Command Replay

Table of Contents

The "replay" library (introduced in 1.16.1) allows commands on a *master* server to be replayed against a *slave* server.

Typical use case

One use case for this is for regression testing or A/B testing of two different versions of an application (eg after a system upgrade, or after a refactoring of business logic).

The *master* and *slave* are set up in a test environment with the *master* pointing at a database backup a few days in the future from the *slave*. For example, the *master* database could be restored with a copy of Wednesday night backup, while the *slave* could be restored with a copy of Monday night's backup. Master is therefore two days "ahead" of the *slave*.

The *master* instance is just a regular instance of the application, but the *slave* instance is started with a number of additional configuration parameters. These are used to allow the *slave* to copy commands from the *master* using the REST API, and then to execute them (similar to the way that background commands are executed).

An alternative configuration is to use production instance as *master*, with the *slave* being in the test environment. The *slave* instance would be restored from the previous night's database backup. As the online users make changes to the production (*master*), these are applied to the *slave* also.

Obviously using production as *master* introduces additional load: the *slave* continually polls the *master* for new commands and so there will be additional load that should be taken into account.

Design

The replication of `Command`s from *master* is *slave* is based on the idea of a "high water mark" (HWM). This represents the most recently executed command on the *slave*. The algorithm is as follows:

- the *slave* polls the *master* for the "next" **foreground** command after its HWM (by timestamp);
- if there is one then this is copied over the *slave* and persisted, at which point it becomes the new HWM;
- the command is then executed, and any background commands that might have been created as a result of this are also executed immediately.

Note that background commands are never replicated. When the replicated foreground command is executed, it will create its own background commands on the *slave* and these are what are executed.

If the command was replayed successfully then the *slave* loops around looking for the next command, continuing until all commands have been copied over from *master* to *slave*, one by one.

But, if the command failed to replay, then the *slave* stops until the issue is resolved. More on this topic below.

"Patching" Mementos

On master the persisted commands (that is, <code>CommandJdo</code> entity instances) hold an XML memento that is a serialization of <code>CommandDto</code> (as per the <code>cmd.xsd</code> schema). This serializes the state of values and of entity references automatically. However, for space reasons it does _not serialize any <code>Blob</code> or <code>Clob</code> arguments; the values are simply missing in the XML.

To address this a `CommandDtoProcessor` implementation can be specified on a case-by-case basis. This interface is defined as:

```
public interface CommandDtoProcessor {
    CommandDto process(final Command command, CommandDto commandDto);
}
```

This is specified using the `@Action#commandDtoProcessor()` or `@Property#commandDtoProcessor()` attribute. For example:

```
@Action(domainEvent = IncomingDocumentRepository.UploadDomainEvent.class,
            commandDtoProcessor = DeriveBlobFromReturnedDocumentArg0.class
    )
    @MemberOrder(sequence = "3")
    public Document upload(final Blob blob) {
        final String name = blob.getName();
        final DocumentType type = DocumentTypeData.INCOMING.findUsing
(documentTypeRepository);
        final ApplicationUser me = meService.me();
        String atPath = me != null ? me.getAtPath() : null;
        if (atPath == null) {
            atPath = "/";
        }
        return incomingDocumentRepository.upsertAndArchive(type, atPath, name, blob);
    }
```

where `DeriveBlobFromReturnedDocumentArg0` (simplified) is:

```
public class DeriveBlobFromReturnedDocumentArg0
        extends DeriveBlobFromReturnedDocumentAbstract {

    @Override
    public CommandDto process(
            final Command command,
            final CommandDto commandDto) {
        final Bookmark result = command.getResult();
        if(result == null) return commandDto;
        final Document document = bookmarkService.lookup(result, Document.class);
        if (document != null) {
            ParamDto paramDto = getParamDto(commandDto, 0);
            CommonDtoUtils.setValueOn(
                paramDto, ValueType.BLOB, document.getBlob(), bookmarkService);
        }
        return commandDto;
    }
    @Inject
    BookmarkService bookmarkService;
}
```

So, in effect, the `CommandDtoProcessor` allows the `CommandDto` to be "patched" with the missing information that was not persisted in the XML. This missing information *does* end up being persisted in the command's memento on the *slave*, but is not persisted in the *master*.

If the `CommandDtoProcessor` returns null, then the `Command` is effectively excluded from the result set, and so is not replicated. The `CommandDtoProcessor.Null` class is such an implementation. For example:

```
@Action(
        commandDtoProcessor = CommandDtoProcessor.Null.class
)
public TurnoverImportManager uploadSpreadsheet(
        @Parameter(fileAccept = ".xlsx")
        final Blob spreadsheet) {
    ...
}
```

Defining Failures

A "replay failure" does not necessarily mean that executing the command threw an exception; it could be the case that the command also threw an exception on the *master*. In such a case, "replay success" means that the same exception was thrown.

So instead, for flexibility, there is a pluggable SPI that allows "command replay analysers" to be installed. If any such analyser detects an error, then the command replay is marked as failed and replication ceases.

This SPI is defined as:

```
public interface CommandReplayAnalyser {
    String analyzeReplay(
            final Command command,      ①
            final CommandDto dto);      ②
}
```

① The `Command` that has just been executed. This will have its `startedAt`, `completedAt` properties set, and either a `result` (unless void) or an `exception` (if any occurred).

② The DTO obtained from the `Command`, being the memento copied over from the *master*. This is passed in as a small performance optimisation to save each analyser having to unmarshal the DTO from XML.

The command replay module provides a number of pre-built implementations.

- `CommandReplayAnalyserResultStr`

  If both *master* and *slave* produces a result, then checks they are the same

- `CommandReplayAnalyserExceptionStr`

  If both *master* and *slave* produces an exception, then checks that the first 500 characters of that exception's stack trace are the same.

- `CommandReplayAnalyserNumberBackgroundCommands`

  Checks that the number of background commands created by both *master* and *slave* are the same.

All of these in-built implementations can be disabled, by setting the appropriate configuration option. For example, to disable the ResultStr analyser, add:

- `isis.services.CommandReplayAnalyserResultStr.analysis=disabled`

Similarly for the other analysers.

New implementations of this SPI can be also registered simply by annotating as a domain service within a module loaded by the application's manifest.

To facilitate this the replay module also defines a more general SPI, called `CommandDtoProcessorService`. This is almost identical to the `CommandDtoProcessor` (which is defined on an action-by-action basis), except it runs for all commands:

```
public interface CommandDtoProcessorService {
    CommandDto process(Command command, CommandDto commandDto);
}
```

This can therefore be used to capture additional information within the memento on the *master*, and then have the replay analyser on the *slave* check. In fact, the same service implementation can implement both the `CommandDtoProcessorService` and `CommandReplayAnalyser` SPIs; the in-built `CommandReplayAnalyserNumberBackgroundCommands` does precisely this.

Detecting Failures

If the *slave* hits an issue when executing the command, then the process stops. The development team can check for a failure by either:

- executing "findReplayHwmOnSlave" action (on the *slave*)

  This returns the HWM with a mixed-in collection showing the most recently executed commands; or

- monitoring the console/log of the *slave*

  The LOG for `ReplayableCommandExecution` class will write out an info message if it has hit a replay error.

If a command fails to replay then the development team can either exclude it or they can retry it (using mixin actions on the `CommandJdo` entity). If a command is excluded then the *slave*'s job will continue onto the next command. If a command is retried then the *slave* simply removes details of the failure condition, and attempts again.

Hints and Tips

There are several reasons why a command may fail to replay.

- First and most obviously, the *master* and *slave* must be compatible with each other.

If the *slave* has changed implementation (eg has a bug fix or new feature) which produces different results, then replaying a command is expected to fail to replay. Indeed, this could be considered a successful test because it confirms that the *slave* implementation is indeed different. Providing appropriate implementations of `CommandReplayAnalyser` will allow useful information about the variation to be obtained.

- the `Command` memento captures the target and object references as "OIDs", which encodes the object type (alias for the concrete class) as well as its identity.

  If database-generated surrogate/artificial keys are in use, then sequencing becomes important. Indeed, this is the main reason that the replication job only works one command at a time, to minimize the chance that the replay happens in a different order to that of the *slave*.

  For more reliability of replays (but arguably more complexity), consider using business natural keys.

- A related and more severe issue is if the identity of objects is specified using a random UUID. Unless the generation of such UUIDs is made deterministic, replaying such commands is unlikely to be do-able.

Implementation

Classpath

Update your classpath by adding this dependency in your webapp project's `pom.xml`:

```
<dependency>
    <groupId>org.isisaddons.module.command</groupId>
    <artifactId>isis-module-command-replay</artifactId>
    <version>1.16.1</version>
</dependency>
```

Check for later releases by searching Maven Central Repo.

For instructions on how to use the latest `-SNAPSHOT`, see the contributors guide.

Bootstrapping

In the `AppManifest`, update its `getModules()` method, eg:

```
@Override
public List<Class<?>> getModules() {
    return Arrays.asList(
            ...
            org.isisaddons.module.command.replay.CommandReplayModule.class,
    );
}
```

Configuration Properties

The *master* and the *slave* are distinguished simply by the presence of a number of configuration properties on the *slave*:

- `isis.command.replay.master.user`
- `isis.command.replay.master.password`
- `isis.command.replay.master.baseUrl`

  For example `http://localhost:8080/restful/`

These allow the *slave* to query the *master* through its REST API.

There are also two optional configuration properties:

- `isis.command.replay.master.baseUrlEndUser`

  If configured, this contributes a mixin on `CommandJdo` entity to open a new web browser tab on

the *master* for the same entity. This can be helpful for debugging a command that has failed to replay.

For example `http://localhost:8080`.

Note that this can't necessarily be derived from the `isis.command.replay.master.baseUrl` configuration property; the *slave* may communicate with *master* using a different network to the one used by the end-user's web browser (eg if Docker stacks are in use).

- `isis.command.replay.master.batchSize`

  The *slave* executes only one command at a time, but it actually requests to fetch several commands. This is because (because of the `CommandDtoProcessor` SPI) not every command is necessarily replicated. The batch size is intended to ensure that "enough" commands are requested that at least one is returned.

On the *slave* it might sometimes be worth specifying a different manifest (using `isis.appManifest` configuration property), for example to disable publishing module.

Quartz job

The commands themselves are replayed through the `RunBackgroundCommandsWithReplicationAndReplayJob` job, scheduled using Quartz. This job *replaces* the `RunBackgroundCommandsJob` that is usually used to run background jobs; it runs on both *master* and *slave*:

- if the job is run on the *master* then it simply does the same thing as the old `RunBackgroundCommandsJob`
- if the job is run on the *slave*, then it uses the REST API to copy commands from the *master* to *slave*, and executes. This continues until all commands have been copied and executed.

The job determines whether it is running on *master* or *slave* simply by checking for the presence of the configuration properties listed above.

The quartz configuration should be something like:

quartz-config.xml

```
<job-scheduling-data ... version="1.8">
  <schedule>
    ...
    <job>
      <name>RunBackgroundCommandsWithReplicationAndReplayJob</name>
      <group>CommandModule</group>
      <description>
        Retrieves executed commands (as XML) from master and
        persists as new unexecuted commands on slave
      </description>
      <job-class>
org.isisaddons.module.command.replay.impl.RunBackgroundCommandsWithReplicationAndRepla
yJob
      </job-class>
      <job-data-map>
        <entry>
          <key>user</key>
          <value>scheduler_user</value>
        </entry>
        <entry>
          <key>roles</key>
          <value>admin_role</value>
        </entry>
      </job-data-map>
    </job>
    <trigger>
      <cron>
        <name>RunBackgroundJobsEvery10Seconds</name>
        <job-name>RunBackgroundCommandsWithReplicationAndReplayJob</job-name>
        <job-group>CommandModule</job-group>
        <cron-expression>0/10 * * * * ?</cron-expression>
      </cron>
    </trigger>
    ...
  </schedule>
</job-scheduling-data>
```

Manual Debugging

By default the quartz job (details below) which replicates commands starts immediately when the webapp is bootstrapped, polling for commands to replicate from master to slave.

Optionally the job can be paused by implementing the `ReplayCommandExecutionController` SPI:

```
public interface ReplayCommandExecutionController {
    enum State {
        RUNNING,
        PAUSED
    }
    State getState();       ①
}
```

① The current state, or `null` if the service implementing this SPI has not yet been initialized.

If an implementation has been provided then the state will be checked. Unless "RUNNING" is returned, the replay job will do nothing.

This can be useful if wishing to manually debug replication. With the replay job paused, commands can instead be manually copied from *master* to *slave* using the "download" and "upload" actions, and then executed one at a time:

- On the ***slave***, use `findReplayHwmOnSlave` action (from the `CommandReplayOnSlaveService`) to obtain the "high water mark".

  If the backup has just been restored, then this will be the most recent foreground command that was run on the production system before the backup was taken.

  If any commands have been replayed, then this will be that most recent replayed command.

- Use the "replayNext" mixin action on the `Command`. This performs the same general steps as the replay job, but for a single command:
  - it fetches the next command via REST
  - persists the command
  - executes the command
  - executes any resultant background commands
  - analyses the result.

This "replayNext" action also returns the next command retrieved. Therefore, the administrator/developer can continue to invoke "replayNext" on each successive command.

There are also a couple of other actions that are worth being aware of:

- On the *master*, the `downloadCommandsOnMasterSince` action returns the commands in XML format, where the "since" is intended to be the transaction id of the high water mark obtained from the

slave

- On the *slave*, the `uploadCommandsToSlave` action allows XML commands to be persisted on the slave.

Dependencies

Maven can report modules dependencies using:

```
mvn dependency:list -o -pl modules/spi/command/replay -D excludeTransitive=true
```

which, excluding Apache Isis itself, returns these compile/runtime dependencies on other modules in the Incode platform:

```
org.incode.module.jaxrsclient:incode-module-jaxrsclient-dom
org.isisaddons.module.quartz:isis-module-quartz-dom
```

For further details on these dependencies, see:

- JAX-RS Client library
- Quartz extension