

Command SPI Implementation + Replay

Table of Contents

Screenshots	2
Commands.....	2
Commands created for action invocations	3
Background Commands using the Background Service	5
Background Commands scheduled implicitly	8
How to configure/use	11
Classpath	11
Bootstrapping.....	11
Configuration Properties.....	11
API	13
CommandService	13
BackgroundCommandService	13
BackgroundCommandExecutionFromBackgroundCommandServiceJdo	14
Supporting Services and Mixins.....	15
Known issues	17
Related Modules/Services	18
Dependencies	19

This module (`isis-module-command`) provides an implementation of Apache Isis' `CommandService` SPI that enables action invocations (``Command`s`) to be persisted using Apache Isis' own (JDO) objectstore. This supports two main use cases:

- profiling: determine which actions are invoked most frequently, what is the elapsed time for each command)
- enhanced auditing: the command represents the "cause" of a change to the system, while the related [Audit module](#) captures the "effect" of the change. The two are correlated together using a unique transaction Id (a GUID).

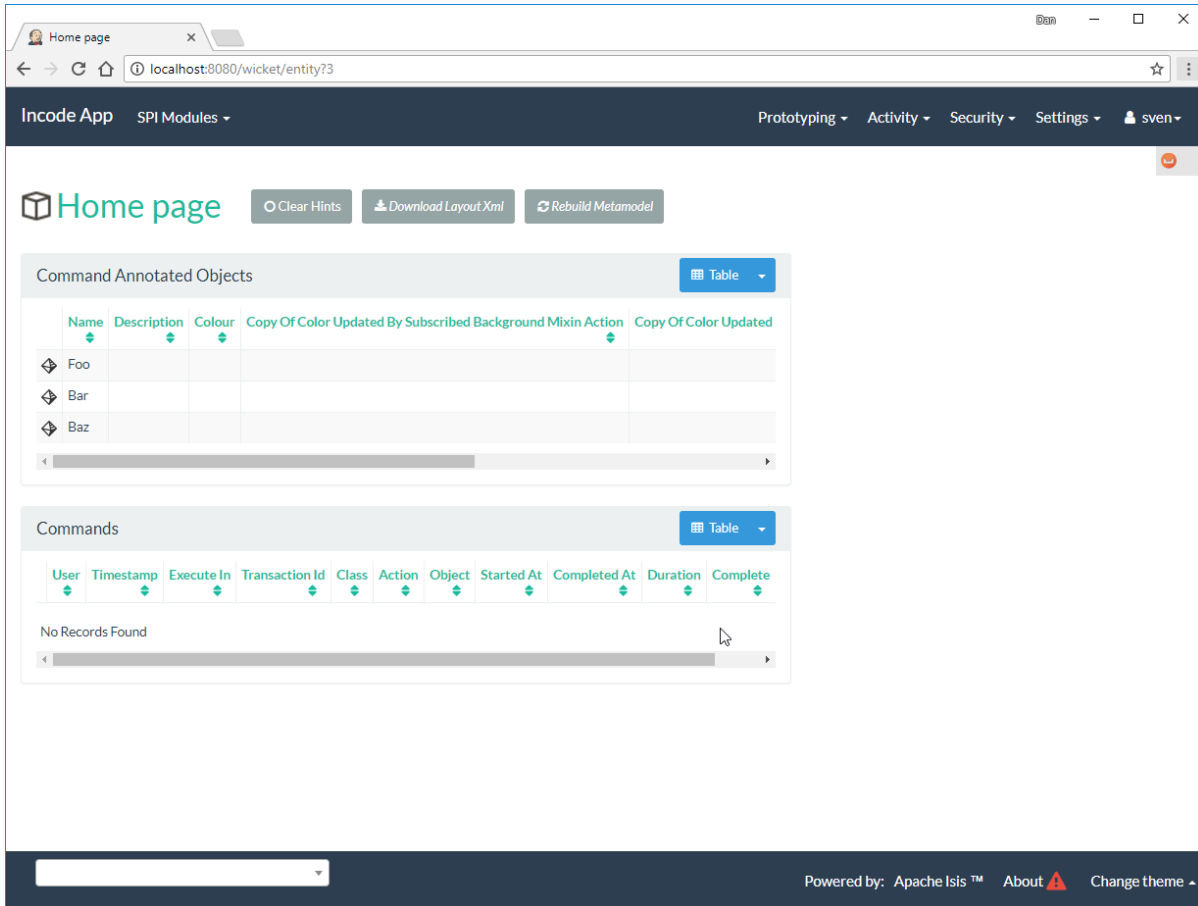
In addition, this module also provides an implementation of the `BackgroundCommandService` API. This enables commands to be persisted but the action not invoked. A scheduler can then be used to pick up the scheduled background commands and invoke them at some later time. The module provides a subclass of the `BackgroundCommandExecution` class (in Isis core) to make it easy to write such scheduler jobs.

As of [1.16.1](#) this module also provides a companion "replay" library. This allows commands on a master server to be replayed against a slave. 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). For more details, see the [replay library](#) page.

Screenshots

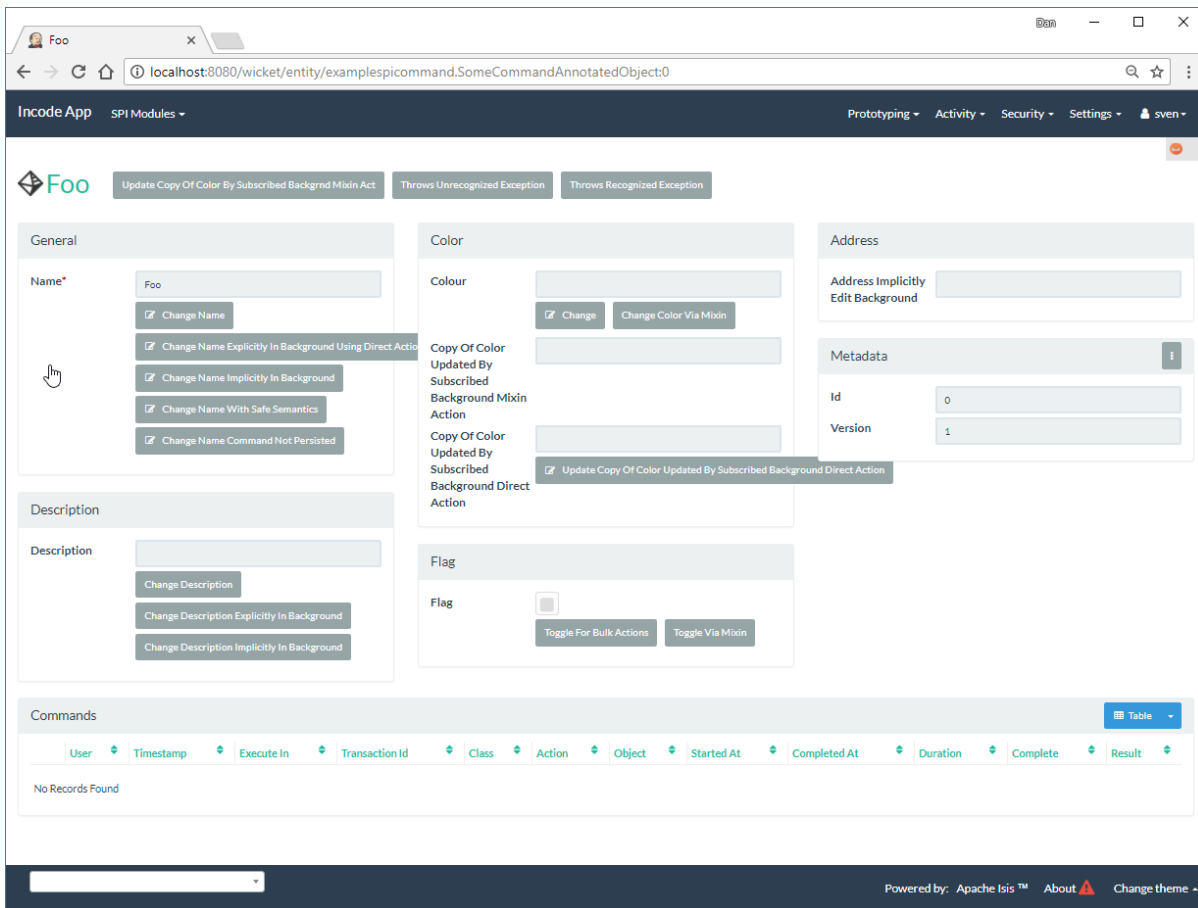
The module's functionality can be explored by running the [quickstart with example usage](#) using the `org.incode.domainapp.example.app.modules.ExampleDomSpiCommandAppManifest`.

A home page is displayed when the app is run:



Commands

Commands can be associated with any object (as a polymorphic association utilizing the `BookmarkService`), and so the demo app lists the commands associated with the example entity:

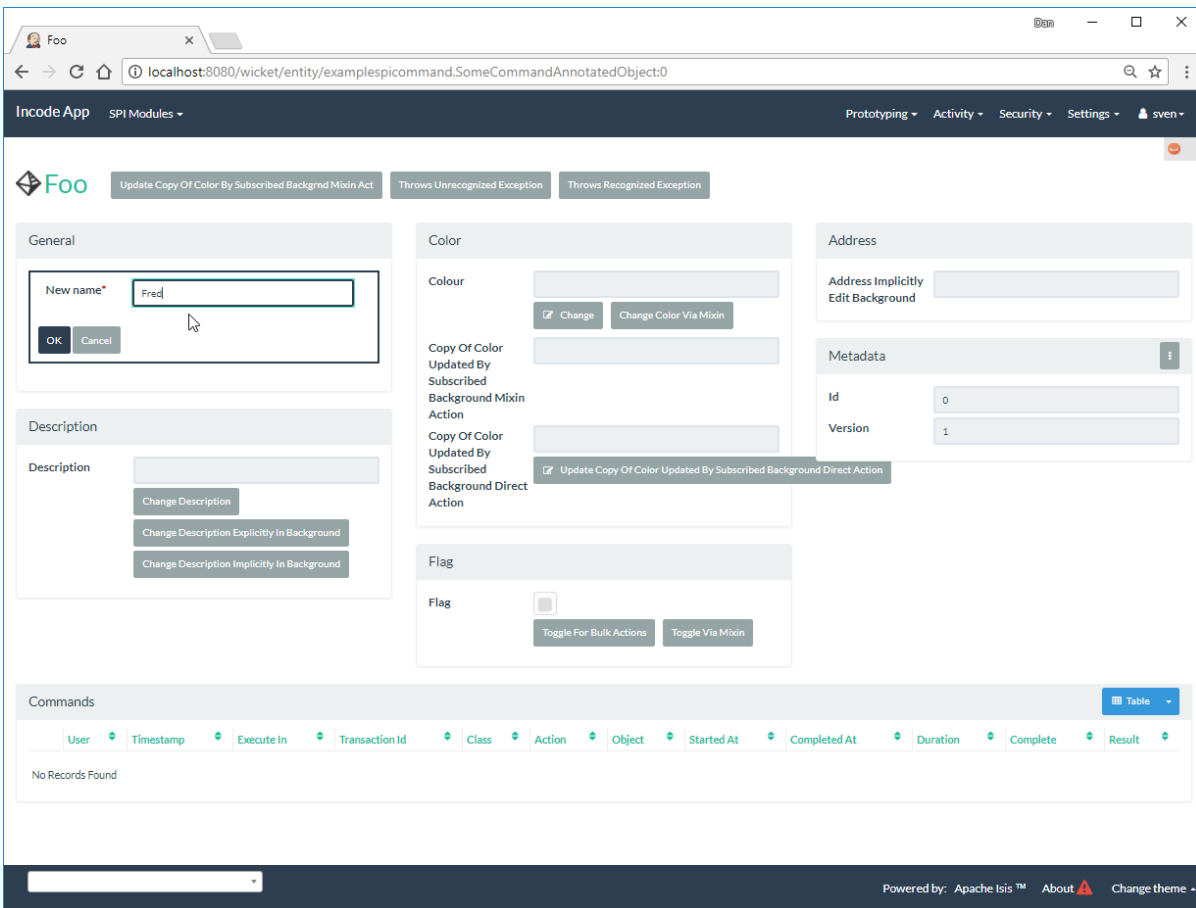


Commands created for action invocations

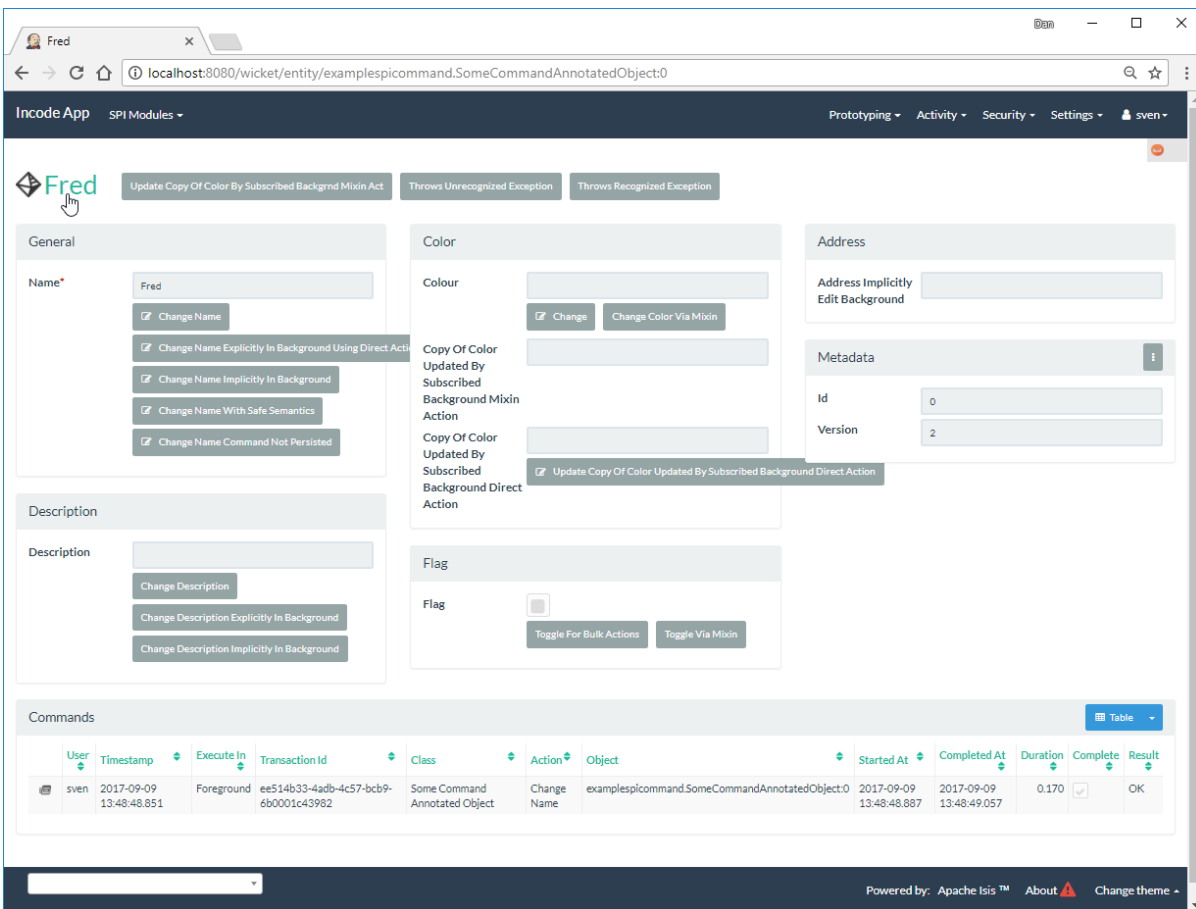
In the example entity the `changeName` action is annotated with `@Action(command=CommandReification.ENABLED)`:

```
@Action(
    semantics = SemanticsOf.IDEMPOTENT,
    command = CommandReification.ENABLED
)
public SomeCommandAnnotatedObject changeName(final String newName) {
    setName(newName);
    return this;
}
```

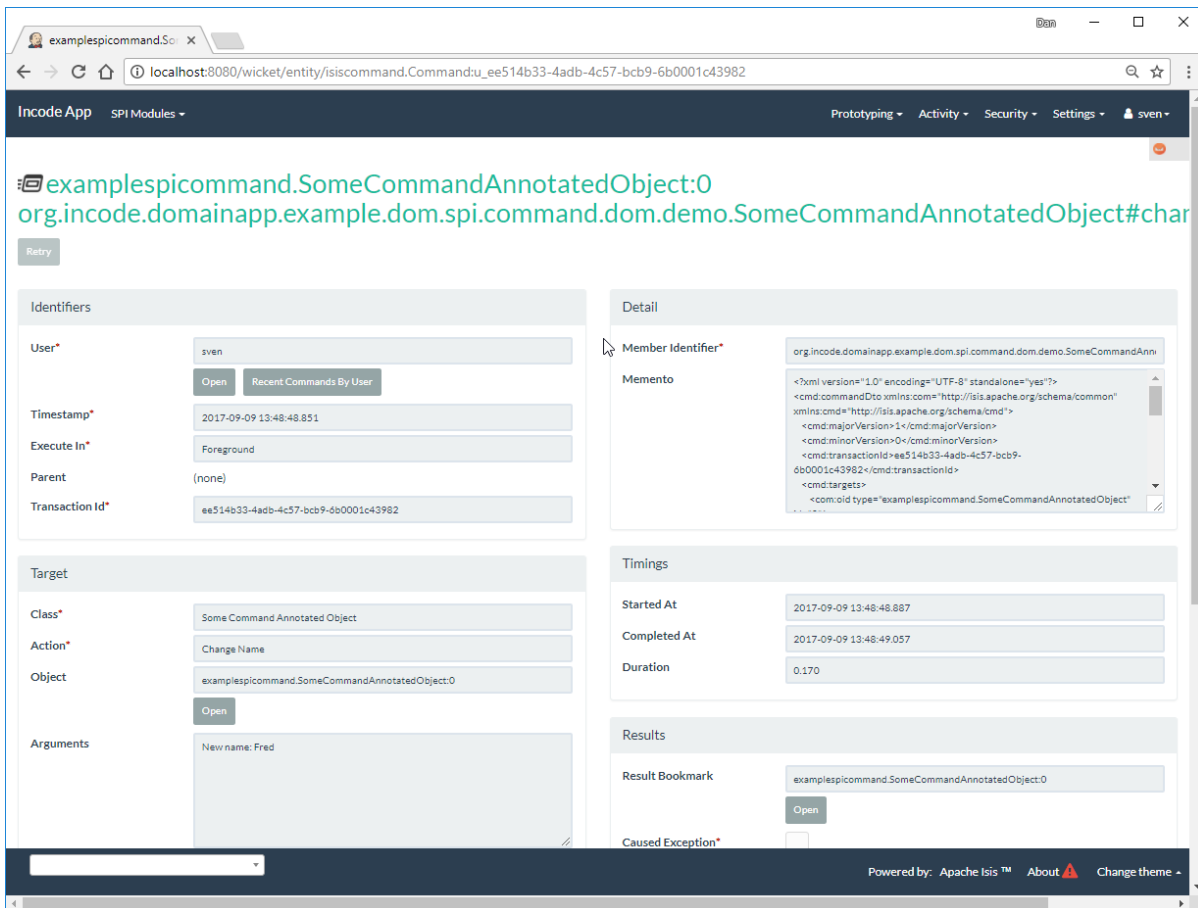
which means that when the `changeName` action is invoked with some argument:



then a command object is created:



identifying the action, captures the target and action arguments, also timings and user:



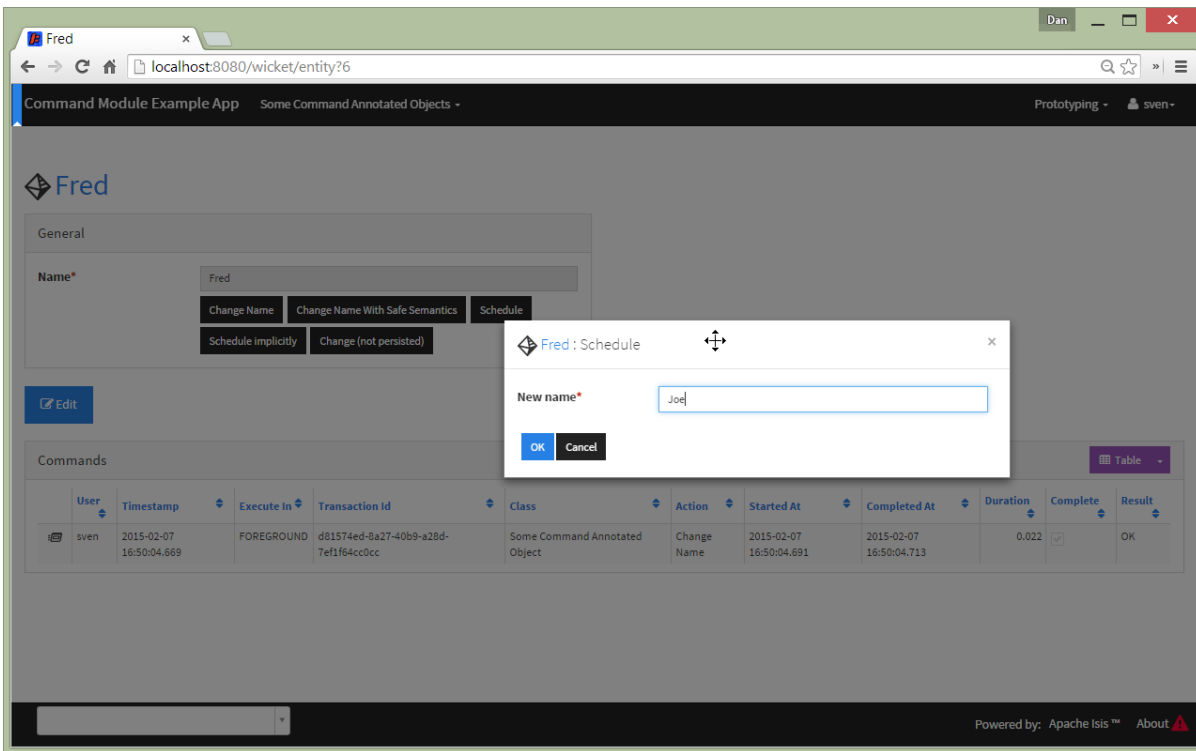
The remaining screenshots below **do** demonstrate (some of) the functionality of this module, but are out of date in that they are taken from the original isisaddons/incodenhq module (prior to being amalgamated into the incode-platform).

Background Commands using the Background Service

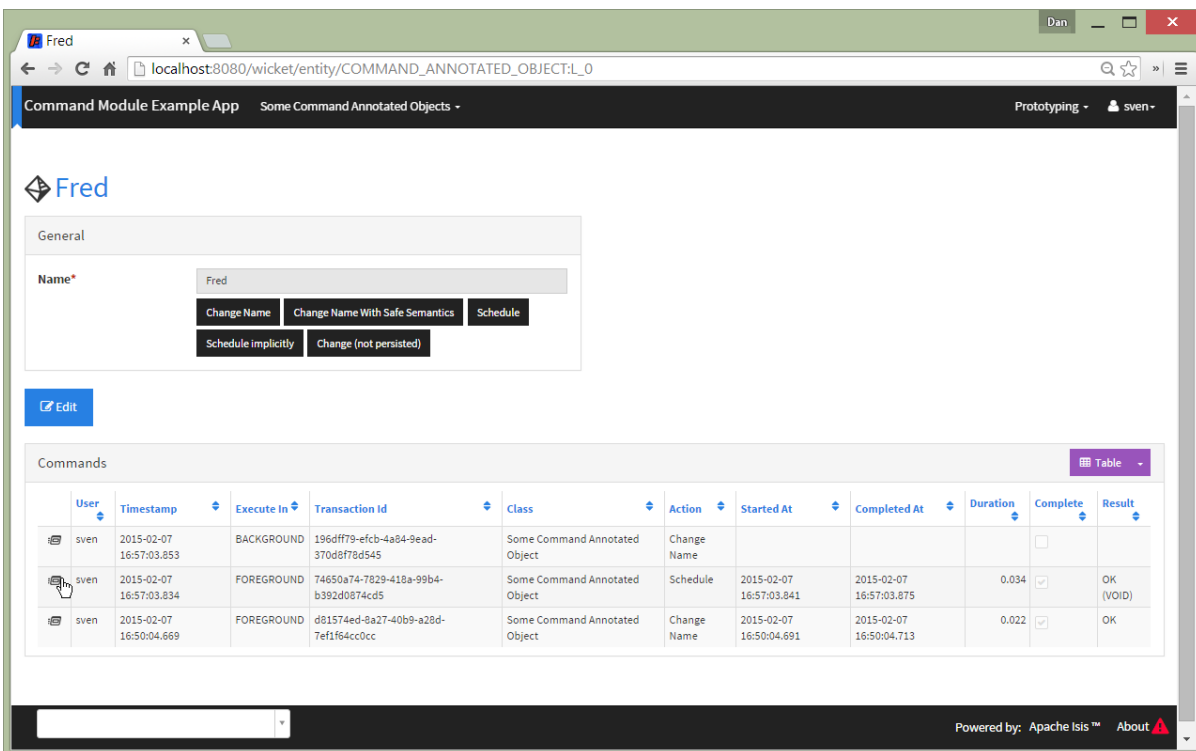
Commands are also the basis for Isis' support of background commands. The usual way to accomplish this is to call Apache Isis' **BackgroundService**:

```
@Action(
    semantics = SemanticsOf.IDEMPOTENT,
    command = CommandReification.ENABLED
)
@ActionLayout(
    named = "Schedule"
)
public void changeNameExplicitlyInBackground(
    @ParameterLayout(named = "New name")
    final String newName) {
    backgroundService.execute(this).changeName(newName);
}
```

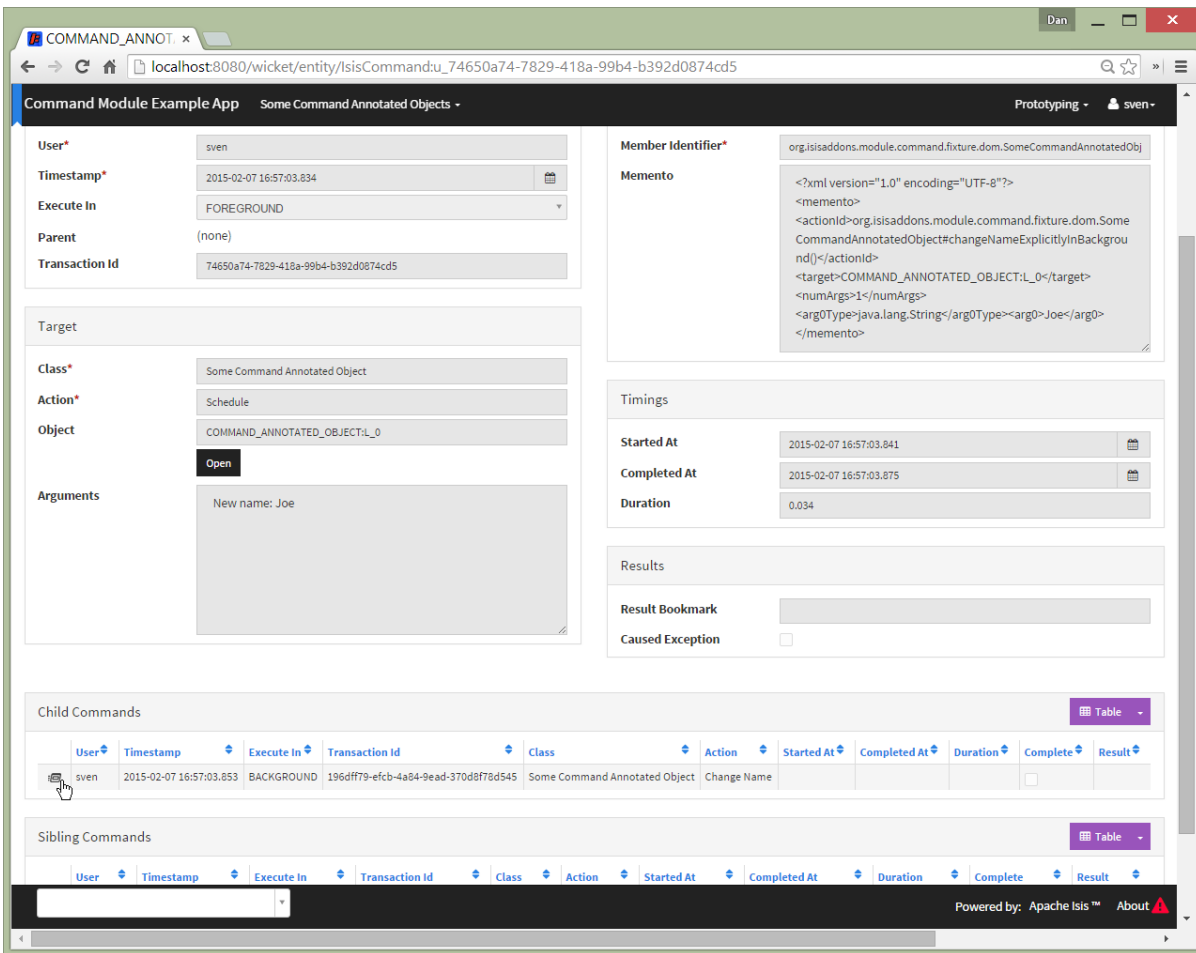
In the screenshots below the action (labelled "Schedule" in the UI) is called with arguments:



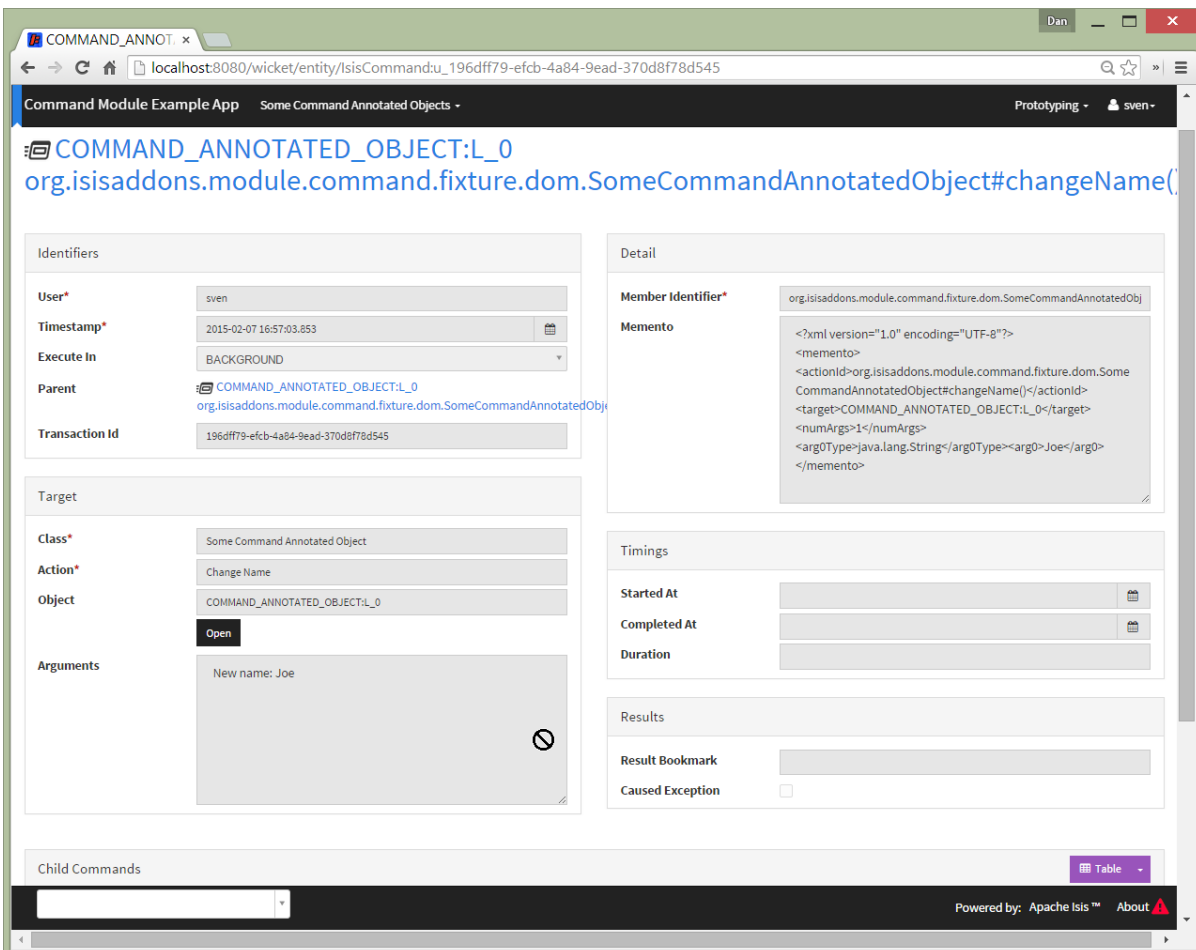
This results in *two* persisted commands, a foreground command and a background command:



The foreground command has been executed:



The background command has not (yet):



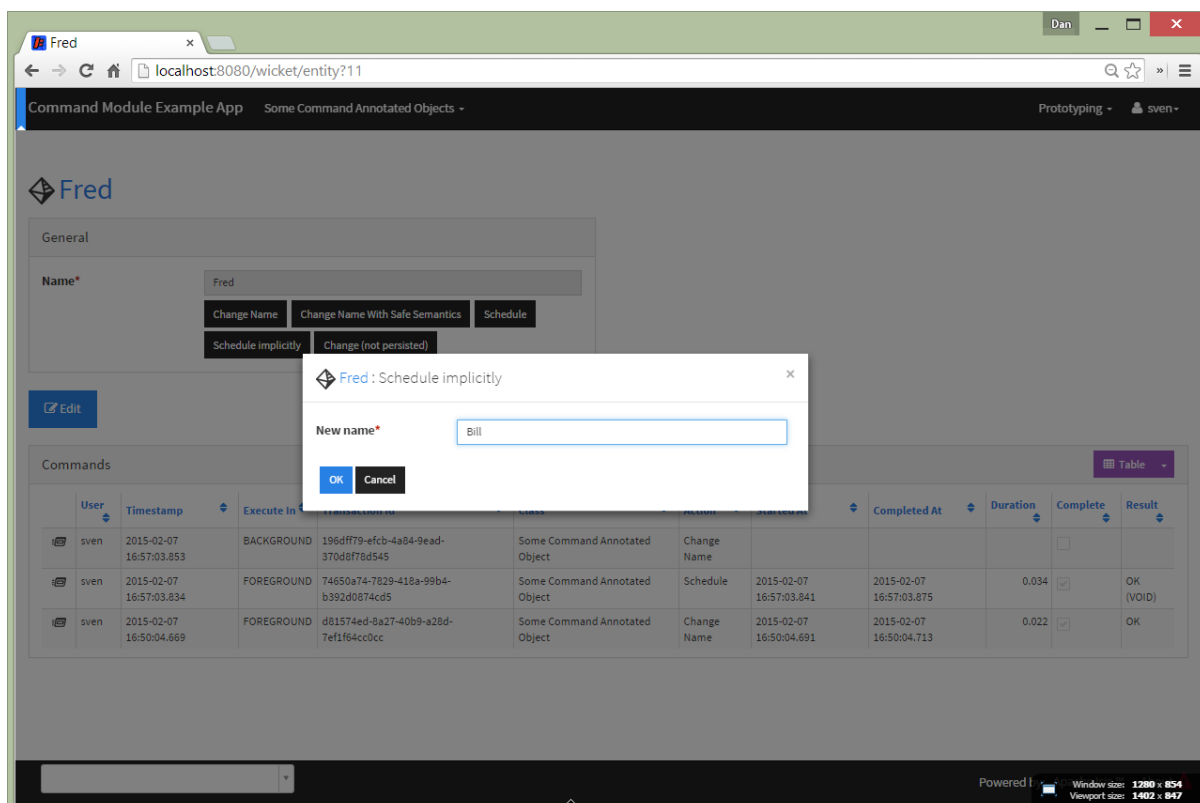
The background command can then be invoked through a separate process, for example using a Quartz Scheduler. The module provides the `BackgroundCommandExecutionFromBackgroundCommandServiceJdo` class which can be executed periodically to process any queued background commands; more information below.

Background Commands scheduled implicitly

The other way to create background commands is implicitly, using `@Action(commandExecuteIn=CommandExecuteIn.BACKGROUND)`:

```
@Action(  
    semantics = SemanticsOf.IDEMPOTENT,  
    command = CommandReification.ENABLED,  
    commandExecuteIn = CommandExecuteIn.BACKGROUND  
)  
@ActionLayout(  
    named = "Schedule implicitly"  
)  
public SomeCommandAnnotatedObject changeNameImplicitlyInBackground(  
    @ParameterLayout(named = "New name")  
    final String newName) {  
    setName(newName);  
    return this;  
}
```

If invoked Apache Isis will gather the arguments as usual:

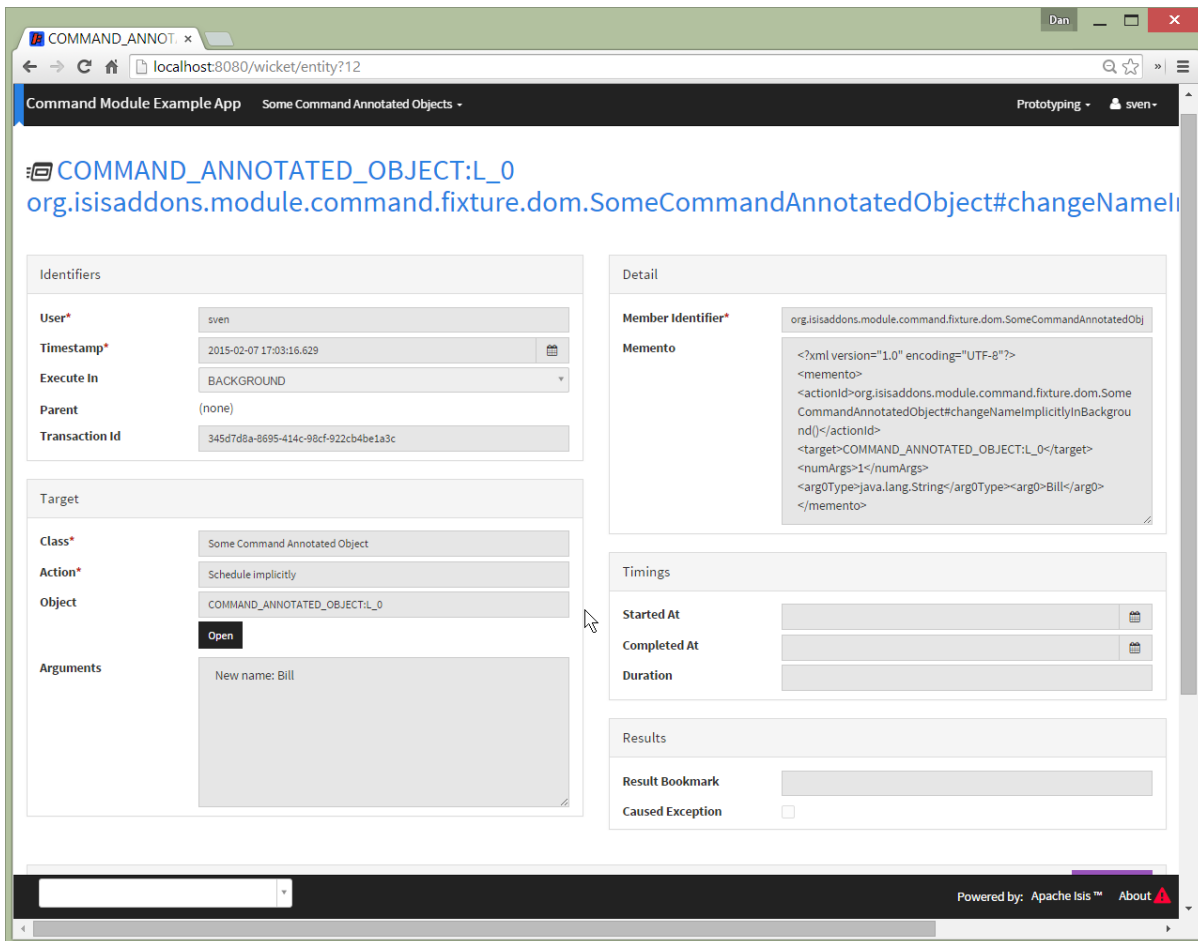


The screenshot shows the Apache Isis web interface for a 'Fred' entity. The main view displays the entity's name as 'Fred' and several buttons: 'Change Name', 'Change Name With Safe Semantics', 'Schedule', 'Schedule implicitly', and 'Change (not persisted)'. A modal dialog box titled 'Fred: Schedule implicitly' is open, showing a 'New name*' field with the value 'Bill' and 'OK' and 'Cancel' buttons. Below the dialog, a table of commands is visible, showing a list of executed commands with columns for User, Timestamp, Execute In, Command, and Result.

User	Timestamp	Execute In	Command	Result
svn	2015-02-07 16:57:03.853	BACKGROUND	196dff79-efcb-4a94-9ead-370d8f78d545 Some Command Annotated Object	Change Name
svn	2015-02-07 16:57:03.834	BACKGROUND	74650a74-7829-418a-99b4-b392d0874cd5 Some Command Annotated Object	Schedule
svn	2015-02-07 16:50:04.669	BACKGROUND	d81574ed-9a27-40b9-a28d-7ef1f64cc0cc Some Command Annotated Object	Change Name

but then does *not* invoke the action, but instead creates the and returns the persisted background

command:



As the screenshot below shows, with this approach only a single background command is created (no foreground command at all):

Fred

localhost:8080/wicket/entity?13

Command Module Example App Some Command Annotated Objects - Prototyping - sven

Fred

General

Name* Fred

Change Name Change Name With Safe Semantics Schedule

Schedule implicitly Change (not persisted)

Edit

Commands Table

	User	Timestamp	Execute In	Transaction Id	Class	Action	Started At	Completed At	Duration	Complete	Result
	sven	2015-02-07 17:03:16.629	BACKGROUND	345d7d8a-8695-414c-98cf-922cb4be1a3c	Some Command Annotated Object	Schedule implicitly				<input type="checkbox"/>	
	sven	2015-02-07 16:57:03.853	BACKGROUND	196dff79-efcb-4a84-9ead-370d8f78d545	Some Command Annotated Object	Change Name				<input type="checkbox"/>	
	sven	2015-02-07 16:57:03.834	FOREGROUND	74650a74-7829-418a-99b4-b392d0874cd5	Some Command Annotated Object	Schedule	2015-02-07 16:57:03.841	2015-02-07 16:57:03.875	0.034	<input checked="" type="checkbox"/>	OK (VOID)
	sven	2015-02-07 16:50:04.669	FOREGROUND	d81574ed-8a27-40b9-a28d-7ef1f64cc0cc	Some Command Annotated Object	Change Name	2015-02-07 16:50:04.691	2015-02-07 16:50:04.713	0.022	<input checked="" type="checkbox"/>	OK

Powered by: Apache Isis™ About

How to configure/use

Classpath

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

```
<dependency>
  <groupId>org.isisaddons.module.command</groupId>
  <artifactId>isis-module-command-dom</artifactId>
</dependency>
```

Check for later releases by searching [Maven Central Repo](#).

Bootstrapping

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

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

Configuration Properties

For commands to be created when actions are invoked, some configuration is required. This can be either on a case-by-case basis, or globally:

- by default no action is treated as being a command unless it has explicitly annotated using `@Action(command=CommandReification.ENABLED)`. This is the option used in the example app described above.
- alternatively, commands can be globally enabled by adding a key to `isis.properties`:

```
isis.services.command.actions=all
```

This will create commands even for query-only (`@ActionSemantics(Of.SAFE)`) actions. If these are to be excluded, then use:

```
isis.services.command.actions=ignoreQueryOnly
```

An individual action can then be explicitly excluded from having a persisted command using `@Action(command=CommandReification.DISABLED)`.

API

This module implements two service APIs, `CommandService` and `BackgroundCommandService`. It also provides the `BackgroundCommandExecutionFromBackgroundCommandServiceJdo` to retrieve background commands for a scheduler to execute.

CommandService

The `CommandService` defines the following API:

```
public interface CommandService {
    Command create();

    void startTransaction(
        final Command command,
        final UUID transactionId);

    void complete(
        final Command command);

    boolean persistIfPossible(
        final Command command);
}
```

Isis will call this service (if available) to create an instance of (the module's implementation of) `Command` and to indicate when the transaction wrapping the action is starting and completing.

BackgroundCommandService

The `BackgroundCommandService` defines the following API:

```
public interface BackgroundCommandService {
    void schedule(
        final ActionInvocationMemento aim,
        final Command command,
        final String targetClassName,
        final String targetActionName,
        final String targetArgs);
}
```

The implementation is responsible for persisting the command such that it can be executed asynchronously.

BackgroundCommandExecutionFromBackgroundCommandServiceJdo

The `BackgroundCommandExecutionFromBackgroundCommandServiceJdo` utility class ultimately extends from Isis Core's `AbstractIsisSessionTemplate`, responsible for setting up an Isis session and obtaining commands.

The [quartz extension](#) module can be configured to run a job that uses this utility class.

Supporting Services and Mixins

As well as the `CommandService` and `BackgroundCommandService` implementations, the module also a number of other domain services and mixins.

The domain services are:

- `CommandServiceJdoRepository` provides the ability to search for persisted (foreground) `Command`s`. None of its actions are visible in the user interface (they are all `@Programmatic`) and so this service is automatically registered.
- (As of 1.8.x) the `CommandServiceMenu` provides actions to search for `Command`s`, underneath an 'Activity' menu on the secondary menu bar.
- `BackgroundCommandServiceJdoRepository` provides the ability to search for persisted (background) `Command`s` and `replayable` commands. None of its actions are visible in the user interface (they are all `@Programmatic`) and so this service is automatically registered.
- (As of 1.16.1) the `CommandReplayOnMasterService` and `CommandReplayOnSlaveService` menu services provide actions to support the `replay` library.
 - `CommandReplayOnMasterService` provides actions to find and to manually download `Commands` since a specified transaction Id (ie as determined from the slave)
 - `CommandReplayOnSlaveService` provides actions to find the most recently replicated `Command` on the slave (the so-called "high water mark") and to manually upload `Commands` obtained from the master

The mixins are:

- `HasTransactionId_command` mixin provides the `command` action to the `HasTransactionId` interface. This will therefore display all commands that occurred in a given transaction, in other words whenever a command, or also (if configured) a published event or an audit entry is displayed.
- `CommandJdo_childCommands` mixin provides the `childCommands` contributed collection, while `CommandJdo_siblingCommands` mixin provides the `siblingCommands` contributed collection
- `CommandJdo_retry` mixin allows commands to be resubmitted.
- `CommandJdo_exclude` mixin allows failing replayable commands to be ignored.

Once any replayable command hits an exception, no further replayable commands are run. This mixin is intended to allow the administrator to choose to skip/ignore any such command.

In addition, the `T_backgroundCommands` abstract mixin can be used to contribute a `backgroundCommands` collection to any object that can be used as the target of a command, returning the 30 most recent background commands. For example:

```
@Mixin
public class SomeObject_backgroundCommands extends T_backgroundCommands<SomeObject> {
    public SomeObject_backgroundCommands(final SomeObject someObject) {
        super(domainObject);
    }
}
```

where `SomeObject` is the class of the target domain class.

(As of 1.8.x and later) these various services are automatically registered, meaning that any UI functionality they provide will appear in the user interface. If this is not required, then either use security permissions or write a vetoing subscriber on the event bus to hide this functionality, eg:

```
@DomainService(nature = NatureOfService.DOMAIN)
public class HideIsisAddonsAuditingFunctionality extends AbstractSubscriber {
    @Programmatic @Subscribe
    public void on(final CommandModule.ActionDomainEvent<?> event) { event.hide(); }
}
```

Known issues

None known at this time.

Related Modules/Services

As well as defining the `CommandService` and `BackgroundCommandService` APIs, Isis' applib defines several other closely related services. Implementations of these services are referenced by the [Isis Add-ons](#) website.

The `AuditingService3` service enables audit entries to be persisted for any change to any object. The command can be thought of as the "cause" of a change, the audit entries as the "effect".

The `PublishingService` is another optional service that allows an event to be published when either an object has changed or an actions has been invoked. There are some similarities between publishing to auditing, but the publishing service's primary use case is to enable inter-system co-ordination (in DDD terminology, between bounded contexts).

If the all these services are configured - such that commands, audit entries and published events are all persisted, then the `transactionId` that is common to all enables seamless navigation between each. (This is implemented through contributed actions/properties/collections; `Command` implements the `HasTransactionId` interface in Isis' applib, and it is this interface that each module has services that contribute to).

Dependencies

Maven can report modules dependencies using:

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

which, excluding Apache Isis itself, returns no direct compile/runtime dependencies.