Cinnober Financial Technology AB
Kungsgatan 36
SE-111 35 Stockholm
Sweden
+46 8 503 047 00
cinnober.com

Ciguan Developer's Guide

Developer’s guide for the Ciguan application framework

Table of contents

1 Introduction 9

2 Standard framework functionality 10
2.1 Standard configuration 10

3 Common concepts 11
3.1 Business types 11
3.1.1 Adding custom business types 11

4 A configuration primer 12
4.1 Deciding on an initial approach 12
4.2 Inherit standard configuration or not? 12
4.3 File naming guideline 12
4.4 Structuring your own configuration 12
4.5 Testing your configuration 12

5 Basic configuration structure 13
5.1.1 Ciguan based applications 13
5.2 Bootstrap implementation class 14
5.2.1 Ciguan based applications 14

6 Common configuration 15
6.1 Configuration file header 15
6.2 Configuration format version 15
6.3 Module inheritance 15
6.4 Plug-in identification 15
6.5 Dictionary 16
6.6 Locales and default formatting patterns 16
6.6.1 Formatting pattern 16
6.6.2 Default formatting patterns 16
6.6.3 Locales 17

7 Metadata configuration 18
7.1 Search packages 18
7.1.1 Namespaces 18
7.2 Get methods 18
7.2.1 Class-based get methods 18
7.2.2 Read consistency for class based get methods 19
7.2.3 Data source lookup get methods 19
7.2.4 Elvis operator get methods 20
7.3 Metadata definition 20
7.3.1 Specifying server request names 20
7.3.2 Mapping explicit business types 20
7.3.3 State attributes 21
7.4 Suppression 21
7.4.1 Directional suppression 21

8 Application server configuration 23
8.1 AS plug-ins 23
8.2 Bean factory configuration 23

8.2.1 Classes without an interface 24
8.2.2 Replacing standard beans 24
8.2.3 Singleton beans 24
8.3 Transport configuration 24
8.3.1 Transport plug-ins 24
8.3.2 Request transformer 25
8.3.3 Broadcast flow handlers 25
8.3.4 Broadcast processors 25
8.3.5 Broadcast class filters 26
8.3.6 Broadcast flow subscriptions 26
8.4 Data sources 26
8.4.1 Lists 26
8.4.2 List filters 28
8.4.3 Trees 29
8.4.4 List trees 30
8.4.5 Structuring data sources 31
8.4.6 Super user mechanism 31
8.5 Cache lookups 32
8.6 Parameters 32
8.7 Components 33

9 Advanced application server configuration 34
9.1 Using data source factories 34
9.1.1 Handling of referential errors during data source creation 35
9.2 Programmatically populated lists using factories 35
9.3 Using Set and Map attributes in filters 36
9.4 Hand-coded data source item classes 36
9.5 Using annotations for metadata 36
9.6 Populating list contents from broadcast processors 37
9.7 Replacing or extending application server components 37
9.8 Referential lookup translations 37
9.8.1 How to disable a referential lookup 38

10 Application server API 39
10.1 Hub classes 39
10.2 Bootstrap 39
10.3 Application server components 39
10.3.1 Writing an application server component 40
10.3.2 Configuring your component 40
10.4 Bean factory 40
10.4.1 Singleton beans 41
10.4.2 Beans with parameters 41
10.5 Dictionary handler 41
10.5.1 Dictionary header 41
10.5.2 Dictionary tags 41
10.6 Broadcast mapper 42
10.7 Broadcast flow handlers 42
10.7.1 Writing a broadcast flow handler 43
10.7.2 Reference data dependency 44
10.8 Broadcast handler 44
10.8.1 Broadcast dispatching logic 44
10.8.2 Listening to broadcasts 44
10.8.3 Writing a broadcast processor 45
10.8.4 Submitting broadcasts 45

10.9 Transport plug-ins 45
10.9.1 Writing a transport plug-in 46
10.9.2 Receiving client messages 46
10.9.3 Session invalidation processing 46
10.9.4 Returning client responses 46
10.9.5 Adding transport plug-ins 47
10.9.6 Enabling the user session inactivity functionality 47
10.10 Authorization handler 47
10.10.1 Checking access to services 48
10.11 Formatter 48
10.12 Application server connection 48
10.12.1 Data source service 48
10.12.2 Request service 49
10.12.3 Transport service 49
10.12.4 Current session locale 49
10.12.5 Session data 49
10.12.6 Populating the client user session 49
10.13 Data source containers 50
10.14 AS plug-ins 50
10.14.1 Plug-in life cycle 50
10.14.2 Writing a plug-in 51
10.15 Summary handlers 51
10.15.1 Writing a custom summary handler 51
10.16 User properties 52
10.16.1 Persistence in applications 52
10.16.2 Creating or updating a user property 53
10.16.3 Removing a user property 53
10.16.4 Persisting client side objects as user properties 53
10.16.5 Restoring client side objects from user properties 53
10.17 File upload 54
10.17.1 Execution flow 54
10.17.2 File upload request 54
10.17.3 Upload file handlers 55
10.17.4 Upload service servlet 55
10.17.5 File upload plug-in 55
10.18 Data source export 56
10.18.1 Execution flow 56
10.18.2 Request parameters 56
10.19 Form handlers 56
10.19.1 Implementing a form handler 57
10.19.2 Configuring a form handler for use 57
10.20 Metrics counters 57
10.21 Services 57
10.21.1 Service configuration 58
10.21.2 Service handler 58
10.21.3 Service implementation 58
10.22 Task scheduler 59
10.22.1 Writing tasks 59
10.22.2 Submitting a task for execution 59
10.22.3 Cancelling a task 59
10.23 Response classes 59

11 Writing application server code 61
11.1 Transport plug-in routing 61

12 Configuring the user interface 62
12.1 A primer on context objects 62
12.2 Special context objects 63
12.2.1 SessionModel 63
12.2.2 MenuParameters 63
12.2.3 TypeAheadFilter 63
12.2.4 The array context 64
12.3 View definitions 64
12.3.1 Data addressing 65
12.4 Viewport views 66
12.4.1 Viewport fields 67
12.4.2 Using other display types than table 68
12.4.3 Filtering viewport data based on context objects 68
12.4.4 Pre-sorting viewports 69
12.4.5 Using column charts 69
12.4.6 Viewports as part of an input widget 69
12.4.7 Trees as part of an input widget 70
12.4.8 Query viewports 71
12.4.9 Viewport summaries 71
12.5 Form views 72
12.5.1 Form initialization configuration 73
12.5.2 Form layout configuration 76
12.6 Form views with editable tables 80
12.7 File upload views 81
12.8 Tree views 82
12.9 Detail views 82
12.10 Search views 82
12.11 Chart views 83
12.11.1 Chart data 84
12.11.2 Chart options 84
12.11.3 Complete example 85
12.12 Displays 86
12.13 Perspectives 86
12.13.1 Perspective definition 86
12.13.2 Perspective slot templates 88
12.13.3 Perspective role mappings 88
12.13.4 Global menu items 89
12.13.5 Menus 89
12.13.6 Context menus 90
12.13.7 Menu items 91
12.13.8 Menu parameter handling 92
12.13.9 Menu item filtering 92
12.14 View interactivity 93

13 Advanced user interface configuration 95
13.1 Extending views 95
13.2 Modifying standard user interface configuration 96
13.2.1 Modifying perspective view references 96
13.2.2 Modifying data sources 96
13.2.3 Modifying views 96
13.3 Removing standard user interface configuration 96
13.3.1 Removing perspectives 96
13.3.2 Removing perspective view references 97
13.3.3 Removing menus 97
13.3.4 Removing menu items 97

13.3.5 Removing data sources 97
13.3.6 Removing views 97
13.3.7 Removing view field sets 98
13.3.8 Removing view fields 98
13.4 Enabling and disabling form items 98
13.5 Filtering lists based on other lists 98

14 Performance considerations 100
14.1 Out-of-dispatcher processing 100
14.2 Broadcast processors and broadcast listeners 100
14.3 Data source filters 100
14.4 Get methods 101
14.5 Business type formatters 101
14.6 Data source listeners 101

15 APPENDICES 102
15.1 Configuration test 102
15.1.1 Specifying constant group name as data source ID 102
15.1.2 Handling duplicate definitions 102
15.2 Adding a business type 103
15.2.1 Common steps 103
15.2.2 Server side steps 103

About this document

This document aims to be the starting point for developers who will make use of the Ciguan framework.

Caution: This document is a work in progress. Originally this was the developers guide for the whole web framework used by Cinnober but now it aims to cover just the core functionality. Lots of things in this document is subject to change and should not be considered “ready” until the first official release.

1 Introduction

The Ciguan framework and application server is a powerful framework for building client applications. The Ciguan framework has no dependency to any particular UI implementation.

2 Standard framework functionality

2.1 Standard configuration

Ciguan ship with configuration module functionality which you can either choose to use as it is by including the supplied configuration files, or you can choose to build your own configuration from scratch. There is no best practice recommendation on how to set up your configuration since it is highly depending on the nature of the client being built.

The standard Ciguan configuration modules and their content are as follows:

Module	Contents
As	Standard application server bean configuration Standard application server transport configuration Standard application server data sources and getters Standard views and menus

3 Common concepts

3.1 Business types

The business type concept is a means to associate a data type with what the value is representing, in order to validate input and format output according to the business rules. For instance, both an amount and a quantity might be stored as an integer or a long value, but they should most likely be formatted in different ways. To help with this, Ciguan features a concept called business type.

Below is a list of valid business types and their meaning.

Business type	Data type	Ciguan Annotation	Meaning
Amount	Long	@CwfAmount	Monetary amount
BasisPoint	Long	@CwfBasisPoint	Basis point (1/100' percent)
Date	String	@CwfDate	Date (YYYY-MM-DD)
DateTime	String	@CwfDateTime	Date and time (YYYY-MM-DDTHH:MI:SS.CCC)
Decimal	Long		Decimal number
InterestRate	Long	@CwfInterestRate	Interest rate (percent)
MultiLineText	String	@CwfMultiLineText	Display and input as multi-line text
Percent	Long	@CwfPercent	Percent
Price	Long	@CwfPrice	Price
Reference	String	@CwfReference	An attribute which references the ID attribute of an object. See section 9.8 for more information.
SecondsSinceTime	Integer	@CwfSecondsSinceTime	Seconds since an arbitrary time
Text	(Any)	@CwfText	Display as text, no formatting
Time	String	@CwfTime	Time (HH:MI:SS.CCC)
URL	String	@CwfUrl	Hyperlink

Business types can currently only be used for attributes. This may change in future Ciguan versions.

Caution: Business types are intended for simple types such as the Java primitive types, the simple types from the `java.lang` package. Do not attempt to use a business type annotation for complex types such as custom classes as this is not supported. Complex types should instead have business types for its attributes.

3.1.1 Adding custom business types

If you need to add your own business types, see section 15.1.2 for a check list on what needs to be done.

4 A configuration primer

4.1 Deciding on an initial approach

Configuring the application server and the user interface might appear to be quite a daunting task at first, but once you get into it, it is fairly straightforward. Below is an outline describing the steps you need to go through when you are about to decide on how to set up the configuration used in your customer project.

4.2 Inherit standard configuration or not?

If you want to make changes to the standard views without creating local copies of the files, Ciguan features powerful mechanisms to do just that, so in general, you should avoid making local copies of the standard configuration modules if possible.

4.3 File naming guideline

A recommendation is to always give configuration modules a name that hints about their content. Also, always use the double suffix `.cwf.xml` to indicate that they are Ciguan module XML files.

4.4 Structuring your own configuration

Configuration files tend to become quite bulky after a while, so a recommendation is to divide them into manageable units and make sure they contain only items of a certain type. Below is a list of suggested module names and their content. Do not see this as the ultimate solution though; your project may have special needs that require other approaches.

Module	Contents
XYZBeanConfiguration	Bean factory configuration (if needed)
XYZDataSources	Data sources and getters
XYZMenus	Menus
XYZPerspectives	Perspectives
XYZViews	Views

Note: In this case, “XYZ" is an example short name for the customer.

If your client project uses plug-in modules you may need to put definitions shared between the plug-ins and your main configuration in a separate module which both the main configuration and the plug-in configuration then inherit. A suggested module name would in this case be XYZCommon.

If your client will be large, it might be worth considering splitting the view-, perspective- and menu configuration into functional areas.

4.5 Testing your configuration

Since XML schema validation is sometimes a blunt tool, Ciguan features a special Junit test class which you can use to test your configuration. See section 15.1 for more details on how to use this test.

5 Basic configuration structure

The following section describes the actual sequence in which the various sections should be placed. Details on each specific configuration can be found later in this document, and in the reference guide.

Section	Description
Configuration file header	Each file should always have a standard header, declaring the root element and schema definitions.
Configuration format version	Each file should indicate what version of the CWF configuration it uses. (The current version is 9.2)
Module inheritance	Each module may inherit other modules. Specify all inherited modules here.
Locales	All locales that should be available to the end users.
Metadata	All metadata for the application
Plug-in identification	If the module is a plug-in, it should specify its identity, version and maker here.
Dictionary	All dictionaries that should be loaded for use in translations to foreign languages should be defined here.
Application server plug-ins	All application server plug-ins should be defined here.
Bean factory	All beans that you want to make available for creation via the bean factory should be defined here.
Transport configuration	All definitions related to the transport layer should be defined here.
Data sources	All data sources that you wish to use in your client should be defined here.
Get methods	All custom get methods you wish to use in your client should be defined here.
MVC definitions	All views you want to use in your client should be defined here.
Perspectives	All perspectives you want to use in your client should be defined here.
Menus	All menus you want to use in your client should be defined here.
Cache references	All wanted referential cache lookups should be defined here
Parameters	All server specific parameters should be defined here
REST	Resources accessible through REST as well as custom JSON serialization rules should be defined here.
Components	All configured application server components should be defined here.

5.1.1 Ciguan based applications

For stand-alone Ciguan based web applications, the starting module is defined in your web.xml deployment descriptor as an initiation parameter to the context listener as follows:

```
<context-param>
    <param-name>cwf.loadModule</param-name>
    <param-value>com.cinnober.as.conf.TE</param-value>
</context-param>
```

The context listener itself is defined as follows:

```
<!-- CWF context listener -->
<listener>
    <listener-class>
        com.cinnober.as.servlet.AsServletContextListener
    </listener-class>
</listener>
```

5.2 Bootstrap implementation class

In order for Ciguan to boot its components properly, you need to configure a bootstrap implementation class. Ciguan ships with a default bootstrap implementation.

Module	Class	Note
Ciguan	com.cinnober.as.impl.AsBootstrap	Default Ciguan bootstrap
Custom	A class that extends AsBootstrap, overrides start and adds customer specific application server components	Customer project specific bootstrap

5.2.1 Ciguan based applications

For stand-alone Ciguan based web applications, the bootstrap implementation is defined as an initiation parameter to the context listener as follows:

```
<context-param>
    <param-name>cwf.bootstrap</param-name>
    <param-value>com.cinnober.as.impl.AsBootstrap</param-value>
</context-param>
```

6 Common configuration

6.1 Configuration file header

Each configuration module should have a standard file header as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration id="Configuration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="
http://xml.cinnober.com/xsd/cwf-9.2.xsd"
    xmlns:xi="http://www.w3.org/2001/XInclude">
```

Note: The location of the CWF XML schema should not contain line breaks; it has only been done to improve readability in the example.

6.2 Configuration format version

In order to simplify migrations to new Ciguan configuration format versions, each configuration module should always specify its version. This done through the `Version` element:

```
<Version major="9" minor="2"/>
```

Always let the version element be the first element in each document.

6.3 Module inheritance

To let a module inherit another module, which in reality means that the inherited module is loaded ahead of the inheriting module so that definitions in it can be referenced, you place one or more `inherits` elements at the top of the module, immediately after the `Version` element:

```
<inherits name="com.cinnober.as.conf.MarketInstrumentModel"/>
<inherits name="com.cinnober.xx.conf.XXTransportConfiguration"/>
```

In the example above, the module inherits both the standard MarketInstrumentModel module, as well as the customer specific XXTransportConfiguration module. Note that the suffix is omitted in the example, which assumes that the files carry the recommended `.cwf.xml` extension.

6.4 Plug-in identification

If your configuration module is a plug-in contributing to the user interface, you need to add a `CwfPlugin` identification element. This element should be placed immediately following the inheritance definitions:

```
<CwfPlugin
    id="XYZPlugin"
    version="1.0"
    author="Cinnober Financial Technology AB"/>
```

The plug-in ID should be unique in the system, so make sure you pick a value which is likely to not clash with other plug-in IDs.

Ciguan contains a standard view, `CwfPluginsViewportView`, which can be used to view all loaded plug-in modules.

Note: Plug-in modules are used primarily in the client where server side code and user interface definitions are delivered as units commonly referred to as plug-ins. This plug-in concept is primarily used for custom instrument specific functionality.

6.5 Dictionary

To load a dictionary file containing translations to foreign languages, you place one or more `AsDictionary` elements, one for each dictionary you wish to load:

```
<AsDictionary
path="/com/cinnober/xyz/plugin/conf/dictionary_xyz.properties"/>
```

Note: Adjust the path to where on the class path the file is located.

For more information on the content of a dictionary, see section 10.5.

Caution: If you use more than one locale in your locale configuration, you need to make sure you supply dictionary translations for at least one of the locales.

6.6 Locales and default formatting patterns

To define locales, including default formatting patterns, you define an `AsDefaultPatterns` element and one or more `AsLocale` elements inside an `AsLocales` container element:

```
<AsLocales>
    <!-- Locales and default patterns go here -->
</AsLocales>
```

6.6.1 Formatting pattern

A formatting pattern is a template that controls how Ciguan will format data for its business types. A pattern is defined as follows:

```
<pattern
    type=""
    value=""/>
```

The attributes for a pattern have the following meaning:

Attribute	Meaning
type	The business type that the pattern is to be used with
value	The pattern to use for formatting

The formatting patterns are defined according to the Java `DateFormat` and `DecimalFormat` specifications.

6.6.2 Default formatting patterns

While there is no requirement to specify default formatting patterns, it will save you some XML configuration. A default pattern is a pattern which is used across all defined locales, unless a locale explicitly defines its own pattern. Each pattern is tied to a specific business type.

A default formatting pattern section is defined as follows:

```
<AsDefaultPatterns>
    <pattern type="Amount" value="#,##0.00"/>
    <pattern type="Volume" value="#,##0.######"/>
    <pattern type="Price" value="#,##0.00"/>
    <pattern type="Decimal" value="#,##0.00"/>
    <pattern type="Percent" value="#,##0.00'%'"/>
    <pattern type="InterestRate" value="#,##0.000"/>
    <pattern type="BasisPoint" value="#,##0'bps'"/>
    <pattern type="Date" value="yyyy-MM-dd"/>
    <pattern type="Time" value="HH:mm:ss"/>
    <pattern type="SecondsSinceTime" value="HH:mm:ss"/>
    <pattern type="DateTime" value="yyyy-MM-dd HH:mm:ss"/>
```

```
    <pattern type="Integer" value="#0"/>
    <pattern type="Long" value="#0"/>
    <pattern type="Double" value="#0.0#"/>
</AsDefaultPatterns>
```

The patterns used in the configuration must adhere to the standard Java `NumberFormat` and `DateFormat` token constructs.

See earlier in this document for a description on the business type concept and a list of valid values.

6.6.3 Locales

A locale is defined as follows:

```
<AsLocale
    id=""
    group=""
    decimal="">
    <!-- Locale specific patterns go here -->
</AsLocale>
```

The attributes for a locale have the following meaning:

Attribute	Meaning
id	The ID of the locale. Must be a valid and available Java `Locale` identifier.
Group	The character used as group separator.
Decimal	The character used as decimal separator.

Below is an example of a multi-locale configuration:

```
<AsLocale id="en_US" group="," decimal=".">
    <pattern type="Date" value="MM/dd/yyyy"/>
    <pattern type="Time" value="hh:mm:ss a"/>
    <pattern type="DateTime" value="MM/dd/yyyy hh:mm:ss a"/>
</AsLocale>
<AsLocale id="sv_SE" group=" " decimal=","/>
<AsLocale id="pt_BR" group="," decimal=".">
    <pattern type="Date" value="dd/MM/yyyy"/>
</AsLocale>
```

Note: If you do not specify patterns for each business type in a locale, it will automatically inherit the default patterns, so you only need to define the patterns you want to specifically define for a locale.

7 Metadata configuration

To define metadata, you define an `AsMeta` element:

```
<AsMeta>
    <!-- Metadata definitions go here -->
</AsMeta>
```

7.1 Search packages

The `SearchPackage` element is placed as a child to `AsMeta`. It defines packages which are searched when an attempt to instantiate an object is made. The search packages are iterated in their order of definition, and as soon as a successful instantiation is done, the iteration stops.

Example:

```
<!-- Packages to search when looking for classes to create -->
<SearchPackage
     packageName="com.cinnober.rtc.generated.te.tap.am"/>
<SearchPackage
     packageName="com.cinnober.rtc.generated.te.tap.cache"/>
<SearchPackage
     packageName="com.cinnober.rtc.generated.te.tap.cd"/>
```

7.1.1 Namespaces

With Ciguan you can assign a namespace to a search package. Namespaces deal with situations where you have multiple classes with the same simple name and you wish to use more than one of them in your configuration. In this case you will need to assign namespaces to the locations where the classes reside. Below is an example of a situation where you have two classes named "Account" which you want to use in your configuration:

```
<SearchPackage
    packageName="com.cinnober.rtc.risk" namespace="ri"/>
<SearchPackage
    packageName="com.cinnober.rtc.clearing" namespace="cl"/>
```

In this case you need to add the namespace to every location where the classes are referenced:

```
<view id="Foo" type="form" model="ri:Account">
...
</view>
```

7.2 Get methods

Get methods can be seen as a way to add virtual attributes to objects. When referencing the attribute, the get method code is invoked, and can then look up and return its data from virtually any location.

Get methods come in two forms, class based getters, and data source lookup based getters. The latter are defined through configuration, while the class based getters require code to be written.

Caution: Since get methods can be invoked quite frequently, particularly if the attribute is displayed in a table and is being used for sorting, the code should be made as fast as possible.

7.2.1 Class-based get methods

A class based get method is a class which implements the `AsGetMethodIf` interface. There is a base class which you can extend, `AsGetMethod`, which is

the preferred way to code a getter. Both the interface and the base class are parameterized, where the parameterization represents the object type which is passed as the item to retrieve the value from.

Note that `AsGetMethodIf` features two different value retrieval methods which are subject to customization:

```
Object getObject(T pItem);
```

This method should always return the raw non-translated value in its real data type. There is no default implementation, so a get method always needs to implement this method. Note that if the return value is something other than a String, you will need to override `getBusinessType`, and possibly `getBusinessSubtype` to allow proper formatting.

```
String getText(T pItem, AsDataSourceServiceIf pService);
```

This method by default calls `getObject` and attempts to format its return value if the `pService` parameter contains a non-null value. The formatting is controlled via the business type supplied by the get method itself.

Note: The default business type is always `Text`, so you must override `getBusinessType` and supply the business type of the `getObject` method if the returned value is something other than a string.

If you override `getBusinessType` and return either `Object` or `Constant`, you must also override `getBusinessSubtype` and return a value according to the following rules:

Business type	Business subtype
Object	The simple name of the `getObject` return type
Constant	The simple name of the constant group class which the `getObject` return value belongs to

7.2.2 Read consistency for class based get methods

The paragraph below is extremely important, so you should read it carefully.

Caution: If an item attribute used for sorting a data source is a get method, that get method must adhere to an important rule: It must be read consistent. Being read consistent means that the get method must return the same value for each call with the same item. If the method returns a derived or calculated value, it should only base the calculation on data available in the item itself. It should never use peripheral information that may change during the period over which the item is stored in the data source. Failure to comply with the read consistency rule may lead to assertion failures and/or corrupt data sources.

Also note that the object passed into the get method must not be modified in any way by the get method as this can lead to assertion failures and/or corrupt data sources.

7.2.3 Data source lookup get methods

Data source lookup get methods are defined solely through configuration. Each get method has a source from which the value is retrieved, and also a definition of the key used for looking up the source item. Note that in order for the lookup to work properly, you need to define a data source list which contains the source items.

A data source lookup get method is defined as follows:

```
<AsGetMethodSource source="View" name="viewName">
    <AsGetMethod type="ViewElement" field="parentInternalId"/>
</AsGetMethodSource>
```

In this example, you can refer to an attribute named “viewName" on objects of type `ViewElement`. Doing so will result in a lookup of a View object with its key value equal to the value of the ViewElement’s parentInternalId attribute. The data source “text" attribute will then be returned. To pick a different attribute, you can add a “field" attribute to the `AsGetMethodSource` element which specifies what field you wish to return.

Changing the getter name

You can add a `name` attribute to the `AsGetMethod` element to change the get method name, which by default is the `name` attribute from the `AsGetMethodSource` element. This concept must be used if several attributes in an object are references to the same foreign object.

7.2.4 Elvis operator get methods

Elvis operator get methods are a very special form of get methods used only for state metadata. Below is an example of an Elvis operator get method:

```
<AsGetMethod type="User" name="enabled"
    expression="isDisabled?disabled:enabled"/>
```

7.3 Metadata definition

By defining metadata, you make the client aware of the structure of a server side object, thus enabling it to for example automatically assign relevant input fields to attributes. Every class that you intend to use as a model object in a view must have metadata created.

Ciguan automatically creates metadata for all view models and their associated BLOB mappings, all context object classes, and all data source item classes. Normally you do not need to define metadata unless you want to explicitly assign a business type to an attribute that either does not have a business type, or if you for some reason want to assign a different business type.

7.3.1 Specifying server request names

By default, each model being submitted from a form will have a name that is `ServerRequest`. This ensures that the request is processed by the request plug-in. While this is ok for all requests passed to the back-end, you may sometimes want to prevent a request from being sent to the back-end and instead let it be processed by another plug-in. In this case you can explicitly specify a server request name as follows:

```
<MetaData className="LogonReq"
    serverRequestName="SessionRequest"/>
<MetaData className="LogoutReq"
    serverRequestName="SessionRequest"/>
```

In the above example, the logon and logout requests have a specific server request name that allows them to be processed by the session plug-in instead of the standard request plug-in.

7.3.2 Mapping explicit business types

For objects without annotated attributes, you can explicitly map a business type to an attribute. In order to do that, you add `Attribute` child elements to the `MetaData` element, one per attribute that you need to map a business type for.

```
<MetaData className="Foo">
    <Attribute attributeName="myPrice" businessType="Price"/>
</MetaData>
```

See earlier in this document for a description of the business type concept and valid business type values.

7.3.3 State attributes

State attributes are special attributes which are intended for client side CSS styling. These attribute values are sent along with all attributes requested by the subscription, but the client does not need to specify them when subscribing. Instead, it is the metadata configuration that defines which state attributes are sent.

You can specify several attributes to use as states, separated by commas.

A state attribute can for example be a flag indicating if the object is enabled or not, in order for the client to display it in a grayed out fashion.

```
<MetaData className="Foo" state="myState"/>
```

In the example above, the value of the attribute `myState` will be sent to the client and used as a CSS class on every column in a viewport view that displays data in a data source that contains objects of type `Foo`.

The `myState` attribute can be one of the following:

- An attribute in the `Foo` class
- A name of a get method operating on the `Foo` class
- An Elvis operator get method

Note: You can return more than one CSS class from an attribute or get method if you return a string with tokens separated by spaces. In that case, every token becomes a separate CSS class in the DOM tree.

7.4 Suppression

You can choose to suppress entire classes, attributes, or combinations of classes and attributes when generating the metadata. To do this, use the following elements:

Element	Use
SuppressClass	`<SuppressClass className=""/>` The `SuppressClass` item can be used to suppress entire classes when transforming data between the client and the server. The item also disables metadata creation for that class.
SuppressAttribute	`<SuppressAttribute attributeName=""/>` The `SuppressAttribute` item can be used to suppress an attribute when transforming data between the client and the server. The item also disables metadata creation for that attribute, regardless of in which class it appears.
SuppressClassAttribute	`<SuppressClassAttribute className="" attributeName=""/>` The `SuppressClassAttribute` item can be used to suppress an attribute in a specific class when transforming data between the client and the server. The item also disables metadata creation for that combination of attribute and class.

7.4.1 Directional suppression

A special case for the suppression is when you specify an additional `direction` attribute to the suppression tags. This means the following:

- Metadata should be created as per normal
- The suppressed attribute should not be copied when transforming between server and client format

Valid values for the direction attribute are:

Direction	Description
In	Value is not copied to client side format
Out	Value is not copied to server side format

Example of suppressing an attribute from the server to the client:

```
<SuppressClassAttribute
    className="User"
    attributeName="hashedPassword"
    direction="in"/>
```

8 Application server configuration

This section covers standard configuration of the application server for all areas not covered by the previous two sections. More elaborate configuration concepts are covered in the next section.

8.1 AS plug-ins

To define an AS plug-in, you add one `AsPlugins` container element, followed by one `Plugin` element for each plug-in you wish to include:

```
<AsPlugins>
    <Plugin
        pluginClass="com.cinnober.as.plugin.impl.AsRssPlugin"
        parameters="url=http://www.cinnober.com/press.xml,
                    pollInterval=60"/>
</AsPlugins>
```

Caution: There should not be any line feeds in the parameter attribute. This has merely been added for readability purposes.

The `parameters` attribute is a string containing all parameters that should be passed to the plug-in. The parameters can be separated by either a comma or a semicolon, and they are structured as name/value pairs separated by an equal sign.

It is up to the plug-in developers to decide what parameter names to use, so you need to check the plug-in documentation to determine what parameters a specific plug-in handles.

If you want to remove an existing plug-in, you can remove it through the following definition:

```
<AsPlugins>
    <Plugin remove="com.cinnober.as.plugin.impl.AsRssPlugin"/>
</AsPlugins>
```

The example above will remove all plug-ins with the specified class, so if you have multiple definitions of the same plug-in but with different parameters, all of the plug-in instances will databe removed. There is currently no way to remove a single instance of a given plug-in class.

8.2 Bean factory configuration

The application server features a lightweight object factory called the bean factory. Its primary use is to simplify interface use instead of referencing implementation classes, but it can also be used to replace one implementation class with another. The latter is quite useful when you want to extend the standard bean implementations.

To configure bean factory mappings, you add one `AsBeanFactory` container element, followed by one `bean` element for each bean you want to configure:

```
<AsBeanFactory>
    <!-- bean definitions go here -->
</AsBeanFactory>
```

If you want to replace one of the standard bean implementations with your own, make sure your bean configuration is loaded after the standard modules. You replace a standard implementation by specifying the same interface as the original bean, but replace the implementation with your own:

```
<bean
    interface="com.cinnober.as.AsConnectionIf"
```

```
    class="com.cinnober.xyz.impl.MyOwnReplacementBean"
    singleton="false"
    parameters="false"/>
```

See the Application server API section for details on how to use the bean factory in your code.

8.2.1 Classes without an interface

Note that even if a class does not implement an interface, you can still use the bean factory to create it. In this case, set both the interface and class attributes to point to the implementation class.

8.2.2 Replacing standard beans

You can replace standard beans which are defined via an interface by configuring a new bean with the same interface but a different implementation class. In this case, the new bean can either choose to extend the one being replaced, or completely replace it by supplying a different implementation.

8.2.3 Singleton beans

By configuring the singleton attribute to a value of “true", each call to the bean factory for the same interface will return the same instance of the bean. This can for example be used instead of Compis, or when you want to avoid the traditional static `Foo.getInstance()` pattern which makes it harder to mock or replace an implementation of an interface.

8.3 Transport configuration

The transport layer is an extensive section of the application server which covers the following areas:

- Transport plug-ins
- Request transformer
- Metadata
- Broadcast flow handlers
- Broadcast processors
- Broadcast class filters
- ACC definitions
- Broadcast flow subscriptions

The transport configuration is always located inside an `AsTransportConfiguration` container element:

```
<AsTransportConfiguration>
    <!-- Transport configuration goes here -->
</AsTransportConfiguration>
```

Below is a description on how to configure each area of the transport configuration.

8.3.1 Transport plug-ins

A transport plug-in is a class which processes incoming data from the client. To configure transport plug-ins, add a `Plugin` element for each plug-in:

```
<Plugin
    pluginClass="com.cinnober.xyz.plugin.XYZPlugin"/>
```

A transport plug-in needs to extend the `AsTransportPlugin` base class. If you want to customize a standard plug-in with your own implementation, let it extend the standard plug-in you wish to customize. Finally, configure a

removal of the standard plug-in followed by the addition of your own plug-in. You also need to make sure that your configuration is loaded after the standard configuration in order for this to work.

Example of a replacement of the standard AS session plug-in:

```
<Plugin
    remove="com.cinnober.as.transport.plugin.AsSessionPlugin"/>
<Plugin
    pluginClass="com.cinnober.xyz.plugin.XYZSessionPlugin"/>
```

8.3.2 Request transformer

The request transformer configuration is configured inside a `RequestTransformerConfiguration` container element which is a child to the `AsTransportConfiguration` container element.

The request transformer has the following configurable items.

Item	Use

GenerateUniqueId	`<GenerateUniqueId`
	`  className=""`
	`  attributeName=""/>`
	The `GenerateUniqueId` item defines that a unique ID should be generated and populated in the specified attribute of a class during data transformation between the client and the server. The unique ID generation currently only supports character data, and the data is always generated in the form of an UUID.

CopyAttribute	`<CopyAttribute`
	`  className=""`
	`  fromAttributeName=""`
	`  toAttributeName=""/>`
	The `CopyAttribute` item defines that a value of an attribute in a certain class should be copied to another location in the same class during data transformation between the client and the server.

8.3.3 Broadcast flow handlers

To configure a broadcast handler, add a `BdxFlowHandler` element:

```
<BdxFlowHandler
    className="com.cinnober.as.bdx.AsBdxFlowAlertHandler"/>
```

Broadcast handlers are components responsible for setting up synchronization sequences and taking care of broadcasts and current value responses related to a particular broadcast flow. There needs to be a broadcast flow handler for every flow that the application server subscribes to, and you also need to configure each subscription. The latter is covered in a separate section below.

See the application server API section on information about how to write a broadcast flow handler.

Note: If you want to extend or replace a standard broadcast flow handler, simply configure your new or extended flow handler in your own configuration in a section which is parsed after the section where the standard flow handler is configured, and it will automatically replace the standard handler.

8.3.4 Broadcast processors

To configure a broadcast processor, add a `BdxProcessor` element:

```
<BdxProcessor
    className="com.cinnober.xyz.MyBdxProcessor"/>
```

Broadcast processors are components which receive a callback every time a broadcast is received, thus allowing them to take action on specific events. A typical use for a broadcast processor is to derive data from existing events and feed them into new data sources.

See the application server API section on information on how to write a broadcast processor.

8.3.5 Broadcast class filters

To configure a broadcast class inclusion filter, add a `BdxClass` element:

```
<BdxClass
    className="com.cinnober.xyz.MyEvent"/>
```

Broadcast class filters are a way to define which events should be passed through the broadcast reception code.

Note: As of this point, Ciguan does not use the broadcast filter configuration. Future releases may however do that.

8.3.6 Broadcast flow subscriptions

Tbd

8.4 Data sources

The data source configuration is more complex than other areas and holds its own structure inside an `AsDataSources` container element:

```
<AsDataSources>
    <!--Data source configuration goes here -->
</AsDataSources>
```

8.4.1 Lists

To define a list data source, add an `AsList` element.

```
<AsList
    id=""
    factory=""
    source=""
    type=""
    key=""
    text=""
    filter=""
    query=""
    memberFilter=""
    userFilter=""/>
```

The list attributes and their meaning are as follows:

Attribute	Meaning
Id	The ID of the data source, must be globally unique
factory	Fully qualified name of a class that must implement the `AsDataSourceFactoryIf` interface.
source	The ID of a list which will be the source of this list. Lists without a source are typically populated via broadcasts, while lists with a source are populated via notifications from the source.
type	The name of the class stored in the list. Can either be a fully qualified class name or a simple class name. In the latter case, the class must be locatable via the configured search package definitions.

Attribute	Meaning
key	The attribute which uniquely identifies an item in the list. Can be an attribute or a comma separated combination of attributes. Get methods are valid as attributed.
text	An attribute of combination of attributes which constitute the textual representation of an item. Typically used in list boxes and in trees.
filter	A filter expression or a fully qualified name of a filter class.
query	Boolean flag that indicates if the data source is used to hold a query response or not. See below for more details.
memberFilter	A filter class used at the member level.
userFilter	A filter class used as the user level.

There are primarily two ways to configure a list, either with or without a factory. Using a factory eliminates the need of all other attributes except the ID. The valid attribute combinations are as follows:

```
<id> <factory> | <id> [<source>] <type> <key> <text> [<filter>]
[<memberFilter>] [<userFilter>]
```

The source attribute is used to make a list a child of another list, which means that it will be notified about all events in the parent list. You typically configure a top level list which is unfiltered, and then you add child lists to it with relevant filters based on for example object state or similar.

Child lists can either use the source attribute, or be placed as child elements to its parent list definition. In both cases, they automatically inherit the type, text and key attributes, so you do not need to specify them. However, you need to specify member and user level filters explicitly for each level.

Note: If you want to create a child list to a list defined in another module, you must specify the source attribute.

Natural list keys vs. composite keys

The attribute defined as the list's key is used as key in the data source, so make sure it uniquely identifies an item. Some objects have their own unique key, but in case you need to construct your own key, you primarily have two options:

- Define a composite key by separating each key component attribute by a comma. This is the preferred option, and enables you to use the original objects as list items.
- Use a hand coded object in the list, preferably by extending the standard class, and maintain the list via a broadcast processor. This requires code to be written, but also gives you complete freedom in constructing keys.

Example of a list definition with a composite key:

```
<AsList
    id="MY_LIST" type="MyObject"
    key="attribute1,attribute2" text="name"/>
```

This example assumes that the class `MyObject` has two attributes named attribute1 and attribute2 which if combined will uniquely identify an item.

Note: There is a maximum limit of three attributes for configured composite keys. If you need more, it is an indication that you should switch to using a hand coded extension class which contains a unique key that you construct yourself.

Query data sources

A query data source is a special data source which typically holds items which are part of a query response. A query data source only exists on user level, and it is tightly bound to the search forms. You should only use query data sources as part of a search form.

Note: If you need a general purpose data source which only exists on user level, you should create it using a factory which only creates the user level list.

8.4.2 List filters

Expression based filters

Expression based filters are definitions of type "`<attribute><operator><value>`", where the attribute is either a name of an attribute or a get method available via the object stored in the data source. The operator is one of the following:

Operator	Meaning
=	Equals
!=	Not equals
=null	Is null
!=null	Is not null
~=	Starts with
>=	Greater than or equal to
<=	Less than or equal to
|=	Contains - used in conjunction with attributes containing comma separated lists of ID values

Class based filters

The list filter definition can also be a class name. In this case, an instance of the filter class will be instantiated when the data source is created. The class name can either be fully qualified, or via the simple name of the class. In the latter case, the class is assumed to be located in the `com.cinnober.as.datasource.filter` package.

A class based filter typically extends the `AsFilter` class and implements the `include` method.

Note: Class based filters do not have access to the current member or user ID. This is intentional, since regular filters are not intended for access based filtering.

Member and user filters

Member and user filters are always class based and are only intended for access based filtering. Do not attempt to use these filters for any other purpose, as it will most likely give you undesirable side effects. Use regular filters in combination with sub lists, and then apply member and user filters to do the access filtering.

Caution: In the current version of Ciguan, you must specify the member filter on all lists in a hierarchy as the member and user filters are not automatically inherited. This will most likely be corrected in future releases.

A member based filter typically extends `AsMemberFilter` while a user based filter extends `AsUserFilter`. Both filters must implement the `include` method. The current member ID and user ID are available through get methods in the respective super classes.

8.4.3 Trees

Trees are data sources where the data is built up in a top-down fashion. Each sub-level is linked to its parent level via a filter condition. To define a tree data source, add an `AsTree` element.

```
<AsTree id="MY_TREE">
    <root>
        <child
            source="MEMBERS_ALL"
            filter="associatedMemberId="/>
    </root>
    <node type="Member">
        <child
            source="MEMBERS_ALL"
            filter="associatedMemberId=../memberId"/>
        <child
            source="USERS_ALL"
            filter="memberId=../memberId"/>
    </node>
</AsTree>
```

There can only be one root element in a tree. However, a root can have more than one child element. Each child specifies a data source to populate data from, and a filter condition used to link the data to the parent node.

If a level in a tree is a many-to-many link object, you can hide that level. To do that, specify the item as a link element instead of a node element. Note that the syntax is slightly different in this case. There is no filter attribute on the child element of a link. Instead, the link element has a key attribute which defines the link condition.

Below is an example where a link is used.

```
<AsTree id="POSITIONS_BY_RISK_TREE">
    <root>
        <child
            source="RISK_CALC_NODES_ALL"
            filter="parentRiskCalculationNodeIdRef="/>
    </root>
    <node type="RiskCalculationNode">
        <child source="RISK_CALC_NODES_ALL"
filter="parentRiskCalculationNodeIdRef=../riskCalculationNodeId"
/>
        <child source="RISK_GROUPS_ALL"
filter="riskCalculationNodeIdRef=../riskCalculationNodeId"/>
    </node>
    <node type="RiskGroup">
        <child source="RISK_GROUP_ACCOUNT_REFERENCES_ALL"
            filter="riskGroupId=../riskGroupId"/>
    </node>
    <link type="RiskGroupAccountReference" key="accountId">
        <child source="POSITION_ACCOUNTS_ALL"/>
    </link>
    <node type="PositionAccount">
        <child source="POSITIONS_ALL"
            filter="accountId=../accountId"/>
    </node>
</AsTree>
```

Tree folders

Trees can also contain folders. A folder can be seen as a child node under which child nodes related to the folder parent node are listed. In essence it is a list mounting point. The folder is shown as a folder in the tree, and it also shows the number of child nodes as a counter next to the folder label. The following example shows a tree that uses folders:

```
<AsTree id="REFERENCE_DATA_TREE">
    <root>
        <folder type="Member" filter="associatedMemberId="/>
    </root>
    <node type="Member">
        <folder type="Member"
            filter="associatedMemberId=../memberId"/>
        <child type="User" filter="memberId=../memberId"/>
    </node>
</AsTree>
```

Conditional children and folders

The `folder` and `child` elements can take an optional `condition` attribute which in essence is a filter expression that defines whether or not the folder or child list should be present, based on attributes in the parent item.

Example of a tree where the member level has conditional subfolders:

```
<AsTree id="REFERENCE_MEMBER">
    <root>
        <folder type="Member" filter="memberType=1" />
        <folder type="ClearingAndTradingMembers"
            source="TRADING_AND_CLEARING_MEMBERS_ALL"/>
        <folder type="CustodianMembers"
            source="CUSTODIAN_MEMBERS_ALL" />
    </root>
    <node type="Member">
        <child type="Member"
            filter="associatedMemberId=../memberId"/>
        <folder type="User" filter="memberId=../memberId" />
        <folder type="PositionAccount"
            condition="dgcxIsTradingUnitMember=true"
            filter="ownerId=../memberId" />
        <folder type="CollateralAccount"
            condition="memberType=102"
            filter="custodian=../memberId"/>
    </node>
</AsTree>
```

In the example above, the position account and collateral account folders only occur where the member is of a relevant type for those categories.

8.4.4 List trees

List trees are data sources where the data is built up in a bottom-up fashion, which means that the lowest level items are the leaves, and each leaf can only be attached to one and only one branch. The main differences between a list tree and a tree are that a list tree will only contain branches for the actual leaves, and a list tree must also contain a known number of levels.

To define a list tree data source, add an `AsListTree` element as follows:

```
<AsListTree
    id="POSITIONS_BY_CLEARER_TREE"
    source="POSITIONS_ALL">
    <node key="clearerId" source="CLEARERS_ALL"/>
    <node key="accountId" source="POSITION_ACCOUNTS_ALL"/>
</AsListTree>
```

In the example, the data source POSITIONS_ALL is the leaf level, while POSITION_ACCOUNTS_ALL is the middle level, and CLEARERS_ALL is the top level.

Building list trees from flat structures (group-by trees)

While list trees normally would be built from a hierarchical object model where every level has a parent object, there is sometimes a need to build a tree from objects with no parents. An example is viewing members in a tree

where the parent level is the member type. Model-wise there is no member type object since it's in reality just an attribute on each Member object. Ciguan can build these trees using a single data source. Below is an example of how to configure a member type tree:

```
<AsListTree id="MEMBERS_BY_TYPE_TREE" source="MEMBERS_ALL">
    <node key="memberType"/>
</AsListTree>
```

Note: You cannot mix group-by and true reference mappings, so you need to make sure your configuration is consistent in this respect.

You can specify up to three group-by levels, but each level must include the key from the parent level and add a new attribute to it:

```
<AsListTree id="USERS_BY_MEMBER_AND_STATE_TREE"
    source="USERS_ALL">
    <node key="memberId"/>
    <node key="memberId,enabled"/>
</AsListTree>
```

8.4.5 Structuring data sources

In order to configure data sources in the most efficient way, you need to study your user interface requirements to get a good picture of how many sub sets of the same data you will need. For example, if an event contains an object with state information, you might want to create a hierarchy of lists where the top level contains all events regardless of any state. Then, you create sub-lists for each set of states that have a distinct business meaning. An example based on trades could for example be trades where the state requires some sort of action from a member. This would constitute a set called "alleged trades". Another set could for instance be all cancelled trades.

In order to create sub-lists, you either use the source attribute, or place the sub list definition as a child element to its source list, and then use the filter attribute to qualify the contained items.

Caution: Do not attempt to use user or member filters to filter items based on state information. User and member filters are only intended for access filtering.

8.4.6 Super user mechanism

The data source structure is normally filtered according to access rights on both member and user levels. However, if you need users with extended access rights, Ciguan features a super user mechanism that you can use. This mechanism allows you to bypass the member level access filtering.

The super user mechanism works by linking data sources on the global level to corresponding data sources on the user level. This way, super users are effectively allowed to look at all data.

Using your own super user conditions

If you have other conditions that apply to super users, you can write your own user data source implementation. Below is an example of a custom user data source implementation:

```
public class FooUserDataSources extends AsUserDataSources {

    public FooUserDataSources(String... pParameters) {
        super(pParameters);
    }

    @Override
    protected boolean isSuperUser() {
        return ...;
    }
```

```
}
```

To use your custom user data source class, simply configure it through the bean factory:

```
<bean

interface="com.cinnober.as.datasource.owner.AsUserDataSources"
    class="some.package.FooUserDataSources"
    singleton="false"
    parameters="true"/>
```

8.5 Cache lookups

In Ciguan there is a singleton cache component. This component allows data searches as follows:

- Object lookup based on type and key value
- Collection lookup based on type
- Referential lookup based on type, attribute and value

You access the cache lookup component as follows:

```
AsCacheIf tCache = AsCacheIf.SINGLETON.get();
```

In order for object and collection lookups to work, you must make sure that there is a broadcast populated data source for every type you want to look up.

In order for referential lookups to work, you also need to configure the lookup paths similar to the following example:

```
<AsCache>
    <cache-reference type="User" field="memberId"/>
</AsCache>
```

Then, in your code, search for referencing objects as follows:

```
Collection<User> tUsers =
    AsCacheIf.SINGLETON.get().getReferencingObjects(
        User.class, "memberId", "AAA");
```

The above example will return all users where the `memberId` attribute has the value "AAA".

Summary: A cache reference definition is basically a way to divide all objects of a certain type into groups based on each unique value of the specified attribute, and the `getReferencingObjects` method is the way to retrieve all objects in a group which matches the specified attribute value.

8.6 Parameters

In Ciguan you can define application server parameters. These parameters are used by specific implementations, so you need to make sure that the parameter names are unique.

Parameters are defined as follows:

```
<AsParameters>
    <parameter name="foo.bar" value="4711"/>
</AsParameters>
```

Apart from in the normal Ciguan XML configuration, application server parameters can also be defined in a section in FwConfiguration.xml.

8.7 Components

In Ciguan you can configure application server components instead of via code in the bootstrap module. This eliminates the need to write a custom bootstrap module only to add components.

Components are defined as follows:

```
<AsComponents>
    <component className="com.example.FooBarComponent"/>
</AsComponents>
```

Note: Configured components are always created and started after all standard components.

9 Advanced application server configuration

9.1 Using data source factories

Data source factories are pieces of code that is used to create a data source when you need more control than the standard configuration can offer. Examples of situations when a factory may be used is for instance when you wish to programmatically populate the list, add special filtering, etc.

To use a data source factory, you specify the factory attribute when configuring your data source:

```
<AsList id="MY_LIST"
    factory="com.cinnober.xyz.XYZMyListFactory"/>
```

A data source factory must implement the `AsDataSourceFactoryIf` interface. This interface contains a number of methods:

```
AsListIf<T> createGlobalList(String pId);
AsListIf<T> createMemberList(
    String pId, AsDataSourceOwnerIf pMemberDataSources);
AsListIf<T> createUserList(
    String pId, AsDataSourceOwnerIf pUserDataSources);
Class<T> getItemClass();
boolean isRootList();
```

The first three are methods for creating lists at the global level, the member level and the user level. It is optional to create a list on any level, so if a level does not apply, you should return null.

The `getItemClass` method should return the class of the item stored in the list.

The `isRootList` method decides whether the list is considered a root list or not. Root lists are lists which receive their data through the normal broadcast dispatching mechanism, while non-root lists must rely on either setting themselves up as a child to another list, or have items inserted via a broadcast listener or broadcast processor.

Example of a factory that creates a global list:

```
public class AsTimeZonesListFactory
    implements AsDataSourceFactoryIf<AsTimeZone> {

    @Override
    public AsListIf<AsTimeZone> createGlobalList(String pId) {
        AsListIf<AsTimeZone> tList =
            new AsEmapiTreeMapList<AsTimeZone>(
                pId, AsTimeZone.class, "key", "key");
        return tList;
    }

    @Override
    public AsListIf<AsTimeZone> createMemberList(
        String pId, AsDataSourceOwnerIf pAsMemberDataSources) {
        return null;
    }


    @Override
    public AsListIf<AsTimeZone> createUserList(
        String pId, AsDataSourceOwnerIf pAsUserDataSources) {
        return null;
    }

    @Override
    public Class<AsTimeZone> getItemClass() {
```

```
        return AsTimeZone.class;
    }

    @Override
    public boolean isRootList() {
        return false;
    }

}
```

9.1.1 Handling of referential errors during data source creation

If a factory attempts to look up another data source while creating its data source and that lookup fails, you should throw an `AsDataSourceNotFoundException` in your factory code. This exception is picked up by Ciguan, which will then automatically retry the call to the factory as soon as all other data sources have been created. The maximum retry count is currently 3, after which the application server is terminated with a log entry informing about the failed lookup.

Lookup failures are normally caused by sequence errors where the data source being looked up is not yet created, so you should take care in defining your data sources in dependency order whenever possible.

Note: The global level data source creation is not retried, so throwing an `AsDataSourceNotFoundException` in the `createGlobalList` method will result in abort of the application server process start-up.

9.2 Programmatically populated lists using factories

Sometimes there may be a need to manually populate a list with items, for example if you need to aggregate or filter data in certain ways. In this case you can use a data source factory, and in the topmost level where you choose to create the list, include code to programmatically insert items in the list.

Example of a factory populating the list with items:

```
public class AsTimeZonesListFactory
    implements AsDataSourceFactoryIf<AsTimeZone> {

    @Override
    public AsListIf<AsTimeZone> createGlobalList(String pId) {
        AsListIf<AsTimeZone> tList =
            new AsEmapiTreeMapList<AsTimeZone>(
                pId, AsTimeZone.class, "key", "key");
        populateTimeZones(tList);
        return tList;
    }

    private void populateTimeZones(AsListIf<AsTimeZone> pList) {
        pList.add(new AsTimeZone("Europe/Stockholm"));
        pList.add(new AsTimeZone("America/New_York"));
        pList.add(new AsTimeZone("Hongkong"));
        pList.add(new AsTimeZone("Brazil/East"));
    }

    @Override
    public AsListIf<AsTimeZone> createMemberList(
        String pId, AsDataSourceOwnerIf pAsMemberDataSources) {
        return null;
    }


    @Override
    public AsListIf<AsTimeZone> createUserList(
        String pId, AsDataSourceOwnerIf pAsUserDataSources) {
        return null;
```

```
    }

    @Override
    public Class<AsTimeZone> getItemClass() {
        return AsTimeZone.class;
    }

    @Override
    public boolean isRootList() {
        return false;
    }

}
```

Note: There are other ways to programmatically populate data in a data source. You can for instance write a plug-in that constructs the objects and submits them through the broadcast dispatching. In this case you would not need a factory at all; the list could be configured as per normal.

9.3 Using Set and Map attributes in filters

The application server features a special mechanism for comparing attributes. A typical equals-filter may look like the following:

```
filter="tradeTypes=1"
```

Normally, this would test for equality between the `tradeTypes` attribute and a value of "1". However, if the `tradeTypes` attribute is an instance of `java.util.Set` or `java.util.Map`, a test is instead made if the value "1" is present in the set, or exists as a key in the map. This is a powerful mechanism that you can use, particularly in combination with hand coded data source classes described in the next section.

In case your object is not hand coded, you can still use the set mechanism via a get method that returns a set or map.

9.4 Hand-coded data source item classes

In case you want to have complete control over your data sources, you can create your own classes to use as items. There are a few rules for what you can and cannot do in a class used as a data source item:

- Attributes used by the client must be declared as public and must not be declared as final, to honor serialization contracts.

If you wish, you can use private attributes and associated get methods instead of public attributes. However, the business type annotation currently only applies to attributes. In case you use a getter, the attribute name in the client is the name of the get method minus the "get" prefix, and the first character in lower case.

Example:

```
public String getUserName();
```

The getter above must be referred to as `userName` in the client configuration, filter criteria expressions, etc.

9.5 Using annotations for metadata

There are a number of annotations available which affect how metadata is created. These annotations are as follows:

Annotation	Use
@BusinessType	Specifies what business type an attribute is to be handled as. Affects input validation and formatting rules.

Annotation	Use
@ConstantGroup	Specifies what range of valid values an integer attribute has. Affects input widget choice.
@IdField	Specifies that an attribute is a unique ID. (Not currently handled by Ciguan)
@DisplayName	Specifies that an attribute is a display name. (Not currently handled by Ciguan)
@IdRefToObjectOfType	Specifies that an attribute is an ID reference that points to an object of a specified type. See section 9.8 for more information.

9.6 Populating list contents from broadcast processors

An easy way to create or derive data based on received broadcasts is to write a broadcast processor and use it to create new objects which are then submitted to the broadcast dispatcher. See later sections in this document for instructions on how to create and configure a broadcast processor.

To submit an object to the broadcast dispatcher mechanism, do this:

```
As.getBdxHandler().broadcast(tItem);
```

9.7 Replacing or extending application server components

If you want to extend or replace standard application server components, you can do so by altering the necessary configuration. Since practically all application server components are created through the bean factory, you replace a component by configuring a new implementation which uses the same interface class as the bean you want to extend. Also make sure your configuration is loaded after the standard configuration.

Below is an example of how to replace a standard bean:

```
<bean
    interface="com.cinnober.as.transport.AsRequestTransformerIf"
    class="
com.cinnober.as.transport.util.AsRtcRequestTransformer"
    singleton="false"
    parameters="false"/>
```

The original bean is configured as follows:

```
<bean
    interface="com.cinnober.as.transport.AsRequestTransformerIf"
    class="
com.cinnober.as.transport.util.AsRequestTransformer"
    singleton="false"
    parameters="false"/>
```

In this case, `AsRtcRequestTransformer` extends `AsRequestTransformer` and overrides a number of methods, but in theory it could supply an entirely new implementation.

9.8 Referential lookup translations

Ciguan supports referential lookup translations through annotations. If an attribute is annotated as a reference to the ID attribute of another object, Ciguan attempts to look up the referenced object, and if it can be found, its text attribute will be displayed instead of the ID. This means that there is no longer a need to write or configure get methods just to show the display name instead of the ID of a referenced object. In order for this to work, the following requirements must be met:

- There must be a data source which contains the referenced object type

- The reference attribute must be annotated using either `@IdRefToObjectOfType` or `@CwfReference`

Note: Formatting of referential lookups is currently only supported in the client, such as in detail views, local array tables and in the viewport box widget. Server side support for viewports and trees will be added to a future Ciguan version.

9.8.1 How to disable a referential lookup

While an automatic translation is what you would want most of the time, there could still be situations where you would want to display the value of the reference attribute. To prevent the referential lookup from translating an ID reference, you can use the format attribute on the field as follows:

```
<field name="barId" format="false"/>
```

10 Application server API

In case you need to write customized code in the application server, there are a number of functions available for you, to simplify the most common tasks. You reach most of them through the application server root. How to retrieve the application server root, and which functions it contains is described below.

10.1 Hub classes

The application server consists of a number of components accessible as singleton instances. There is one important hub class, the `As`. The hub class gives you access to virtually all of the singleton application server components.

Some components however, are instantiated in a per-session fashion rather than as global components. The session level components can be retrieved via the `AsConnectionIf` interface which you receive as part of the calls to the transport layer.

10.2 Bootstrap

The bootstrap is a singleton that is responsible for starting up all application server components. The bootstrap instance can be accessed as follows:

```
AsBootstrapIf tBootstrap = AsBootstrapIf.Singleton.get();
```

See chapter 5.2 for information on how to configure the bootstrap implementation class.

Below is an example of a custom bootstrap implementation that instantiates a custom application server component:

```
public class MyBootstrap extends AsBootstrap {

    @Override
    public void start(String pStartModuleName)
        throws AsInitializationException {
        super.start(pStartModuleName);
        MyComponentIf.Singleton.create();
        MyComponentIf.Singleton.get().startComponent();
    }

}
```

Note that you only need to call `startComponent` explicitly after creating your component. The other life cycle methods are called automatically by Ciguan.

10.3 Application server components

Application server components are code units that run as singletons. The application server components have a specific life cycle that involves the following stages:

- Start component
- All components started
- Synchronize external data
- Reload configuration
- Stop component
- All components stopped

10.3.1 Writing an application server component

Writing an application server component is relatively straightforward. You write a class that extends `AsComponent` and override any of the methods in the `AsComponentIf` interface as needed.

To make your component accessible to other code, you should also write an interface as follows:

```
/**
 * Component interface
 */
public interface MyComponentIf {

    // Service methods go here

    /**
     * Singleton instance of me
     */
    public static class Singleton {

        static MyComponentIf cInstance;

        public static MyComponentIf get() {
            return cInstance;
        }

        public static void create() {
            cInstance = AsBeanFactoryIf.Singleton.get()
                .create(MyComponentIf.class);
        }
    }

}
```

In order for this example to work, you also need to configure a bean in the bean configuration.

10.3.2 Configuring your component

Finally, you need to configure your component in order for it to be created and started. You configure your component as follows:

```
<AsComponents>
    <component className="com.example.FooBarComponent"/>
</AsComponents>
```

10.4 Bean factory

The bean factory is a lightweight object factory which you can use to configure which implementation that should be created for a certain interface. In general terms it's just a way to de-couple interfaces from implementations by deciding their binding at runtime, rather than at compile time. You configure it through XML, and you then programmatically create the beans by calling the `create` method. The bean factory does not currently support automatic instance creation during startup, so you will need to take care of that yourself if it is needed.

To use the bean factory in your own code, simply retrieve the bean factory from the application server root and execute the `create` method, supplying either an interface or an implementation class:

```
AsConnectionIf tConnection =
    As.getBeanFactory().create(AsConnectionIf.class);
```

You can also replace standard bean classes using the bean factory. In order to do that that, you need to configure the bean factory entry to have both the `interfaceName` and `className` attributes set to regular classes.

10.4.1 Singleton beans

You can use the bean factory for singleton purposes if you set the `singleton` attribute to true. The first call to `create` will return a new instance of the class while repeated calls to the `create` method will return the same instance as the first time.

10.4.2 Beans with parameters

You can let beans created by the bean factory take parameters when they are constructed. To do that, set the `parameters` attribute to `true`, add a constructor that takes a `String...` parameter, and invoke the bean factory like this:

```
MyBeanIf tBean = As.getBeanFactory().create(
    MyBeanIf.class, "foo", "bar");
```

Note that it is up to the bean implementation to interpret and validate the parameters that are passed.

10.5 Dictionary handler

The dictionary handler can be used to retrieve translations for texts. It is mainly the client which uses the translations, but in some cases it is needed for server side code too. One example is custom getters, which sometimes need to return localized strings. To translate a text, you use the following code:

```
String tText = As.getDictionaryHandler().getTranslation(
    ".message.myMessage", pLocale);
```

Note that you need to know the locale that the client uses when looking up translations. Custom getters will receive the locale as part of the call to `getValue()`. Other code will need access to the HTTP connection in order to find the current locale:

```
Locale tLocale = pHttpConnection.getAsConnection().getLocale();
```

10.5.1 Dictionary header

A translation dictionary must always start with a header which defines which locales the translations belong to. A header should always be the first row, and has the following layout:

```
NULL ; en_US
```

The NULL marker is a dummy key that identifies the header, and the value is a locale ID which should match a configured AS locale.

If your client has support for multiple languages, you can choose to either have translations for all languages in the same dictionary, or use multiple dictionaries, one per language.

10.5.2 Dictionary tags

Each translation dictionary uses keys to map the translations. The first part of the key is always a tag which classifies the entry. Tags always start and end with a period character.

You are free to define your own tags if needed. However, there are a number of standard tags already in use by the client. The standard tags and their use are as follows:

Tag	Use
button	Button texts
constant	Constant group values
field	Object fields
fieldset	Form field sets
menu	Menu items
message	Alert and information messages
tooltip	Button tooltips

There are more tags present in many dictionaries, and there might even be a prefix before the tag. For server side translations, this is however not normally the case.

10.6 Broadcast mapper

The broadcast mapper is an application server bean that is responsible for delivering an incoming message to one or more data sources. A broadcast mapper implements the `AsBdxMapperIf` interface, which has the following method:

```
boolean onBroadcast(
    final Object pMessage,
    final Map<Class<?>, Set<AsEmapiTreeMapList<?>>>
pClassToListMap);
```

While the message itself needs no explanation, the map is intended to be used for looking up the list(s) to which the message should be added. During start-up, the application server inspects all configured data sources as follows:

- An entry is added to the map for the exact type that the data source contains.
- A walk upwards in the class hierarchy is done, and if a data source is found, an entry is added to the map for that type.

The default implementation of the broadcast mapper performs a lookup of the type of the incoming message, and if a set of lists is found, the message is added to each list in the set.

Caution: While this logic is mostly sufficient, there are situations where the mapping needs to be extended. The most common one involves sub classing of messages. An example of this is the Platform `AlertMessage` class, which is extended in several levels. In this case it is not feasible to have a data source for the lowest subclass level, particularly not when the sub classing involves several branches. Instead, you would write a new implementation of the broadcast mapper that walks upwards in the class hierarchy, testing for an entry in the map for each level, and as soon as a match is found, add the message to all lists in the set, and then stop the walking. A word of caution though, any form of iteration in the broadcast reception can potentially slow down the message reception, so you should not walk too many levels in the class hierarchy. A recommendation is also to not use sub classing for messages if it can be avoided.

10.7 Broadcast flow handlers

Broadcast flow handlers are code units responsible for receiving raw broadcast data from the TEPS layer. A broadcast flow handler implements the `AsBdxFlowHandlerIf` interface, and thus handles the following tasks:

- Declares the broadcast class (envelope) being handled
- Declares the current value response being handled (If applicable)

- Defines the synch sequences needed for the flow in combination with a specific broadcast partition
- Processes broadcast reception
- Processes current value response reception

There is always one broadcast flow handler per broadcast flow.

10.7.1 Writing a broadcast flow handler

Writing a broadcast flow handler is as easy as writing a class that extends the `AsBdxFlowHandler` base class, and configuring it to be used. The hardest part to get right however, is the synch sequence response. To return the correct response, you need to know a little about the flow you are processing data from, namely what its primary publishing subject is.

An example of how to assemble a synch sequence for subscription groups might look like this:

```
@Override
public SynchPrimarySequence[] getSynchSequence(
    String pPartition, int pBdxPartition) {
    SubscriptionGroup[] tSubscriptionGroups =
        AsCacheIf.SINGLETON.get().get(SubscriptionGroup.class);
    List<String> tSubscriptionGroupKeys =
        new ArrayList<String>();
    int tPartitionNumber = getPartitionNumber(pPartition);

    for (SubscriptionGroup tSubscriptionGroup :
         tSubscriptionGroups) {
        if (tSubscriptionGroup.getPartitionId() != null &&
            tSubscriptionGroup.getPartitionId().intValue() ==
tPartitionNumber) {
            tSubscriptionGroupKeys.add(
tSubscriptionGroup.getSubscriptionGroupId().toString());
        }
    }

    SynchPrimarySequence[] tSequences =
new SynchPrimarySequence[tSubscriptionGroupKeys.size()];
    for (int i = 0; i < tSequences.length; i++) {
        tSequences[i] = new SynchPrimarySequence(
            new SynchSequence(tSubscriptionGroupKeys.get(i), 0),
null, pBdxPartition);
    }

    return tSequences;
}
```

Caution: It is important to get the synch sequence response correct; otherwise the publishing layer will need to issue retransmission requests if it detects a sequence number gap. If you see log entries related to retransmission requests, you should review the synch sequence setup code. Note however that incorrect synch sequence setup is not the only reason for retransmissions. Publishing on incorrect primary subjects can also trigger retransmissions, so you may need to review your application code too.

Setting up the broadcast and current value response classes is far simpler. Note that a flow may or may not have a current value response depending on its implementation. In case the flow does not have a current value response, you should return `null` from the `getCvResponseClass` method, and use the generic `SynchDataResponse` class for the R parameterization.

When receiving broadcasts and current value responses, you should generally iterate over the broadcast envelope content and submit each `MessageIf`

component through the broadcast layer. Below is an example from the Ciguan standard alert broadcast flow handler:

```
@Override
public void receiveBdx(String pSubject, BdxAlert pBdx) {
    if (pBdx.data != null) {
        for (BdxAlertData tData : pBdx.data) {
            getBdxHandler().broadcast(tData.bdx.mMessageIf);
        }
    }
}
```

Current value responses are generally structured in the same way as broadcast envelopes, so the processing is very similar. Each contained `MessageIf` component needs to be submitted through the broadcast layer.

10.7.2 Reference data dependency

In the `AsBdxFlowHandler` base class, there is a method named `isReferenceDataDependant()` which you can override if you are certain that neither sync sequences nor data direct processing and all its descendants are depending on reference data in any way:

```
@Override
public boolean isReferenceDataDependant() {
    return false;
}
```

In theory, this can speed up the application server start-up slightly, but the recommended way is to not override this method.

10.8 Broadcast handler

The broadcast handler is the application server entry point for broadcasts and cache updates. While it takes care of broadcast dispatching automatically, it has a few utility methods you can use in your code.

You locate the broadcast handler via the application server root:

```
AsBdxHandlerIf tBdxHandler = As.getBdxHandler();
```

10.8.1 Broadcast dispatching logic

The broadcast handler dispatches each incoming message to all its listeners. These listeners include:

- The global data source container – when a message arrives, the container attempts to locate a root level data source with the same class as the message, and if one is found, the message is fed into it.
- All registered broadcast processors
- All registered broadcast listeners

10.8.2 Listening to broadcasts

If you need to listen to broadcasts in your own code, there are two ways you can achieve this:

- By configuring a broadcast processor
- By adding a broadcast listener

The broadcast processor mechanism is more automated, and it also enables you to configure parameters which are passed to the processor instance. A broadcast processor is active throughout the up time of the application server and cannot be removed. A broadcast listener is more lightweight, requiring no configuration. Broadcast listeners can be added and removed during the up

time of the application server. Which one you should choose depends on the situation.

To add a broadcast listener programmatically, do something similar to the following:

```
MyBdxListener tListener = new MyBdxListener();
As.getBdxHandler().addBdxListener(tListener);
```

The example assumes that your `MyBdxListener` class implements the `AsBdxListenerIf` interface.

Caution: If you add a broadcast listener for temporary use, you must make sure that the listener is removed when it is no longer needed. Otherwise it will keep being called. Adding temporary broadcast listeners and forgetting to remove them will eventually slow down the broadcast dispatching.

10.8.3 Writing a broadcast processor

You write a broadcast processor by creating a class that extends the `AsBdxProcessor` base class, and override either the `onBroadcast` or the `onReferenceData` method, or both, depending on your needs.

When your class is done, you configure it by adding a `BdxProcessor` definition in your configuration:

```
<AsTransportConfiguration>
    <BdxProcessor
        className="com.cinnober.xyz.bdx.XYZBdxProcessor"
        parameters="foo=1,bar=2"/>
```

In the example, the two parameter values will be available to the broadcast processor by calling the `getParameter` method:

```
String tFoo = getParameter("foo");
String tBar = getParameter("bar");
```

A broadcast processor also receives a start signal when it is launched, via the `start` method. You can override this method and add code to be executed during the start if needed.

10.8.4 Submitting broadcasts

One area of use for a broadcast processor might for instance be to create new data based on existing broadcasts. It is a good way to for example create summary data sources this way. When you have created a summary item in your code, you can submit it through the normal broadcast processing flow by calling the broadcast handler:

```
MySummaryItem tItem = new MySummaryItem();
tItem.setFoo(pEvent.getMemberId());
tItem.setBar(pEvent.getQuantity());
As.getBdxHandler().broadcast(tItem);
```

10.9 Transport plug-ins

A transport plug-in is a class which processes incoming data from the client. The general idea behind transport plug-ins is to divide functionality into separate units, so the standard application server configuration includes the following plug-ins:

Plug-in	Responsibility
`AsClientLogPlugin`	Client logging
`AsDataSourcePlugin`	Data source related requests.
`AsFileUploadPlugin`	File upload processing

Plug-in	Responsibility
`AsFormHandlerPlugin`	Form handler processing
`AsRequestPlugin`	Server side request-response handling.
`AsSessionPlugin`	Session related functionality, login, logout, query session state.

Each configured transport plug-in will receive a call for each message according to the `AsTransportServicePluginIf` interface. It is then up to the plug-in to decide whether or not to take action on the message or not.

10.9.1 Writing a transport plug-in

Writing a transport plug-in is as easy as writing a class which either implements the `AsTransportServicePluginIf` interface, or extends the `AsTransportPlugin` class, where the latter is the preferred way.

Note: If you want, you can explicitly declare a constructor that takes an `AsConnectionIf` parameter, in case you wish to explicitly keep a handle to this parameter.

10.9.2 Receiving client messages

Client messages are received through the `onMessage` method:

```
void onMessage(AsConnectionIf pConnection, CwfMessage pMessage);
```

`CwfMessage` is a wrapper object that contains the actual client request data, a client message handle, and the name of the request. In general, the request name can be found either as a component in the `MvcRequestEnum` or `MvcModelNames` enumerations.

10.9.3 Session invalidation processing

If your transport plug-in needs to do special processing when a user session is being terminated, you can override the reset method:

```
void reset(AsConnectionIf pConnection);
```

The reset method is called as the last step during HTTP session invalidation.

Normally there should be no need to override the reset method, unless your transport plug-in has explicitly allocated resources that need to be disposed of, such as broadcast listeners, data source listeners, etc.

10.9.4 Returning client responses

If the plug-in needs to return a response to the client, you need to take care in using a proper response name, and to pass the request handle. The application server transport layer contains a response queue to which you add a response according to the following example:

```
CwfData tData = new CwfData(MvcModelNames.ServerResponse);
tData.setProperty("foo", "Example data passed to the client");

CwfMessage tResponse = new CwfMessage(
    tData, pMessage.getHandle());
pConnection.getTransportService()
    .addClientMessage(tResponse);
```

Note: In the example, `pConnection` and `pMessage` refer to the `onMessage` call parameters.

In some cases, the constructor argument to `CwfData` (which is the message name) is not of any importance to the client. However, there are situations where the message name does have importance, like when sending a response to a form submission, so you should always consider using a value from the

`MvcEventEnum` enumeration, or whatever a suitable value would be depending on the situation.

Sending simple responses

The base class for a transport plug-in features two methods which you can use to send a simple response to the client:

```
protected void sendOkResponse(
    AsConnectionIf pConnection, int pHandle);

protected void sendErrorResponse(
    AsConnectionIf pConnection, int pHandle,
    int pErrorCode, String pErrorMessage);
```

Mimicking a real back end response

If your application server code is called from a form, the response in the first example will not suffice. Using the simple responses above will work, but if you need even better control, there are two ways to achieve this:

- Instantiate a `ResponseMessage` object (or an appropriate subclass) with an attached `TapStatus` object and transforming it to a `CwfData` object through the request transformer. This gives you the most control of the returned data, including type safety etc.
- Set the `ATTR_STATUS_CODE` and `ATTR_STATUS_MESSAGE` attributes on the `CwfData` object in the previous example. The `ATTR_STATUS_VALUE` attribute should have a value of 3001 to indicate success, according to the `TapStatusCode` class hierarchy. The `ATTR_STATUS_MESSAGE` should be a message in English. A translation of the message is then automatically done on the client side. This is similar to calling `sendErrorResponse` but gives you exact control of the code and the message, and you can also set other properties you wish to pass to the client.

10.9.5 Adding transport plug-ins

You add transport plug-ins by configuring them as part of the transport configuration section:

```
<AsTransportConfiguration>
    <Plugin
        pluginClass="com.cinnober.xyz.plugin.XyzSessionPlugin"/>
```

10.9.6 Enabling the user session inactivity functionality

Ciguan has the capability to supervise user inactivity, and disconnect a user who has been inactive too long. To use the inactivity functionality, you must do the following:

- Make sure you have a customer specific session plug-in extension.
- Override the `populateSessionData` method and set the desired inactivity in minutes according to the example below.

```
@Override
protected void populateSessionData(
    AsConnectionIf pConnection, CwfDataIf pData) {
    super.populateSessionData(pConnection, pData);
    // 15 minutes of idle time
    pData.setProperty(ATTR_MAX_INACTIVE_INTERVAL, 15);
}
```

10.10 Authorization handler

The authorization handler is a component which manages CFT platform ACC rights tokens and supplies utility methods to check for access to these rights.

You locate the authorization handler through the application server root:

```
AsAuthorizationHandlerIf tHandler =
     As.getAuthorizationHandler();
```

10.10.1 Checking access to services

The authorization handler has two methods to check for access to a certain token:

```
boolean checkAccess(String pUserId, String pServiceRef);
boolean checkAccess(String pUserId, FwServiceRef pServiceRef);
```

The first of these methods is primarily used by the client, when granting or revoking access to views. The parameter string is the name of the `FwServiceRef` attribute with its "ACC_" prefix removed.

The second method takes an `FwServiceRef` attribute and is intended to be used by server side code.

Note: In order for the client side method to work properly, you need to configure each ACC definition class which you intend to use in your MVC configuration.

10.11 Formatter

The application server provides a formatting component which you can use to format return values targeted for the client. A typical use for this is when a custom get method returns a computed value of some kind and wants it formatted according to the rules dictated by the client session's locale.

You find the formatter via the following method:

```
AsFormatIf tFormatter = As.getFormatter();
```

Note: The standard Ciguan formatter is created through the bean factory, so if you want to replace it with your own, make sure you configure it through the bean factory configuration.

10.12 Application server connection

The `AsConnectionIf` interface is the point from which you can access all the application server session level functionality. You will need a HTTP servlet request in order to retrieve the application server connection associated with it:

```
AsConnectionIf tHttpConnection =
    As.getAsConnection(pServletRequest);
```

The connection features a generic data storage in which you can place your own data. In essence it is a map with a class as the key and an arbitrary object as the data. You can use it through the following methods:

```
<T> void setConnectionData(Class<T> pType, T pObject);
<T> T getConnectionData(Class<T> pType);
```

The connection can be used to access the per-session functionality that the transport layer provides. In the following sections, each part of the transport layer is described in detail.

10.12.1 Data source service

The data source service is a function rich component that provides access to data sources. Most of the functions in the `AsDataSourceServiceIf` interface are intended for application server internal use, but one is geared towards custom application server code.

To send a data source related message or event to the client, you use the following method:

```
void addPendingDataSourceEvent(CwfMessage pEvent);
```

Note: If the message is a response to a client request, you need to make sure that the handle in the client request is passed back in the response.

10.12.2 Request service

The request service is a component that handles requests to and responses from the back end. It has a number of functions to send requests with or without returning a response to the client, and with or without transforming a client request. The latter form is intended when custom code needs to instantiate TAP messages directly and send them to the back-end, for instance when a request from the client should trigger several requests to the back-end.

For more details on the request service, see the `AsRequestServiceIf` interface.

10.12.3 Transport service

The transport service is a component that handles client requests and administers the transport plug-ins. Most of its functions are for internal use, but there is one method which can be used to send a message to the client:

```
void addClientMessage(CwfMessage pMessage);
```

Like with data source events and responses, you need to make sure that if the message is a response to a request from the client, that you pass the request handle back to the client.

10.12.4 Current session locale

You can retrieve the current locale associated with the HTTP session from the application server connection through the following method:

```
Locale getLocale();
```

Caution: The locale can change if the user selects a new language, so you should not keep a permanent reference to it.

10.12.5 Session data

You can retrieve the session data structure from the application server connection through the following method:

```
AsSessionDataIf getSessionData();
```

The session data gives you access to information about the session, like if the session is logged in, what member and user that is using the connection, if the session is act on behalf etc.

Caution: You should always consider the session data as read-only data even if its interface contains methods to modify the content.

10.12.6 Populating the client user session

While the user session is maintained and managed on the application server, there is also a client version of it. See section 12.2.1 for a description.

The client user session is normally populated by the standard `AsSessionPlugin` transport plug-in. If a customer project wants to populate additional values that gets sent to the client, you need to do the following:

- Write a custom transport plug-in that extends `AsSessionPlugin`, override the `populateSessionData` method and set any property you need to make available in the client.

- Configure the custom transport plug-in to be used.

10.13 Data source containers

In case you want to locate the data source containers for the different levels, you can do that through the following methods in the application server root:

```
AsDataSourceOwnerIf getGlobalDataSources();

AsDataSourceOwnerIf getMemberDataSources(
    String pMemberId);

AsDataSourceOwnerIf getUserDataSources(
    String pMemberId, String pUserId);
```

Each of the three levels allows you to look up a data source based on its ID, and optional filter and sort criteria. Be careful though, specifying a filter and/or sort criteria may cause the data source to be created if it does not already exist. This is by design, but it might not be what you intended. Also note that it might create the container itself, if it does not already exist. This is also by design.

The only container which is always present is the global data source container. The two other levels are normally created when the first user at a member logs in for the first time, and when a user logs in the first time.

Caution: Custom code should normally only access the global data source level, and never specify any filter or sort parameters.

10.14 AS plug-ins

AS plug-ins are code units started during the application server startup. They are not intended for any particular use, so a plug-in developer can construct a plug-in for pretty much any task. A typical use for an application server plug-in is to collect data from an external system and feed it into data sources.

Plug-ins support parameter configuration via its XML configuration:

```
<AsPlugins>
    <!-- RSS channels -->
    <Plugin
        pluginClass="com.cinnober.as.rss.plugin.AsRssPlugin"
        parameters="url=http://di.se/rss,pollInterval=60"/>
</AsPlugins>
```

Parameters are "name=value" pairs separated by commas.

10.14.1 Plug-in life cycle

All AS plug-ins are started during the "synchronize external data" stage of the boot process. The AS plug-ins are started one by one, in the order in which they are defined. A plug-in goes through the following life cycle stages:

- The `start()` method is called with a reference to the broadcast dispatcher. At this point, all parameters are loaded and available to the plug-in.
- A thread is created and its `run()` method is called. The plug-in can choose to either stay dormant in the `run()` method, or let it return once the processing is done.
- The `stop()` method is called, which sets the "stop ordered" flag in the base class. If the plug-in is dormant in the run method, it should return from it as soon as the stop ordered flag has been set, in order not to leave any stray threads. When `stop()` is called and the thread is still active, an interrupt is sent to the thread. If you call `Thread.sleep()` in your plug-in

and catch an `InterruptedException`, you should then call `isStopOrdered()` to determine if it's time to terminate the thread.

10.14.2 Writing a plug-in

Writing a plug-in is quite simple. You write a class which extends `AsPlugin` and implements the run method. You can also optionally override the start method and do any initialization you might need, including parameter parsing.

Note: All configured parameters are guaranteed to be available when start is called.

When your plug-in class is finished, you configure it through the `AsPlugin` configuration element.

All plug-ins always run as separate threads in order not to inadvertently block the application server start-up thread. In general you should avoid placing code that performs time consuming tasks in the start method since this will block the start-up. Place such code in the run method instead.

Caution: Remember to always place all your code in the `run` method inside a try/catch block where `Throwable` is caught.

Sending broadcasts from a plug-in

To send a broadcast from a plug-in, you simply do the following:

```
MyBroadcast tBdx = new MyBroadcast();
getMessageHandler().broadcast(tBdx);
```

10.15 Summary handlers

Summary handlers are functions which operate on all items in the underlying data source. They are instantiated on a per column basis in the code that listens to data source events and forwards data to the client. All summary data is computed per event and the current values are sent to the client when data in the data source changes.

10.15.1 Writing a custom summary handler

To implement a custom summary class, you write a class that extends `AsViewportSummaryHandler` and specify the T parameter as the class it operates on. (Always the same class as the items in the data source)

Override `getBusinessType` and possibly `getBusinessSubtype` if needed. The base class assumes that the summary has the same business type as the column values, but this is not always true. (Count is one exception)

Implement `onDataSourceEvent` and handle the various types of events that can occur.

Implement `getHandlerType`.

Implement `getValue` and return a "raw" (unformatted) value. This value will then automatically be run through the formatter to produce a formatted value.

Below is an example of the count implementation:

```
public class AsViewportSummaryCountHandler<T>
    extends AsViewportSummaryHandler<T> {

    protected long mValue;

    public AsViewportSummaryCountHandler(
        AsGetMethodIf<T> pValueGetter,
        AsDataSourceServiceIf pService) {
        super(pValueGetter, pService);
```

```
    }

    @Override
    public Object getValue() {
        return mValue;
    }

    @Override
    public HandlerType getHandlerType() {
        return HandlerType.count;
    }

    @Override
    public CwfBusinessTypeIf getBusinessType() {
        return CwfBusinessTypes.Long;
    }

    @Override
    public void handleViewportEvent(
        AsDataSourceEventIf<T> pEvent) {
        switch (pEvent.getType()) {
            case ADD:
                mValue += 1;
                break;

            case UPDATE:
                // No action, count is still the same
                break;

            case REMOVE:
                mValue -= 1;
                break;

            case CLEAR:
            case DESTROY:
                mValue = 0;
                break;

            case SNAPSHOT:
                mValue = pEvent.getSnapshot().size();
                break;

            default:
                break;
        }
    }

}
```

Note: The count implementation overrides `getBusinessType` since a count is just a number and should be formatted as such, regardless of which column it is applied to.

10.16 User properties

10.16.1 Persistence in applications

At this point, Ciguan does not supply any form of persistence, so in this case, user properties will not be persisted by default. You will need to write a custom implementation of the `AsUserPropertyPersisterIf` interface and configure your implementation class as a bean in order to save user properties.

You will also need to write a plug-in that reads the user properties and submits them through the broadcast dispatching mechanism in order to load them into the application server data source.

10.16.2 Creating or updating a user property

To create or update a user property, you simply construct an `AsUserProperty` object and set all properties to the desired values. Finally you submit it through the normal broadcast processing, which takes care of updating the data source. Below is an example of creating a user property:

```
AsUserPreference tPref = new AsUserPreference(
    pConnection.getUserId(),
    tPerspective, tSlot, tView, tItem, tPreference, tValue);
As.getBdxHandler().broadcast(tPref);
```

The user preference object takes the following parameters:

Parameter	Meaning
pUserId	The ID of the user currently logged in
pPerspectiveId	The ID of the perspective currently being displayed
pSlotId	The ID of the perspective slot for which the preference is valid
pViewId	The ID of the view for which the preference is valid
pItemId	The ID of the item for which the preference is valid
pPreference	The ID of the preference
pValue	The value of the preference

There are different constructors available, depending on how many of the parameters that are relevant for the property you are creating. At a minimum, the user ID, preference and value need to be specified.

The interface `MvcUserPreferenceAttributesIf` contains a PREF_NONE attribute that all not explicitly defined parameters will be set to.

10.16.3 Removing a user property

Sometimes there may be a need to remove a certain user property. This is done in the same way as when creating or updating a user property, but setting the `deleted` property of the object to `true`.

Note: There is no method available for setting the deleted property at this point, so you need to set the attribute directly.

10.16.4 Persisting client side objects as user properties

While a user property is normally just a string value, there may be a need to save an entire client side object as a user property. This can be achieved by serializing the `CwfData` container into text and putting it into a property.

Below is an example of how to turn a `CwfData` object into text:

```
CwfData tData = pMessage.getData();
String tText = serializationUtil.toJson(tData);
AsUserPreference tPref = new AsUserPreference(
    pConnection.getUserId(), "MyPreferenceID", tText);
As.getBdxHandler().broadcast(tPref);
```

In the example, the `pMessage` and `pConnection` variables refer to the transport plug-in `onMessage` method parameters.

10.16.5 Restoring client side objects from user properties

If you have chosen to persist a client side object as a user property, you will need to restore it once it is read back. This is done as follows:

```
AsUserPreference tSettings = getPreference();
CwfDataParser tParser = new CwfDataParser(tSettings.getValue());
CwfData tData = tParser.parseCwfData());
```

Note: The `CwfDataParser` class can be used both on the server and in the client, but the `AsUserPreference` class is a server side only class.

10.17 File upload

Ciguan supports file upload through the conventional HTML multipart form data mechanism. The functionality consists of the following components:

- File upload forms (see later sections on how to configure this)
- An upload request that serves as the model for the file upload form.
- Upload file handlers
- Upload service servlet
- File upload plug-in

10.17.1 Execution flow

While a file upload might seem like just another form, it's quite different behind the scenes. This due to the fact that standard Ciguan forms are not true HTML forms, they are merely a collection of widgets where the values are sent via AJAX when the submit button is pressed. Since file uploading cannot be done over AJAX, Ciguan does the following in a file upload form:

- A HTML form is injected into the dialog.
- The form model is submitted via AJAX as per normal, but with no receiver in the application server. The submission is only done to obtain a handle to which the final return status is sent.
- The handle is inserted into the HTML form that wraps the dialog, and the form is submitted to the file upload service servlet via a POST action.
- The file upload service servlet parses the form data and assembles an object model.
- Each defined upload handler is called with its associated file data, plus all conventional form fields.
- The file upload service servlet assembles a file upload response message which is submitted through the application server transport layer and picked up by the file upload plug-in.
- The file upload plug-in examines each upload handler result and assembles a response where each upload handler result is passed as a sub code.
- The file upload plug-in sends the response to the client using the original handle in order to indicate the result in the client.

10.17.2 File upload request

Like any other form, a file upload form requires a model to hold the widget values. While the model will not be processed by the application server, you still need it in order for the MVC mechanisms to work. At this point you need to explicitly configure the model metadata using a specific server request name as follows:

```
<AsMeta>
    <MetaData
        className="MyFileUploadRequest"
        serverRequestName="FileUploadRequest"/>
</AsMeta>
```

Note: This requirement will be dropped in future Ciguan releases, but for now you need to do it.

Ciguan features a very basic class, `AsFileUploadReq`, which you can use as the form model if the only field you will have in the form is a file.

10.17.3 Upload file handlers

A file upload handler is a Java class that receives metadata about the file being uploaded, along with the raw file data. The handler should perform any actions you plan to do with the file data, and set a return code and message based on the outcome of the processing.

To write a file upload handler, write a class that extends `AsFileUploadHandler` and implement the `doProcess` method:

```
public abstract void doProcess(
    AsConnectionIf pConnection,
    AsFileUploadParam pFile,
    List<AsFileUploadParam> pFormFields);
```

The `pConnection` parameter is the application server connection through which you can read most of the standard application server functionality. The `pFile` parameter is the file associated with the form field that the handler is defined for, and the `pFormFields` parameter is a list of data for the other form fields.

Since the file upload functionality is based on the Apache commons file upload library, you can retrieve the file data via a `FileItem` instance that you obtain via a call to `pFile.getFileItem();`

When you have processed the file, you should always call the `setStatusCode` and `setStatusMessage` methods to indicate whether processing was successful or not. If you plan on using a configuration where you open a form on success or error, you should also call the `setResponseName` method to indicate the name of the response returned to the client.

10.17.4 Upload service servlet

The file upload service servlet is a standard application server component responsible for receiving the form data and calling each defined upload handler one by one. When all handlers are called, the servlet assembles an upload status request which is submitted to the application server transport layer. The status request is a message that will by default be processed via the standard application server file upload plug-in.

Note: There should normally be no need to extend this component.

10.17.5 File upload plug-in

The file upload plug-in is a standard application server transport plug-in that receives the upload status requests and assembles a response message which is sent back to the calling form. The plug-in can be extended to add functionality by overriding the following method:

```
public void handleFileUploadResult(
    AsConnectionIf pConnection,
    int[] pSubCodes,
    String[] pSubMessages,
    String pModelName);
```

If you want to return your own custom status codes based on the overall result which is available in the `pSubCodes` array, you can call the `setStatusCode` and `setStatusMessage` methods. You can also specify a name of the response message via the `setResponseName` method, in case you plan on using a form configuration where you open a new view upon success or failure.

Note: The response name should primarily be set by the upload handler, so setting it in the upload plug-in is a last resort option.

The `pModelName` parameter is the name of the file upload form model.

If you want to customize the return codes, or perform additional processing, write a class that extends the `AsFileUploadPlugin` class and replace the standard plug-in with your own through the following transport configuration:

```
<Plugin remove=
    "com.cinnober.as.transport.plugin.AsFileUploadPlugin"/>
<Plugin
    pluginClass="com.cinnober.xyz.plugin.XYZFileUploadPlugin"/>
```

10.18 Data source export

Ciguan supports viewport data source export functionality, also known as Excel Export. The functionality consists of the following components:

- A client side view for selecting the output format
- A data export servlet
- A data source export bean

10.18.1 Execution flow

The export is started when the user clicks the "Export to Excel" icon on the viewport toolbar. This triggers the following steps:

- An EXPORT event is sent to the viewport controller
- The controller assembles a parameter context
- The "DataSourceExportView" form view is launched with the parameter context
- When the user submits the form, a HTML POST request is sent to the data export servlet
- The export servlet parses the input parameters (see below), creates and calls the export bean, and finally writes the data to the output stream in the form of an attachment, which will prompt the user to save it as a file.

10.18.2 Request parameters

The following request parameters are passed to the data export servlet:

Parameter	Description
handle	The client request handle to which the response should be returned. Possible values are: ▪ CSV - Comma separated values ▪ XLS - Microsoft Excel (Not implemented) ▪ XML - Not implemented ▪ JSON - Not implemented
returnType	The type of file to be produced.
viewId	The ID of the view from which the export request originates.

10.19 Form handlers

Form handlers are server side code units dedicated for a form view. They are intended for server side tasks associated with forms, and provide functionality associated with various life cycle states of a form. These states are currently as follows:

- Form start-up
- Form submission

- Form shutdown

Note: Think of form handlers as server side assistants. Form handlers are not a server side replacement for the client, so even if a form handler task should fail, the client is not interrupted by it.

Note: You cannot replace the client side form submission with the `onFormSubmit` method; it is simply an additional helper task.

10.19.1 Implementing a form handler

Writing a form handler is as easy as writing a class that extends `AsFormHandler`, and implementing one or more of the methods below.

Method	Description
`onFormStartup`	Called when the form starts. You can choose to either return null, or a transformed object. If you choose to return a transformed object, it will be passed as a context object to the view.
`onFormSubmit`	Called just before the form is submitted.
`onFormDestroy`	Called when the form is closed normally.
`destroy`	Optional - called when the page has been reloaded. Think of this method as a `finally` clause in a `try/catch`, the method is normally called right after `onFormDestroy`, but if the form is not closed the normal way, the form handler will still be allocated, so this method is called as part of a page reload when all leftover form handlers are removed.

10.19.2 Configuring a form handler for use

To tie a form handler to a certain form, you also need to add an extra attribute to the view definition, as follows:

```
<view id="Foo" type="form" model="Bar"
    formHandler="com.cinnober.example.FooHandler">
```

10.20 Metrics counters

If you want to count occurrences of events and make the counter visible in the JMX MBean, you can do so through functions defined in the `AsMetricsIf` interface. Below is an example of how to increment a counter which will be visible in an MBean browser:

```
AsMetricsIf.Singleton.get().incrementCounter(
    MX_FAILED_READ_ATTEMPTS);
```

10.21 Services

Starting with Ciguan 10.4.0, there is now a structured way to implement services in the application server. Earlier, it has been somewhat difficult to write pure application server service implementations, mainly because there has not been an infrastructure for it. Various attempts have included extensions of the request service, the request plug-in, the data source service, using separate transport plug-ins, etc. Simply put, the implementations have been very different, thus making them harder to maintain and expand than it should be.

10.21.1 Service configuration

The new service infrastructure uses a new XML configuration section to define the service implementations and their input parameters:

```
<AsServices>
    <service
       requestClass="..."
       serviceClass="..."/>
</AsServices>
```

The attribute of the service element are as follows:

Attribute	Description
requestClass	The type of the request which should be handled by the service.
serviceClass	The implementation class which is responsible for handling the service calls.

10.21.2 Service handler

The service handler is an application server component which is responsible for parsing the service configuration and instantiating the various service implementations. The service handler is an application server component, and it is accessible through the `AsServiceHandlerIf` singleton instance. There should normally be no need to extend the service handler component, but if you wish to do so, you can use the bean factory configuration to replace the standard implementation with your own.

10.21.3 Service implementation

A service implementation typically extends `AsService` and supplies one service method per request type. You can either supply one service implementation per request type, or bundle several service methods in the same implementation, it all depends on what is the most practical case for the implementation being developed. On one hand, it is generally good to have small single-task implementation classes, but on the other hand it might also be good to keep a multi-request service in one implementation.

Note that there is no interface defining the specific service methods. The base class has a default service method which looks like this:

```
public final Object service(
    AsConnectionIf pConnection, Object pRequest);
```

The default service method will however never be called, your own methods will be called instead. This is done through reflection, which means that if you configure a service called `foo.bar.FooService` which takes a `FooReq` request and returns a `FooRsp` response, you would need to implement the following service method:

```
public FooRsp service(
    AsConnectionIf pConnection, FooReq pRequest);
```

Note: If you do not implement this method, the application server will not start, so you need to make sure your service methods match your service configuration.

An important fact to remember about service implementations is that they are singletons. Even though you can configure several services with different requests pointing to the same implementation, there will only be one running instance of the service implementation. This implies two things:

- The service implementation must be thread safe
- The service implementation should not keep state information

If you need to keep service state information associated with a user session, you can store it in the `AsConnectionIf` object. See the `setConnectionData()` and `getConnectionData()` methods.

Below is an example of a simple service implementation:

```
public class DemoService extends AsService {

    /**
     * @param pConnection
     * @param pRequest
     * @return the response
     */
    public ResultRsp service(
        AsConnectionIf pConnection, DemoServiceReq pRequest) {
        return new ResultRsp("Success", new TapStatus(
            3001, "Service execution OK"));
    }

}
```

The configuration for the above service would look like this:

```
<AsServices>
    <service
       requestClass="DemoServiceReq"
       serviceClass="com.cinnober.as.demo.service.DemoService"/>
</AsServices>
```

10.22 Task scheduler

The task scheduler is an application server component responsible for running scheduled tasks, which are either executed once, or on a repeated basis. The scheduler can be retrieved through the `AsSchedulerIf` singleton instance.

10.22.1 Writing tasks

You implement a task by writing a class that implements the `AsScheduledTaskIf` interface. A task is always executed in a separate thread, to avoid having long running tasks blocking the scheduler thread. Therefore, you should always give your task a good and descriptive name to recognize it easier in IDE stack views and stack dumps.

If you want to have your task executed on a repeating basis, simply let the `getIntervalMs()` method return a positive value. A zero or negative value means that the task should only execute once.

10.22.2 Submitting a task for execution

You submit your task for execution as follows:

```
AsScheduledTaskIf tTask = new FooBarTask();
AsScheduledTaskHandleIf tHandle =
    AsSchedulerIf.Singleton.get().schedule(tTask);
```

10.22.3 Cancelling a task

If you need to cancel a task, call the `cancel()` method on the handle object returned by the scheduler when you submitted your task for execution.

10.23 Response classes

If you need to send a generic response to the client from code which works with the server side object format, such as application server services, there is a generic response class called `AsResponse` which you can instantiate and

return. It contains an attribute of type `AsStatus` which is used to set the status code and the status message.

11 Writing application server code

This section contains information on how to write various types of application server code.

11.1 Transport plug-in routing

If you have a need for application server specific functionality and want to split it between transport plug-ins, you can use the server request name as a routing switch. Normally, all classes used as models in forms are using `ServerRequest` as their model name. This implies that the request gets picked up by the standard request plug-in and gets sent to the back end. For functionality purely implemented in the application server however, you will normally want to prevent this from happening. This can be achieved in many ways, but the recommended approach is as follows:

- Write your own request class
- Configure a `serverRequestName` attribute through explicit metadata configuration. (See below for an example)
- Check the `getName()` value of the incoming message in the call to `onMessage()` and only take care of the message if it matches your configured `serverRequestName` attribute value

Below are some examples of explicitly configure server request names:

```
<MetaData className="ChatConnectReq"
    serverRequestName="ChatConnectReq"/>
<MetaData className="ChatDisconnectReq"
    serverRequestName="ChatDisconnectReq"/>
<MetaData className="ChatEndReq"
    serverRequestName="ChatEndReq"/>
<MetaData className="ChatMsgReq"
    serverRequestName="ChatMsgReq"/>
<MetaData className="ChatStartReq"
    serverRequestName="ChatStartReq"/>
```

12 Configuring the user interface

12.1 A primer on context objects

Context objects are the key to transferring information from one view to another in the client. Context objects are automatically passed to views by the Ciguan framework. Some context objects are always available, while the presence of others depends on the user operation. For example, when right-clicking on an item in a viewport or tree, the corresponding object is automatically fetched from the server and passed into the view which is opened by selecting an item from the context menu that appears.

Context objects are also passed to views through a number of other operations:

- When dragging an item and dropping it on a view
- When opening a view or forwarding to a view from a form button
- As a result of a context lookup defined in a view
- When selecting a value in a widget which is configured for context update

In order for a view to accept a context of a certain type, the view definition must define that particular context type, and what to do with it. This is done through the `context` element. You can repeat the `context` element as many times as you need in order to copy all relevant parts.

Note: When Ciguan passes a context object of a certain type to a view that does not define it via at least one `context` element, it will be silently ignored.

A model can only hold one and exactly one instance of a context object of a certain type. As soon as a context object of the same type as an existing one is passed to the controller, the current object is replaced by the new one.

There are a number of ways to access the context objects in a view:

- Copying selected information into the model via the `context` definition. There is however no need to copy anything into the model, so if you only want to accept a certain context object type, you can omit the `source` and `target` attributes.
- Using it in filter expressions through the `$context.<type>.<field>` addressing notation
- Mapping a field to it through the `context` attribute
- Programmatically, by doing a call to `getContextObjects()` on the model

A word on multiple selections

When multiple rows are selected in a table, and the user right-clicks on one of them, all selected objects are passed to the client as an array. It is then up to the receiving view to either declare a single context, an array context, or both. Sending an array context to a view that only declares a single context will result in multiple launches of that view for each item in the array. This is typically used in conjunction with menu items of type auto-submit.

Caution: You should use multiple selections carefully. If the objects in the data source displayed by the table view are large and a user can select a large number of them, the time it takes to send the selection to the client can be long, and in extreme cases even lead to script execution warnings.

12.2 Special context objects

12.2.1 SessionModel

The `SessionModel` context object is an object passed to a view when it is opened from a menu item. The object is in reality a `CwfData` object with the following attributes:

Attribute	Content
memberId	The ID of the member for the user currently logged in
userId	The ID of the user currently logged in
isLoggedIn	Boolean flag indicating if a user is logged in or not
date	Today's date in yyyy-mm-dd format
isActOnBehalf	Boolean flag indicating if the session is act on behalf or not
actingUserId	If the session is act on behalf, this is the ID of the user performing the act on behalf
languageCode	The code of the language (locale) that the session is currently using
perspective	The ID of the perspective currently being displayed

Note: The session can contain additional attributes if your customer project has decided to populate more data into the session. This is typically done on the application server, in a specialized subclass of the `AsSessionPlugin` transport plug-in class.

12.2.2 MenuParameters

The `MenuParameters` context object is an object passed to a view when it is opened from a menu item that has a parameter attribute. The `MenuParameters` object is in reality a `CwfData` object where the configured menu parameters have been assigned to properties.

For example, if the parameters attribute looks like this:

```
<menuitem view="MyView" parameters="foo=0,bar=1"/>
```

Then, `MyView` needs to declare a `MenuParameters` context as follows:

```
<context type="MenuParameters"/>
```

Finally you can address the parameters, either directly in the view via the context syntax:

```
<field name="foo" context="MenuParameters"/>
```

You can also programmatically address parameters by retrieving all context objects from the model, locate the `MenuParameter` object, and for example call `getProperty("foo")`.

12.2.3 TypeAheadFilter

The `TypeAheadFilter` context object is a context object submitted via the viewport input box widget. Every time a character is typed in the widget, a `TypeAheadFilter` object is passed to the controller. The object has a single attribute named expression which you can use in filters, preferably in conjunction with the "starts with" operator:

```
<context
    type="TypeAheadFilter"
    filter="displayName~=expression"/>
```

12.2.4 The array context

The special array context object is used when multiple rows are being selected. This object has the `objectName` property set to the class name of each item with an array suffix "[]" appended to it. Each item in the array is located in a list named `arrayItems`.

Array contexts can be used in the form model, and as pure context objects, in the same fashion as regular context objects.

In the form model, an array context can be used in two ways:

- As an object array, resulting in a pre-populated array input widget.
- As an array of properties, resulting in a comma separated list of values.
- As an object array context, resulting in a pre-populated read-only table widget.

Example 1, copying the context into an object array:

```
<form id="MyArrayForm" type="form" model="MyArrayModel">
    <context type="MyObject[]" source="." Target="myObjects"/>
    <!-- form layout goes here -->
</form>
```

In this case, the `myObjects` attribute must be declared as an array of `MyObject` in the model class.

Example 2, copying a property from the context into a string:

```
<form id="MyArrayForm" type="form" model="MyArrayModel">
    <context type="MyObject[]"
        source="id" target="myObjectIds"/>
    <!-- form layout goes here -->
</form>
```

In this case, the `MyObject` class is assumed to have a String attribute called `id` which is appended to a comma separated list of values, and finally set in the `myObjectIds` String attribute.

When used as a pure context, the array is displayed in a table similar to what the array input widget uses, but without the input fields and the add button.

Example 3, using an array context as display-only information:

```
<form id="MyArrayForm" type="form" model="MyArrayModel">
    <layout>
        <fieldset id="MyObjects" context="MyObject[]">
            <field name="id"/>
        </fieldset>
    </layout>
</form>
```

In this case, the field set becomes read-only, displaying a table with properties from the `MyObject` class.

12.3 View definitions

The MVC definition section is quite an extensive area which will likely contain a lot of configuration, particularly for a large client application. You should spend some time thinking about how to structure the definitions, particularly if your client will feature plug-ins contributing to the user interface. An example of this is instrument plug-ins in the Real Time Clearing client application.

MVC view definitions are always added as child elements to an `AsMvc` container element:

```
<AsMvc>
    <!-- View definitions go here -->
</AsMvc>
```

There are many different view types, each one with its own definition syntax. Each view type is described in the following sections.

12.3.1 Data addressing

Addressing attribute data

To address an attribute with a given object type, you use the standard dot notation:

```
[<path>.]*.<attribute-name>
```

To address sub structures, you just append as many paths as needed to reach the attribute you want.

Addressing model data

Model data can be addressed via the following macro syntax:

```
$model.<attribute-path>
```

`<attribute-path>` is the location within the model.

A special model addressing scenario which cannot use the $model macro is if you have an array where an attribute at a certain array index needs to address another attribute at the same array index. This is usually needed when you have list box filters which depend on list box data on the same grid row. In this case you need to use the following syntax:

```
$row.<attribute-path>
```

In this case, `$row` will expand to the path of the enclosing field set and the current array index.

Addressing context data

Context object data can be addressed via the following macro syntax:

```
$context.<object-type>.<attribute-path>
```

`<object-type>` corresponds to the type of the context object, and `<attribute-path>` is the location within the object.

Addressing session data

Session data can be addressed via the following macro syntax:

```
$session.<attribute-name>
```

`<attribute-name>` is the name of the session attribute. See section 12.2.1 for details on standard session attributes.

12.4 Viewport views

The top level viewport definition looks like this:

```
<view
    id=""
    type="table"
    dataSourceId=""
    [model=""]
    [sortable="false"]
    [filterable="false"]
    [multiSelect="false"]
    [dragSource="false"]
    [singleton="true"]>
    [<context type="" source="" target="" filter=""/>]*
    [<sort field="" order=""/>]
    <display
        type="table"
        [allowRemove=""]
        [addView=""]
        [width=""]
        [rows=""]>
        <!-- Viewport field definitions go here -->
    </display>
</view>
```

The view attributes have the following meaning:

Attribute	Meaning
id	The ID of the view. Must be unique across all loaded modules.
type="table"	The type of the view.
dataSourceId	The ID of the data source containing the viewport data.
model	An optional model for submission upon view start-up, if the viewport is intended to show a search result. See section 12.4.8 for more details.
sortable="false"	The table cannot be sorted. (Pre-sorting still applies)
filterable="false"	The table cannot be filtered
multiSelect="false"	Multi select is not possible
dragSource="false"	Dragging objects from the table is not possible
singleton="true"	There will be only one instance of the view at any given time.

A viewport view can be seen as a table where the client displays only as many rows as it can fit on the screen. All data is kept on the application server, and the client only receives updates for the visible data, along with data about the total table size, which is needed to size and position the scroll bars properly.

Note: Fields that are included but configured as not visible can still be displayed through the toolbar "choose columns" function.

Viewport views support view interactivity. See section 12.14 for details.

The display attributes have the following meaning:

Attribute	Meaning
type="table"	The display is a read-only table. See section 12.4.2 for more types.

Attribute	Meaning
width	Only for tables displayed in a dialog. Defines how wide the view and its surrounding dialog will be initially.
rows	Only for tables displayed in a dialog or embedded in a form field set. Defines how many rows the table will initially expand to before a vertical scroll bar appears.

12.4.1 Viewport fields

A viewport field (column) is defined as follows:

```
<field
    name=""
    [width=""]
    [displayRef]
    [visible="false"]
    [readOnly="true"]
    [required="false"]
    [summary=""]
    [dataSourceId=""]
    [viewref=""]
    [position=""]/>
```

The attributes have the following meaning:

Attribute	Meaning
name	The name of the column. Can be specified in `x.y.foo` syntax if necessary.
width	An optional initial column width given in pixels. The default column width is 100 pixels.
displayRef	For tables only, an optional reference to a display definition, if you want to display the value as a graph. (The current version of Ciguan only supports horizontal bar charts)
visible	For tables only, a Boolean flag indicating if the column should be initially hidden. Hidden columns can be displayed via the Manage Columns dialog.
readOnly	For editable tables only, an optional Boolean attribute specifying that the column values should be read only.
required	For editable tables only, an optional Boolean attribute specifying that the column values should be considered optional. By default, all values are required in an editable table, so you only need to specify `required="false"` for the fields you want to make optional.
summary	For tables only, an optional attribute specifying that the column should be included in the summary row, and what summary function to apply. See section 12.4.9 for more details on column summaries.
dataSourceId	For editable tables only, an optional ID of a data source specifying valid values for the field. This implies that a list box input widget will be used.
viewref	For editable tables only, an optional reference to a viewport view from which values are selected. This implies that a viewport box or tree viewport box input widget will be used.

Attribute	Meaning
position	For view extension only, an optional attribute which specifies the position of the column in relation to a column in the base view. When position is not specified, the column is placed at the end of the column collection. The following position variants are available: ▪ `position="first"`, insert the column before the current first column. ▪ `position="before:foo"`, insert the column before the column named "foo". ▪ `position="after:foo"`, insert the column after the column named "foo".

12.4.2 Using other display types than table

A viewport view always requires a display of type `table`, but you can configure other display types too. The other supported display types are as follows:

Display type	Meaning
editable	An editable table

Editable tables are defined as follows:

```
<display type="editable" [allowRemove="false"] [addView=""]>
    <field name="foo" readOnly="true"/>
    <field name="bar"/>
</display>
```

The following display attributes are only valid for editable displays:

Attribute	Meaning
type="editable"	The type of the display.
allowRemove="false"	Optional Boolean attribute which disables the option to remove rows from the table.
addView	Optional attribute which defines a form view for creating new rows in the table.

See section 12.6 for more information on editable tables.

12.4.3 Filtering viewport data based on context objects

Context objects passed to a viewport view can be used for filtering the data in the viewport based on attribute values in the context object. The filter syntax is as follows:

```
filter="<item attribute><operator><context attribute>"
[condition="<context attribute><operator><value>"]
```

Example:

```
<context
    type="Member"
    filter="memberId=memberId"
    condition="memberType=1"/>
```

In the example above, the data source for users is filtered by a member, so the filter refers to the `memberId` attribute in the data source `User` item objects, where they have to equal the value of the `memberId` attribute of the

Member context object. The specified filter is however only applied if the `memberType` attribute of the Member context object has a value of 1.

Caution: Filtering cannot be performed with array contexts. This is intentional since there is currently no way to configure "or"-based filters.

12.4.4 Pre-sorting viewports

A viewport can be pre-sorted in case the default sort order provided by the server is not sufficient. To specify a pre-sorted order, you can add a `sort` element to the viewport view definition:

```
<sort field="riskNodeMarginPercent" order="descending"/>
```

The current version of Ciguan only allows sorting on one field, so even if you define multiple sort fields, only the first one will be used.

Note: The value of the `field` attribute must reference the name of a column in the table.

12.4.5 Using column charts

You can use a special type of chart in viewports, namely the column chart. It is a simple chart in the form of a horizontal bar. To use column charts, you need to do two things:

- Configure a chart display
- Configure a field in the viewport to reference the chart display

Example of a chart display:

```
<display id="myHorizontalBar" type="horizontal-bar">
    <threshhold value="0" color="green"/>
    <threshhold value="50" color="orange"/>
    <threshhold value="80" color="red"/>
    <threshhold value="100" color="purple"/>
</display>
```

Example of a field that references the chart display:

```
<field
    name="riskNodeMarginPercent"
    displayRef="myHorizontalBar"/>
```

There are a few rules of thumb related to using column charts:

- You must match the threshold values to the value range of the attribute
- The attribute must be numeric and is assumed to be a percentage value

12.4.6 Viewports as part of an input widget

The special form input widget `MvcViewportBox` uses a viewport as an aid when selecting data. It is typically used when the amount of possible values exceeds the practical limit for a list box. A rough estimate of this limit is 50 rows. Attempting to use a list box with too many rows will cause the browser to slow down considerably, and possibly even hang for a long time.

To use a viewport as part of an input widget, you do the following:

- Configure a viewport view based on a data source where the object's key attribute is the value you wish to populate into the underlying model value
- Configure a viewport view context object of type `TypeAheadFilter` and set a filter that will select items based on the type ahead filter expression
- Configure the input field to have a `viewref` attribute that points to the viewport.

Below is an example of a form field that uses a viewport for data selection:

```
<field
    name="orderBook"
    viewref="SelectFromOrderBooksViewportView"/>
```

The viewport is defined as follows:

```
<view
    id="SelectFromOrderBooksViewportView"
    type="table"
    dataSourceId="ORDER_BOOKS_ALL">
    <context
        type="TypeAheadFilter"
        filter="displayName~=expression"/>
    <display
        type="table">
        <field
            name="displayName"
            width="200"/>
        <field
            name="currency"
            width="60"/>
    </display>
</view>
```

12.4.7 Trees as part of an input widget

The special form input widget `MvcTreeViewportBox` uses a tree as an aid when selecting data. To use a tree as part of an input widget, you do the following:

- Configure a tree view based on a data source which is either a tree or a list tree.
- Configure the input field to have a `viewref` attribute that points to the tree view.
- Configure context elements for all selected object types you are allowed to select by clicking on them in the tree. This is due to how the tree selection widget works, it passes the selected node as a context object, and then it is up to the form definition to decide what node types to populate data from.

Below is an example of a form field that uses a tree for data selection:

```
<field
    name="fooId"
    viewref="MarketTreeView"/>
```

The tree view is defined as follows:

```
<view
    id="MarketTreeView"
    type="tree"
    dataSourceId="MARKET_TREE"/>
```

In the example we assume that the market tree has four different node types, Market, Market List, Segment and Instrument, but the form field should only be populated when clicking on an instrument, not the other levels. This implies that the form must specify at least one context definition for the Instrument type.

Multiple fields with the same tree

If you have more than one field in your form that uses the same tree to select data from (or another tree with the same selectable objects), you need to be able to detect which widget that has triggered the selection in order to copy the value to the relevant field. This is done by adding a `condition` attribute

on the context definition and defining expressions that compare the value of the `_contextsource` attribute with the field names:

```
<context type="TradableInstrument"
    condition="_contextsource=foo"
    source="internalId" target="fooId"/>
<context type="TradableInstrument"
    condition="_contextsource=bar"
    source="internalId" target="barId"/>
```

The example above assumes that the form has two fields; `foo` and `bar`, which both reference the same tree view.

12.4.8 Query viewports

A viewport can be defined to execute a query and show the result, similar to the search form, but without possibilities to change the query or refresh the contents. To achieve this, you must point the data source to a query data source, and you must also configure the model attribute to the request you want to submit. Finally, you must populate the request through data available in the context objects passed to the viewport view.

A typical use for this is a detail view which shows the master record while an embedded viewport shows the detail records. In this case you would define a detail form with a field set referencing a viewport view defined as a query viewport, and in the query viewport definition, populate the request through the master item, which will automatically be available as a context object.

Example of a query viewport:

```
<!-- Pending gross settlement instructions viewport -->
<view id="PendingGrossSettlementInstructionsViewportView"
    type="table"
    model="SeQueryGrossInstructionsReq"
    dataSourceId="PENDING_GROSS_SETTLEMENT_QUERY">
    <context type="SeNetSettlementInstructionEvent"
        source="netSettlementId"
        target="netSettlementInstructionId"/>
    <context type="SeNetSettlementInstructionEvent"
        source="settlementSystemId"
        target="settlementSystemId"/>
    <context type="SeNetSettlementInstructionEvent"
        source="clearingMemberSettlementAccount"
        target="settlementAccountId"/>
    <display type="table">
        <field name="clearerId"/>
        <field name="positionAccount"/>
        <field name="paymentAmount"/>
        <field name="deliveryProduct"/>
        <field name="deliveryAmount"/>
        <field name="sourceEventType"/>
        <field name="sourceId"/>
        <field name="instructionType"/>
        <field name="affectCollateralBalance"/>
        <field name="grossSettlementId"/>
    </display>
</view>
```

12.4.9 Viewport summaries

Viewport summaries are functions which operate on all items in the underlying data source. They are displayed at the bottom of the table, and are automatically updated when the data source changes, when the table is filtered, etc.

To apply a summary function to a column, you configure the field as follows:

```
<view
    id="TradesViewportView"
    type="table"
    dataSourceId="Trades">
    <display type="table">
        <field name="price" summary="avg"/>
        <field name="quantity" summary="sum"/>
    </display>
</view>
```

The example above will display the average trade price and the total quantity as summary information.

Ciguan supports the following summary types:

Type	Description
avg	Average value
count	Item count
max	Maximum value
min	Minimum value
sum	Total value
custom:<class-name>	Specifies a custom summary function. The <class-name> parameter should specify a fully qualified name of the class implementing the summary. See section 10.15 for more details on how to implement a custom summary handler.

12.5 Form views

Form views are views intended for filling out information which is then submitted to the application server for further processing. A form is typically based on a request object. All forms are based on a model which is either automatically created via metadata, or hand coded.

Form views have an extensive configuration which gives you a lot of control over the form behavior. Each configuration element is described below.

The top level form definition looks like this:

```
<view
    id=""
    type="form"
    model=""
    [accService=""]
    [pinnable="true"]
    [resizable="false"]
    [focus=""]
    [onError=""]
    [onSuccess=""]
    [dockable="false"]
    [modal="true"]
    [formHandler=""]
    {singleton="true"]>
    <!--view content goes here -->
</view>
```

The attributes have the following meaning:

Attribute	Meaning
id	The ID of the form view. Must be unique across all loaded modules.
type="form"	The type of the view.

Attribute	Meaning
model	The type of object used as model.
accService	The ACC service required to access the form view. Usually the same as the ACC token associated with the back-end service which will be executed when submitting the form.
pinnable="true"	A Boolean flag indicating if the form is pinnable or not. Pinnable forms have a pin button In the window caption which can be used to keep the form open after submit. Defaults to "false".
resizable="false"	A Boolean flag indicating if the form dialog can be resized or not. Defaults to "true".
focus	The name of the input field that will receive focus when the form is opened. Default is the first focusable input field.
onError	Defines the action if the form submission failed: When set to `text:key`, an information popup is displayed if the server response to the submitted form indicated an error. The "key" part is a key which identifies a translation in the dictionary, using the `.message.` prefix. When set to `view:id`, the view with the given ID is opened if the form submission failed. The current form model and all context objects will be passed as contexts to the new view.
onSuccess	Defines the action if the form submission was successful: When set to `reload`, the application will be reloaded if the server response to the submitted form indicated successful processing. Typically used when selecting a language or changing some other configuration that affects the look and feel of the client. When set to `text:key`, an information popup is displayed if the server response to the submitted form indicated successful processing. The "key" part is a key which identifies a translation in the dictionary, using the `.message.` prefix. When set to `view:id`, the view with the given ID is opened if the form submission was successful. The current form model and all context objects will be passed as contexts to the new view.
dockable="false"	A Boolean flag indicating if the form can be docked or not. A dockable form can be dragged into a slot, which will then display it in a tab, rather than as a dialog.
modal="true"	A Boolean flag indicating if the form will be modal when displayed in a dialog. A modal dialog blocks access to all application functionality except the view itself.
formHandler	An optional fully qualified name of a form handler implementation class. See section 10.18 for more information on form handlers.
singleton="true"	There will be only one instance of the view at any given time.

12.5.1 Form initialization configuration

When a form is launched, Ciguan runs through a number of configuration items as part of the form initialization. These are processed in the order which they are described below.

Context definitions

In order for a form to accept a context object of a certain type, you need to configure the context object and what to do with it. Typically, you configure a context because you want to copy information from the object into one or more form fields.

A form context definition looks like this:

```
<context type="" [condition=""] [source="" target=""]>
    [<context-lookup type="" key="" [useFormHandler="true"]/>]
</context>
```

The condition, source and target attributes are optional, so if you only want to accept a certain context object type, you simply specify its type. Situations when this might be applicable are for example to allow menu parameters, or to allow a context object assigned through context lookup.

The condition attribute is a Boolean expression which, if specified, is evaluated and tested for a match before the context object is processed further.

The source attribute refers to data in the context object, while the target attribute refers to a location in the form model. They can refer to individual attributes as well as entire sub-objects. There is also a special value, ".", which indicates the entire context object or model.

Caution: When using the special "." source and target references, you need to make sure they are valid in respect to what you are copying, and to what location. Copying an object of one type into a model of another type works, but will likely result in a model with data in incorrect locations. You may also accidentally overwrite model data this way.

Context lookups

A context lookup is an operation that enables you to look up and populate a context object based on information in the current context object. This is useful when you want to display information about objects related to the form without having them as context objects upon form creation.

The key attribute refers to an attribute in the parent context object, and the type attribute defines which type of object that should be looked up. This implies that there is a data source that contains objects of the type you need to look up, and that the data source from which the object is to be fetched has a key which is logically equal to the value of the context lookup key attribute.

An alternative to using a property from the context object as key is to use a hardcoded value. This is done by surrounding the key value with single quotes.

Note: If for example a trade is to be looked up, its data source key is the trade ID, so you need to make sure that the key attribute you are doing the lookup via contains a valid trade ID value.

In order to use context lookups, you basically need to do the following:

- Make sure you have defined a data source which holds objects of the type you want to look up, and that the data source is properly populated
- Configure a context of the type you want to look up. This context does not require any source/target definitions
- Configure the actual context lookup
- Optionally configure one or more form fields displaying values from the new context object

Below is an example excerpt of a context lookup configuration.

```
<view
    id="TradeDetailsAndLogView" type="detail" model="Trade">
    <context type="Trade" source="." Target=".">
        <context-lookup type="TradableInstrument"
            key="tradableInstrumentId"/>
    </context>
    <context type="TcTradeEvent" source="trade" target=".">
        <context-lookup type="TradableInstrument"
            key="tradableInstrumentId"/>
    </context>
    <context type="AmAccountTradeEvent"
        source="trade" target=".">
        <context-lookup type="TradableInstrument"
            key="tradableInstrumentId"/>
    </context>
    <context type="TradableInstrument"/>
    <layout>
        <fieldset id="Common">
            <field name="tradableInstrumentDisplayName"/>
            <field name="price"/>
            <field name="currencyId"
                context="TradableInstrument"/>
```

In the example above, the `Trade` context object will trigger a context lookup based on its `tradableInstrumentId` attribute value, resulting in a new context object of type `TradableInstrument` from which the `currencyId` attribute is finally displayed.

Caution: Context lookups cannot be used with array contexts. This is intentional.

Using form handlers to perform the context lookup

Starting with Ciguan 10.5.0, you can let a form handler do the context lookup, rather than applying the default data source lookup method. To do this, you need to define a form handler and implement the `onFormContextLookup()` method. Use the new `sendClientContext()` method to pass back the object to the client:

```
@Override public void onFormContextLookup(
    AsConnectionIf pConnection, CwfMessage pModel) {
    Currency tCurrency = new Currency("FHC", "FHC",
        "Form handler lookup currency");
    sendClientContext(
        pConnection, pModel.getHandle(), tCurrency);
}
```

Set definitions

Set definitions are basically assignment of values to model fields. A set is configured as follows:

```
<set target="" value=""/>
```

The target attribute must point to a valid attribute in the model, and the value must be specified in its raw format.

Clear definitions

Clear definitions are used to set model attributes to null. This is useful in some cases where a context definition has copied an entire structure into the model, but you do not want to keep certain transient information such as identifiers, keys or other information that must be passed as null.

A clear is defined as follows:

```
<clear target=""/>
```

The target attribute must point to a valid attribute in the model.

12.5.2 Form layout configuration

The layout section of a form view specifies the field sets, optional custom button definitions, and optional field set groupings. The default field set grouping is vertical, where the field sets are laid out in the order in which they are defined.

Note: Field sets are optional, so it is possible to define a form with only buttons, in case that is needed.

Field sets

A field set is defined as follows:

```
<fieldset
    id=""
    [path=""
    [width=""]
    [context=""]
    [required="true"]>
    <!-- Fields or view references go here -->
</fieldset>
```

The attributes have the following meaning:

Attribute	Meaning
id	The ID of the field set. Used as a key when looking up a translation in the dictionary.
path	An optional path which specifies a starting point location in the model where the fields are located.
width	An optional width in pixels used when rendering the field set. Typically used when the field set contains a table, or if you want more control over the layout. Specified as a numeric value.
context	Declares that the field set is read-only and should display an array context of the given type, which must be specified with the array brackets suffix. When specifying the context attribute, the path attribute is ignored, and all fields in the field set must reference attributes in the array component type.
required="true"	For array input only, specifies that the array must have at least one row.

A field set is a logical grouping of fields in a form. The field set is rendered as a border surrounding the fields, and a label in the top left corner. The label text is translated from the ID of the field set using a dictionary lookup. The path attribute specifies a location in the model where the fields in the field set are located. The path is optional but it greatly simplifies readability, so it is recommended that you always qualify a path when applicable.

Fields

A form field is defined as follows:

```
<field
    [name=""]
    [text=""]
    [context=""]
    [dataSourceId=""]
    [filter=""]
    [updateContext="true"]
    [align="left|right"]
```

```
    [disabled=""]
    [clearDisabled="false"]
    [readOnly="true"]
    [required="false"]
    [multiSelect="true"]
    [viewref=""]
    [default=""]
    [upload-handler=""]
    [position=""]/>
```

The attributes have the following meaning:

Attribute	Meaning
name	The name of the attribute. Can be specified in `x.y.foo` syntax if the containing field set does not specify a path.
text	Optional, do not create an input widget. Instead, create a label and populate it with the dictionary translation for a field with an ID corresponding to the text attribute value. Can be combined with the align attribute.
context	If the field should display information from a context object, this optional attribute specifies the type of context object from which to fetch the value. Fields displaying context information are always read only.
dataSourceId	An optional ID of a data source specifying valid values for the field. This implies that a list box input widget will be used.
filter	An optional attribute that specifies a filter used to reduce the number of available items in a list box or viewport box widget.
updateContext="true"	An optional Boolean attribute specifying that when a value is selected in a list box or viewport box, the selected object should be fetched and passed into the form as a context object. Typically used in combination with a filter attribute on another field. **Note:** In order for this to work, you also need to allow the object as a context in the form definition
align	An optional alignment. Used in conjunction with the `text` attribute. Defines how the text should be placed. If the align attribute is not specified, the text will span the label and widget columns. Allowed values are: ▪ `left` - the text will be placed in the label column ▪ `right` - the text will be placed in the widget column.
disabled	An optional expression used to define when a field should be disabled. For example used to disable a list box when a value is selected in another list box, also known as mutual exclusion. **Note:** By default, disabled items are always set to null in the model. This behavior can be changed, see below for details.
clearDisabled="false"	An optional Boolean attribute specifying that the item should not be cleared if the field is disabled. Must be used in conjunction with the `disabled` attribute.
readOnly="true"	An optional Boolean attribute specifying that a field should be read only.
required="false"	An optional Boolean attribute specifying that a field should be considered optional. By default, all form fields are required in a form view, so you only need to specify `required="false"` for the fields you want to make optional.

Attribute	Meaning
multiSelect="true"	An optional Boolean attribute specifying that a list box should allow multiple selections. Only valid for fields where a data source ID is specified.
viewref	An optional reference to a viewport view from which values are selected.
default	An optional default value which will be set when the underlying model value is null
upload-handler	For file upload views only, should contain a fully qualified name of a class containing a file upload handler implementation. See section 12.7 for more details on file upload views.
position	For view extensions only, an optional attribute which specifies the position of the fi in relation to a column in the base view. When position is not specified, the column is placed at the end of the column collection. The following position variants are available: ▪ `position="first"`, insert the column before the current first column. ▪ `position="before:foo"`, insert the column before the column named "foo". `position="after:foo"`, insert the column after the column named "foo".

Multi-line text input fields

Normally, all fields which are mapped to attributes of type Text will use a text box for data entry. However, Ciguan also supports multi-line text entry and presentation in forms. In this case you need to ensure that the model attribute is configured to use the `MultiLineText` business type.

Views

You can include another view inside a field set. This is typically used to view related information in the form of a table. To include a view in a field set, use the following configuration:

```
<view ref=""/>
```

A field set that contains a view cannot contain any other items.

Caution: Do not attempt to include a form view in another form view, or a detail view in another detail view. This will not work, even though the configuration currently allows it.

Buttons

By default, Ciguan adds a submit button to a form view. You can however configure the buttons manually for more control. One reason to do this is to utilize the special types of buttons provided by Ciguan. The available button types for use in forms are as follows:

Type	Meaning
close	Close the form.
forward	Forward to another form and close the currently open form. See below for more details.
open	Open another form, keeping the current form open. See below for more details.
reset	Reset the form. (Search forms only)

Type	Meaning
submit	Submit the form contents for processing.

To configure a button, use the following definition:

```
<buttonpanel>
    <button
        name=""
        type=""
        [view=""]
        [autoSubmit="true"]
        [confirmMessage=""]/>
</buttonpanel>
```

The button attributes have the following meaning:

Attribute	Meaning
name	The name of the button. Used as a key when translating the button text.
type	The type of the button. See earlier explanation.
view	Only for type "open" or "forward". The ID of the view to open or forward to.
autoSubmit="true"	Only for type "open" or "forward". Indicates whether or not the next view should automatically submit its content or not.
confirmMessage	If set, the value is used to look up a message shown in a dialog, which must be confirmed in order for the action to be executed.

You can add as many buttons as you like, and they will be placed in a left to right fashion, so the first defined button will be leftmost, and so on. You can also define an empty button panel, which results in no buttons at all.

Field set grouping

By default, all field sets in a form are stacked vertically in the order they are defined. If you want to lay out the field sets in other ways, you can define field set groupings similar to this example:

```
<group direction="vertical">
    <group>
        <fieldset ref="A"/>
        <fieldset ref="B"/>
    </group>
    <group>
        <fieldset ref="C"/>
    </group>
</group>
```

The group direction is either horizontal or vertical, and the next group direction is automatically the opposite of the containing group's direction, unless you explicitly define it.

In the example above, field sets A and B will be laid out next to each other, above field set C.

Note: Tables might not calculate their size properly when laid out in field sets which do not occupy the full width, so try to avoid placing a table in a field set which is in a horizontal group, at least if the table is wide.

Using Open or Forward buttons in a form

When you use an Open or Forward button in a form, the following objects are being passed as contexts to the opened or forwarded view:

- All current context objects that are of a different type than the current form's model
- The definition of the button being pressed
- A copy of the current form's model

Note: A forward or open is a client only operation, so no get methods are evaluated on the passed objects.

12.6 Form views with editable tables

An editable table is a special form feature that requires the underlying table data source to be a query data source. Editable tables will create input widgets for all columns which are not read only.

Editable tables are typically used when a request contains an array that can hold many entries, and to improve performance, the data is kept on the application server. When the form is submitted, the request is pre-processed in the application server in order to copy the array elements into the request before passing it to the back-end.

To define a form with an editable table, you must do the following:

- Define a query data source that holds items of the same type as the array in the request
- Define a viewport view with a display of `type="editable"`, optionally specifying `allowRemove="false"` if you do not want to allow rows to be deleted by the user. If you want to allow the user to add new items you can also configure the `addView` attribute and let it point to a form view for entering new items.
- Configure all non-editable fields in the viewport view as `readOnly="true"`
- Define a form view with a field set that contains a reference to the viewport view
- Add code in the application server that populates the data source with relevant entries

The last step is needed since adding rows is not possible from the client at this point. Editable tables are typically used in conjunction with file uploads or other bulk processing.

Below is a complete example of a form with an editable table.

The query data source:

```
<AsList id="BVMF_CANCEL_TRADES_INFOS_QUERY"
    type="BvmfCancelTradeData" key="tradeId" text="tradeId"
    query="true"/>
```

The editable viewport:

```
<view id="BvmfCancelTradeInfoObjectsViewportView" type="table"
    dataSourceId="BVMF_CANCEL_TRADES_INFOS_QUERY"
    model="BvmfCancelTradesReq">
    <context type="BvmfCancelTradesReq" source="." Target="."/>
    <display type="editable">
        <field name="accountOwnerName" readOnly="true"/>
        <field name="accountName" readOnly="true"/>
        <field name="accountClientDocId" readOnly="true"/>
        <field name="amount" readOnly="true"/>
        <field name="quantity" readOnly="true"/>
        <field name="buyOrSell" readOnly="true"/>
```

```
        <field name="cancelQuantity"/>
    </display>
</view>
```

The form view:

```
<view id="BvmfCancelTradeWizardView2" type="form"
    model="BvmfCancelTradesReq"
    accService="TcBvmfTradeServices_tcBvmfCancelTrades">
    <context type="BvmfCancelTradesReq" source="." Target="."/>
    <layout>
        <fieldset id="Origin">
            <field name="tradeDate" readOnly="true"/>
            <field name="matchedTradeId" readOnly="true"/>
            <field name="externalTradeId" readOnly="true"/>
            <field name="symbol" readOnly="true"/>
            <field name="originalQuantity" readOnly="true"/>
            <field name="price" readOnly="true"/>
            <field name="amount" readOnly="true"/>
            <field name="cancelReason" />
        </fieldset>
        <fieldset id="TradesToBeCancelled"
            path="bvmfTradeCancellations">
            <view ref="BvmfCancelTradeInfoObjectsViewportView"/>
        </fieldset>
        <buttonpanel>
            <button name="finish" type="submit"/>
            <button name="cancel" type="close" />
        </buttonpanel>
    </layout>
</view>
```

In this example, the viewport has a model property defined, which will actually send the `BvmfCancelTradesReq` context object to the server as if it was an actual query to the back-end, which it in reality is not. Instead, the data source service component has been extended in the following way:

```
@SuppressWarnings("unchecked")
@Override
protected void populateDataSourceFromQuery(
    AsConnectionIf pConnection, AsListIf pDataSource,
    CwfDataIf pQuery, int pRequestHandle) {

    if (pQuery.getProperty(ATTR_OBJECT_NAME).equals(
        BvmfCancelTradesReq.class.getSimpleName())) {
        populateCancelTradesFromQuery(
            pConnection, pDataSource, pQuery, pRequestHandle);
        return;
    }
    super.populateDataSourceFromQuery(
        pConnection, pDataSource, pQuery, pRequestHandle);
}
```

As you may have figured out, the `populateCancelTradesFromQuery` method clears the data source and populates relevant items in it. These items will be displayed in the viewport, and the `cancelQuantity` attribute is then editable.

12.7 File upload views

File upload views are identical to form views in respect to their configuration, with two exceptions:

- The view type must be `fileupload` instead of `form`.
- Each field that refers to a file must use an additional `upload-handler` attribute which should specify a fully qualified name of a file upload handler class. See the application server API section for more details on this.

Below is an example of a very simple file upload view:

```
<view id="FileUploadView" type="fileupload"
    model="AsFileUploadReq">
    <layout>
        <fieldset id="Common">
            <field name="file" upload-handler=
                "com.cinnober.as.impl.MyHandler"/>
        </fieldset>
    </layout>
</view>
```

In the example, the `AsFileUploadReq` class is a Java class that features one public String attribute called `file`.

12.8 Tree views

Tree views are the easiest views to configure since most of the configuration is specified by the underlying data source.

A tree view definition looks like this:

```
<view
    id=""
    type="tree"
    dataSourceId=""/>
```

The attributes and their meaning are as follows:

Attribute	Meaning
id	The ID of the tree view. Must be unique across all loaded modules.
type="tree"	The type of view.
dataSourceId	The ID of the data source that defines the tree. Must point to an `AsTree` or `AsListTree` data source.

Note: Tree views support view interactivity definitions. See section 12.14 for details.

12.9 Detail views

Detail views are form views where the view is considered read only, and there is no submit button. Instead, there is by default a close button. Other than that, the functionality is identical to forms in terms of configuration, with the following exceptions:

- Fields only need to be specified by their name, since no input is possible

12.10 Search views

Search views are a combination of a form view and a viewport view, where the form view is used to populate a query, and the viewport view displays the corresponding search result. When a search is submitted, the search form is hidden, but can be displayed again using a button on the viewport toolbar. The viewport toolbar also features a refresh button which repeats the previous search without displaying the search form.

A search form is defined as follows:

```
<view
    id=""
    type="search"
    model=""
    targetView="">
```

```
    <!-- Form content goes here -->
</view>
```

The attributes have the following meaning:

Attribute	Meaning
id	The ID of the tree view. Must be unique across all loaded modules.
type="search"	The type of view.
model	The type of object used as model.
targetView	The ID of the result view. This view must be mapped to a query data source that contains objects of the same type as an array in the query response.

When the result view is closed, the search form view is automatically closed too, and after that point it cannot be opened again.

12.11 Chart views

Caution: Chart functionality and its stability will improve over time, but at this point, the chart functionality is to be considered experimental.

Chart views use graphics to display information. A chart view obtains its data from a data source, and you configure which attributes to display on which axis. Starting with version 10.3.0 of Ciguan, all chart views use Highcharts as chart library.

A chart view is defined as follows:

```
<view
    id=""
    type="chart">
    <chart type="">
        <chart-data>
            <!-- Chart data goes here -->
        </chart-data>
        <chart-options>
            <!-- Chart options go here -->
        </chart-options>
    </chart>
</view>
```

The attributes have the following meaning:

Attribute	Meaning
id	The ID of the tree view. Must be unique across all loaded modules.
type="chart"	The type of view.

Ciguan currently supports the following chart types:

Type	Description
area	Area chart
bar	Bar chart with horizontal bars
column	Column chart with vertical bars
columnrange	Column chart with free floating columns where each end has a value
donut	Donut chart, similar to pie chart but with concentric circles for breakdown purposes

Type	Description
line	Line chart
pie	Pie chart with slices
stackedbar	Stacked version of the bar chart

12.11.1 Chart data

Chart data contains data source and field mappings. It is defined as follows:

```
<chart-data dataSourceId="">
    <chart-field
        name=""
        [axis=""]
        [pointIndex="true"]
        [series="true"]/>
    [<chart-field ... />]*
</chart-data>
```

The `dataSourceId` attribute defines the data source from which the chart data should be fetched.

The chart field attributes have the following meaning:

Attribute	Meaning
name	The name of the field. Must correspond to an attribute, a get method or a configured get method on the object in the data source.
axis	The axis to which the attribute value should be mapped. Allowed values are: ▪ `x[:<n>]` - The X axis, or the <n>'th X axis if the extended syntax is used. ▪ `y[:<n>]` - The Y axis, or the <n>'th Y axis if the extended syntax is used. **Note:** The extended axis syntax is not yet supported
series="true"	A Boolean flag indicating that the attribute is a series identifier. All values belonging to a bar, line etc share the same series identifier value.
pointIndex="true"	A Boolean flag indicating that the value of the attribute should be used as an index number for the point. Used for sorting purposes where it is not possible to sort on the X axis value, which is the default.

12.11.2 Chart options

The chart options should always contain a CDATA section containing a JSON string. The contents of the JSON string should be definitions of Highcharts options configurable for the particular type of chart being created.

The JSON string is equivalent to the value passed into the `highcharts()` JavaScript function. See the examples at http://www.highcharts.com/demo and click the "View options >>" button to see the script examples.

Only a limited set of options are supported, and there are some important restrictions:

- Series should not be defined; these are covered by the chart data section.
- The chart type should not be defined; it is covered by the chart section.
- Functions are typically not supported. The axis label formatter function can however be used.

Chart option expansion macros

The CDATA block in the chart options section has support for expansion of macros. The following constructs can be used:

Macro	Description
`$text:<key>`	A translated text from the dictionary. The key is a full key to any section in the dictionary.
`$session.<attribute>`	A value from the session model. See section 12.2.1 for details.
`$context.<type>.<attribute>`	A value from a context object.
`$model.<attribute>`	A model value.

12.11.3 Complete example

Below is an example of a complete pie chart definition.

```
<view id="DemoPieChartView" type="chart">
    <chart type="pie">
        <chart-data dataSourceId="PieChart">
            <chart-field name="series" series="true"/>
            <chart-field name="text" axis="x"/>
            <chart-field name="yValue" axis="y"/>
        </chart-data>
        <chart-options><![CDATA[
             {
                title: {
                    text: '$text:DemoPieChartView.title'
                },
                tooltip: {
                    headerFormat: '',
                    pointFormat: '{point.name}:
<b>{point.percentage:.1f}%</b>'
                },
                plotOptions: {
                    pie: {
                        allowPointSelect: true,
                        cursor: 'pointer',
                        dataLabels: {
                            enabled: true,
                            color: '#000000',
                            connectorColor: '#0000FF',
                            format: '<b>{point.name}</b>:
{point.percentage:.1f} %'
                        }
                    }
                }
            }
        ]]></chart-options>
    </chart>
</view>
```

12.12 Displays

To define a display, which is used for chart purposes, add a display element as follows:

```
<display id="chart" type="horizontal-bar">
    <threshhold value="0" color="green"/>
    <threshhold value="50" color="orange"/>
    <threshhold value="80" color="red"/>
    <threshhold value="100" color="purple"/>
</display>
```

12.13 Perspectives

Perspectives are defined as child elements to an `AsPerspectives` element:

```
<AsPerspectives>
    <!-- Perspective and role mappings go here -->
</AsPerspectives>
```

12.13.1 Perspective definition

To define a perspective, add a perspective element as a child of the `AsPerspectives` element, as follows:

```
<perspective
    id=""
    [deviceType=""]
    [maxWidth=""]>
    <content
        [deviceType=""]
        [screenOrientation=""]>
    </content>+
</perspective>
```

The perspective element attributes have the following meaning:

Attribute	Description
id	The ID of the perspective. Must be unique within the perspective definitions.
deviceType	Comma separated list of applicable device types. Can be seen as a filter which determines whether a perspective applies or not for a certain device type. Valid device types are `mobile`, `tablet` and `desktop`.
maxWidth	Can be used to control the maximum width for a perspective, measured in pixels. If the browser window exceeds the maximum specified perspective width, the perspective content will center itself in the browser window. This is useful for web page like clients.

The content element attributes have the following meaning:

Attribute	Description
deviceType	A comma separated list of device types for which the particular content applies. Valid device types are `mobile`, `tablet` and `desktop`.
screenOrientation	The screen orientation for which the particular content applies. Valid screen orientations are `portrait` and `landscape`.

Content

In Ciguan you can define different contents for different device types and/or screen orientations. Simply repeat content elements using different device type and/or screen orientation values as needed.

The content of a perspective is a hierarchy of groups and slots. Both groups and slots are container elements that hold references to views which will be displayed. The difference between a group and a slot is that a group can only contain slots, while a slot can either contain a single group, or one or more views. The outermost container in a perspective must always be a group, never a slot.

Group

A group element has the following attributes:

Attribute	Meaning
Direction	An optional attribute that defines whether the content of the group is stacked vertically or horizontally. (Valid values are "vertical" or "horizontal")

Slot

A slot element has the following attributes:

Attribute	Meaning
Id	The ID of the slot. Must be unique within the perspective definition.
Columns	Only for display mode grid. Number of columns used when laying out the views.
columnSpacing	Only for display mode grid. The spacing between each column, in pixels.
displayMode	The display mode of the slot. See below for details on valid values.
Height	An optional width of the slot. Only relevant for slots in a vertical group.
Width	An optional width of the slot. Only relevant for slots in a horizontal group.
static="true"	Only for display modes split and tab. Setting the static attribute means the following: ▪ For a split slot, the splitter between the contained slots cannot be moved. ▪ For a tab slot, the tabs cannot be rearranged, tabs cannot be dragged out of the slot, and dialogs cannot be docked into the slot
Template	A reference to a perspective slot template which will be injected into the slot. See section 12.13.2 for more information on slot templates.
customSlotType	Only for the custom display mode. A user defined string defining which slot type to instantiate.

Valid values for the display mode are:

Display mode	Meaning
Custom	Custom slot, you need to implement the slot and instantiate it in the `createSlot()` MVC factory method.

Display mode	Meaning
Disclosure	The views in the slot are accessed through a disclosure mechanism, where only one view is visible at a time. (Not yet implemented)
Grid	The views in the slot are laid out next to each other in a chess board style. **Note:** In the first iteration, all views have the same size.
Single	The slot can only hold a single view. An attempt to open a new view in the slot will close the existing view.
Split	The views in the slot are separated by movable splitters. Causes all views to be visible and divides the available space evenly among all views.
Tab	The views in the slot are accessed through tabs, where each tab contains one view.
Tray	The views share the space occupied by the slot according to their sizes, and placement is decided through CSS.

Height and width

The height and width attributes of slots are normally just relative numbers, so two slots in the same group having a height of 1 is the same as two slots in the same group having a height of 100. They will be equally high. The exception to this relational size model is the "tray" display mode, where the height (or width) is not a relative size but instead, the size in pixels.

Slot template references

A slot which references a template cannot contain a nested structure, but you can still specify the slot attributes, thus adding to or overriding corresponding attributes defined on the slot template itself.

12.13.2 Perspective slot templates

In Ciguan it is possible to define perspective slot templates and reference these templates in the perspective slot definitions. This will save you from repeating identical slot definitions over and over, and will result in a more compact perspective layout. Also, a slot template may reference other slot templates in its content, to further allow structural improvements to the definitions.

Slot templates are defined as follows:

```
<AsPerspectives>
    <slot-template id="" ... >
    </slot-template>
</AsPerspectives>
```

The slot-template element takes the same attributes as the slot element, minus the template attribute. In all other aspects it is identical to a slot.

12.13.3 Perspective role mappings

A perspective requires one or more role mappings to be available for users in the system. You map perspectives to roles by adding role elements as child elements to the `AsPerspectives` element, like this:

```
<role name="CLEARING_HOUSE_SUPER_USER">
    <perspective ref="Trading"/>
    <perspective ref="PositionManagement"/>
    <perspective ref="CollateralManagement"/>
    <perspective ref="SettlementManagement"/>
```

```
    <perspective ref="RiskManagement"/>
    <perspective ref="ReferenceDataManagement"/>
</role>
```

In order for a perspective to be available to a user, the user must be assigned at least one of the roles to which the perspective is mapped. Menus

Menus are somewhat tricky to configure, since the structure has more depth than other configuration areas. Menus are always added as child elements to an `AsMenus` container element.

```
<AsMenus>
    <!-- Menu definitions go here -->
</AsMenus>
```

12.13.4 Global menu items

Some menu items are defined globally and can be referenced from other menus. Currently, only the menu separator is defined like this. The menu separator item has an ID of "separator" which you can reference in other menus, like this:

```
<menuitem ref="separator"/>
```

Note: The separator functionality is subject to change within the near future where it will become a unique element rather than a menu item with a specific ID.

12.13.5 Menus

Menus are traditional drop-down menus attached to a menu bar in the same way as you would normally find in desktop applications. Menus can be oriented either horizontally or vertically. A menu can contain both menu items and sub-menus.

A menu is defined as follows:

```
<menu
    id="" | remove=""
    [position=""]
    [perspective=""]
    [view=""]>
    <!-- Menu items and submenus go here -->
</menu>
```

The menu attributes have the following meaning:

Attribute	Meaning
Id	The ID of the menu; must be globally unique.
Remove	Mutually exclusive with id and should refer to an ID of an already existing menu. If specified, it means that the original definition of the menu should be removed. Usually immediately followed by a new definition of the same menu.
Position	An optional position where the menu should be inserted. When position is not specified, the item is appended after the other menus. The following position variants are available: ▪ `position="first"`, insert the item before existing menus. ▪ `position="before:foo"`, insert the menu before the menu with ID "foo". ▪ `position="after:foo"`, insert the menu after the menu with ID "foo".
Perspective	An optional comma separated list of perspective ID's for which the menu will appear.

Attribute	Meaning
View	An optional view for which the menu will appear.

12.13.6 Context menus

Context menus are typically menus available when right-clicking on a row in a table, or on a node in a tree. Context menus can hold both menu items and menus. For simplicity reasons however, only one level of menus is currently supported.

You define a context menu for an item of a particular type as follows:

```
<contextmenu
    type=""
    [perspective=""]
    [view=""]>
    <!-- Menu items and menus go here -->
</contextmenu>
```

The context menu definition attributes have the following meaning:

Attribute	Meaning
type	The type of the item for which the context menu is valid. Usually the simple name of the item class, but a special type is "Session". See below for details. The type attribute can also contain an array suffix "[]" in which case the context menu definition will be used when multiple objects of the indicated type are selected.
perspective	An optional comma separated list of perspective ID's for which the menu will appear.
view	An optional view for which the menu will appear. If the view attribute is set, the context menu will replace the global context menu for the given type.

The session context menu

There is one special context menu that has the user session as its context. This context menu is the one to which you attach the menu bar that is displayed in the top portion of the screen. This context menu definition contains a more complex menu structure, including as many menu levels you like.

A session context menu is defined as follows: (snippet)

```
<AsMenus>
    <contextmenus>
        <contextmenu type="Session">
            <menu id="system">
                <menuitem view="ActOnBehalfView"
                    filter="isActOnBehalf=false"/>
                <menuitem view="ExitActOnBehalfView"
                    filter="isActOnBehalf=true"
                    autoSubmit="true"/>
            </menu>
            <menu remove="trading"/>
            <menu id="trading"
                perspective="Trading,PostTrade,
                             PositionManagement,RiskManagement,
                             SettlementManagement">
                <menuitem view="TradeInsertView"/>
                <menuitem view="BrokerTradeInsertView"/>
                <menuitem view="TradeReportInsertView"/>
```

```
                <menuitem ref="separator"/>
                <menuitem view="SetMarketPriceView"/>
                <menuitem view="SetInterestRateView"/>
                <menuitem view="QueryOpenInterestView"/>
                <menuitem view="OpenInterestsViewportView"/>
                <menuitem ref="separator"/>
                <menuitem view="AddInstrumentView"/>
                <menuitem view="AddMarketView"/>
            </menu>
            <menu id="position"
                perspective="ClearingHouseRiskManagerPosition">
                <menuitem view="CalculateMarkToMarketView"/>
            </menu>
```

Since the session context menu is in essence a menu, please see the menu section for a complete reference on each element.

Caution: For consistency reasons, the session context menu is likely to be replaced by a dedicated menu bar element in a future Ciguan release.

12.13.7 Menu items

Menu items are generally actions, and most commonly links to views which open when you select the item. Menu items can also be references to other menu items, like the separator. Finally, menu items can also take parameters and some other options.

A menu item is defined as follows:

```
<menuitem id="" | ref="" | remove="" | view=""
    [accService=""]
    [autoSubmit="true"]
    [confirmMessage=""]
    [expand=""]
    [filter=""]
    [parameters=""]
    [perspective=""]
    [position=""]/>
```

The menu item attributes have the following meaning:

Attribute	Meaning
id	The ID of the menu item. Used only when defining a global menu item to be referenced by other menu items. When specifying the ID, no other attributes should be specified.
ref	A reference to an ID of a global menu item which will be inserted at the point of the reference. When referencing a global menu item, no other attributes should be specified.
remove	The ID of a menu item to remove. Must refer to an existing menu item. When removing a menu item, no other attributes should be specified.
view	A reference to a view with the given ID. When the menu item is selected, the referenced view will open.
accService	An optional ACC service token to which the user must have access in order for the menu item to appear.
autoSubmit="true"	An optional flag indicating that the opened view should be submitted without displaying the view. Must only be used in conjunction with form views.
confirmMessage	An optional value used to populate a text in a confirmation dialog that appears when the menu item is selected. In order for the menu item to be executed, the user must select the Ok button.

Attribute	Meaning
expand	An optional value used to expand the current menu item into multiple menu items pointing to the same view, but with different parameters, one per item in the data source associated with the field referenced by the expand attribute. Primarily used for the perspective selection menu.
filter	A filter expression or fully qualified name of a class used to decide whether or not the menu item should be displayed or not. See below for more information on filtering menu items.
parameters	An optional parameter string of the form "name=value[,name=value]*". See below for more information on how parameters are handled.
perspective	An optional ID of a perspective for which the menu item should be displayed.
position	An optional position where the menu item should be inserted. When position is not specified, the item is appended to the end of the menu. The following position variants are available: ▪ `position="first"`, insert the item at the top of the menu. ▪ `position="before:foo"`, insert the item before the menu item with ID "foo". ▪ `position="after:foo"`, insert the item after the menu item with ID "foo".

12.13.8 Menu parameter handling

When a menu item is selected, all its defined parameters, plus the values of the `autoSubmit` and `confirmMessage` attributes will be placed in an object of type `MenuParameters`, which is then passed as a context to the view which is opened as a result of selecting the menu item.

12.13.9 Menu item filtering

Menu items can be filtered on several criteria:

- The specified list of perspectives
- The specified view
- The defined ACC service
- Programmatic filters, either expression- or class based

Expression based filters

Expression based filters are Boolean expressions defined on the format:

```
<attribute><operator><value>
```

An example of an expression based filter on an item of type `User` which aims to only show the menu item for a disabled user will look like this:

```
<contextmenu type="User">
    <menuitem view="EnableUserView" filter="isDisabled=true"
</contextmenu>
```

Class based filters

When expression based filters are not enough, you can write a small Java class that is used to filter the menu item. The preferred way is to extend the

`AsMenuItemFilter` class and supply the class to which the context menu is defined as the parameter type.

Below is an example of a class based menu filter:

```
/**
 *
 * Example menu item filter for menu items defined in a
 * context menu defined for objects of type Member
 *
 */
public class TeMemberMenuItemFilter
    extends AsMenuItemFilter<Member> {

    public TeMemberMenuItemFilter(AsConnectionIf pConnection) {
        super(pConnection);
    }

    @Override
    public boolean include(Member pObject) {
        // Make a clever decision here by looking at the member
        // object and possibly the AS connection which you can
        // retrieve by calling getConnection()
        return false;
    }

}
```

Multiple selections and filtering

If multiple objects are selected, all objects must pass the filtering in order for a menu item to appear in the context menu. If the selected objects do not share any menu items where they all pass the filter, no context menu will be available.

Example:

```
<contextmenu type="User[]">
    <menuitem view="EnableUserView" filter="isDisabled=true"
    <menuitem view="DisableUserView" filter="isDisabled=false"
</contextmenu>
```

If multiple users are selected and one of them is disabled while the rest are enabled, no context menu will appear.

12.14 View interactivity

View interactivity is a concept through which you can let one view, also known as the source view, interact with other views. View interactivity is defined through configuration and does not require any custom code to be written.

View interactivity is currently supported by the following source view types:

- Viewport views
- Tree views

A left-click on a node in a tree view, or a left click on a row in a viewport view will trigger the view interactivity by fetching the underlying object from the server and sending it as a context object to the defined target views, in the same fashion as a drag and drop operation would. It is then up to the target view to take action on the context object.

Below is an example of view interactivity configuration in the risk perspective in the standard cCran client:

```
<view id="PositionsByRiskTree" type="tree"
    dataSourceId="POSITIONS_BY_RISK_TREE">
    <click-event type="RiskCalculationNode">
```

```
        <target view="RiskNodeValuesViewportView"/>
        <target view="RiskGroupValuesViewportView"/>
        <target view="CollateralPositionsViewportView"/>
        <target view="PositionsViewportView"/>
    </click-event>
    <click-event type="RiskGroup">
        <target view="RiskGroupValuesViewportView"/>
        <target view="PositionsViewportView"/>
    </click-event>
    <click-event type="PositionAccount">
        <target view="PositionsViewportView"/>
    </click-event>
</view>
```

Note: By holding down the control key and left-clicking on the same item as you previously clicked on, you revert the interaction. Technically, this removes the context object from the target views.

13 Advanced user interface configuration

13.1 Extending views

View extension is a powerful concept that allows you to define a new view based on another view and adding or removing field sets. When extending a view, the new view inherits all definitions from the base view, but you can also add new contexts and field sets as well as redefining various properties.

If you want to redefine the layout grouping of the base view, simply define a new grouping in the new view which will then replace the original definition. You can reference all the base view field sets as well as the ones you add.

If you wish to add or change buttons, you can add a button panel definition which will then replace the original buttons.

Below is an example of how to extend a view:

```
<view id="TI_FutureDetailsView"
    extends="TradableInstrumentDetailsView">
    <blob path="tradableInstrumentExt.
                instrumentBlob.localObject"
        model="TestFuture"/>
    <layout>
        <fieldset id="TestFuture"
            path="tradableInstrumentExt.
                  instrumentBlob.localObject">
            <field name="displayName"/>
            <field name="assetClass"/>
            <field name="underlyingId"/>
            <field name="expiry"/>
            <field name="lastTrading"/>
            <field name="isPhysical"/>
            <field name="identifier"/>
            <field name="identifierType"/>
            <field name="contractSize"/>
            <field name="collateral"/>
            <field name="collateralType"/>
            <field name="collateralSubType"/>
        </fieldset>
        <group direction="horizontal">
            <group>
                <fieldset ref="Common"/>
                <fieldset ref="RtcDetails"/>
            </group>
            <group>
                <fieldset ref="TestFuture"/>
            </group>
        </group>
    </layout>
</view>
```

In the above example, the following is done:

- A BLOB object is added
- A new field set is added to the original layout
- The original field sets are grouped with the new field set

You can extend views in several steps, and you can even simulate abstract views by removing the base view as a final configuration directive.

13.2 Modifying standard user interface configuration

13.2.1 Modifying perspective view references

Since a perspective definition is a fairly nested structure, perspective modification is restricted to adding or removing views from an already existing slot. Below is an example on how to add new views to an existing perspective:

```
<AsPerspectives>
    <perspective modify="PositionManagement">
        <slot ref="positions">
            <view id="TI_TestFutureViewportView"/>
            <view id="TI_TestOptionOnFutureViewportView"/>
            <view id="TI_TestFuturePositionsViewportView"/>
        </slot>
    </perspective>
</AsPerspectives>
```

13.2.2 Modifying data sources

To modify a data source, you simply add an `AsList` element as follows:

```
<AsList
    modify=""
    [factory=""] |
    [[memberFilter=""][text=""][type=""][userFilter=""]]
```

Note that the modifiable attributes are limited. The modify attribute must also reference a valid data source ID.

Caution: There is no validation if a modified data source has children, so when you modify either the factory or the type, you need to make sure that this does not affect any child data sources.

13.2.3 Modifying views

To modify a view, you add a view element as follows:

```
<AsMvc>
    <view modify="UserDetailsView">
        <!-- Modifications go here -->
    </view>
</AsMvc>
```

The modify attribute must point to a valid view ID.

The syntax for modifying a view is similar to the syntax for extending a view, the difference is that you do not specify a new view ID. You can add contexts, add or remove field sets, fields, buttons, etc. Please see the following sections for information on how to remove field sets and fields from a view.

13.3 Removing standard user interface configuration

13.3.1 Removing perspectives

To remove a perspective, you simply add a `perspective` element as follows:

```
<AsPerspectives>
    <perspective remove="Trading"/>
</AsPerspectives>
```

The remove attribute must reference an existing perspective ID.

13.3.2 Removing perspective view references

To remove a view from a perspective, you add a view removal reference in the same slot where the view is originally placed, inside a perspective modification definition, as follows:

```
<AsPerspectives>
    <perspective modify="PositionManagement">
        <slot ref="positions">
            <view remove="PositionsViewportView"/>
        </slot>
    </perspective>
</AsPerspectives>
```

13.3.3 Removing menus

To remove a menu, you simply add a `menu` element as follows:

```
<AsMenus>
    <contextmenus>
        <contextmenu type="Session">
            <menu remove="system"/>
        </contextmenu>
    </contextmenus>
</AsMenus>
```

The remove attribute must reference an existing menu ID.

13.3.4 Removing menu items

To remove a menu item, you simply add a `menuitem` element as follows:

```
<AsMenus>
    <contextmenus>
        <contextmenu type="User">
            <menuitem remove="UserDetailsView"/>
        </contextmenu>
    </contextmenus>
</AsMenus>
```

The remove attribute must reference an existing menu item ID. Since a menu item without an explicit ID inherits its referenced view ID, you need to specify the view ID.

13.3.5 Removing data sources

To remove a data source, you simply add an `AsList` element as follows:

```
<AsList remove="CURRENCIES_ALL"/>
```

The remove attribute must reference an existing data source ID.

Caution: There is currently no validation that you could potentially remove a data source with children, so you need to take care of that manually.

13.3.6 Removing views

To remove a view, you add a view element as follows:

```
<AsMvc>
    <view remove="UserDetailsView"/>
</AsMvc>
```

The remove attribute must point to a valid view ID.

Removing a view is useful for many things, such as:

- When you use standard configuration but wish to shrink its scope within your customer project

- When you use standard configuration but want to completely change a particular view. (Removing it and defining it again)
- When you extend standard views and wish to prevent the base views from being used

13.3.7 Removing view field sets

To remove a field set from a view, you add a view element as follows:

```
<AsMvc>
    <view modify="TradableInstrumentDetailsView">
        <layout>
            <fieldset remove="Common"/>
        </layout>
    </view>
</AsMvc>
```

The remove attribute must reference an existing field set ID.

13.3.8 Removing view fields

To remove a field from a view, you add a view element as follows:

```
<AsMvc>
    <view modify="TradableInstrumentDetailsView">
        <layout>
            <fieldset id="Common">
                <field remove="subscriptionGroupId"/>
            </fieldset>
        </layout>
    </view>
</AsMvc>
```

The remove attribute must reference an existing field ID.

13.4 Enabling and disabling form items

You can configure a form field to be enabled or disabled based on other form fields. This is useful for instance if you want to allow selection of a value in a list box only if another list box in the form is empty. For example, configuring both list boxes to be enabled based on each other would make them mutually exclusive.

Example:

```
<field
    name="scheduleState"
    dataSourceId="TRADING_STATES_ALL"
    required="false"
    disabled="$model.scheduleEntries.scheduleEvent!=null"/>
<field
    name="scheduleEvent"
    dataSourceId="SCHEDULE_EVENTS_ALL"
    required="false"
    disabled="$model.scheduleEntries.scheduleState!=null"/>
```

In the example, you can select either a trading state or a schedule event, but not both.

Note: The disabling of items work with all widget types, not just list boxes.

13.5 Filtering lists based on other lists

Some list boxes in a form might need to be filtered based on other lists in the same form. For instance, if selecting a value in one list might lessen the number of valid values in another list. By using the filter attribute, you can link one list to another:

```
<!-- Add settlement account clearer ref -->
<view id="AddSettlementAccountClearerRefView" type="form"
    model="CdAddSettlementAccountClearerRefReq"
    accService=
        "CdRefDataServices_addSettlementAccountClearerRef">
    <context type="Clearer"
        source="memberId" target=
            "settlementAccountClearerRef.clearerId"/>
    <context type="SettlementAccount"
        source="settlementSystemId" target=
            "settlementAccountClearerRef.settlementSystemId"/>
    <context type="SettlementAccount"
        source="accountId" target=
            "settlementAccountClearerRef.settlementAccount"/>
    <layout>
        <fieldset id="Reference"
            path="settlementAccountClearerRef" width="300">
            <field name="clearerId"
                dataSourceId="MY_CLEARERS_ALL"/>
            <field name="settlementSystemId"
                dataSourceId="SETTLEMENT_SYSTEMS_ALL"/>
            <field name="settlementAccount"
                dataSourceId="SETTLEMENT_ACCOUNTS_ALL" filter=
"settlementSystemId=$model.settlementAccountClearerRef.settlemen
tSystemId"/>
        </fieldset>
    </layout>
</view>
```

In the example, the `settlementAccount` field is filtered using the selected value of the `settlementSystemId` field, thus decreasing the number of valid entries to only the accounts related to the selected settlement system.

Note: Filtering works with list boxes as well as with viewport boxes.

14 Performance considerations

There are a number of areas where you need to consider performance when writing code in the application server. The most important part involves being lean on the message dispatching logic. For complexity reasons, the broadcast dispatcher is single threaded, and it will likely remain single threaded in the future.

The message dispatcher calls all registered broadcast processors and broadcast listeners one by one, so a delay in one processor or listener will delay the message to all the other. Therefore you need to make sure that all code executed in the context of the message dispatcher is as fast as possible. Code executed in the context of the message dispatcher involves the following:

- Broadcast processors
- Broadcast listeners
- Data source filters
- Data source listeners
- Get methods

Note: Get methods are normally not executed as part of the message dispatcher, but we have seen instances where getters are explicitly executed from filters.

14.1 Out-of-dispatcher processing

If you have to do complex processing when messages arrive, you should aim for a design that performs the processing outside the message dispatcher context. At this point there is no generic queue and associated thread pool concept within the application server, so you need to build your own if you need multi-threaded processing. The recommended approach is as follows:

Build an application server component which is accessible via a singleton interface, in the same way as the `AsBdxHandlerIf` component.

Include the new component in the application start-up by writing a custom bootstrap class which also creates and starts your new component.

In the relevant broadcast flow handlers, call your new component with the received message in the same way as you already do with the `AsBdxHandlerIf` component.

In your new component, put relevant incoming messages on a queue and create as many dequeuer instances as you need. Then, do your processing in the dequeuer.

14.2 Broadcast processors and broadcast listeners

Broadcast processors and broadcast listeners are executed in the context of the message dispatcher, so you should generally do a minimal amount of work in them. A common use for broadcast processors and listeners is to maintain derived data by creating new objects and submitting them on the broadcast queue. While this is generally not an issue, you should avoid creating multiple objects unless you have to, since creating multiple objects every time a message of a certain type arrives effectively multiplies the inbound message rate by the number of objects you create.

14.3 Data source filters

Data source filters are units of code that make a decision whether or not to filter an object. Filters are executed in a number of situations:

- When a message is to be added, updated or removed
- When a list is being created from another list, as part of receiving the parent list snapshot
- When a user filters a viewport view

While the first scenario is on a per message basis, the second and third situations are multi-message scenarios that can potentially span large amounts of data. Therefore it is of utmost importance that a filter is as fast as possible:

- Do not loop through collections of unknown or large sizes, and never make assumptions that a collection is small unless you are absolutely sure
- Carefully examine all called utility methods to make sure they are as fast as possible
- Do not read data from locations where you may be blocked by locks

14.4 Get methods

Get methods are primarily executed when a client fetches viewport data, but we have also seen instances of getters being explicitly executed as part of filters, so getters fall under the same performance rules as previously mentioned:

- A getter must be as fast as possible
- A getter should typically always compute its result from the parameter object, and therefore not read data from the surrounding environment
- A getter used in conjunction with data sorting must adhere to the read consistency rule described in section 7.2.2

14.5 Business type formatters

Business type formatters are normally invoked when the client fetches viewport data, either directly, or from get methods. A formatter therefore fall under the same performance rules as previously mentioned.

- If you use formatters from the `java.text` package, do not create a new formatter for every call, create one instance, keep a reference to it, and re-use it for all successive calls
- If a re-used formatter is not inherently thread safe, you must make sure that you synchronize the calls to it

14.6 Data source listeners

Data source listeners are units of code that are executed when data is added, updated or removed from the list being listened. Other events include when the list being listened to is being cleared or destroyed.

In general, the performance consideration rules mentioned earlier also apply to data source listeners. The considerations are perhaps even more important to data source listeners as data source listeners are not as visible as other message dispatching components - they are added programmatically, usually from data source factories, but sometimes from other locations.

You should typically do as little as possible in a data source listener. Do not perform complex calculations, data aggregations, data transformations or other CPU consuming work, as this will slow down the transaction in the list to which the data source listener is attached.

15 APPENDICES

15.1 Configuration test

To aid in validating your Ciguan configuration, there is a base class, `AsConfigurationTest`, which you can extend, and point it to your top level configuration file. (Also known as the boot module) Below is a simple example of a customer specific configuration test:

```
public class RtcExConfigurationTest extends AsConfigurationTest
{

    @BeforeClass
    public static void before() {
        cModule = "com.cinnober.as.conf.RtcExConfig";
        AsConfigurationTest.before();
    }

}
```

The example above will test the module `com.cinnober.as.conf.RtcExConfig.cwf.xml` and all its inherited modules.

15.1.1 Specifying constant group name as data source ID

In rare cases there might be a need to use a list box widget with multiple selections to populate an integer array property with values from a constant group. To do this, you need to do two things:

- Make sure the array attribute is not annotated as a constant group
- Use `dataSourceId="<constant group class name>"` in the UI configuration

While this works, it will cause the configuration test to fail since there is no data source definition for the constant group list. (These lists are created silently during metadata creation at server start-up) To allow the tests to pass, you can now specify the constant group class name(s) in your test class as follows:

```
public class RtcExConfigurationTest extends AsConfigurationTest
{

    @BeforeClass
    public static void before() {
        cModule = "com.cinnober.as.conf.RtcExConfig";
        cReferencedConstantGroupDataSourceIds =
            "MemberType,TRADE_STATE";
        AsConfigurationTest.before();
    }
}
```

The `cReferencedConstantGroupDataSourceIds` string is a comma separated list of constant group class names (without package) which are referenced by form fields.

15.1.2 Handling duplicate definitions

Generally speaking, there should be no need for duplicate definitions of views, data sources, etc. You should always aim to either extend or modify standard configuration, or if that is not possible, make a copy of the original definition and give it a new name. That said, in some cases the quickest solution is still to copy the original definition into your own configuration and then make changes to it. This will however cause the configuration test to generate warning messages.

There is a way around this by specifying how duplicate definitions should be reported in the test. You do this by setting the global `cReportDuplicateDefinition` attribute to either `error`, `warning` or `ignore` in the `before` method in your test class as follows:

```
public class DemoConfigurationTest extends AsConfigurationTest {

    @BeforeClass
    public static void before() {
        cModule = "com.cinnober.as.conf.Demo";
        cReportDuplicateDefinition = Result.ignore;
        AsConfigurationTest.before();
    }

}
```

The default duplicate definition status is currently `warning`, which is probably sufficient in most cases.

15.2 Adding a business type

Adding a business type can be an extensive task if you have no existing business type collection to add it to. The list below assumes that you do not have an existing business type collection, so in case you do have a collection, some items can be omitted.

15.2.1 Common steps

1. Create a new enumeration that implements `CwfBusinessTypeIf` - use `CwfRtcBusinessTypes` as a template. The `isOrdinal` method defines whether values are subject to range filtering or not. (>= / <=)

Note: The static block where all enumeration instances are added to the `CwfBusinessTypes` enumeration is very important.

2. Add components to the new enumeration, one per business type.

15.2.2 Server side steps

1. Reference the new enumeration somewhere in code that is referenced during the application server startup - see `AsRtcMetaDataHandler.java` and `RtcBeanConfig.cwf.xml` for an example.
2. Create a new formatter implementation and add formatting to the new business types - See `AsRtcFormat` for an example.
3. Configure the new formatter to be used - See `RtcBeanConfig.cwf.xml` for an example.