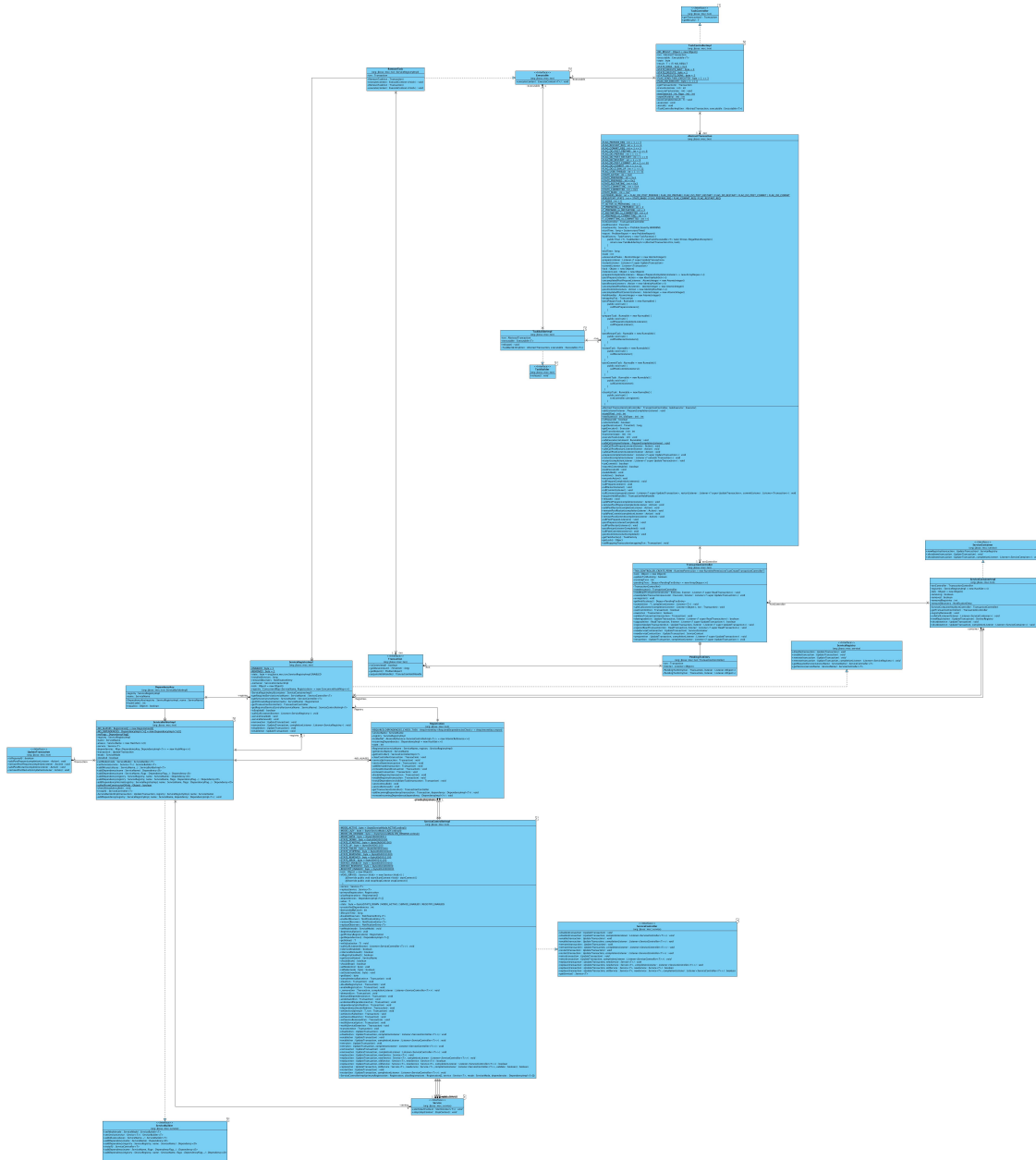


DRAFT - An introduction to the JBoss Modular Service Container: Part 2 - Transaction layer

Weinan Li

Here is the diagram that shows both the transaction layer and the container layer:



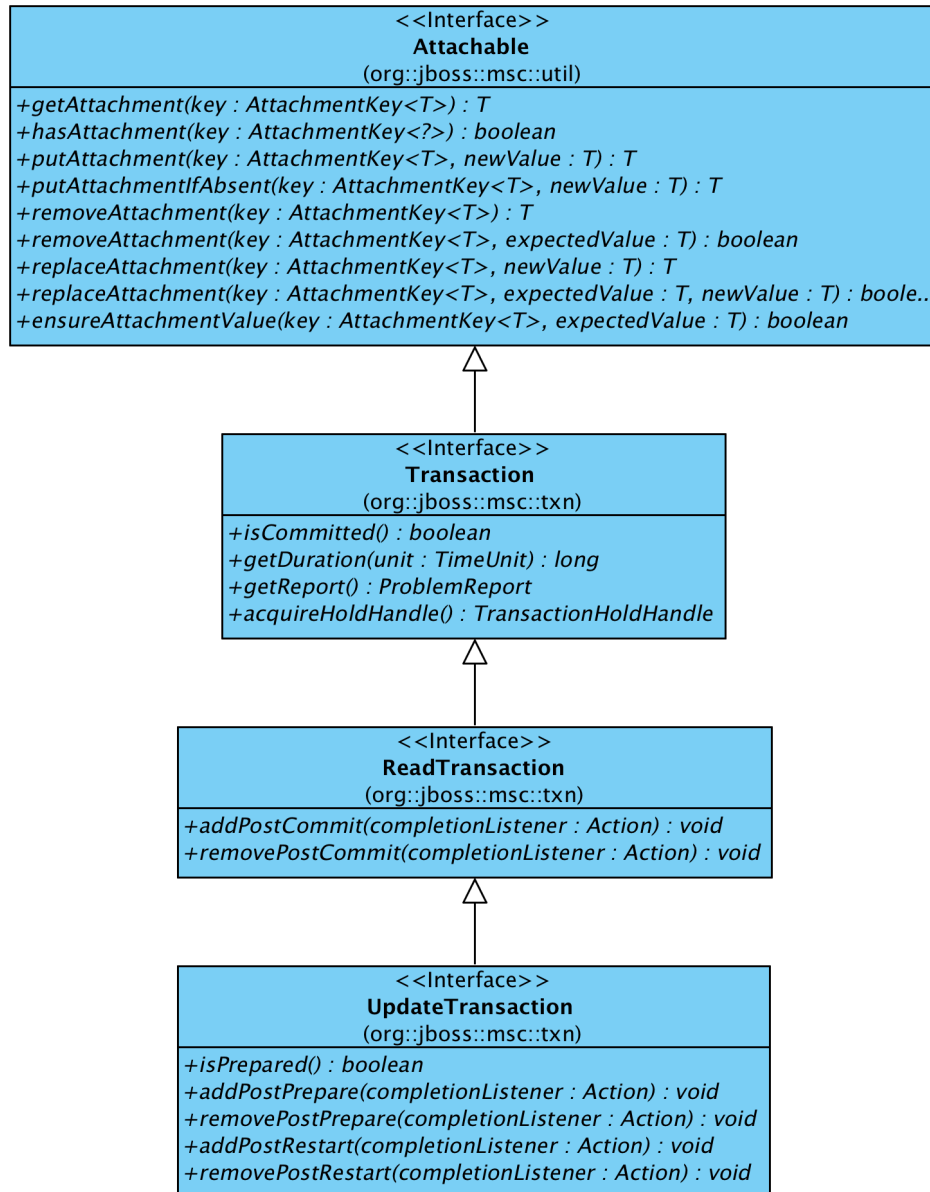
pictures/transaction_container.jpg

From the diagram, we can see whole system can be roughly divided into two parts. The bottom left corner is the container layer, and the up right corner is the transaction layer. The connection point is `ServiceContainerImpl` <-> `TransactionController`.

Here is the diagram that shows the transaction layer only:

pictures/msc/transaction_layer.jpg

There are two kinds of transactions by design. One is ReadTransaction and the other is UpdateTransaction. Here is the relationship of them:



pictures/msc/transaction.png

The above diagram shows that the ReadTransaction interface extends the Transaction interface, and UpdateTransaction extends from the ReadTransaction. When we want to use the transaction controller to create the service container, or adding services into the container, or removing services from the container, then we need the UpdateTransaction as a handle to request the controller to take these actions. We will see the detail usage of the update transaction later.

A sample container

Here is the demo code and my comment:

```
package io.weli.jboss.msc2;

import org.jboss.msc.service.*;
import org.jboss.msc.txn.Transaction;
import org.jboss.msc.txn.TransactionController;
import org.jboss.msc.txn.UpdateTransaction;
import org.jboss.msc.util.CompletionListener;

import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

/**
 * Created by weli on 16/05/2017.
 */
public class Play {
    public static void main(String[] args) {
```

```
        TransactionController controller =
TransactionController.newInstance();
```

The above line will create a new instance of the TransactionController.

```
final CompletionListener<UpdateTransaction> updateListener = new
CompletionListener<>();
```

```
controller.newUpdateTransaction(Executors.newSingleThreadExecutor(),
updateListener);
```

```
final UpdateTransaction transaction =
updateListener.awaitCompletionUninterruptibly();
```

Nearly all the actions in the container are asynchronous. The above code creates a transaction. You can think a transaction as a handler provided by TransactionController to control the whole lifecycle of the service container. The transaction will be used to create service container, register services into container, and it is associated with an executor. We will see the usage of the UpdateTransaction later.

The above transaction creation action is executed asynchronously, so we need to create a CompletionListener in above code to wait for the transaction creation to be done by the controller, and finally we get the UpdateTransaction from the listener.

```
        try {
            ServiceContainer container =
controller.newServiceContainer(transaction);
```

We get the service container from the service controller as shown above. Please see the usage of transaction. The update transaction is associated with every action issued by the controller, and the transaction will be prepared and committed at last to modify the service container statuses. We will see this in following code.

```
ServiceRegistry registry = container.newRegistry(transaction);
```

The service registry is created from the container, and it will be used to include services.

```
ServiceBuilder<Void> serviceBuilder =  
controller.newServiceContext(transaction).addService(registry,  
ServiceName.of("foo"));
```

The above code will add a service named foo into the service registry, and it will return a service builder for the service name to be connected with a real service. The following code defines a service:

```
class FooService implements Service<Void> {  
  
    @Override  
    public void start(StartContext<Void> startContext) {  
        System.out.println("Foo service started.");  
    }  
  
    @Override  
    public void stop(StopContext stopContext) {  
        System.out.println("Foo service stopped.");  
    }  
}
```

The above code defines a FooService that implements the Service interface. It's a service just does nothing but just output some log.

```
serviceBuilder.setService(new FooService());
```

The above code used the service builder to connect the service to its service name.

```
serviceBuilder.install();
```

The above code used the service builder to install the service into registry. Until now we have finished the work to create a service container and put a service registry into the container, and we have created our service and associated it with the registry. Now we should commit the transaction and all the above actions will be persisted in container. Here is the final code:

```
    } finally {  
        final CompletionListener<Transaction> prepareListener = new  
CompletionListener<>();  
        controller.prepare(transaction, prepareListener);  
        prepareListener.awaitCompletionUninterruptibly();
```

```

        final CompletionListener<Transaction> commitListener = new
CompletionListener<>();
        controller.commit(transaction, commitListener);
        commitListener.awaitCompletionUninterruptibly();
    }
}
}

```

We can see the `prepare(...)` and `commit(...)` actions are all related with the update transaction, and these actions are all executed asynchronously. We can see the container relies on the transaction handle to manage its consistency of the internal state. All the actions are associated with the update transaction, and after the update transaction is committed, the actions will alter the container state.

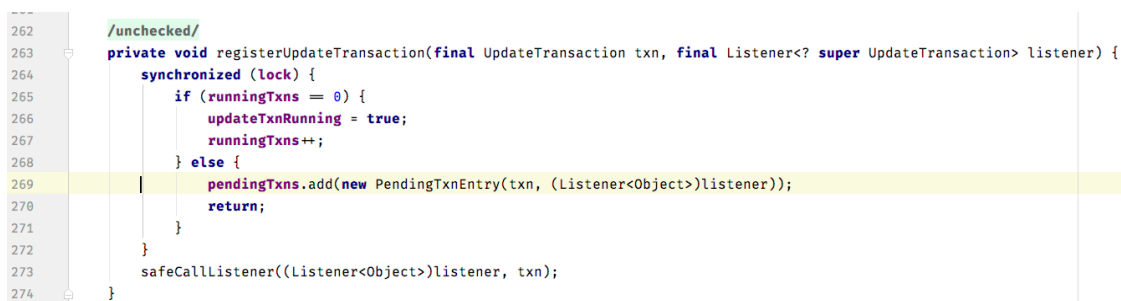
In addition, there is only one update transaction can be in action at one time. It does not allow multiple threads to get many update transactions to alter container internal services and their statuses at one time. If there are many threads requesting the update transactions, only one thread will succeed and others need to wait.

Here is the relative code in `org.jboss.msc.txn.AbstractTransaction.registerUpdateTransaction(...)` method:

```

---
262
263
264
265
266
267
268
269
270
271
272
273
274

```



```

/unchecked/
private void registerUpdateTransaction(final UpdateTransaction txn, final Listener<? super UpdateTransaction> listener) {
    synchronized (lock) {
        if (runningTxns == 0) {
            updateTxnRunning = true;
            runningTxns++;
        } else {
            pendingTxns.add(new PendingTxnEntry(txn, (Listener<Object>)listener));
            return;
        }
    }
    safeCallListener((Listener<Object>)listener, txn);
}

```

pictures/msc/registerUpdateTransaction.png

From the above diagram, we can see if there are more than one update transaction creation requests, the requested update transactions will be put into pendingTxns queue, and there is only one running update transaction allowed and the listener will be called to return the transaction to the caller. The other callers will be blocked if they call their completion listeners.

Here is the complete code:

```

import org.jboss.msc.service.*;
import org.jboss.msc.txn.Transaction;
import org.jboss.msc.txn.TransactionController;
import org.jboss.msc.txn.UpdateTransaction;
import org.jboss.msc.util.CompletionListener;

import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

```

```

/**
 * Created by weli on 16/05/2017.
 */
public class Play {
    public static void main(String[] args) {

        TransactionController controller =
TransactionController.newInstance();

        final CompletionListener<UpdateTransaction> updateListener = new
CompletionListener<>();

        controller.newUpdateTransaction(Executors.newSingleThreadExecutor(),
updateListener);

        final UpdateTransaction transaction =
updateListener.awaitCompletionUninterruptibly();

        class FooService implements Service<Void> {

            @Override
            public void start(StartContext<Void> startContext) {
                System.out.println("Foo service started.");
            }

            @Override
            public void stop(StopContext stopContext) {
                System.out.println("Foo service stopped.");
            }
        }

        try {
            ServiceContainer container =
controller.newServiceContainer(transaction);
            ServiceRegistry registry = container.newRegistry(transaction);
            ServiceBuilder<Void> serviceBuilder =
controller.newServiceContext(transaction).addService(registry,
ServiceName.of("foo"));
            serviceBuilder.setService(new FooService());
            serviceBuilder.install();
            controller.downgrade(transaction, null);

        } finally {
            final CompletionListener<Transaction> prepareListener = new
CompletionListener<>();
            controller.prepare(transaction, prepareListener);
            prepareListener.awaitCompletionUninterruptibly();
        }
    }
}

```



```
        final CompletionListener<Transaction> commitListener = new  
CompletionListener<>();  
        controller.commit(transaction, commitListener);  
        commitListener.awaitCompletionUninterruptibly();  
    }  
}  
}
```