eHealth Framework

Implementing an Import Processor

Imprint

InterComponentWare
Altrottstraße 31
69190 Walldorf
Tel.: +49 (0) 6227 385 0
Fax.: +49 (0) 6227 385 199

Notice

The wording in this document applies equally to women and men. The masculine form was selected to ease the comprehensibility and legibility of the text.

All company logos are a registered trademark of InterComponentWare AG.

The product names mentioned in this documentation are either trademarks or registered trademarks of the respective owners and are stated for identification purposes only.

This documentation and the software components are protected by copyright © 2006-2010 InterComponentWare AG.

Note:

The current version of this document has a draft status and various chapters are still in review.

The document is collaboratively built with the use of the Darwin-Information-Typing-Architecture (DITA) and has therefore a draft status concerning styles and layout. The necessary adaptations are currently also in a developmental stage.

Contents

1 Overview

Purpose

The purpose of this howto is to provide a step-by-step guide of how to implement a custom implementation of an import processor and its configuration as well as its integration in modules and assemblies.

Scope

2 Implementing an Import Processor

2.1 What is an Import Processor?

An import processor helps you to import data into the database during build and deploy time. We distinguish two phases:

- Bootstrap
- Test data import

Both phases use classes implementing the interface ImportProcessor. It is possible to use one implementation for both phases. However it is recommended to use a separate implementation for each phase, as they have a completely different purpose, are executed at different points in time and different rules apply to them. In the following sections the differences are explained.

2.1.1 Bootstrap

Bootstrap is the phase of the build or the deployment when data which is essential for the application to work is written to the database. This phase is executed during build **and** deployment.

The bootstrap may operate both on empty, newly created databases and productive once containing customer data. Therefore one important rule for this phase is that the bootstrap must be repeatable. The subsequent runs of the bootstrap phase should not cause an inconsistent state of data. For example the same data should not be added twice. Instead the import processor executing the bootstrap should recognize the existing data and update it or skip the processing. This must be accounted for during implementation.

2.1.2 Test Data Import

The test data import (or short "import") is the phase when test data is written to the database. This phase is executed only during the build and not during deployment. It is used to prepare the database for the use in integration tests. Integration tests can access this test data in a later phase of the build during test execution.

Usually the database is cleaned during a build. Because of this the test data import does not need to be repeatable.

2.2 Implementing the Interface

Regardless if used during bootstrap or test data import all classes have to implement `com.icw.ehf.commons.process.ImportProcessor`.

It has two methods that need to be implemented. The other ones have all been deprecated in the meantime. Here is the documentation from the code.

```
/** * The import processor interface defines methods for importing
data of any kind * into the database. The import processor is used
at build and deployment to * bootstrap the database, to import test
* data or any other kind of information into the system. */ public
interface ImportProcessor {

/** * Execute the processor for the provided import parameter. * *
```

```
                    @param importParameter The {@link ImportParameter} instance contains
                    * all required information for the processor to execute. */ void
                    process(ImportParameter importParameter);

                    /** * Clear all data, so this processor can be run anew. */ void
                    reset(); }
```

The class `ImportParameter` is a wrapper class to take one ore more Resources (a Spring defined interface for a resource descriptor).

You can choose any sort of implementation you like. In the eHF we often use ImportProcessors that read via xstream test data from XML files directly into our domain objects and then use the services of the module to persist them. But you can use any other way to generate test data.

2.3 Configuration

There are two parts of the configuration. A Maven and a Spring related part.

2.3.1 Maven configuration

The Maven related part is quite short and uses properties for configuration. These can be specified as command line argument, using the -D option or more conveniently via the `project.properties`. The type of the import (bootstrap, import or both) has to be specified as well as the location of the corresponding Spring configuration files. A sample `project.properties` looks like this:

```
                # for local bootstrapping maven.ehf.db.initialize=true
                maven.ehf.db.initialize.bootstrap=true
                maven.ehf.db.initialize.bootstrap.pattern=classpath:/META-INF/<
yourmodule>-bootstrap-context.xml
                maven.ehf.db.initialize.import=true
                maven.ehf.db.initialize.import.pattern=classpath:/META-INF/<
yourmodule>-import-context.xml
```

2.3.2 Module Spring Configuration

In the previous section you have seen how the module Spring context is propagated to the build plugin that executes the bootstrap and the import.

The Spring configuration consists of two parts. One is the core which is used in the module and is also propagated to the assembly. The other part is different for execution each in the assembly and the module. The additional configuration in the module adds those parts of the Spring configuration that are typically provided by the assembly.

Here an example to make it more clear: The ehf-document needs codes from the ehf-codesystem and ehf-codesystem-repository. This means that the ehf-document is dependent on the bootstrap of the ehf-codesystem and its repositories.

If you want to use the ehf-document inside the assembly, the assembly takes care of triggering the bootstrap of the ehf-codesystem (We will see how this is done in a later section). The rest is just configuring the ImportProcessor itself. On the other hand, if you intend to run the bootstrap during the module build you need to explicitly add the bootstrap dependencies. Here is an example of the core configuration from the ehf-document (`src/main/resources/META-INF/ehf-document/ehf-document-bootstrap.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
```

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean class="com.icw.ehf.commons.process.executor.ImportProcessExecutor"
        lazy-init="false" init-method="process">

        <property name="resourceLocation" value="classpath:/META-INF/ehf-document/
bootstrap" />
        <property name="resourceLoader">
            <bean class="com.icw.ehf.commons.process.resource.
DefaultResourceLoader">
                <property name="subPattern" value="**/*dat,**/*xml" />
            </bean>
        </property>
        <property name="importProcessor" ref="documentProcessor" />
    </bean>

    <bean id="documentProcessor" class="com.icw.ehf.document.dbimport.
DocumentModuleImportProcessor">
        <property name="transactionManager" ref="transactionManager" />
        <property name="documentDocumentTypeMetaDataSecureService" ref=
"documentDocumentTypeMetaDataSecureService" />
        <property name="documentModuleSecureService" ref=
"documentModuleSecureService" />
        <property name="documentUnitIdentifier" ref="documentUnitIdentifier" />
        <property name="codesystemResolver" ref="codeResolver"/>
        <property name="localeProvider" ref="localeProvider"/>
        <property name="hibernateTemplate" ref="hibernateTemplate"/>
    </bean>

</beans>
```

Here, an ImportProcessExecutor is configured. During build all these executors are gathered from Spring and executed. This executor has the Resources and the `documentProcessor` injected. This implementation of an ImportProcessor needs some dependencies to do its job like, e.g., `documentModuleSecureService` . They are injected in the bean definition of the processor. Beans both from the module and from the system context are used. But where do they come from? Normally from the assembly, but when you build the module? This is where the second part of the Spring configuration comes into play.

The file resides in `src/test/resources` and is therefore only available during the build of the module and not during the build of the assembly or the deployment. It is the file that was specified in the `project.properties` mentioned above. Here again the example from ehf-document:

```xml
<?xml version="1.0" encoding="utf-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
springframework.org/schema/beans/spring-beans-2.5.xsd">

    <import resource="classpath:/META-INF/ehf-codesystem-module-context.xml" />

    <import resource="classpath:/META-INF/ehf-codesystem-repository/ehf-
codesystem-repository-bootstrap.xml" />

    <import resource="classpath:/META-INF/document-system-context.xml" />

    <import resource="classpath:/META-INF/ehf-document-runtime-context.xml" />
```

4

```
    <import resource="classpath:/META-INF/ehf-document/ehf-document-bootstrap.
xml"/>

</beans>
```

As we can see here the bootstrap of the document is apparently dependent on the bootstrap of the codesystem. This is why its bootstrap configuration is included here. It gets also executed during the build. If your module is not dependent on another module's bootstrap data you can omit this.

Next thing the document system context and the runtime context is imported. The system context provides default configuration which is normally provided by the assembly (like TransactionManager, SecurityService, etc.). The runtime context contains all the services provided by the document module. That's where the ImportProcessor gets its dependencies from, e.g., the services like `documentModuleSecureService` mentioned above.

In a last step the configuration for the ImportProcessor is imported.

So in this file you do what usually is the job of the assembly: Bring together all the different Spring configuration files that are needed to build the Spring context necessary for the bootstrap process. The configuration for the import process looks similar. The setup for the module is now complete. Let's have a look how this is done in the assembly.

2.3.3 Assembly Configuration

The assembly configuration is relatively simple. This is because you can skip the step of gathering all the contexts. It is the job of the assembly to have all the Spring contexts in place.

The assembly also has a `project.properties` file. There, the bootstrap is usually already configured.

```
                maven.ehf.db.initialize.bootstrap.pattern=classpath:/META-INF/ehf-
assembly-bootstrap.xml
                maven.ehf.db.initialize.import.pattern=classpath:/META-INF/ehf-
assembly-import.xml
                maven.ehf.security.policy.file=${basedir}/src/main/resources/META-
INF/ehf-assembly/bootstrap/bootstrap.policy
```

All you have to do is to add to the file `src/main/resources/META-INF/ehf-assembly-bootstrap.xml` the following line:

```
                <import
                resource="classpath:/META-INF/yourmodule/yourmodule-bootstrap.
xml"/>
```

It should point to the file located in your `src/main/resources`. The setup for the import is again similar.