Natural for Ajax

Custom Controls

Version 9.1.4

October 2021

This document applies to Natural for Ajax Version 9.1.4 and all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: CIT-CUSTOMCONTROLS-914-20210925

Table of Contents

Preface

This documentation provides information on how to develop your own custom controls with Natural for Ajax. It is organized under the following headings:

Overview	General information about custom controls and when to use them.
Control Concept	Details about the control concept and how to create custom controls.
Creating Macro Controls Out of Existing Controls	How to create macro controls from existing controls.
Creating New Controls	How to create completely new controls.
Special Issues	Additional advanced topics for control creation.

1 About this Documentation

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
`Monospace font`	Identifies service names and locations in the format *`folder.subfolder.service`*, APIs, Java classes, methods, properties.
`Italic`	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at **https://documentation.softwareag.com**.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at **https://empower.softwareag.com/**.

You can find product information on the Software AG Empower Product Support website at **https://empower.softwareag.com**.

To submit feature/enhancement requests, get information about product availability, and download products, go to **Products**.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the **Knowledge Center**.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at **https://empower.softwareag.com/public_directory.aspx** and give us a call.

Software AG Tech Community

You can find documentation and other technical information on the Software AG Tech Community website at **https://techcommunity.softwareag.com**. You can:

- Access product documentation, if you have Tech Community credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

2 Overview

This documentation provides information on the Natural for Ajax control concept. It is recommended that you first become familiar with the “normal development” of screens inside Natural for Ajax.

When do you need custom controls? In general there are two cases:

1. You want to combine existing controls to form complex controls with a certain dedicated task. Maybe you want to define an “address area” control which consists of a certain arrangement of fields and labels that form an address. These kinds of controls are called “macro controls” in this documentation - you take what is available and group it into certain units.
2. You want to create new controls - maybe you need some special kind of icon with a certain behavior.

While macro controls do not require to deal with JavaScript and HTML, creating completely new controls requires knowledge of JavaScript and HTML and the use of the JavaScript library functions that are available via the Natural for Ajax framework.

Natural for Ajax supports a flexible and open control framework. The basic concepts of this control framework are XML as the layout definition format and interfaces for integrating control definitions into the page generation process of Natural for Ajax. This framework is not specific to custom controls. All Natural for Ajax controls are using this control framework themselves.

The first concept is the definition of controls, that is, control tags with certain attributes which you can integrate via a tag library concept into layout definitions. The second concept is the binding of the control to server-side Natural adapter fields and events. Following the strict Natural for Ajax architecture, dynamic controls have to transfer their data at runtime to/from Natural adapter fields. This binding concept is important for new controls and for macro controls:

- When creating new controls, you want to bind the control to corresponding Natural adapter fields and events.

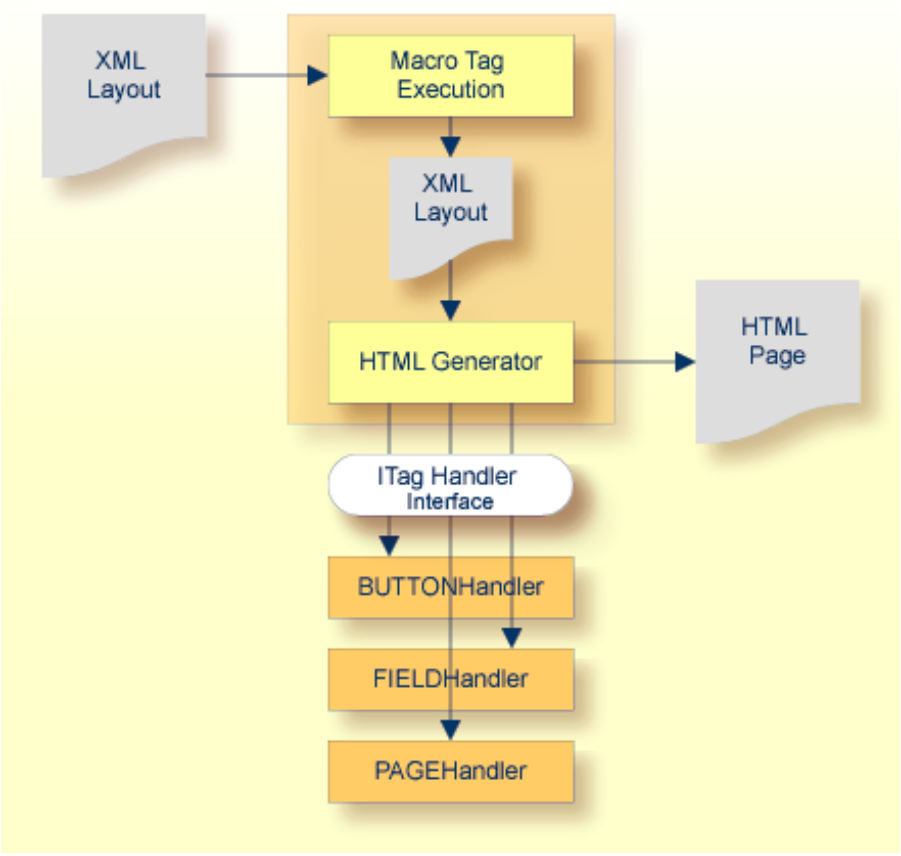
- When creating macro controls (for example, an address area), you sometimes want your control to provide some additional server-side logic. For advanced controls, this logic may go beyond the logic of the single controls which make up the macro control.

See *Binding between Page and Adapter* in the *Special Development Topics* (part of the Application Designer documentation) for detailed information, especially about the hierarchical name binding concept (access path).

3 Control Concept

Page Generation

The page generation is the process of transferring an XML layout definition into an HTML/JavaScript page. It is automatically executed inside the Layout Painter when previewing a layout. It can also be called from outside.



A generator program (`com.softwareag.cis.gui.generate.HTMLGenerator`) is receiving a string which contains the XML layout definition. The generator program parses this string with an XML parser and as a consequence processes the string tag by tag.

The generation of HTML pages is done in two steps:

- **Macro Execution**
  First, each tag of the XML layout is checked if it is a so-called “macro tag”. A macro tag is a tag which does not produce HTML/JavaScript output itself but which itself produces XML tags. Imagine a control rendering an address input: this control is using existing controls in order to create some defined output area representing an address. The HTML/JavaScript is not produced by the address control directly - the address control internally creates other controls (such as fields or buttons) which themselves produce corresponding HTML/JavaScript code.

The execution of macro tags is recursively done until no macro tag is contained in the XML layout anymore; that is, macro tags themselves can internally use macro tags.

- **HTML Generation**
  After having executed the macros, the rendering of HTML/JavaScript is started. This is done by calling corresponding tag handlers for each tag. A tag handler is a Java class and is applied to the corresponding tag via naming conventions. The HTML generator instantiates objects of a tag handler class for the tags and calls corresponding methods of these tag handlers.

  Each tag handler is called via a defined interface (`com.softwareag.cis.gui.generate.ITagHandler`) and can contribute to the generation process. All tag data, including the properties from the layout definition, is passed to the tag handler. In addition, a string with the current HTML/JavaScript is passed and the tag handler can add its control-specific HTML/JavaScript to this HTML/JavaScript string.

A tag handler instance is called at three different points of time by the generator:

- when the tag is starting (for example, the generator finds "<page…>"),
- when the tag is closing (for example, the generator finds "</page>"),
- when the generator creates a defined JavaScript method which is called at runtime in the browser when the page is loaded.

It is now the task of the tag handler to create HTML/JavaScript statements at the right point of time.

Tag Handlers and Macro Tag Handlers

As described above for the HTML generation, control-specific tag handlers are called. The macro execution is either completely done based on XML definitions (see *Creating Macro Controls Out of Existing Controls*) or you can also implement a specific macro tag handler. This macro tag handler is then called during the macro execution. In case the macro execution is completely done based on XML definitions, a general macro tag handler is used internally.

Macro tag handlers and tag handlers are Java classes which implement specific interfaces. The corresponding classes are applied to the tags via the following naming convention: for a tag `<mytag>`, the corresponding Java class must have the name "MYTAGHandler".

The following topics describe the tag handler and the macro tag handler interfaces in more detail:

- Macro Tag Handlers (IMacroTagHandler)
- Tag Handlers (ITagHandler)
- Call Sequence (IMacroTagHandler and ITagHandler)

- Extensions of IMacroTagHandler and ITagHandler

Macro Tag Handlers (IMacroTagHandler)

The interface `com.softwareag.cis.gui.generate.IMacroTagHandler` contains two methods which represent the different points of time when the generator calls the tag handler during the macro execution phase.

```
package com.softwareag.cis.gui.generate;

import org.xml.sax.AttributeList;
import com.softwareag.cis.gui.protocol.ProtocolItem;

public interface IMacroTagHandler
{
    public void generateXMLForStartTag(String tagName,
                                AttributeList attrlist,
                                StringBuffer sb,
                                ProtocolItem pi);
    public void generateXMlForEndTag(String tagName,
                                     StringBuffer sb);
}
```

Detailed information about the methods can be found inside the Javadoc documentation which is part of your Natural for Ajax installation.

Tag Handlers (ITagHandler)

The interface `com.softwareag.cis.gui.generate-ITagHandler` contains three methods that represent the different points of time when the generator calls a tag handler during the HTML generation phase.

```
package com.softwareag.cis.gui.generate;

import org.xml.sax.AttributeList;
import com.softwareag.cis.gui.protocol.*;

public interface ITagHandler
{
    public void generateHTMLForStartTag(int id,
                                        String tagName,
                                        AttributeList attrlist,
                                        ITagHandler[] handlersAbove,
                                        StringBuffer sb,
                                        ProtocolItem protocolItem);

    public void generateHTMLForEndTag(String tagName,
                                      StringBuffer sb);
```

```
    public void generateJavaScriptForInit(int id,
                                          String tagName,
                                          StringBuffer sb);
}
```

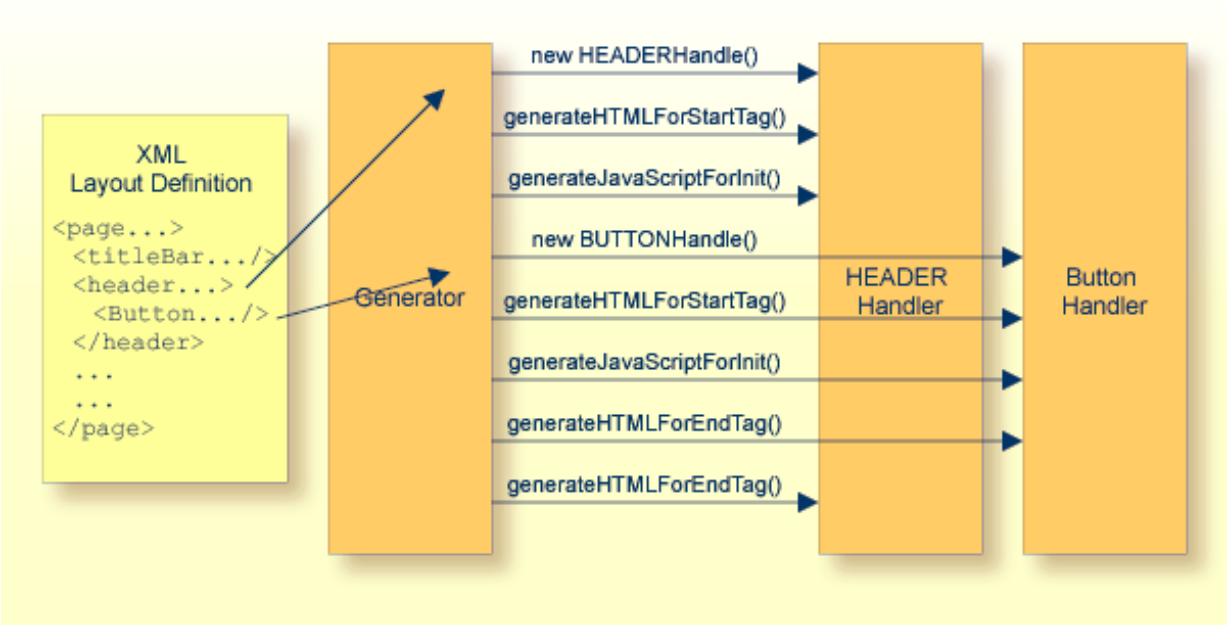
Detailed information about the methods can be found inside the Javadoc documentation which is part of your Natural for Ajax installation.

Call Sequence (IMacroTagHandler and ITagHandler)

A tag is processed by the generator in a certain way that is now described for the HTML generation phase. (The macro execution phase is processed in a similar way.)

- The generator finds the tag, reads its properties and assigns an ID. The ID is unique inside one page.
- The generator creates a new instance of the tag handler which is responsible for processing the tag.
- The generator calls the `generateHTMLForStartTag` or `generateXMLForStartTag` method correspondingly. It passes the list of properties, the string buffer which represents the HTML/JavaScript or XML string correspondingly and a protocol item in which the tag handler can store further information.
- For tag handlers only: The generator calls the `generateJavaScriptForInit` method. It passes as main parameter a string representing the method body of the `initialisation` method. You can append JavaScript statements to this string.
- (If the generator finds tags below the current tag, these tags are processed in the same way now.)
- The generator finds the end tag and calls the `generateHTMLForEndTag` or `generateXMLForEndTag` method correspondingly.

The following image illustrates the call sequence for tag handlers:



Be aware of the following:

- There is one instance of a corresponding tag handler per tag. If there are three button definitions inside a layout definition, then during generation there are three instances of the `BUTTONHandler` class.
- There is one instance of a protocol item which is passed as parameter per tag. Each tag has its own protocol item. All the protocol items are collected at generation point of time to form one generation protocol.

Extensions of IMacroTagHandler and ITagHandler

There are certain interfaces which extend the framework for specific situations:

- `com.softwareag.cis.gui.generate.IMacroHandlerWithSubTags` - this is an extension of `IMacroHandler` and provides the possibility to also receive subtags of a tag.
- `com.softwareag.cis.gui.generate.ITagWithSubTagsHandler` - this is an extension of the `ITagHandler` interface and provides the possibility to also receive the subtags of a tag.
- `com.softwareag.cis.gui.generate.IRepeatCountProvider` and `com.softwareag.cis.gui.generate.IRepeatBehaviour` - these interfaces are responsible for controlling a special management for the REPEAT processing, which you use, for example, inside grids (ROWTABLEAREA2).

You do not need to know anything about these extensions to create your first controls. Documentation is provided inside the Javadoc documentation.

Library Concept

The library concept is responsible for defining the way how the generator finds a tag handler class for a certain tag. There are two situations:

1. The generator finds a tag without a ":" character. This indicates that this is a control from the Natural for Ajax product - the according tag handler is found inside the package `com.softwareag.cis.gui.generate`, the class name is created by converting the tag name to upper case and appending "Handler".

   For example, if the generator finds the tag "header", it tries to use a tag handler class `com.softwareag.cis.gui.generate.HEADERHandler`.

2. The generator finds a tag with a ":" character, for example, `demo:address`. This indicates that an external control library is used. There is a central configuration file (*<installdir>/config/control-libraries.xml*) which contains the external control library names and the corresponding Java package names. Besides this central configuration file, also corresponding configuration files on the user interface level are supported. This is explained later. After having found the package name, the class name is built in the same way as with standard Application Designer controls.

   For example, if the generator finds the tag `demo:address` and in the configuration file the demo prefix is assigned to the package *com.softwareag.cis.demolibrary*, then the full class name of the tag handler is `com.softwareag.cis.demolibrary.ADDRESSHandler`.

What happens if the generator does not find a valid class for a certain tag? In this case, it just copies the tag of the layout definition inside the generated HTML/JavaScript string. Via this mechanism, it is possible to define, for example, HTML tags inside the layout definition which are just copied into the HTML/JavaScript generation result.

When writing your own controls, be sure to use a tag name with your own prefix (such as `test:mycontrol`) and use your own Java package name which must not start with `com.softwareag`. The tag names without prefixes and the Java package `com.softwareag` are reserved for the Natural for Ajax product.

Control Libraries

A control library is a Java library containing `ItagHandler/IMacroTagHandler` implementations. The corresponding *.jar* or *.class* files can be copied either to the central *WEB-INF/lib* or *WEB-INF/classes* directory of the *cisnatural* web application, or they can be copied to the *./appclasses/lib* or *./appclasses/classes* subdirectory of a user interface component.

The central control file for configuring control libraries in your installation is the file *<webapp-dir>/cis/config/controllibraries.xml*. An example of the file looks as follows:

```
<controllibraries>
 <library package="com.softwareag.cis.demolibrary"
          prefix="demo">
 </library>
</controllibraries>
```

Each library is listed with its tag prefix and with the package name in which the generator looks for tag handler classes.

As an alternative to this central control file, you can have a file *./cisconfig/controllibraries.xml* in your user interface component (for example, *njxdemos/cisconfig/controllibraries.xml*). The format of this file is identical to the central control file *controllibraries.xml*.

Binding Concept

The normal binding concept between a page and a corresponding class is:

- Controls refer to properties and methods.
- For pages implemented with Java, properties and methods are directly implemented as `set/get` methods or as straight methods inside the adapter class.

  As you might already have read in the part *Binding between Page and Adapter* of the *Special Development Topics* (part of the Application Designer documentation), the binding is much more flexible. You can define hierarchical access paths for both methods and properties.

  Note: The Application Designer documentation is included in the Natural for Ajax distribution package.

- For pages implemented with Natural, properties and methods are mapped to an internally used XML representation of the data. The Service Data Objects technology (SDO) is used for manipulating the XML internally (see also ***http://download.oracle.com/otndocs/jcp/sdo-2_1_1-fr-oth-JSpec/***). The XML is then bound to fields and events in the generated Natural adapter.

  If you do not delegate the data binding to already existing controls, you need to call specific methods in your handler class which add the corresponding data structures to the SDO. The custom control examples in the Natural for Ajax demos contain corresponding guidelines.

  For complex bindings, you sometimes need to implement a corresponding binding class. A binding class is a Java class which extends the class `com.softwareag.cis.adapter.ndo.NDOCustomControlInfoBase`. You sometimes use a binding class if you want to implement some control functionality which is not appropriate to be implemented in the JavaScript layer. There is also a naming convention for these binding classes. For a tag `demo:address`, the name of the binding class would be ADDRESSInfo. The custom control examples in the Natural for Ajax demos contain corresponding guidelines.

Integrating Controls into the Layout Painter

Once having created new controls, you want to use them inside the Layout Painter. The Layout Painter is configured by a set of XML files.

The files with the name *editor_*.xml* are used to extend the Layout Painter with your own custom controls. These *editor_*.xml* files can either be located centrally in *<webappdir>/cis/config/* or in a subfolder *cisconfig* of a user interface component.

There is a central file *editor.xml* which defines the central controls that come with the Natural for Ajax framework. For each control, the properties and how the control fits into other controls is defined. In addition, data type definitions to provide value help for the properties are defined inside this file.

In short: *editor.xml* controls the way in which controls are presented inside the Layout Painter.

When creating new controls, you want to integrate your controls into the Layout Painter, that is, you want to register them inside *editor.xml* as well. Instead of letting you directly manipulate *editor.xml*, there is an extension concept - in order to keep your definitions untouched by release upgrades. Again naming conventions are used: for a control library named "demo", you would define your controls in a file with the name *editor_demo.xml*. This means that you will have one *editor_*.xml* file per control library.

Have a look at the *editor_demo.xml* file:

```
<!-- DEMO:ADDRESSROWAREA2 -->
  <tag name="demo:addressrowarea2">
    <attribute name="addressprop" mandatory="true"/>
    <protocolitem>
    </protocolitem>
  </tag>
  <tagsubnodeextension control="pagebody" newsubnode="demo:addressrowarea2"/>
```

In this example, a new control `demo:addressrowarea2` is defined:

- It provides one property `addressprop`.
- It can be placed into the existing Application Designer control `pagebody`.

Or have a look at the following section:

```
<!-- DEMO:ADDRESSROWAREA3 -->
  <tag name="demo:addressrowarea3">
    <attribute name="addressprop" mandatory="true"/>
    <taginstance>
      <rowarea name="Address">
          <itr>
              <label name="First Name" width="100">
              </label>
              <field valueprop="$addressprop$.firstName" width="150">
              </field>
          </itr>
          <itr>
              <label name="Last Name" width="100">
              </label>
              <field valueprop="$addressprop$.lastName" width="150">
              </field>
          </itr>
          <vdist height="10">
          </vdist>
          <itr>
              <label name="Street" width="100">
              </label>
              <field valueprop="$addressprop$.street" width="300">
              </field>
          </itr>
          <itr>
              <label name="Town" width="100">
              </label>
              <field valueprop="$addressprop$.zipCode" width="50">
              </field>
              <hdist width="5">
              </hdist>
              <field valueprop="$addressprop$.town" width="245">
              </field>
          </itr>
          <vdist height="10">
          </vdist>
          <itr>
              <hdist width="100">
              </hdist>
              <button name="Clear" method="$addressprop$.clearAddress">
              </button>
          </itr>
      </rowarea>
    </taginstance>
    <protocolitem>
        <addproperty name="$addressprop$" datatype="ADDRESSInfo" ↩
useincodegenerator="true"/>
    </protocolitem>
  </tag>
  <tagsubnodeextension control="pagebody" newsubnode="demo:addressrowarea3"/>
```

The control `demo:addressarea3` has the following features:

- It provides one property `addressprop`.
- It contains the macro XML (between `<taginstance>` and `</taginstance>`) for building the control out of existing controls.
- It binds to an address property of type `ADDRESSInfo` (between `<protocolitem>` and `</protocolitem>`).
- It can be positioned below the `pagebody` control.

Whenever possible we recommend to use the Control Editor to create and edit the *editor_*.xml* files.

Note: If you are working with NaturalONE, see *Ajax Developer* in the NaturalONE documentation for details on using the Control Editor. If you are working with the standalone version of Natural for Ajax, see *Development Workplace* in the Application Designer documentation, which is included in the Natural for Ajax distribution package, for details on using the Control Editor.

Summary

When defining new controls, there are the following resources:

- *controllibraries.xml* - to define control library prefixes and their binding to a certain Java package holding control implementations.
- *editor_*.xml* - to define the controls and how they fit into existing controls.
- `IMacroTagHandler` - implementations that transfer XML control definitions into other XML control definitions. Remember: Implementing your own macro tag handler is optional. If a control does not have its own macro tag handler, a general macro tag handler is used internally.
- `ITagHandler` - implementations that transfer XML control definitions into HTML/JavaScript.
- `NDOCustomControlInfoBase` - base class to implement complex data binding or control functionality for execution at runtime.

The next section contains examples for building macro controls and new controls.

4 Creating Macro Controls Out of Existing Controls

General Information

There are two types of controls that you can create with Natural for Ajax:

- **Graphical Controls**
  A graphical control transforms an element of an XML layout definition into HTML/JavaScript code. You can either create completely new controls by writing your own HTML/JavaScript, or you can reuse existing controls and compose the HTML/JavaScript of these controls to new controls. The latter is called “macro control”. The recommendation is to use macro controls if possible and to write your own HTML/JavaScript only if needed.

  Example: You can define an address area that comprises several existing controls (such as ITR, LABEL and FIELD). You call the address area "NADC:ADDRESS" where "NADC" is the library prefix. The control is a kind of macro that expands a short XML layout definition, which just contains the NADC:ADDRESS control tag, into a more complex layout definition containing all the single control tags (such as ITR, LABEL and FIELD).
- **Non-visual Controls**
  A non-visual control adds some data binding to the layout. It allows your Natural program to exchange data with the Natural for Ajax framework and the browser client. The control can decide whether the data is available in the browser or only available in the Natural for Ajax framework of your web application. Non-visual custom controls are usually defined as macro controls from existing non-visual controls.

  Examples: You would like to define a specific data structure. This structure should be the same for multiple layouts of your application. To do so, you would define a macro control composed of several XCIDATADEF controls. Or you would like multiple of your layouts to exchange context data. To do so, you could define a macro control composed of XCICONTEXT controls.

The following two aspects of macro controls are important:

- **Layout Aspect**
  From the layout aspect, macro controls help to be flexible regarding design changes. Macro controls make sure that a certain graphical arrangement of existing controls is not applied to various page layouts by using copy-and-paste, but by using a proper control definition. When changing the control definition and re-generating the layout definitions that use the control, all changes in the control are automatically propagated to the page layouts.
- **Server-side Aspect**
  From the server-side aspect, a macro control may have pre-designed server-side Natural data fields and events that can be associated with it. For example, an address control may trigger an event on the Natural server to check the validity of a zip code that the user has specified.

Creating Macro Controls

Creating a macro control consists of the following steps:

- Defining a New Control Library
- Defining the Control Attributes and Control Hierarchy (Subtags, Container)
- Definining/Implementing the Rendering of the Control
- Implementing the Control's Server-Side Processing (Optional)
- Putting Things to Work

The topics below describe a sample control which is used to specify an address.

We recommend that you use all-lowercase letters for prefixes, control names and attributed names. A good example for this is:

nadc:zipcodecity

A bad example would be the following:

NaDc:zipCodeCity

Defining a New Control Library

Each control that is not supplied by Software AG requires a prefix that ensures that controls supplied by different providers can be used within one page. In our example, we use the prefix "nadc" for “Natural for Ajax Demo Controls”.

The setup is done in the file <*project*>/*cisconfig*/*controllibraries.xml*.

An example for registering the nadc library would be:

```
<library prefix="nadc"
         package="mycontrols.nadc">
</library>
```

The package that is included in the definition is the Java package that contains corresponding tag handlers (optional).

Note: In order to activate new control libraries, you must restart the Tomcat server. For NaturalONE, this means restarting Eclipse.

Defining the Control Attributes and Control Hierarchy (Subtags, Container)

For our nadc control library, we create the file *editor_nadc.xml*. This file can be created using the Control Editor.

Note: If you are working with NaturalONE, see *Ajax Developer* in the NaturalONE documentation for details on using the Control Editor. If you are working with the standalone version of Natural for Ajax, see *Development Workplace* in the Application Designer documentation, which is included in the Natural for Ajax distribution package, for details on using the Control Editor.

In the Control Editor, set up the file *editor_nadc.xml*. One file can contain a number of controls. For each control, you specify the following:

The name of the control
In our example, this is "nadc:address". Be sure to add the prefix when entering the name.

The control's attributes
This is the list of attributes that a user of the control must specify when using the control inside a page. In our example, the control "nadc:address" has the following attribute:

addressprop
A reference to the runtime property implementation.

Positioning definitions
These specify where the control can be added inside a layout definition. They are used by the layout editor, which only allows controls to be placed in the specified positions.

The positioning definitions include:

Name of the section in the controls palette
The layout editor arranges all controls within the controls palette. This palette is structured into sections. If the name of a section does not yet exist, a new section is created automatically. In our example, the name of the section is "NJXDemos".

Embedding containers
A list of all controls which allow the new control to be positioned inside it. In our example, we decide to position the controls below "pagebody", "rowarea", "colarea" and "splitcell".

After maintaining the information in the Control Editor, the content of the file *editor_nadc.xml* is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<controllibrary>
  <editor>
    <tag name="nadc:address">
      <attribute name="addressprop" mandatory="true"/>
      <taginstance>
      </taginstance>
      <protocolitem>
      </protocolitem>
    </tag>
    <tagsubnodeextension control="pagebody" newsubnode="nadc:address"/>
    <tagsubnodeextension control="rowarea" newsubnode="nadc:address"/>
    <tagsubnodeextension control="colarea" newsubnode="nadc:address"/>
    <tagsubnodeextension control="splitcell" newsubnode="nadc:address"/>
    <taggroupsubnodeextension group="NJXDemos" newsubnode="nadc:address"/>
  </editor>
</controllibrary>
```

Definining/Implementing the Rendering of the Control

You can define the rendering either in a descriptive way (XML) in the *editor_nadc.xml* file or you can define it by implementing a tag handler. The Natural for Ajax demos contain examples for both options. Here, we will describe how to implement the rendering using a tag handler. The tag handler is a Java class which defines how the control's "short XML" (for example, `<nadc:address addressprop='person'/>`) is transformed into an XML layout definition which itself contains standard controls or own controls. The tag handler class must extend the interface `IMacroTagHandler`. For details, see the corresponding Java API documentation. The tag handler has to take care of two main issues:

- defining the rendering of the control;
- defining data bindings (advanced usage).

First, we discuss the sample "nadc:address" control. The tag handler is implemented in the package `com.softwareag.cis.test.customcontrols`; this is the package that was defined with the control library prefix "nadc" in the configuration file *controllibraries.xml*. The name of the tag handler class follows the convention `<controlInUpperCase>Handler`, in our case "ADDRESSHandler".

```
package com.softwareag.cis.test.customcontrols;

import org.xml.sax.AttributeList;

import com.softwareag.cis.gui.generate.IMacroTagHandler;
import com.softwareag.cis.gui.generate.IXSDGenerationHandler;
import com.softwareag.cis.gui.protocol.Message;
import com.softwareag.cis.gui.protocol.ProtocolItem;

public class ADDRESSHandler
    implements IMacroTagHandler
{
```

```
    public void generateXMLForStartTag(String tagName,
                                       AttributeList attributes,
                                       StringBuffer xml,
                                       ProtocolItem protocolItem)
    {
        // read attributes
        String ap = attributes.getValue("addressprop");
        // rendering
        xml.append
        (
            "<rowarea name='Address'>" +
              "<itr>" +
                "<label width='120' name='First/ Last Name'/>" +
                "<field valueprop='"+ap+".firstName' width='150'/>" +
                "<hdist width='5'/>" +
                "<field valueprop='"+ap+".lastName' width='150'/>" +
              "</itr>" +
              "<itr>" +
                "<label width='120' name='Street'/>" +
                "<field valueprop='"+ap+".street' width='305'/>" +
              "</itr>" +
              "<itr>" +
                "<label width='120' name='Zip Code/ City'/>" +
                "<field valueprop='"+ap+".zipCode' width='80'/>" +
                "<hdist width='5'/>" +
                "<field valueprop='"+ap+".city' width='220'/>" +
                "<hdist width='10'/>" +
                "<button name='Check' method='"+ap+".onCheck'/>" +
              "</itr>" +
            "</rowarea>"
        );
        IXSDGenerationHandler xga = protocolItem.findXSDGenerationHandler();
     xga.addControlInfoClass(protocolItem, ap, ADDRESSInfo.class);
 }


    public void generateXMlForEndTag(String arg0, StringBuffer arg1)
    {
    }

}
```

Here are the major processing steps of the tag handler:

- The attribute `addressprop` is read from the control definition and stored in the variable `ap`.
- The XML layout definition is appended by adding controls such as ROWAREA and FIELD. Note that inside the rendering definition, property and method references are prefixed with "ap" and "." (period).

- The ADDRESS control implements some advanced binding (optional). As seen later, some part of the control functionality is implemented in a corresponding binding class. This binding class `ADDRESSInfo` is registered as the server-side counterpart of the control at an `IXSDGenerationHandler` interface.

Implementing the Control's Server-Side Processing (Optional)

You can apply binding objects to controls. The association of a control to a binding object is specified in the control's tag handler via the method `addControlInfoClass` of the interface `IXSDGenerationHandler`.

The server binding objects are optional. They may encapsulate functions with the control which are executed for the control on the server side. Example: In our address control, a zip code validity check will be added. This check is automatically executed within the control when the user presses the **Check** button that is part of the control's rendering.

This server-side processing for a control is defined in a class that extends the existing class `NDOCustomControlInfoBase`. See the following code:

```
package com.softwareag.cis.test.customcontrols;

import com.softwareag.cis.adapter.ndo.NDOCustomControlInfoBase;
import commonj.sdo.DataObject;

public class ADDRESSInfo
    extends NDOCustomControlInfoBase
{
    public void onCheck()
    {
        // first execute the control's logic
        DataObject address = getDataObject();
        String zipCode = address.getString("zipCode");
        String city = address.getString("city");
        if ("64297".equals(zipCode) &&
            (city == null || city.length() ==0))
        {
            address.set("city","Darmstadt");
        }
        // delegate the method to the normal event processing
        super.invokeMethod("onCheck");
    }
}
```

We recommend the following guidelines:

- The class's package should be the same as with the handler class.
- The name of the class is `<controlInUpperCase>Info`.

The class extends the class `NDOCustomControlInfoBase` which is supplied with Natural for Ajax. The base class provides some useful functions:

- It provides the properties for the contained content (that is, "firstName", "lastName", "street", etc.).
- It provides a binding to the SDO to which the control refers. In the address example, the control binds to an "addressprop", all detail properties are defined to be "<*valueOfAddressProp*>.firstName", "<*valueOfAddressProp*>.lastName", etc. The method `getDataObject()` returns the SDO object that represents the control.
- It provides a function to map methods to events in the Natural adapter code. If you control a specific execution for a method inside the control (in the example, this is the `onCheck()` method), you can delegate the event to the normal Natural adapter event handling after having handled the control-specific matters.

Putting Things to Work

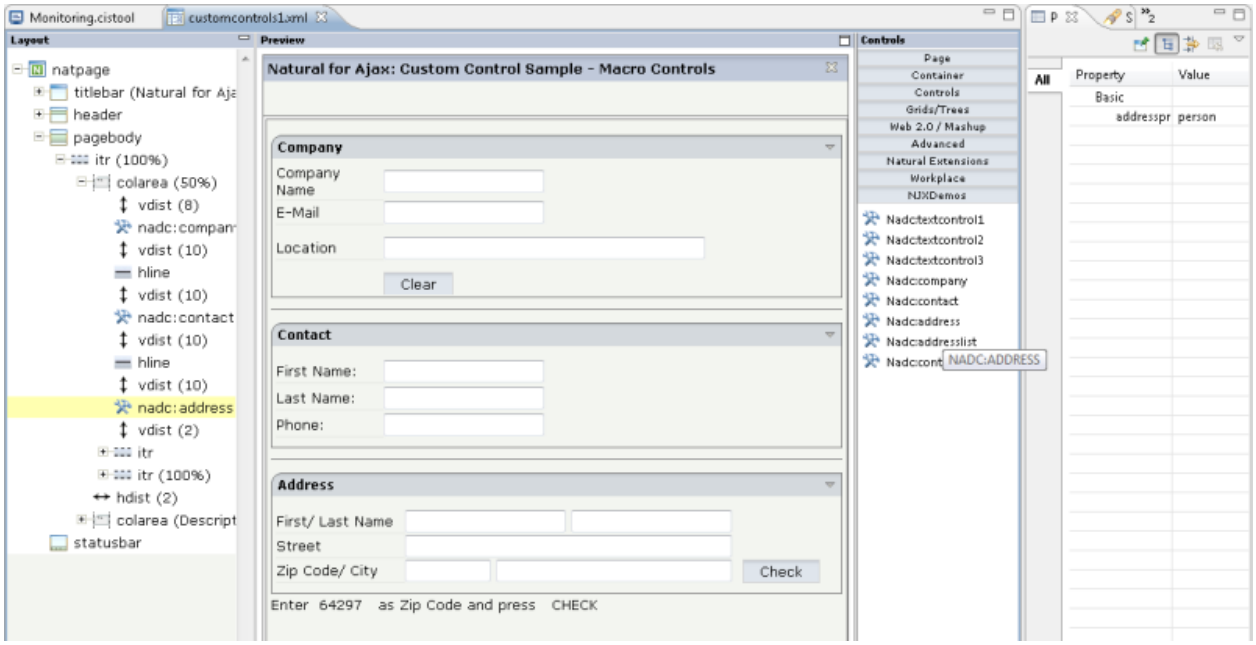
Now we show how to use the control within a page.

When you open the Layout Painter, you can see a new section in the controls palette which has the name "NJXDemos". In this section, you can find the control "nadc:address".

If you define a layout as follows:

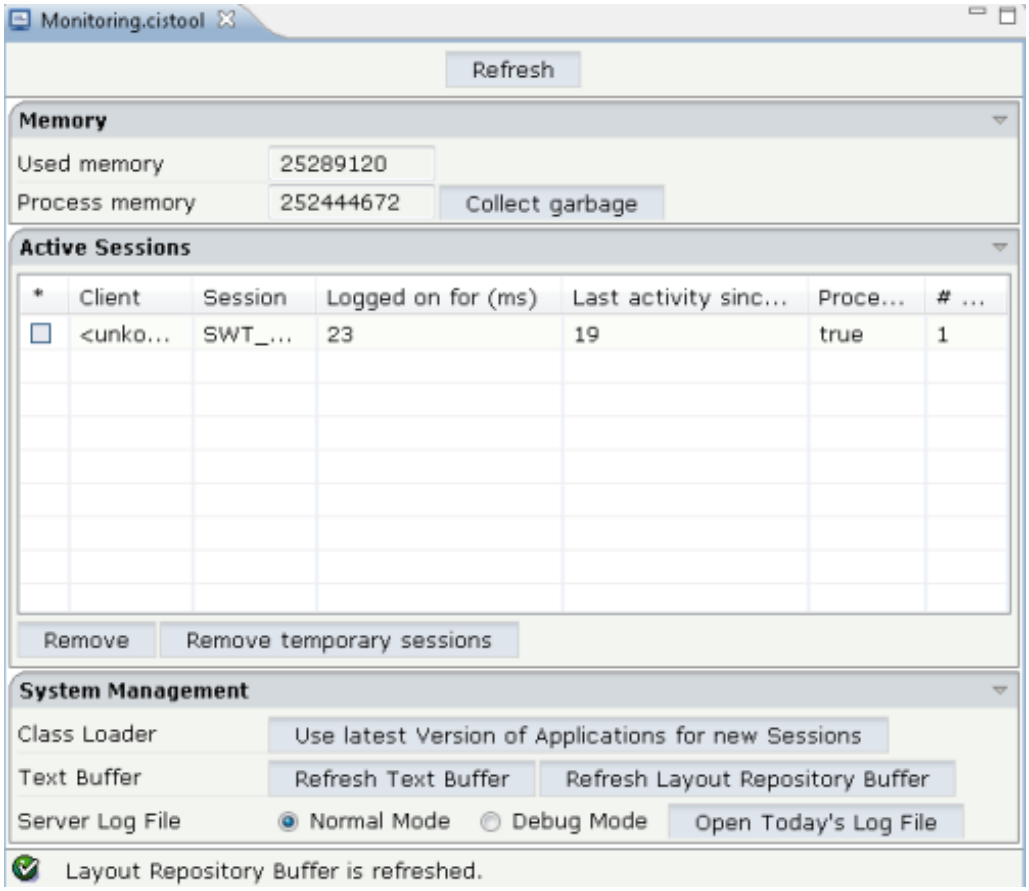
```
<natpage>
   ...
    <pagebody>
        ...
        <nadc:address addressprop="person">
        </nadc:address>
        ...
    </pagebody>
    ...
</natpage>
```

the corresponding page looks as shown below:



When you run a corresponding Natural program with this layout and you then enter "64297" in the field defined by `zipCode` and choose the **Check** button, the name of the corresponding city is “calculated” by the control processing. Note that the event `person.onCheck` is then delegated to the normal Natural adapter event processing. This means that this method can be implemented just as a normal event in the Natural program.

Important: To activate/refresh new/changed controls in existing control libraries, you must choose the **Use latest Version for Applications in new Session**, **Refresh Text Buffer** and **Refresh Layout Repository Buffer** buttons in the monitoring tool.


Monitoring.cistool
Refresh
Memory
Used memory
25289120
Process memory
252444672
Collect garbage
Active Sessions
*
Client
Session
Logged on for (ms)
Last activity sinc...
Proce...
...
<unko...
SWT_...
23
19
true
1
Remove
Remove temporary sessions
System Management
Class Loader
Use latest Version of Applications for new Sessions
Text Buffer
Refresh Text Buffer
Refresh Layout Repository Buffer
Server Log File
Normal Mode
Debug Mode
Open Today's Log File
Layout Repository Buffer is refreshed.

5 Creating New Controls

Concept

In the previous section, you learned how to create macro controls out of existing controls. You will now learn how to build completely new controls which are not yet part of the Application Designer control set.

The concept of building your own controls is to insert corresponding HTML and JavaScript instructions into the HTML page which is the result of the generation process.

A JavaScript function library is available which can be directly accessed inside the HTML code which is generated. This library contains useful methods for accessing properties as well as triggering event execution and/or taking part in the execution of events.

Njxdemos Sample Control: NADC:TEXTCONTROL1

For the sample layout, see the file *customcontrols3.xml* in the *njxdemos/xml* folder. NADC:TEXTCONTROL1 is a quite simple example of a new control: It does nothing else than writing text which is passed via a static tag attribute into the generated HTML page:

The corresponding XML layout definition looks as follows:

```
<rowarea name="Pure JavaScript/HTML...">
  <vdist height="10"></vdist>
  <itr>
    <nadc:textcontrol1 text="Some Static TEXT">
    </nadc:textcontrol1>
  </itr>
  <vdist height="10"></vdist>
</rowarea>
```

You see that the text which is passed inside the text attribute of the NADC:TEXTCONTROL1 tag is displayed inside the control in bold letters.

For details on the corresponding tag handler, see the description and the source code for the `CUSTC3-P.NSP` example in the Natural for Ajax demos.

JavaScript Functions

For more interactive controls - for example, which use certain data coming from the server-side adapter - you need to access certain JavaScript functions which are available inside the client. The generated HTML page contains an object named "csciframe". This object provides a certain set of functions for usage from within custom controls.

It is not possible in JavaScript to arrange a set of published functions in some kind of interface in order to only allow users a dedicated access. Therefore, the functions which are allowed to access are listed in this section. You must not use any other functions of the Ajax framework - even if you may see additional functions in the JavaScript sources. Only the following functions belong to the public Java Script library:

Function	Description

setPropertyValue(pn,pv)	Sets a property value inside the adapter. The value is not directly sent to the server but is buffered first in the client. If there is a synchronization event, then the buffer is transferred.
	
	pn = name of property
	
	pb = value
	
	Examples:
	
	`csciframe.setPropertyValue(companyName,"Software AG");`
	
	`csciframe.setPropertyValue(address.firstName,"John");`
	
	`csciframe.setPropertyValue(addresses[2].firstName,"Maria");`

getPropertyValue(pn)	Reads a property value from the adapter (better: the client representation of the adapter).
	
	pn = name of property
	
	result = string of property value
	
	Examples:
	
	`var vResult1 = csciframe.getPropertyValue("company");`
	
	`var vResult2 = csciframe.getPropertyValue("addresses[2].firstName");`
	
	Pay attention: the adapter value is always passed back as a string.
	
	A boolean value, as a consequence, is returned as "true" string and not as "true" boolean value.

Function	Description
	Null values of the adapter, that is, where the Java adapter class on the server side passes back "null", are returned as an empty string (""). A JavaScript null value is passed back if the property for which you ask does not exist.

registerListener(me)	Passes a method pointer (me value). The method is called every time when a response of a client request is processed. In other words: every time new data comes from the server or if the model is updated in another way (for example, by flush signals of other controls), then the corresponding methods are called. In the method, you can place a corresponding reaction of your control on new data.
	
	 The method which you pass must have a parameter `model` - which is not used anymore, but which has to be defined.
	
	 Example:
	
	 `...`
	`...`
	`function reactOnNewData(model)`
	`{`
	`  var vResult = csciframe.getPropertyValue("firstName");`
	`  alert(vResult);`
	`}`
	`...`
	`...`
	`csciframe.registerListener(reactOnNewData);`
	`...`
	`...`

invokeMethodInModel(mn)	Invokes the calling of a method inside the adapter. As a consequence, the data changes which may have been buffered inside the client are flushed to the server and the method is called. mn = name of adapter method Example: `csciframe.invokeMethodInModel("onSave");`
submitModel(n)	Synchronizes the client with the server. Analogous to the `invokeMethodInModel()` method from the synchronization point of view - but now without calling an explicit method in the adapter. n = name, must be `submit` Example: `csciframe.submitModel("submit");`

Njxdemos Sample Control: NADC:TEXTCONTROL3

The following example is an extension of the above NADC:TEXTCONTROL1 example. Whereas in the NADC:TEXTCONTROL1 control, the text to be output by the control was defined at design time as a static attribute of the tag definition, the text is now dynamically derived from an adapter property. The adapter sets the final text value during a server roundtrip.

The corresponding XML layout definition looks as follows:

```
<rowarea name="With own Data Binding...">
  <vdist height="10"></vdist>
  <itr>
    <label name="Your name" width="100">
    </label>
    <field valueprop="yourname" width="200" flush="server" ↩
flushmethod="onNameChanged">
    </field>
  </itr>
  <vdist height="10"></vdist>
  <itr>
    <nadc:textcontrol3 textprop="hellotext" >
    </nadc:textcontrol3>
  </itr>
  <vdist height="10"></vdist>
</rowarea>
```

The NADC:TEXTCONTROL3 generates an own adapter property. This results in the generation of a data field in the Natural adapter.

For details on the corresponding tag handler, see the description and the source code for the `CUSTC3-P.NSP` example in the Natural for Ajax demos.

Summary

Writing new controls requires a profound knowledge of HTML and JavaScript. In principle, everything is simple, but there are a couple of pieces which have to be put together in order to form a control properly:

- You have to render the control via HTML.
- You have to manipulate the control via JavaScript - in case you have a dynamic control.
- You have to bind the control to adapter properties/methods.
- You have to pay attention to the fact that all controls are living in the same page - and there must not be any confusion with naming of IDs and method names.
- You have to use the JavaScript initialization for registering your control inside the internal eventing when new page content arrives inside the client.
- You have to properly fill the protocol item.

Some topics have been mentioned here, but have not been fully explained. For more information, see ***Special Issues***.

6 Special Issues

Protocol Item

Inside a tag handler, a protocol item is passed in the called methods. There are some mandatory tasks that you have to do with a protocol item:

- You must tell the protocol item every property you are referencing from your control.

  This information is required because only these properties are transferred from the server to the client at runtime which are referenced inside the page.
- You must tell the protocol item every text ID you are referencing from your control.

  Again this information is used to send the right text IDs to the client processing.

In case of using macro controls, one macro control is rendered into many normal controls. Each normal control is treated in the way that it generates corresponding HTML/JavaScript and in the way that it itself tells to which properties it binds; that is, each normal control adds its properties/text IDs itself: when your macro control contains some FIELD controls, then each FIELD control will tell during generation the adapter properties to which it binds - there is no necessity for you to re-tell on macro control level.

But: you might tell on macro control level that all the contained adapter properties are not provided via one-by-one implementation but by implementing a server-side class already providing all sub-properties. We call these classes "binding classes", see also *Creating Macro Controls Out of Existing Controls*. To add a binding class `ADDRESSInfo`, you use the protocol item in the following way:

- Call `IXSDGenerationHandler xga = protocolItem.findXSDGenerationHandler();` to get access to an object which implements the `IXSDGenerationHandler` interface. This object is responsible for the generation of the corresponding data bindings for NATPAGE layouts.
- Call the method `addControlInfoClass` and pass your binding class as the third parameter. The second parameter is the name of the complex property to which you apply the binding: `xga.addControlInfoClass(protocolItem, myproperty, ADDRESSInfo.class);`.

For more information, see the corresponding Javadoc files of your Natural for Ajax or NaturalONE installation.

Bringing Controls into the Layout Painter - Data Types

In a previous section, you learned how to integrate controls into the Layout Painter (see ***Integrating Controls into the Layout Painter***). In addition to the basic concepts described in that section, you can also apply data-type definitions to your control attributes.

The following example shows how to apply data types to an attribute *editor_nadc.xml* file:

```
<!--
Dynamic extension of editor.xml file.
-->

<controllibrary>
 <editor>

    <!-- datatype TEXT -->
    <datatype name="nadc:count">
      <value id="1st" name="First"/>
      <value id="2nd" name="Second"/>
      <value id="3rd" name="Third"/>
    </datatype>

    <!-- control MYCONTROL -->
    <tag name="nadc:mycontrol">
        <attribute name="fixedtext" datatype="nadc:count"/>
    </tag>
    ....
```

Note that both new data types and new control tags are named together with their prefix - in order not to mix them up with standard Application Designer controls or with controls of other control library providers.

Array Binding

As shown in the `CUSTC2-P` example of the Natural for Ajax demos, you can create controls with repeated data structures without having to deal with all the details of array bindings. The control NADC:ADDRESSLIST is an example for this. The idea is to have your own custom controls and simply reuse the repeated concepts and array bindings of the framework together with your own custom controls. We recommend to use this approach whenever possible.

If you really need to implement your own repeated data binding, you will find some details for generating and implementing your own array binding in the description of the `CUST2-P` example and the corresponding Natural for Ajax demos.

Text ID/Multi Language Controls

Please contact Software AG in case you create new controls with language-dependent information - and if you want to use the same translation methods as Application Designer does for these controls.