
    private int timesCalled;
}
```

For example, by using the @ReadLock annotation on the getNumberOfBookings method, we are able to tell the STM subsystem that no state modifications will occur in this object when it is used in the transactional memory. Also, the @NotState annotation tells the system to ignore timesCalled when transactions commit or abort, so this value only changes due to application code.

Please refer to the Narayana guide for details of how to exert finer grained control over the transactional behaviour of objects that implement interfaces marked with the @Transactional annotation.

# Creating STM objects

The STM subsystem needs to be told about which objects it should be managing. The Quarkus (aka Narayana) STM implementation does this by providing containers of transactional memory within which these object instances reside. Until an object is placed within one of these STM containers it cannot be managed within transactions and any state changes will not possess the A, C, I (or even D) properties.

Note, the term "container" was defined within the STM implementation years before Linux containers came along. It may be confusing to use especially in a Kubernetes native environment such as Quarkus, but hopefully the reader can do the mental mapping.

The default STM container (org.jboss.stm.Container) provides support for volatile objects that can only be shared between threads in the same microservice/JVM instance. When a STM-aware object is placed into the container it returns a handle through which that object should then be used in the future. It is important to use this handle as continuing to access the object through the original reference will not allow the STM subsystem to track access and manage state and concurrency control.

```
    import org.jboss.stm.Container;

    ...

    Container<FlightService> container = new Container<>(); ①
    FlightServiceImpl instance = new FlightServiceImpl(); ②
    FlightService flightServiceProxy = container.create(instance);
  ③
```

① You need to tell each Container about the type of objects for which it will be responsible. In this

example it will be instances that implement the FlightService interface.

② Then you create an instance that implements `FlightService`. You should not use it directly at this stage because access to it is not being managed by the STM subsystem.

③ To obtain a managed instance, pass the original object to the STM `container` which then returns a reference through which you will be able perform transactional operations. This reference can be used safely from multiple threads.

# Defining transaction boundaries

Once an object is placed within an STM container the application developer can manage the scope of transactions within which it is used. There are some annotations which can be applied to the STM-aware class to have the container automatically create a transaction whenever a specific method is invoked.

## Declarative approach

If the `@NestedTopLevel` or `@Nested` annotation is placed on a method signature then the STM container will start a new transaction when that method is invoked and attempt to commit it when the method returns. If there is a transaction already associated with the calling thread then each of these annotations behaves slightly differently: the former annotation will always create a new top-level transaction within which the method will execute, so the enclosing transaction does not behave as a parent, i.e., the nested top-level transaction will commit or abort independently; the latter annotation will create a transaction with is properly nested within the calling transaction, i.e., that transaction acts as the parent of this newly created transaction.

## Programmatic approach

The application can programmatically start a transaction before accessing the methods of STM objects:

```
AtomicAction aa = new AtomicAction(); ①

aa.begin(); ②
{
    try {
        flightService.makeBooking("BA123 ...");
        taxiService.makeBooking("East Coast Taxis ..."); ③
        ④
        aa.commit();
        ⑤
    } catch (Exception e) {
        aa.abort(); ⑥
    }
}
```

① An object for manually controlling transaction boundaries (AtomicAction and many other useful

classes are included in the extension). Refer to the javadoc for more detail.

② Programmatically begin a transaction.

③ Notice that object updates can be composed which means that updates to multiple objects can be committed together as a single action. [Note that it is also possible to begin nested transactions so that you can perform speculative work which may then be abandoned without abandoning other work performed by the outer transaction].

④ Since the transaction has not yet been committed the changes made by the flight and taxi services are not visible outside of the transaction.

⑤ Since the commit was successful the changes made by the flight and taxi services are now visible to other threads. Note that other transactions that relied on the old state may or may not now incur conflicts when they commit (the STM library provides a number of features for managing conflicting behaviour and these are covered in the Narayana STM manual).

⑥ Programmatically decide to abort the transaction which means that the changes made by the flight and taxi services are discarded.

# Distributed transactions

Sharing a transaction between multiple services is possible but is currently an advanced use case only and the Narayana documentation should be consulted if this behaviour is required. In particular, STM does not yet support the features described in the Context Propagation guide.