

C24 IO Spring Reference Manual

Adam Skogman, Oliver Gierke

Copyright © 2011

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Preface	iii
1. Project metadata	iii
1.1. Modules	iii
I. Reference documentation	1
1. C24 IO support for Spring	2
1.1. C24 model	2
1.2. HttpMessageConverters	2
1.3. OXM integration	3
2. C24 IO support for Spring Integration	4
2.1. Introduction	4
2.2. Unmarshalling Payloads	4
2.2.1. Using the XML namespace	4
2.2.2. Input to the Unmarshalling Transformer	5
2.3. Marshalling with C24 Sinks	5
2.3.1. Marshalling using the XML namespace	5
2.3.2. Output options for the marshalling transformer	5
2.4. Using C24 Transforms	6
2.4.1. Configuring C24 Transforms using the XML namespace	6
2.4.2. Input payloads	6
2.4.3. Output payload	6
2.5. Using C24 XPath	6
2.5.1. XPath Statements as Beans	6
2.5.2. Transforming Objects using XPath	6
2.5.3. Routing using XPath	7
2.5.4. Filtering using XPath	7
2.5.5. XPath Header enricher	7

Preface

1. Project metadata

- Bugtracker - <https://support.c24.biz/>
- Maven Release repository - <http://repo.c24io.net:8081/nexus/content/groups/public>
- Sources - <https://github.com/C24-Technologies/c24-spring>

1.1. Modules

- Core - Integration with core Spring components
- Integration - Integration with Spring Integration components

Part I. Reference documentation

Chapter 1. C24 IO support for Spring

C24 IO can be integrated into several of the core Spring Framework features as well. C24 IO plugs into Spring-WS for SOAP based services, replacing the XML marshalling framework. It plugs into Spring MVC for building REST interfaces using C24 IO marshalling and unmarshalling, allowing any C24 IO supported data type to be passed seamlessly over REST/HTTP.

1.1. C24 model

The very core of the C24 Spring integration is integrating a C24 model into a Spring application. To do so the core module contains a `C24Model` class that can be constructed by handing it an `Element` from your generated model classes.

Example 1.1. Programmatically setting up a `C24Model` instance

```
C24Model model = new C24Model(YourGeneratedElementClass.getInstance());
```

To ease that process in a Spring application we offer a namespace element that allows you to either define the `Element` class to build the model from or define a base package to scan for C24 classes.

Example 1.2. Using the XML namespace to setup a C24 model

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c24="http://schema.c24.biz/spring-core"
       xsi:schemaLocation="...">

  <c24:model base-package="com.acme.mypackage" />

  <!-- or -->

  <c24:model base-element="com.acme.mypackage.SampleModelDocumentRootElement" />
</beans>
```

If you don't add an `id`-attribute to the model element we will default to `_c24model`. All other namespace elements will reference the model bean by this id by default so that you should need to wire the model into those explicitly. However if you have multiple C24 models in your app, you have to assign an id to them and then refer to them via the `model-ref` attribute of other namespace element.

1.2. `HttpMessageConverters`

Building REST webservice with Spring MVC you might want to easily marshal your C24 data structures defined into IO objects and vice versa. To do so we ship a configurable `HttpMessageConverter`:

Example 1.3. Setting up `HttpMessageconverters` to bind request and response data to C24 objects

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c24="http://schema.c24.biz/spring-core"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="...">

  <c24:model base-package="com.acme.mypackage" />

  <mvc:annotation-driven>
    <mvc:message-converters>
      <c24:http-message-converter>
        <c24:format type="TEXT" />
        <c24:format type="XML" content-type="application/vnd.foo+xml" />
      </c24:http-message-converter>
    </mvc:message-converters>
  </mvc:annotation-driven>

</beans>
```

The `http-message-converter` gets a set of `format` elements configured that define which representation shall be supported and for which content type each of them shall be returned. If you don't configure a content type we assume defaults like `text-plain`, `application/xml` and so on.

1.3. OXM integration

If you are using the Spring OXM module it's easy to set up a `Marshaller/Unmarshaller` instance capable of handling C24 IO data like this:

Example 1.4. Configuring an Spring C24 OXM marshaller

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c24="http://schema.c24.biz/spring-core"
       xsi:schemaLocation="...">

  <c24:marshaller id="marshaller" />

  <c24:model base-package="com.acme.mypackage" />

</beans>
```

Chapter 2. C24 IO support for Spring Integration

Spring Integration provides an extension of the Spring programming model to support the well-known [Enterprise Integration Patterns](#). C24 IO components for Spring Integration makes C24 marshaling and transformation directly usable as Spring Integration components.

2.1. Introduction

C24 IO Spring Integration support extends the Spring Integration Core with implementations of transformers, selector and routers designed to make working with C24 IO Complex Data Object messages in Spring Integration simple. The provided messaging components are designed to work with any data represented as a `ComplexDataObject`.

Also provided are unmarshalling and marshalling transformers for turning String or byte type payloads into C24 IO Complex Data Objects, and turning CDOs back into `Strings` or bytes. We're using transformer to mean Spring Integration transformer, not an IO transform.

This guide will emphasise the use of Spring XML namespaces to configure C24 IO Spring components. The namespaces are configured like this:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-c24="http://schema.c24.biz/spring-integration"
  xsi:schemaLocation="http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-integration-2.0.xsd
    http://schema.c24.biz/schema/spring-integration
    http://schema.c24.biz/spring-integration.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

</beans>
```

2.2. Unmarshalling Payloads

This section will explain how to use the unmarshalling (parsing) capabilities of C24 IO within the Spring Integration Framework. The basis for C24 IO unmarshalling is the `Source` class. In Spring Integration, unmarshalling is implemented as a Spring Integration `Transformer`.

2.2.1. Using the XML namespace

Configuring an unmarshalling transformer using the namespace:

```
<int-c24:unmarshalling-transformer
  source-factory-ref="textualSourceFactory"
  model-ref="inputDocumentModel"
  input-channel="..."
  output-channel="..." />
```

Two specific attributes are required. The `source-factory-ref` is a `SourceFactory` defined as a bean elsewhere in the XML config. There are many `SourceFactory` implementations, each wrapping a C24 IO `Source` implementation. E.g. there is a `XmlSourceFactory`, a `TextualSourceFactory` and a `FixSourceFactory`.

The second attribute is a *model-ref*. When unmarshalling, you must configure which `ComplexDataObject` derived class you want to create. Typically, this is clear from the use case, such as when receiving a message on a specific MQ queue, or to a specific `WebService` endpoint. Please see Section 1.1, “C24 model” for information about configuring a model.

2.2.2. Input to the Unmarshalling Transformer

These are the message payloads supported by the unmarshalling transformer. They will be passed to the C24 `Source` either as a `Reader` (you can't control the encoding) or a `InputStream` (you can configure the encoding on the `SourceFactory`).

- `Reader` - will be passed as-is.
- `String` - will be passed as a `StringReader`.
- `InputStream` - will be passed as-is.
- `byte[]` - will be passed as a `ByteArrayInputStream`.
- `File` - will be passed as a `FileInputStream`.

The actual unmarshalling is determined by the C24 `Source` that is used. The package `biz.c24.io.spring.source` holds a number of `SourceFactory` implementations that wrap different C24 `Sources`. All the configuration properties that you would usually find on the `Source` subclass (such as `setEncoding()`) is available on the individual `SourceFactory` implementations.

The source factories are configured as beans, like so:

```
<bean id="textualSourceFactory" class="biz.c24.io.spring.source.TextualSourceFactory">
  <property name="encoding" value="UTF-8"/>
</bean>
```

2.3. Marshalling with C24 Sinks

This section will explain how to use the marshalling capabilities of C24 IO within the Spring Integration Framework. The basis for C24 IO marshalling is the `Sink` classes. In Spring Integration, marshalling is implemented as a Spring Integration `Transformer`.

2.3.1. Marshalling using the XML namespace

A marshalling `Transformer` is configured like this:

```
<int-c24:marshalling-transformer
  input-channel="..."
  output-channel="..."
  sink-factory="xmlSinkFactory"
  output-type="BYTE_ARRAY" />
```

The `sink-factory` is a reference to a `SinkFactory` that is defined as a bean. The `output-type` is an enum value that determines if the output is a `String` or a byte array.

2.3.2. Output options for the marshalling transformer

The actual marshalling is determined by the C24 `Sink` that is used. The package `biz.c24.io.spring.sink` holds a number of `SinkFactory` implementations that wrap different C24 sinks. All the configuration properties that you would usually find on the `Sink` subclass (such as `setEncoding()`) is available on the individual `SinkFactory` implementations.

The output type determines what kind of payload will be output from the marshaling transformer.

- *STRING* will output a `String` as the payload. It's up to the user to determine if that makes sense given the chose Sink type.
- *BYTE_ARRAY* will output a `byte[]` as the payload. It's up to you to determine if that makes sense given the chose Sink type. Configure the encoding on the `SinkFactory`.

2.4. Using C24 Transforms

2.4.1. Configuring C24 Transforms using the XML namespace

```
<int-c24:transformer
  transform-class="biz.c24.io.examples.transforms.basic.ExampleTransform"
  input-channel="..."
  output-channel="..."
  always-return-array="false" />
```

The attributes to configure are:

- *transform-class* - the class name of the C24 IO generated transform.
- *always-return-array* - if you want the output to always be an array, even if only a single object was the output from the transform. See below.

2.4.2. Input payloads

The Spring Integration IO transformer only accepts a single `Object` as the input payload. In effect, this should be of a `ComplexDataObject` subclass.

2.4.3. Output payload

The result of transforming a single object (see above) can either be a single object, or an array of objects. The default is to only output a single `ComplexDataObject` as the payload. If the transform yields more than one object, or if the attribute *always-return-array* is set to true, then an `Object[]` will be the payload.

2.5. Using C24 XPath

C24 IO has excellent XPath capabilities, and specifically any `ComplexDataObject` can be queried using XPath. The following components leverage that capability by providing Spring Integration routers, selectors (for filters) and transformers. A common trait is that they either take a simple XPath statement as a `String`, or a reference to a XPath Statement defined using a standard Spring bean.

2.5.1. XPath Statements as Beans

In some cases, an XPath statement cannot be expressed just as a simple string. A good example is if it needs to navigate and differentiate between more than one namespace. If so, you can declare and configure a C24 Xpath Statement.

2.5.2. Transforming Objects using XPath

Picking out parts of a larger structure using XPath

```
<int-c24:xpath-transformer input-channel="..."
  evaluation-type="OBJECT_RESULT"
```

```

xpath-statement="/Emails"
output-channel="..." />

```

The input to the transformer must always be a subclass of `ComplexDataObject`.

- *evaluation-type* determines the type of the output payload. Options include `object`, `String`, `boolean` and `list` (for selecting more than one node). See the enum `biz.c24.io.spring.integration.xpath.XPathEvaluationType` for details.
- *xpath-statement* is used to provide a simple XPath expression as a string.
- *xpath-statement-ref* is used to provide a reference to an XPath statement declared as a bean elsewhere.

2.5.3. Routing using XPath

```

<int-c24:xpath-router input-channel="..."
  xpath-statement="//Employee[1]/FirstName" default-output-channel="wrongChannel">
  <int-c24:mapping value="Andy" channel="rightChannel" />
  <int-c24:mapping value="Barry" channel="anotherChannel" />
</int-c24:xpath-router>

```

The input to the router must always be a subclass of `ComplexDataObject`.

- *xpath-statement* is used to provide a simple XPath expression as a string.
- *xpath-statement-ref* is used to provide a reference to an XPath statement declared as a bean elsewhere.

The provided XPath *must* evaluate to a string.

2.5.4. Filtering using XPath

Filtering can be done either by providing an XPath statement that evaluates to a boolean, or by providing both a XPath Statement and a String to compare the result to.

```

<int:filter discard-channel="wrongChannel" ref="boolSelector" />

<int-c24:xpath-selector id="boolSelector"
  evaluation-result-type="boolean"
  xpath-statement="//Employee/FirstName/text()='Andy'" />

<int:filter discard-channel="wrongChannel" ref="stringSelector" />

<int-c24:xpath-selector id="stringSelector"
  evaluation-result-type="string" xpath-statement="//Employee/FirstName"
  string-test-value="Andy" />

```

The input to the selector must always be a subclass of `ComplexDataObject`.

- *evaluation-result-type* determines if this is a boolean or String based selector.
- *string-test-value* is required by the String based selector.
- *xpath-statement* is used to provide a simple XPath expression as a string.
- *xpath-statement-ref* is used to provide a reference to an XPath statement declared as a bean elsewhere.

2.5.5. XPath Header enricher

Before passing a `ComplexDataObject` payload to a channel adapter or similar, it is often needed to pick out certain data and put it into named headers. For this use case, there is an XPath header enricher.

```
<int-c24:xpath-header-enricher input-channel="..." output-channel="...">
  <int-c24:header name="firstName" xpath-statement="//Employee/FirstName" />
  <int-c24:header name="lastName" xpath-statement="//Employee/LastName" />
</int-c24:xpath-header-enricher>
```

Note that several headers can be enriched at the same time. The XPath expressions will be evaluated into Strings. No other data type is currently supported.