Last update: 07.06.2004 18:17:35
Author: Joern Turner <joern.turner@chibacon.de>
Editors: Eric Hanson, Lawrence McCay

# Chiba Cookbook

# 1 Introduction

> *„Forms are an important part of the Web, and they continue to be the primary means for enabling interactive Web applications. Web applications and electronic commerce solutions have sparked the demand for better Web forms with richer interactions. XForms 1.0 is the response to this demand, and provides a new platform-independent markup language for online interaction between a person (through an XForms Processor) and another, usually remote, agent. XForms are the successor to HTML forms, and benefit from the lessons learned from HTML forms."*
> *[XForms 1.0 W3C Candidate Recommendation 12.11.02, 1.1]*

HTML Forms once made the Web interactive, allowing users to talk back to the servers they visit. Since then whole industries have built their profits on HTML forms as they have existed since the very beginning of HTML (Hypertext Markup Language). Whole technologies have evolved around the possibility to interact with users via the Internet.

Despite the immense impact HTML Forms have had on the development of the Web, years of use has shown that they are not suitable for today's applications. Missing validation, data-typing and logic-components must be compensated for by heavy use of scripting which itself leads to bad maintainability and increasing development cost.

XForms is a new emerging standard that addresses all these issues bringing rich interaction, a well-defined processing model and universal data management to improve the situation. It's based on XML and therefore is easy to handle, offers self-describing datastructures and is adaptable to any application or environment.

## 1.1 What is XForms?

XForms is a new W3C recommendation which is the successor of HTML forms and will replace them in XHTML 2.0. It addresses the limitations of HTML forms and adds common functionality found in most form-applications like logic, validation, calculation and data-typing.

Today a diversity of frameworks is used to circumvent the weaknesses of web forms. These are either proprietary and/or commercial which makes it hard to port them to another server-infrastructure and raises the overall cost-of-ownership. While server-infrastructures like J2EE and .NET have evolved and established industry standards for nearly all requirements, the form-processing part will still be hand-coded over and over again.

Once XForms is adopted this picture will change drastically. Forms will be portable from one server to another, serve different clients, provide rich validation and calculation and allow complex logic without a single line of scripting code. Furthermore XForms is maintainable through clear separation of logic, data and presentation which allow a role-based development process.

For further information please check the links at the end of this document, especially the book from Micah Dubinko which gives a good overview and in-detail description of the technology along with some practical tips.

## 1.2 What is Chiba?

The goal of the Chiba project is to provide an Open Source XForms Implementation for the widest possible range of environments and applications. Chiba is based on Java 1.4 and up and uses XSLT for transcoding XForms to the desired target language ((X)HTML, WML, VoiceML,...). It can be run on any Java-enabled platform.

### 1.2.1 Features

- largely conforms to the XForms 1.0 Recommendation
- delivers most XForms functionality to today's browsers
- does not rely on any scriting capabilities on client-side but allows generation of script-code where appropriate
- adaptable to potentially any client by implementation of an XSLT stylesheet
- supports nearly all XForms Actions even with scriptless browsers
- strong data-typing
- validation and calculation with dependency tracking
- DOM Event support
- customizable UI stylesheets (XSLT+CSS), Actions, Connectors and SubmissionDrivers
- based on Java2 and XSLT
- no client installation required
- extensible Connector interface for resouce-loading

Although XForms was introduced as a client-side technology meant to be implemented by browers, we believe XForms will evolve in server-side applications. Up to now it's unlikely that XForms will be implemented soon in major browsers like Internet Explorer and Netscape.

The current status of the implementation is available from http://chiba.sourceforge.net/features.html.

### 1.2.2 Distributions

Chiba currently comes in two distributions:

1. as a typical servlet-based web-application (war-file)

2. as an optional jar-file (chicoon.jar) for integration with Apache Cocoon 2.0.4

Both are available as source or binary package.

# 2 Getting started

## 2.1 Installation

The Chiba processor should run on any Java-compatible operating system. It's tested on Linux (Debian) and Windows 2000 Professional. It has been reported to run with different Tomcat Versions (since Ver. 3.3), Weblogic, Sun One Server and Enhydra Multiserver as webcontainer. If you experience problems on any platform please report to the mailing list.

Requirements:

- JDK/JRE 1.4 or higher
- a Servlet 2.2 compatible webcontainer

The current release was developed and tested with JDK 1.4.2 and Tomcat 4.1.27. When you're using these components installation is easy.

Follow these steps to install Chiba:

- download `chiba-[version].war`
- copy the war-file to your webcontainers' `webapps` directory
- start the webcontainer
- point your browser to `http://localhost:8080/chiba-[version]` to view the sample forms

That's all.

In case nothing happens we would be happy if you give us a note with your system configuration and symptom description.

Note: with older versions of Tomcat you may get classloading problems or version conflicts in the parser classes. In this case move the files `xercesImpl.jar` and `xml-apis.jar` to tomcat's `lib` directory.

## 2.2 Running the sample forms

After you've installed Chiba you may want to have a look at the sample forms. These are constantly maintained with each new release and show some working XForms functionality. They always use the up-to-date namespaces and syntax currently supported. By studying them in detail you may figure out some authoring options.

To view the samples start your webcontainer, browse to the Chiba mainpage and click on the link 'samples'. You'll see a list of files and possibly some subdirs. Execute a sample by clicking on its name.

# 3 Using forms

In this section i'll try a little tutorial by explaining an example which will be detailed out step by step to show the features of XForms and Chiba. Before starting a few words about what will be needed and a warning.

If you simply want to setup a contact form for your website please go away and spend your time better. XForms surely is the future for sophisticated applications but is overdone for a simple form (for now). Of course this situation will change with the arrival of XForms tools like generators and form-designers that will free authors from the burden of the markup. - The power of XForms comes to the price of learning its markup for now but is definitely worth the effort if you want secure, consistent and maintainable forms. And as with other markup languages – its never a fault to know the details.

To start with XForms you should have at least a basic understanding of XML and XPath. For the more advanced features like creating your own styling and advanced validations you'll also need some knowledge of XML Schema, XSLT and CSS.

For experimentation with the samples you'll need:

- a good text editor preferrable with XML support
- a running installation of Chiba

[to be continued...]

# 4 Authoring XForms

XForms emphazises client-independence by using tagnames that avoid any association with visual display such as 'checkbox' or 'border'. This is because a form should be designed for a specific purpose and not for a specific client. This allows the same form to

be used with a browser, a mobile phone or even a voice application by transcoding the logical XForms UI elements to their appropriate equivalents in the target language.

So when authoring XForms you should keep this in mind and avoid excessive mixing of different markup languages. For HTML this means to leave out every HTML tags as far as possible and write only XForms markup. You can style your forms with CSS and this is always the better option if you have more than one form to write.

[say some more?]

## 4.1 UI Transformation process

Chiba internally maintains a DOM tree which represents the XForms. While XForms clearly separates model, constraints and user interface markup, this structure can be complicated to transform into a user-consumable form. To ease the job of creating a satisfying layout, Chiba annotates the input form with the helper elements/attributes described below.

While loading a document that contains XForms markup, the processor will perform the following steps:

all UI controls are iterated and the following logic is applied to each:

- A `<chiba:data>` element will be created as child of the control.
- The binding expression (if any) will be resolved and the resulting value is added as text content to the `<chiba:data>` element.
- The attributes `chiba:name`, `chiba:enabled`, `chiba:readonly`, `chiba:required`, `chiba:type` and `chiba:valid` are added to the `<chiba:data>` element. The values of these attributes will be set to match the current state of their associated instance nodes and their Model Item Properties.
- The chiba:name attribute serves as request parameter-name during http transfers. The names are used by the stylesheets to name controls in the output. When the server receives these parameters during a form-session it maps them back to their originally referenced instance-nodes.
- A chiba:xpath attribute is added which reflects the resolved xpath to the instance node it is bound to.
- Any repeat elements are rolled out; for every instance data entry the children of the repeat are cloned, wrapped into a transient xforms:group and processed as above.

The resulting structure is much easier to transform to the client's language than the original XForms document. All relevant data are directly accessible via the attribute- or child-axis in XSLT.

Examples for every single form control and its result after evaluation are shown below. Note that only elements that are involved in some XForms evaluation are mentioned here. All other elements are simply copied from input to ouput. This way input markup that comes from a namespace different to XForms is preserved and may be used to layout XForms elements.

### The input Element

*The input Element before evaluation*

```
<xf:input xf:bind="myinput1">
        ...
</xf:input>
```

*The input Element after evaluation*

```
<xf:input xf:bind="myinput1" ...>
   ...
   <chiba:data chiba:enabled="true"
               chiba:name="d19c4"
               chiba:readonly="false"
```

```
                chiba:required="false"
                chiba:type="string"
                chiba:valid="true">this is an output from a different
    model</chiba:data>
    ...
</xf:input>
```

## The secret Element

*The secret Element before evaluation*

```
<xf:secret xf:bind="mysecret1" xf:ref="secret1">
        ...
</xf:secret>
```

*The secret Element after evaluation*

```
<xf:secret       xf:bind="mysecret1" xf:ref="secret1">
    <chiba:data chiba:enabled="true"
                chiba:name="d23r3"
                chiba:readonly="false"
                chiba:required="false"
                chiba:type="string"
                chiba:valid="true">a secret of mine</chiba:data>
    ...
</xf:secret>
```

## The textarea Element

*The textarea Element before evaluation*

```
<xf:textarea xf:bind="mytext1" xf:ref="text1">
        ...
</xf:textarea>
```

*The textarea Element after Evaluation*

```
<xf:textarea xf:bind="mytext1" xf:ref="text1">
    <chiba:data chiba:enabled="true"
                chiba:name="d23r3"
                chiba:readonly="false"
                chiba:required="false"
                chiba:type="string"
                chiba:valid="true">a possibly longer text....</chiba:data>
    ...
</xf:textarea>
```

## The output Element

*The output Element before evaluation*

```
<xf:output xf:bind="myoutput1">
        ...
</xf:output>
```

*The output Element after evaluation*

```
<xf:output xf:bind="myoutput1">
    <chiba:data chiba:enabled="true"
                chiba:name="d45z6"
                chiba:readonly="false"
                chiba:required="false"
                chiba:type="string"
                chiba:valid="true"> something to output model</chiba:data>
                ...
</xf:output>
```

## The upload element

*The upload element before evaluation*

```
<xf:upload xf:bind="upload">
        ...
</xf:upload>
```

*The upload element after evaluation*

```
<xf:upload xf:bind="upload">
    <chiba:data chiba:enabled="true"
                chiba:name="d47z8"
                chiba:readonly="false"
                chiba:required="false"
                chiba:type="string"
                chiba:valid="true">mydoc.pdf</chiba:data>
    ...
</xf:upload>
```

## The range Element

*The range element before evaluation*

```
<xf:range xf:bind="upload" xf:ref="upload">
        ...
</xf:range>
```

*The range element after evaluation*

```
ATTENTION: range control is not supported yet.
<xf:range xf:bind="range">
    <chiba:data chiba:enabled="true"
                chiba:name="d56r3"
                chiba:readonly="false"
                chiba:required="false"
                chiba:type="string"
                chiba:valid="true">23.5</chiba:data>
    ...
</xf:range>
```

## The trigger Element

*The trigger element before evaluation*

```
<xf:trigger xf:id="mytrigger">
        ...
</xf:trigger>
```

*The trigger element after evaluation*

```
<xf:trigger xf:id="mytrigger">
    <chiba:data chiba:enabled="true"
                chiba:name="d29s5"
                chiba:readonly="false"
                chiba:required="false"
                chiba:type="string"
                chiba:valid="true" />
    ...
</xf:trigger>
```

## The submit Element

*The submit element before evaluation*

```
<xf:submit xf:id="mysubmission">
        ...
</xf:submit>
```

*The submit element after evaluation*

```
<xf:submit xf:id="mysubmission">
    <chiba:data chiba:enabled="true"
                chiba:name="d78o5"
                chiba:readonly="false"
                chiba:required="false"
```

```
                chiba:type="string"
                chiba:valid="true" />
    ...
</xf:submit>
```

## The select Element

*The select element before evaluation*

```
<xf:select xf:ref="myletter">
       <xf:choices>
              <xf:item>
                     <xf:label>A</xf:label>
                     <xf:value>a</xf:value>
              </xf:item>
              <xf:item>
                     <xf:label>B</xf:label>
                     <xf:value>b</xf:value>
              </xf:item>
              <xf:item>
                     <xf:label>C</xf:label>
                     <xf:value>c</xf:value>
              </xf:item>
       </xf:choices>
       ...
</xf:select>
```

*The select element after evaluation assuming 'myletter' has the value 'a b'. For each token in the string the associated item will be selected.*

```
<xf:select xf:ref="myletter">
       <xf:choices>
              <xf:item xf:selected="true">
                     <xf:label>A</xf:label>
                     <xf:value>a</xf:value>
              </xf:item>
              <xf:item xf:selected="true">
                     <xf:label>B</xf:label>
                     <xf:value>b</xf:value>
              </xf:item>
              <xf:item>
                     <xf:label>C</xf:label>
                     <xf:value>c</xf:value>
              </xf:item>
       </xf:choices>
       <chiba:data chiba:enabled="true"
                     chiba:name="d84n4"
                     chiba:readonly="false"
                     chiba:required="true"
                     chiba:type="string"
                     chiba:valid="true" />
       ...
</xf:select>
```

The data value in this case is not a child of `chiba:data` but is expressed through the `selected` attributes on `xf:item` elements.

## The select1 Element

*The select1 element before evaluation*

```
<xf:select1 xf:id="select" xf:ref="aLetter">
       <xf:choices>
              <xf:item>
                     <xf:label>A</xf:label>
                     <xf:value>a</xf:value>
              </xf:item>
              <xf:item>
                     <xf:label>B</xf:label>
                     <xf:value>b</xf:value>
              </xf:item>
              <xf:item>
                     <xf:label>C</xf:label>
```

```
                    <xf:value>c</xf:value>
                </xf:item>
        </xf:choices>
        ...
</xf:select1>
```

*The select1 element after evaluation assuming the value of 'aLetter' is 'b'.*

```
<xf:select1 xf:id="select" xf:ref="aLetter">
        <xf:choices>
                <xf:item>
                        <xf:label>A</xf:label>
                        <xf:value>a</xf:value>
                </xf:item>
                <xf:item xf:selected="true">
                        <xf:label>B</xf:label>
                        <xf:value>b</xf:value>
                </xf:item>
                <xf:item>
                        <xf:label>C</xf:label>
                        <xf:value>c</xf:value>
                </xf:item>
        </xf:choices>
        <chiba:data chiba:enabled="true"
                    chiba:name="d99n1"
                    chiba:readonly="false"
                    chiba:required="false"
                    chiba:type="string"
                    chiba:valid="false" />
        ...
</xf:select1>
```

The data value in this case is not a child of `chiba:data` but is expressed through the `selected` attributes on `xf:item` elements.

## The repeat Element

*The repeat element before evaluation*

```
<xf:repeat xf:nodeset="/a/b" xf:id="myrepeat">
        <xf:input xf:id="input1" xf:ref=".">
                ...
        </xf:input>
        ...
</xf:repeat>
```

*The repeat element after evaluation assuming the nodeset 'a/b' has two entries in the* `Instance`*.*

```
<xf:repeat xf:nodeset="/a/b" xf:id="myrepeat" chiba:index="1">
        <xf:group chiba:transient="true" chiba:position="1">
                <xf:input xf:ref=".">
                        <chiba:data chiba:enabled="true"
                                    chiba:name="d12a"
                                    chiba:readonly="false"
                                    chiba:required="false"
                                    chiba:type="string"
                                    chiba:valid="false">repeat-value1</chiba:data>
                        ...
                </xf:input>
                ...
        </xf:group>
        <xf:group chiba:transient="true" chiba:position="2">
                <xf:input xf:ref=".">
                        <chiba:data chiba:enabled="true"
                                    chiba:name="d12a1"
                                    chiba:readonly="false"
                                    chiba:required="false"
                                    chiba:type="string"
                                    chiba:valid="false">repeat-value2</chiba:data>
                        ...
                </xf:input>

                ...
```

```
      </xf:group>
</xf:repeat>
```

There are some things worth pointing out:

1. A `chiba:index` attribute will be created/updated to reflect the current position of the repeat-cursor.
2. For every entry in the instance a `xf:group` with the custom `chiba:transient` attribute is created. This is necessary to handle any mixed markup that may occur inside the repeat element correctly. The `chiba:transient` attribute signals the stylesheet writer that this group was not part of the original markup.
3. Every transient group gets a `chiba:position` attribute which holds the nodeset position of the group. This is more a convenience for stylesheet authors than a strict necessity because this position is implicit but may be expensive to compute in your tranformations.

Chiba uses some custom attribute as convenience for stylesheet writers. The following attributes are created on controls:

| | |
|---|---|
| `enabled` | reflects the status of the 'relevant' Model Item Property. Also used to 'activate' alert elements to signal an validation error |
| `readonly` | reflects the status of the 'readonly' Model Item Property |
| `required` | Reflects the status of the 'required' Model Item Property |
| `name` | parameter-name during HTTP transfers |
| `transient` | is only used with repeat elements and signals the transient nature of the group |
| `index` | is only used with repeat elements and reflects the value of the currently selected entry of the repeat (cursor) |
| position | the position of a transient group (repeat-entry) in the nodeset |

*Table 1: Chiba attributes*

All of these attributes use the Chiba namespace

'http://chiba.sourceforge.net/2003/08/xforms'

## 4.2 Authoring patterns

Although discussions about building user interfaces for XForms can reach philosophical dimensions, i'll try to stay pragmatic here and give some usefull hints how to approach this non-trivial task. Don't get this wrong – its non-trivial not because of its complexity but because of the feature-richness and the amount of options to handle it.

Even if i risk to contradict myself there are some facts that have to be kept in mind when authoring XForms container documents:

- XForms is not stand-alone but always embedded in a *container document*
- XForms is layout-agnostic and needs to be styled and layouted by the *host-language*
- XForms is device-independent. When you like this you should keep it in mind when choosing your styling approach (see below)

There are two major styling 'patterns' (in the 'gang of four' sense) which have evolved to approach the styling problem:

1. *direct layout*
   This pattern applies whenever XForms is styled by markup contained in the input document e.g. through placing XForms controls into HTML table tags or . This way the layout/styling is hard-coded in the input. This allows Chiba to use all features of the host-language but limits the device-independence of the form cause not every client will understand this specific language or is limited in its interpretation.
2. *generic layout*
   is signified by a XForms document container that only provides minimal structure to embed the XForms markup. Most notably there will be a header and body section most of the times although this is only a convention. The header will contain the Xforms model while the body embeds the user interface markup. The author will only write down the logical groupings of controls and the sequence in which they appear and avoid mixing host-language markup and XForms markup in the body of the document. By using appearance and CSS class/style attributes the author may give hints which will be used by the UI-Generator to add layout and styling information.

   This pattern is recommended for production environments where device-independence and production efficiency is required and many forms have to be maintained. It allows single-source authoring for multiple clients and is similar to CSS in separating content form styling/layout. Most of the Chiba samples follow this pattern.

   Please note that Chiba uses XHTML container documents as default. When using the generic layout pattern this does not pose any restrictions for the supported clients cause the styling hints can be used to create a complete different output document which is optimized for that client/device. Actually Chiba is completely ignorant about the surrounding markup and does not process it in any way but to pass it on to the UIGenerator.

In real life you sometimes have to break the patterns and of course these mark the 'extreme' cases. In practice you'll sometimes want to mix the approaches and of course you can. As mentioned above, Chiba will preserve all markup contained in the input document and pass it to the UIGenerator. So nothing prevents you from using the generic layout pattern but reverting to direct layout for some fine-tuning.

## 4.3 XSLT stylesheets

When some action has been performed by the Processor (normally triggered by user interaction), the internal DOM will be refreshed and the resulting UI DOM will be fed into the XSLT processor.

By default it will use the standard stylesheet configured in the Chiba config-file to transform into HTML. You may choose to overwrite this setting (see configuring Chiba) or specify a different stylesheet for the current input document only.

If the processor finds a '`chiba:stylesheet`' attribute on the root node of the document, it will use the referenced stylesheet e.g.

```
<mydoc chiba:stylesheet="mystylesheet.xsl" ...>
```

The stylesheet must be located in the stylesheet directory (default: '`web-inf/xslt`') or be referenced relative to this location.

The generation step is responsible for producing the output the client understands. Although Chiba currently only supports the output of HTML, it should be easy to implement a stylesheet for other markup languages too. This is also the place to add custom javascript routines and CSS styling.

There are three options to patch the standard output:

1. You may overwrite some templates in the standard stylesheet to fine-tune your HTML rendering or add scripting. Use `<xsl:import href="html-standard.xsl" />` at the beginning of your stylesheet to import the standard stylesheet.
2. You may replace the standard stylesheet with your own (see configuring Chiba below). If your target client is still HTML, you may choose to keep 'html-form-controls.xsl' which handles the bare form controls. To do that import this file with `xsl:import` as above.
3. You may overwrite the default stylesheet setting for a specific document like described above.

If you have to optimize the user experience by sophisticated styling and reduction of server-turnarounds, here's the place to do it:

Some client-oriented functionalities like 'setfocus' and 'message' cannot be implemented by a server-side implementation but you may still choose to match these elements in the input form and provide script routines that handle these actions on the client-side.

Equally it may be nasty to contact the server for every single operation on the data. Additional scripts for handling repeats and validation could be convenient and sometimes even mission-critical.

## 4.4 Chiba stylesheets

Note: this information applies to the html4.xsl + html-form-controls.xsl stylesheets dated from 26.03.2004

Chiba comes with two standard stylesheets that transform XHTML documents with XForms markup into plain HTML. Both stylesheets are acting as a unit: html-form-controls.xsl transcodes XForms UI controls such as input, select1 or textarea into HTML controls that can be handled by any browser. The templates from this stylesheet are used from html4.xsl to build complete HTML documents that match the XForms input. The following paragraphs explain how these stylesheets are used to add layout to your forms.

As XForms is platform-independent in its nature one should avoid styling forms by adding mixed content to the XForms document cause this ties the document to HTML as output format (well, not strictly but it complicates things significantly when you plan to use them for other clients too). Chiba uses XSLT stylesheets to transcode from native XForms into the target-language of the client. While transforming it assigns certain CSS classes to the UI constructs to allow flexible styling and layout through the use of CSS. This process is referred to as 'CSS annotation'.

Before going on with the CSS explanations is should be mentioned that our experience showed that pure CSS layout can be difficult to achieve especially for complex table layouts and repeated data. html4.xsl uses a mixed approach here; while controls are meant to be styled by CSS, the other constructs group, repeat und switch use a conventional HTML table approach for some appearances. So, it's up to you which way you prefer: a pure HTML output or styling via CSS or a mixture of both. CSS offers you more flexibility in most situations. But it depends ;)

The Chiba stylesheets heavily rely on CSS for enabling layout and sometimes even depend on it to steer e.g. the display/hiding of alert messages.

Chiba tries to mirror the pseudo classes and elements the XForms spec defines. The following table lists these classes and elements.

| Selector defined by XForms | Type | Description | Chiba output after transform |
|---|---|---|---|

| | | | |
|---|---|---|---|
| :required | pseudo-class | selects all required form controls | class="required" |
| :optional | pseudo-class | selects all optional form controls | class="optional" |
| :valid | pseudo-class | selects all valid form controls | class="valid" |
| :invalid | pseudo-class | selects all invalid form controls | class="invalid" |
| :read-only | pseudo-class | selects all read-only controls | class="readonly" |
| :read-write | pseudo-class | selects all writeable controls | class="readwrite" |
| :out-of-range | pseudo-class | selects all elements that are out-of-range | not supported yet |
| :in-range | pseudo-class | selects all elements in-range | not supported yet |
| ::value | pseudo-element | represents the data-entry area of a XForms control, the actual widget excluding the label | class="value" |
| ::repeat-item | pseudo-element | represents a single entry/item in a repeating section | class="repeat-item" |
| ::repeat-index | pseudo-element | represents the currently selected repeat-item | class="repeat-index" |

As these elements are not part of the CSS standard yet today's browser won't interpret these at all. To circumvent this problem Chiba translates these pseudo selectors to 'normal' class selectors that any browser capable of CSS will understand.

An example in pseudo syntax will illustrate this best.

The following XForms input :

```
<xf:input id="my-input" xf:bind="somebind">
  <xf:label>your name</xf:label>
</xf:input>
```

will be transformed into:

```
<span id="my-input" class="input valid readwrite required">
  <span class="label">your name</span>
  <input type="text" id="my-input-value" class="value" />
</span>
```

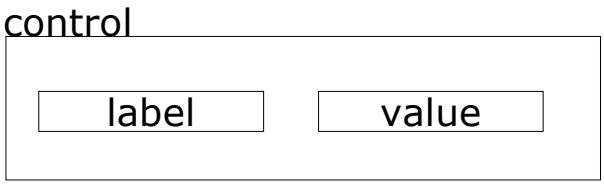
provided that the modelitem-properties the node 'my-bind' points to evaluated to valid, readwrite and required.

In a more general form the output for the class attribute will be (reading '|' as exclusive OR):

class="valid | invalid readonly | readwrite optional | required".

There are several things worth pointing out:

- Note that only XForms elements that have binding expresssions will get these classes. For the purpose of this text 'XForms control' denotes all XForms elements that may be bound to some instance node.
- The class-attribute will always consist of four CSS classes: the local-name of the XForms control, and one class for every modelitem-property.
- controls will have a child class 'value' which represents the control or widget used to edit the data-value
- Notice the id-attributes. These become relevant for individual styling (see below).




Logically every XForms control like input,output, select, select1, textarea,... will be output in a class structure like the above. The control itself will be represented by some HTML substitute like div, span or td. This will be the element that wears the classes for the ModelItem Properties and acts as a placeholder for the original XForms element. Every XForms control will have a label which is output and wrapped with a CSS class named 'label'. Likewise the actual HTML control will get a CSS class 'value' to be styled.

### Styling groups and repeats

For convenience Chiba provides some extra classes for XForms group and repeat elements which match the default appearance attribute values defined by XForms namely 'minimal', 'compact' and 'full'. These are listed here:

| XForms UI | Chiba output<br>after transform |
|---|---|
| <xf:group xf:appearance="minimal"...> | <br> for the group label (if any) |
| <xf:group xf:appearance="compact"...> | <table class="compact-group valid \| invalid readonly \| readwrite optional \| required"> and<br><td class="compact-group-label label"> for the group label (if any) |
| <xf:group xf:appearance="minimal"...> | <table class="full-group valid \| invalid readonly \| readwrite optional \| required"> and<br><td class="full-group-label label"> for the group label (if any) |
| <xf:repeat xf:appearance="minimal"...> | <br> (has additional 'repeat-index' here when selected) |
| <xf:repeat xf:appearance="compact"...> | <table class="compact-repeat valid \| invalid readonly \| readwrite optional \| required"><br><tr class="repeat-item"> (has additional 'repeat-index' here when selected) |

| | |
|---|---|
| <xf:repeat xf:appearance="full"...> | <table class="compact-repeat valid \| invalid readonly \| readwrite optional \| required"><br><tr class="repeat-item"> (has additional 'repeat-index' here when selected) |

To see the effect of these styles in practice run the appearance.xhtml sample form. There are also some additional classes for repeat-selectors but these are not described here. Just look at the generated HTML source to learn more.

### Styling individual controls

The above approach is fine for a generic styling of your forms. It makes re-use of layout- and styling information for multiple forms easy and should always be the preferred way. Nevertheless situations may occur where you need direct access to a certain control or group element for styling purposes.

CSS comes to the rescue again. Chiba will apply the original ids from your input XForms document to the output document following these rules:

- the id of the XForms control will be placed on a wrapper div- or span-element which translates the combined nature of XForms control (control + label in one unit) to HTML.
- the label of the control (if any) will get no id applied cause it will be always possible to style the label by the combination of an ID selector and an Child or Descendant selector e.g.
  #my-input > .label (Child selector)
  #my-input .label (Descendant selector)
- the HTML control resulting from the transform will get the original id with '-value' appended

By carefully assigning ids to your XForms document you lay the ground for fine-grained layout control. Using these ids you now have the handle to write CSS ID Selectors for styling every individual control. Here another example shows how easy it is to write styles with this approach:

Somewhere in your XForms:

```
<xf:input id="title"....>
  <xf:label>Document Title</xf:label>
```

In your CSS file:

```
#title{
  font-weight:bold;
}
#title > .label{
  //put your label styles here
}
#title > .value{
  //put your control styles here
}
```

## 5 Architecture

The following sections are targetted at architects and developers who are interested in using Chiba in their own applications and gives some insight about the inner workings.

The heart of the Chiba project is the Chiba Processor (org.chiba.xml.xforms.ChibaBean) which is the actual implementation of W3C XForms.

Details about the conformance level and currently supported features can be obtained from [1].

## 5.1 Short History

The initial motivation for the Chiba project was to build an environment to share arbitrary documents conforming to different templates or schemas between small groups of people working together in eCommerce projects and avoiding the clutter of different program versions and storage locations. This was back in year 2000 and an initial prototype was built as an web application using (guess?) XML as storage format.

Shortly thereafter I stumbled over XForms as a possible alternative to reach the initial requirements and began to investigate it as a platform. Even back in these days XForms already looked like a key technology in solving problems like the described one but reaching far more than just that. The Chiba project was started on 10 February 2001.

The initial goal at first still was to develop a kind of component to be used in web applications for projects that have some small to medium document-management problems to solve. Structure conformance of documents, consistent storage access and high flexiblility of structures were the main points of interest in these days.

A lot was learned by the great work of the XForms Working Group and the scope of the project quickly broadened and the decision for the implementation of an XForms processor was taken. But against the somtimes client-centered view of the official Spec. back then, Chiba always tried to 'save' the server-centered view and start developing from the least common denominator of net-based applications: deliver browsers with an HTML front-end. Most interesting applications today involve consistent sharing of data in a multi-user environment with possibly multiple different clients so this approach seemed to be reasonably founded and should extend to support more elaborate user interaction through flexible delegation to a client with more capabilities (HTML + Javascript, Flash + Actionscript, Java,...).

The implementation of Chiba closely followed the different versions of the XForms Spec. repeating its errors sometimes but also reaching some stable levels which allowed some early integrations even in commercial products. This refinement process took all of 2002 and most of 2003, still only supporting a subset of the XForms functionalities. In 2003 big effort was put in refactoring the whole codebase to move it even closer to a XFORMS-FULL implementation. Despite the doubts about DOM in server components we fully committed to it and started to implement the more advanced feature of XForms like DOM events, dependency tracking, validations and the like.

From now on Chiba is considered to become independent of its using environment: either client or server or distributed. The Chiba web application serves the purpose of an reference implementation, showing one potential (and hopefully useful) use of Chiba in a networked application. The main focus of development is the Chiba processor while the reference implementation will be extended to become a useful toolkit for building enterprise java applications with multiple client support.

The Chiba project does NOT intend to become the ultimate web application framework or architecture - there are already good frameworks around that meet this need – instead Chiba will provide integrations for several popular frameworks like J2EE, Struts and Cocoon (see Chicoon in Section 'Goodies').

## 5.2 Usage Scenarios

The diagram below shows some different possible scenarios for operating Chiba.

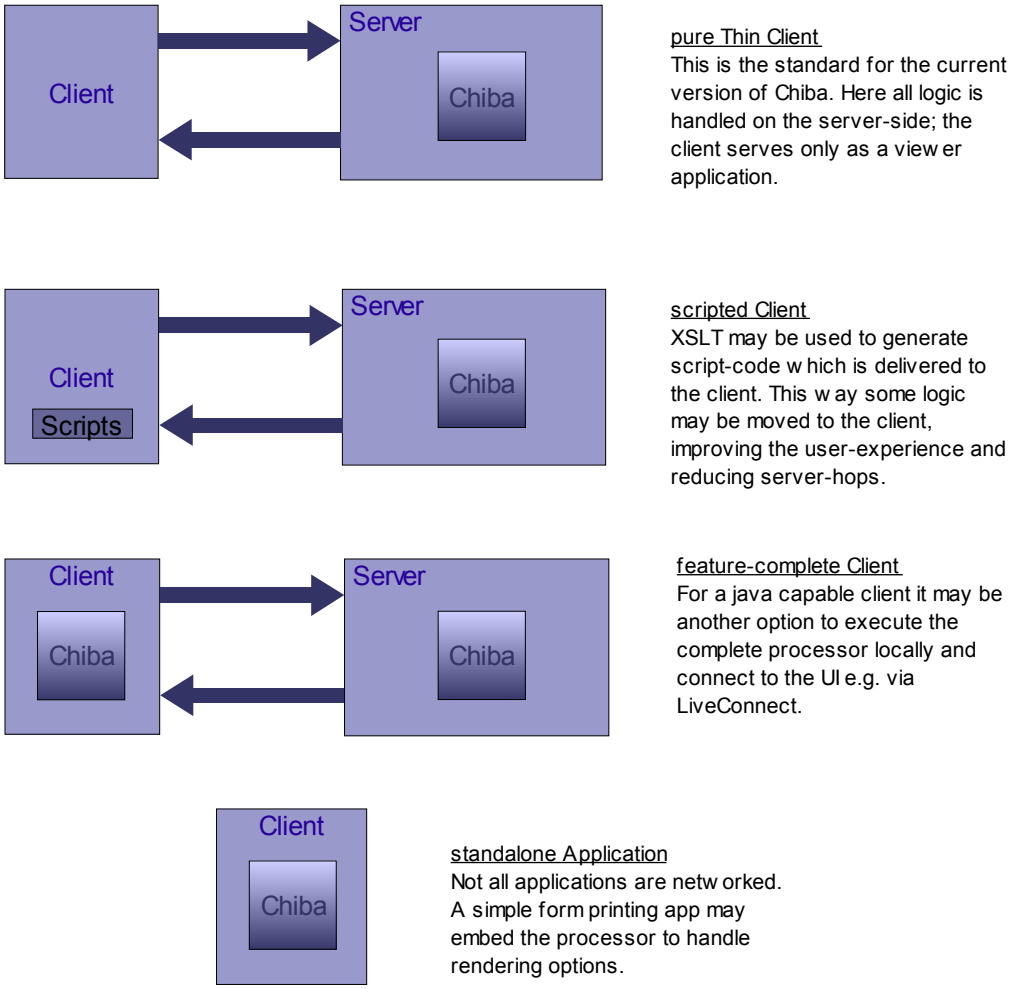



*Illustration 1: Usage Scenarios*

Chiba currently supports the first two scenarios through `html4.xsl` and `dom-browser.xsl` (see Section about Styling) and may be extended to support the others.

Note for 0.9.3: dom-browser.xsl is currently deprecated and will be re-activated in 0.9.4

## 5.3 Applicability

Chiba started as a pure server-side solution to form-processing. The goal was to provide a XForms processor which works even with scriptless browsers[1].

Although only a subset of the XForms functionality has been implemented[2] as of now, this already allows Chiba to use a lot of XForms' powerful features for a wide range of applications. To name a few:

- E-Business
- Inter-, Intra- and Extranet Applications

1 In some real-world scenarios it could always happen that Javascript is blocked by some firewall policy.

2 Client-oriented features like the 'setfocus' Action are not directly supported by the processor but will be preserved and can be implemented through custom client-side scripts.

- Customer Relationship Management Solutions (CRM)
- Content Management Systems (CMS)
- E-Government
- Workflow Applications
- Knowledge Mangement
- Applications that require multi-client support
- As an interface to Webservices

## 5.4 The Chiba webapp

First of all it should be said that the ChibaServlet is not intended to fullfill all requirements for a production-quality webapplication (even if it may be extended in that direction) but to showcase the usage of the Servletadapter, ChibaBean and the UIGenerator. The servlet serves as the glue holding these pieces together providing a demo application for operating the Chiba XForms engine.

Even if we improve ChibaServlet further to make it more useful as a starting point for developing with Chiba, it is recommended to integrate Chiba in production-quality frameworks for serious applications (see short history for details).

The Chiba webapplication is built upon Servlet 2.3 standard and may be executed on any webcontainer conforming to that standard. It has been reported to run on Tomcat (since Ver. 3.3), Jboss, Weblogic and Sun One Server.

The Chiba webapplication is made up of a single servlet which delegates most of the request processing to an adapter component.

The following diagram shows a very high-level overview of the Chiba architecture. The components will be explained in the next sections.

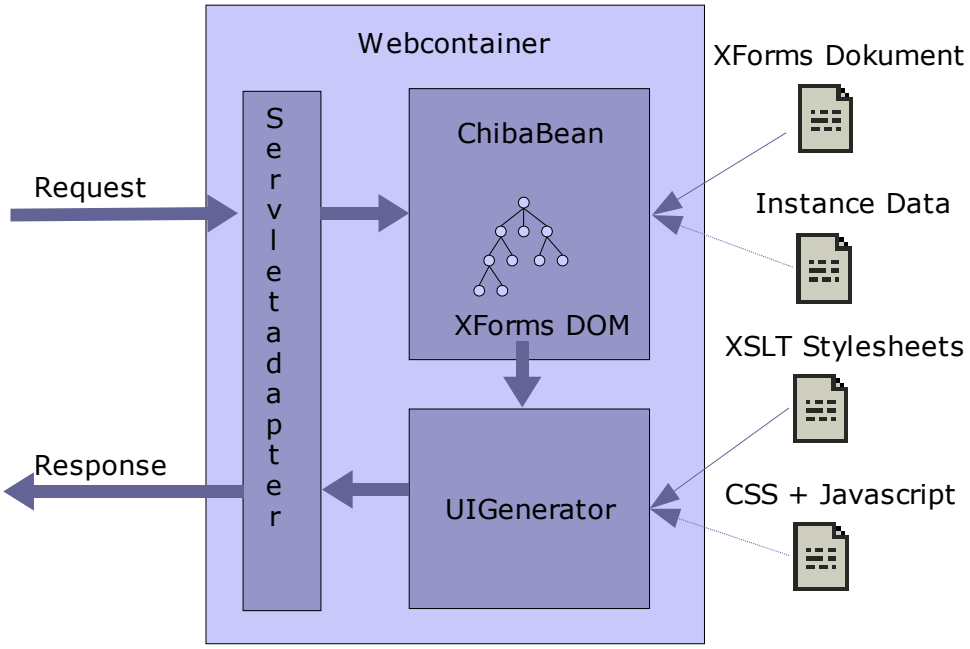



*Illustration 2: high-level view of Chiba architecture*

### 5.4.1 ServletAdapter

The class `org.chiba.adapter.web.ServletAdapter` connects ChibaBean with the servlet-environment. While the servlet is responsible for setting up and triggering Adapter, ChibaBean and UIGenerator the actual request-processing is delegated to the `handleRequest()` method of `ServletAdapter`.

The `ServletAdapter` encapsulates details about request-encoding (url-encoded, multipart), the mapping of http-parameter names to instance-data, the indices of repeats and which XForms trigger to execute.

### 5.4.2 ChibaBean

Class org.chiba.xml.xforms.ChibaBean represents the XForms processor itself and provides API methods to initialize, configure and process forms. In regard to the MVC pattern ChibaBean takes the role of the XForms controller and at the same time acts as a facade to the processors' functionality.

Chiba follows the SoC (separation of concerns) principle and itself is implemented after the MVC pattern.

Once inited, `ChibaBean` holds a reference to a `Container` instance which is the DOM representation of the XForms document and the model in our MVC architecture. All runtime manipulations on the Container are carried out by dispatching DOM events from ChibaBean to the Container through ChibaBeans' `dispatch()` method. It will take the id of the target element and the eventtype as arguments and feeds an appropriate event instance into the internal XForms DOM.

All other methods are used to configure the processor before calling init() or to `shutdown()` the processor.

For details about these methods and their usage please refer to Section 7.

### 5.4.3 UIGenerator

Finally the UIGenerator is the view component of Chiba. It's not coupled to ChibaBean in any way and may well be replaced by other technologies that are adequate for creating user interface markup. Chiba provides one implementation for the UIGenerator interface called XSLTGenerator. It transforms the internal XForms DOM of the processor (aka ChibaBean) into the markup of the target language.

Here's a visualization of the complete MVC model and a summary of what have been said in the preceding paragraphs.

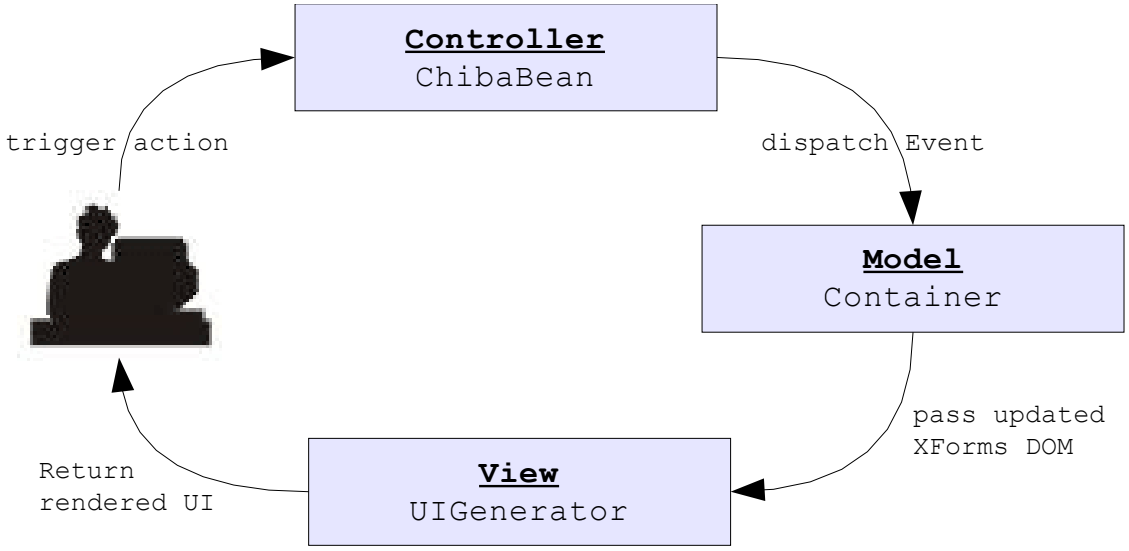



*Illustration 3: Chiba MVC model*

### 5.4.4 Connectors

Connectors call some backend code and return some object to the processor. They will be triggered by the processor at certain events during the life-cycle. Most commonly you'll use UriResolver and SubmissionHandler objects to load data from some URI or to submit data to some target.

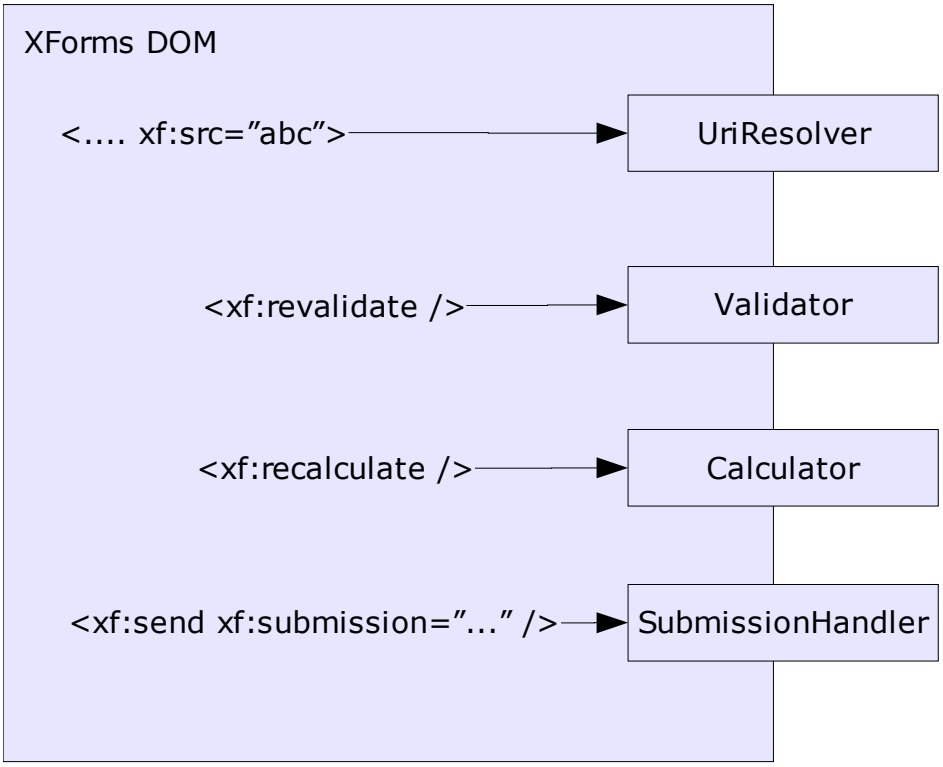



*Illustration 4: different Connector types*

Calculators and Validators are discussed in section 'Extending Chiba'.

#### URIResolver

Loading of external XML normally happens at init-time of the processor. While initializing the `Instance` objects the processor looks for a `xf:src` attribute on element `instance`. If present, an `UriResolver` is instantiated to handle the loading.

For example:

```
<xf:instance xf:src="http://myhost/somedata">...
```

The processor will determine which Resolver by examining the scheme of the `src` attributes' value in this case '`http`'. The `UriResolver` class is looked up in the '`connectors`' section of the config-file. After instantiation of the appropriate driver, the resolve method is called which returns an object to the processor. At the moment only DOM is understood by the processor, so `UriResolver` Implementations should return DOM.

#### SubmissionHandler

SubmissionHandler works the same way as UriResolvers: when the processor hits a submission it's action attribute will be examined and used for the selection of the right driver. This is looked up again from the config-file in section 'connectors'.

Example: the entry for the Http-Driver looks like this:

```
<submission-driver scheme="http"
class="org.chiba.xml.xforms.connector.http.HTTPSubmissionDriver"/>
```

When the `submit()` method is called, the data is validated, serialized and sent to the target denoted by the action URI.

Example: If you have the following submission in your document

<xf:submission xf:action="http://myhost/mytarget" method="post">

your data will be serialized into a http-post (using `HTTPSubmissionDriver`) and sent to 'myhost/mytarget'.

## 5.5 Processing Model and Lifecycle

[todo]

### 5.5.1 Initialization

[todo]

### 5.5.2 Submission

[todo]

### 5.5.3 Destruction

[todo]

## 5.6 Request processing

After a ChibaBean instance has been initialized by a GET request all subsequent requests are POSTs to ChibaServlet. The servlet will delegate the whole processing to the ServletAdapters' handleRequest() method.

The ServletAdapter will distinguish these tree different parameter types by using a prefix which can be configured in the Chiba config-file:

| Type | Description | Config param name | Default prefix |
|---|---|---|---|
| data | maps to a UI control | chiba.web.dataPrefix | d_ |
| trigger | maps to a trigger found in the XForms | chiba.web.triggerPrefix | t_ |
| selector | signals a repeat index position to the processor | chiba.web.selectorPrefix | s_ |

*Table 2: parameter types*

The following diagram shows the phases of request-processing and marks the phases 4-6 as optional. This is because only actions that actually change the state of the instance data need to execute these phases. Simple user interface actions skip these phases and continue directly with a refresh().

Actions that DO trigger these phases are noted below.

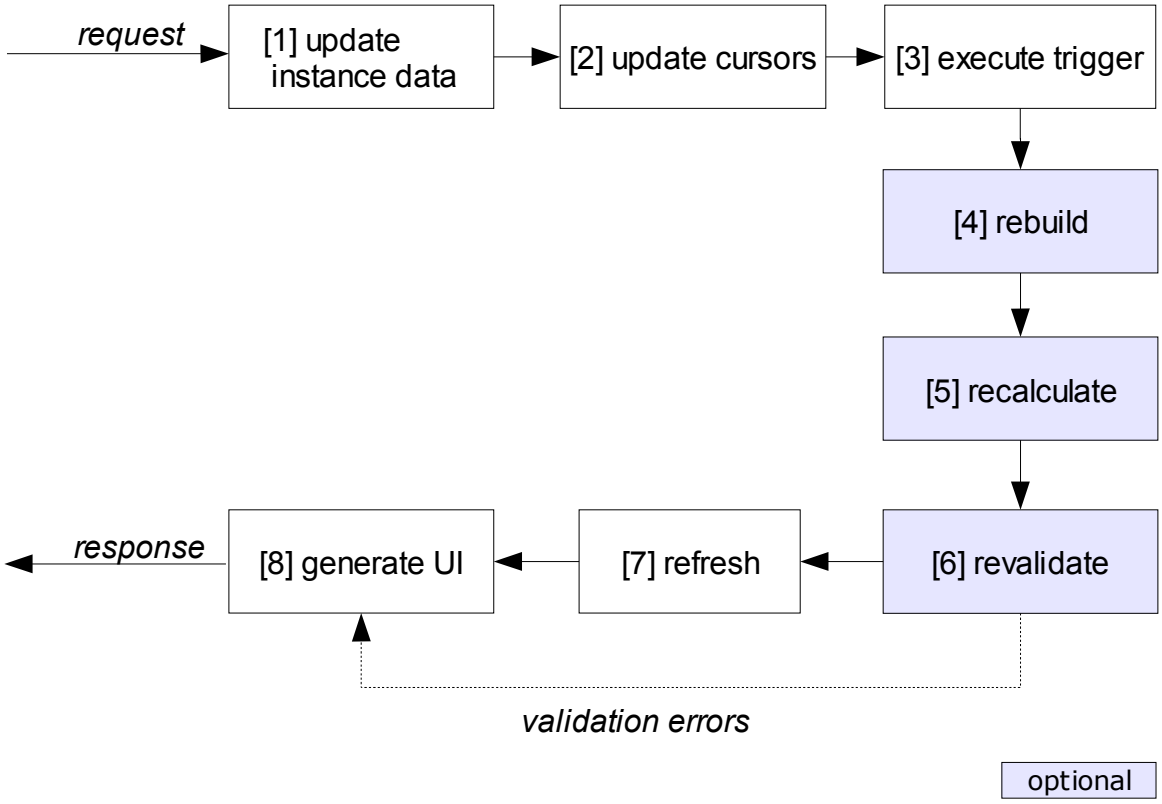



*Illustration 5: Request processing stages*

[1] update instance data

During this step all data parameters are filtered from the request and applied to the instance data. But the request parameters do not contain a direct reference to the instance data node (see explanation below) but to an UI control bound to that node. The ServletAdapter will update the value of the control which will call the setNodeValue() method of the target instance. This way the ServletAdapter simulates the user interaction with controls against a set of instance data.

Because HTTP allows encoding data only in simple name/value pairs and XForms works on XML structures, there needs to be a mapping from simple request-parameters to XPath expressions.

The `WebAdapter` class maps the instance XPaths to ids of UI controls and these are used in HTTP requests and responses as parameter names. A former version of Chiba used the instance xpaths as parameter names directly which is nice for debugging but might be considered a security issue in production scenarios. It exposes your internal datastructures and that's why we change the default here. But its easy to overwrite this default implementation and use whatever you like as parameter names.

[2] update cursors

Each repeat has an index which represents the current position in the nodeset. In a server-side scenario we need a way to tell the server about the users changes to this position. In Chiba we call this position a cursor and after updating the data we need to update these cursors before executing any actions.

During this phase all cursor parameters are filtered out and the corresponding internal repeat objects are updated by calling their setIndex() method.

[3] execute trigger

After data and cursors are up-to-date we can execute the actions the user triggered through interaction with some UI control. The incoming trigger parameter will contain the id of that control as value. After looking up the target element the ChibaBean.dispatch() method is called to execute the action(s) found as child elements of that control.

Chiba also supports 'deffered update behaviour' as described in the XForms recommendation. That means that if there are multiple actions that need to execute a rebuild, revalidate or recalculate these will occur only once for the request.

[4] rebuild

The rebuild phase only occurs if its needed, that is when nodes have been inserted into the instance data or deleted from them. During this phase the dependency graph for a model will be rebuilt.

Only insert and delete actions trigger a rebuild.

[5] revalidate

During this phase all models in the document will be revalidated. All bind elements found in a model are iterated and their model item properties (xpath expressions on bind) will be evaluated.

If an error occurs, the chiba:data element will be updated accordingly. This element is used by the UI generator to trigger the rendering of error messages.

Note that only insert, delete and setvalue trigger a revalidate. All other actions skip this phase.

[6] recalculate

As in phase [5] calculation is done model by model. All calculate attributes found in a model are executed.

Again only insert, delete and setvalue actions trigger a recalculate.

[7] refresh

`XFormsRefresh` works the same way as [5] + [6]. It evaluates the Model Item Properties 'readonly' and 'required' and updates all form controls by setting the correct values for the `chiba:readonly` and `chiba:required`[3] attributes.

In contrast to the preceding phases a refresh happens for each request.

[8] transform UI

In the last phase the UI for the user-agent is generated. ChibaServlet takes control again, gets the unrolled, annotated XForms DOM from ChibaBean and feeds it into the UIGenerator for generating a user interface.

# 6 Using Chiba in your applications

## 6.1 ChibaServlet

ChibaServlet is a rather simple servlet which communicates with a ChibaBean instance through the ServletAdapter. When ChibaServlet receives a GET it will perform the following steps:

1. a ChibaBean instance is created and its base URI is set.
2. a ServletAdapter instance is created and connected to ChibaBean
3. the requested form will be initialized
4. a XSLTGenerator instance is created

3 see section 'custom Chiba attributes' for details

5. the initialized XForms DOM will be fed into the XSLTGenerator which will output its result to the servlet ouputstream

All subsequent requests will be POSTs from a web form and the servlet will execute the actions described in section 'Request processing'.

[todo: get parameter explanation: form, instance,xslt,css]

## 6.2 ChibaBean

ChibaBean is the main class of interest when working with Chiba. It handles configuration, initialization, all XForms interaction and destruction of the container. To instantiate ChibaBean simply call the default constructor:

```
ChibaBean chibaBean = new ChibaBean();
```

### 6.2.1 Loading a form

After creation of ChibaBean you'll want to set a XForms document for processing:

```
chibaBean.setXMLContainer(...);
```

As an alternative you might use a ...

```
chibaBean.setXMLContainer(...);
```

### 6.2.2 Setting the base URI

### 6.2.3 Passing initial instance data

### 6.2.4 Attaching exception listeners

### 6.2.5 Attaching event listeners

### 6.2.6 Initializing the processor

## 6.3 ChibaContext

ChibaContext is a simple class which allows to hold some key/value pairs which have to be passed between Chiba and the using application. It's similar to a ServletContext in holding app-specific parameters and making them accessible via the `getProperty` method.

A '*Context property*' is a key which is stored in an instance of ChibaContext at runtime.

### 6.3.1 Parameterizing Connector URIs

'*Context Property Substitution*' (which is described here) is an extremely simple but flexible way of letting your application speak to Chiba at runtime and pass in some information not present in the form itself but possibly important for the handling of the form (like dynamically determining the source for loading an instance or the target of a submission).

With the help of Context properties URIs can be dynamic by reading parameters from the context. For example in your form you might write something like:

<xf:instance src="http://myhost/path?param1={key1}&param2={key2}"/>

Of course this is not a valid URI and would throw an MalformedURIException if used with a Connector directly. So before the URI is resolved all expressions enclosed in curly brackets are substituted with the values from the Context. If we assume key1 to be 'value1' and key2 to be 'value2' for the above example the resulting URI would be:

<xf:instance src="http://myhost/path?param1=value1&param2=value2"/>

As the Context Property Substitution takes place in `ConnectorFactory` it will be automatically be available in every `Connector` implementation.

## 6.4 Configuration

All configuration parameters are found in a file named 'default.xml' which is located in package `org.chiba.xml.xforms.config` in the classpath. If you don't like this policy you may put the configfile in a different location and reference it from web.xml as follows:

```
<context-param>
      <param-name>chiba-config.xml</param-name>
      <param-value>WEB-INF/chiba.xml</param-value>
      <description>location of config-file</description>
</context-param>
```

ChibaServlet will expect the given location to be relative to the root of the webcontext.

### 6.4.1 default.xml

The configuration file consists of several sections which are grouped by topic. These are:

- <properties>
  this section contains (you guess it) <property> elements which are used to store simple key/value pairs involved in Chiba configuration. For documentation of these properties please see the comment in default.xml.
- <error-messages>
  used to configure the error-messages of the Chiba processor itself. (NOT the XForms messages).
- <stylesheets>
  while its always possible to set the stylesheet at runtime, this section defines the standard stylesheet to be used. For HTML the entry
  `<stylesheet name="html-default" value="html4.xsl"/>`
  defines 'html4.xsl' as the standard stylesheet for HTML clients.
- <connectors>
  in this section the various `Connector` implementations that Chiba uses for URI resolution can be configured. Every `Connector` defines a scheme attribute which defines the URI scheme which identifies a `Connector` and the fully qualified classname of the `Connector` implementation. The `ConnectorFactory` class will use these entries at runtime to instanciate the configured implementation.
- <extension-functions>
  this markup is proposed to configure XPath extension functions that are not part of the XForms standard. This feature has not yet been enabled fully.
- <actions>
  similar to `Connector` this sections defines the handler classes for a specific action. This allows to extend the processor and plug-in custom actions.
  Note: dynamic instanciation of actions has been disabled during a refactoring. This feature will be re-integrated.

# 7 Extending Chiba

Although XForms is extremely flexible and powerful it cannot cover all requirements when it comes to real-world applications. Especially problematic is business-logic that deals with validation or calculation and isn't easily expressed in XForms constraints or XML Schema.

Another area of interest is connectivity. Instance Data may come from a database, the filesystem, a remote location or are dynamically generated. The same is true for the submission. Different requirements may arise for:

- protocol to use (http get/post/put, file, smtp, ldap, jdbc, jndi, soap,...)
- serialization format (XML, urlencoded, multipart,...)

The extension interfaces described in the following sections allow to address all these issues.

## 7.1 Connectors

[tbd]

## 7.2 Calculators

[explain the extension functions for calculation]

## 7.3 Validators

[explain the extension functions for validation]

## 7.4 Actions

XForms already defines the most common set of actions that might occur in form-processing so you'll rarely need to extend the standard set. However there might be situations where this is not sufficient: e.g. if you need some kind of conditional processing in a sequence of actions you can't express this with the generic actionset any more.

Here custom actions come to the rescue. Write your own action by implementing the `Action` interface or easier by extending `AbstractAction` which already implements the most of the methods. This way you'll only need to implement the logic the `perform()` method. This method returns a `boolean` which signals if the execution was successful (in a application-specific interpretation).

Of course, a custom action may fire other actions as well, so they can be used as a kind of macro.

If you've implemented a custom action, you must declare it in the config-file. Browse to the section 'actions' and create a new entry like this:

<action name="[tagname]" class="[fully qualified java class name]“ />

Replace [tagname] with the desired name you want to use in your documents as tagname and [fully qualified java class name] which the name of your javaclass.

NOTE: always make sure that your actions in XForms documents have a correct xf:id attribute. The processor identifies actions through this id and if it is not present, the action will never be fired.

# 8 Goodies

## 8.1 Chicoon

[installation, description of use]

## 8.2 Schema2XForms Builder

The Schema2XForms Builder is a command-line tool to generate complete XForms documents from an XML Schema. If you already have a Schema for the desired form-document you save a lot of tedious coding and get workable forms instantly.

Schema2XForms Builder is implemented as an Ant task that can be called like all other targets in `build.xml`. The following copies the description found in the header of `build.xml`.

> this target generates an XForm from an W3C XMLSchema document.
> A number of parameters need to be specified in order for it to work
> (parameters are passed with -DparameterName=parameterValue):
>
> - schema2XForms.xform: name of the xform to generate: there must be a
> directory with this name under the "xforms" directory, and the schema
> in this directory must be called with this name followed by .xsd
>
> - schema2XForms.rootTagName: the "root" element name of the XML instance
> corresponding to this schema
> schema2XForms.instanceFile and schema2XForms.instanceHref: the instance
> XML document can be specified either as a file, in which case it will be
> included in the generated XForms, or as a URI, in which case an "href" link
> will be set on the "instance" element of the generated XForms
>
> - schema2XForms.action: this parameter will be set in the "action"
> attribute of the "submission" element in the generated XForms document
>
> - schema2XForms.submitMethod: this parameter will be set in the "method"
> attribute of the "submission" element in the generated XForms document
>
> - schema2XForms.wrapperType: specifies the kind of wrapper elements to be
> generated in the XForms document.
> Default will generate default, platform independant elements, while
> "HTML" wrapperType will generate an XHTML document

As an example to illustrate how Schema2XForms works, we will convert the purchaseOrder.xsd schema to XForms. You will find purchaseOrder.xsd in src\org\chiba\tools\schemabuilder\test under the root directory of chiba. First, Schema2XForms will need two directories, the source directory where the schema resides and the target directory where the form will be placed. These two directories are hardcoded to xforms/<nameofschema> and generated/<nameofschema>, respectively.

We therefore need to create the directories xforms/purchaseOrder and generated/purchaseOrder before converting the schema. After copying purchaseOrder.xsd in xforms/purchaseOrder we need to create an instance document that, at least, contains the required elements.

Here is a simple instance document that is named purchaseOrder-instance.xml:

```
<purchaseOrder xmlns="purchaseOrder"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="purchaseOrder purchaseOrder.xsd"
       customerId="">
       <headerInformation>
              <streetAddress/>
              <state_or_province/>
              <name/>
              <rushDelivery>0</rushDelivery>
       </headerInformation>
       <phoneNumber customerType="">
       <areaCode/>
       <number></number>
```

```
      <longEnum/>
      <shortEnum/>
      <longList/>
      <shortList/>
      </phoneNumber>
</purchaseOrder>
```

Note that the instance need not be valid, however all your elements must be included in the instance. If you add some information (like rushDelivery) then it will be displayed as a default value for this field in the resulting HTML form. In this case, rushDelivery will be false (0) by default when the form is rendered.

The next step is to actually convert the schema combined with the instance into a fully functional form. This can be accomplished with this very simple command line:

```
ant schema2XForms \
      -Dschema2XForms.xform=purchaseOrder \
      -Dschema2XForms.rootTagName=purchaseOrder \
      -Dschema2XForms.wrapperType="HTML" \
      -Dschema2XForms.stylesheet="html-standard.xsl" \
      -Dschema2XForms.instanceFile=xforms/purchaseOrder/purchaseOrder-instance.xml
```

This will generate purchaseOrder.xml in generated/purchaseOrder. All that is needed now is to copy that into the build/forms directory and execute `ant deploy`. You will now see your form in the main test page of chiba, next to all the other demos.

# 9 The future

Further 0.9.x versions will:

- add the missing controls and improve client-side interaction through generated scripts
- cover the complete xforms markup
- add schema validation
- conformance testing with the official XForms Test Suite
- tune the footprint and performance
- prepare 1.0

As a consequence of the transition to DOM Events the Chiba Processor wil become an independant component following the common JavaBean patterns and can be plugged in any Java program. The using application may register Eventlisteners for every XForms event happening during form-processing and react in a application specific way. An Adapter acts like a bridge between application and processor. The current RequestController will be substituted by a 'WebAdapter' which translates requests into events and back into responses. While the processor will support the full XForms featureset, the adapter may choose to handle only the subset corresponding to the client's capabilities.

# 10 Appendices

## 10.1 Building Chiba

If you want to extend Chiba or integrate it in your own environment, you'll want to build Chiba from source. Here's how it's done.

You'll need a working installation of Jakarta Ant 1.5 or higher to run the Chiba build scripts. Its also assumed that you're familiar with the basic usage of Ant.

> **Note**: It is assumed that a `tomcat_home` variable is present in your systems environment and points to the Tomcat installation. For other webcontainers you

have to overwrite this setting by editing build.xml or setting the property '`webapps.dir`' via the Ant commandline (see Ant documentation).

These are the main targets:

- '`compile`' – as it says compiles the Chiba sources to a directory named '_build' under the installation root. Also copies all supporting files (xml, xsl) to the build directory.
- '`deploy`' – deploys Chiba into your web container. This is usefull if you want to develop with Chiba and add extensions to the core. Creates the complete structure needed for a web-application and copies all files to their destinations.
- '`distribute`' – packs Chiba into source- and binary distributions.

Further targets are available and documented in the build.xml file itself.

## 10.2 Directory structure

Let's take a look at the directory structure of a Chiba installation. You should find this structure as a subtree of your webcontainer's home directory after you've installed Chiba and started the webcontainer. The Container will expand the Chiba war-file which results in the following layout:

| | |
|---|---|
| **webapps** | **your webcontainers webapps-directory** |
| **chiba-[version]** | **Chiba home directory** |
| **forms** | **sample forms – here are the files browsed with forms.jsp** |
| **images** | **images used by the sample forms** |
| **jsp** | **some jsp pages for browsing forms and showing debug output** |
| **scripts** | **Javascript files used by the XSLT stylesheets** |
| **styles** | **CSS stylesheets used by the XSLT transformations** |
| **WEB-INF** | **Webcontainers WEB-INF directory** |
| **classes** | **Chiba class files** |
| **lib** | **all libraries used by Chiba** |
| **xslt** | **Chiba XSLT transformations** |

Most interesting for the following explanations is the forms directory. Put all form documents you're working on into this directory. When calling the Url http://localhost:8080/chiba-[version]/jsp/forms.jsp the contents of this directory will be listed and you can execute them.

All other directories should be pretty self-explanatory.

## 10.3 Links

- [1] Features page of Chiba project – http://chiba.sourceforge.net/features.html
- The official homepage of the W3C – http://www.w3c.org/markup/forms
- Chiba homepage – http://chiba.sourceforge.net
- Chiba project page at sourceforge – http://www.sourceforge.net/projects/chiba
- Chiba Download – http://sourceforge.net/project/showfiles.php?group_id=20274
- Status of Implementation - http://chiba.sourceforge.net/features.html
- interesting book about XForms from Micah Dubinko - http://dubinko.info/writing/xforms/book.html
- Links to other XForms implementations – http://www.w3c.org/markup/forms in section 'XForms Implementations'

- A short introduction to XForms from Micah Dubinko (Author of the above book and member of the working group) - http://www.xml.com/pub/a/2001/09/05/xforms.html
- Professional XForms + Chiba support – http://www.chibacon.de

## 10.4 Glossary

*document container*

a document in any XML compatible markup language like (XHTML, SVG, VoiceML, FlashML) that embeds XForms markup

*host language*

a XML markup language that embeds XForms markup.