Innowhere

ItsNat

Reference Manual

v0.3

Doc. version 1.0

June 9, 2008

Jose María Arranz Santamaría

TABLE OF CONTENTS

LEGAL

1. INTRODUCTION

1.1 THE AJAX ERA: REVOLUTION AND NIGTHMARE

We are living in the AJAX (Asynchronous JavaScript And XML) era, there is no dude, the AJAX approach is revitalizing the old DHTML stuff, DHTML is not new of course but today is a real option, the times of Navigator 4.x and Internet Explorer 4.x are definitively gone. Fortunately; the current Mozilla/Firefox is the W3C dream, Safari 3 and Opera 9 are highly W3C compliant, Internet Explorer 6 is not doing very bad in the web standard space (of course v7 is better) and some mobile browsers have the same capabilities as desktop counterparts. DHTML and the `XMLHttpRequest` approach to communicate with the server, have introduced the web programming world in the AJAX era, the core technology of Web 2.0, achieving the web RIA (Rich Internet Applications) utopia.

But any “new” paradigm ever drives to “real” FUD: Fear Uncertainty Doubt.

Too many new AJAX frameworks are out there, ajaxpatterns.org lists[1] 50 Java AJAX frameworks at the time of writing!

When a senior Java developer tries to enter in the new AJAX world, fear, uncertainty and doubt are perhaps the first feelings. Of course this document is not done to propagate FUD against AJAX, AJAX has come to stay. ItsNat main objective is to combat the reasonably FUD about AJAX: “AJAX without FUD”.

1.2 WHY ANOTHER (AJAX) FRAMEWORK? CURRENT SCENARIO

This is the question (approximately) that Jonathan Locke, the main author of the famous Wicket framework, said to the world[2] some time ago, Jonathan, a “hard core” Swing contributor, was shocked with the Java web frameworks out there, the wheel was not developed as a circle (we may call this *The Jonathan Locke Syndrome*). ItsNat is the result of similar feelings, ok we already have Wicket, Wicket is nice … but there is another way to build the Java based web… another Java web is possible … another wheel more circular.

Most of the current AJAX Java web frameworks share these characteristics (all characteristics are not applied to every framework of course):

1.2.1 JavaScript centric

The revitalized DHTML technology invites to create new cutting-edge JavaScript libraries, plenty of cool and sophisticated widgets.

This approach has problems like: JavaScript is a weak language, hard to code, test, manage and debug, error prone (no errors are detected in a compilation phase), with a poor and strange object orientation, very slow, no refactoring tools, too many browser differences, too much code sent to the client, defensive programming on the server (no confidence in the client

[1] http://ajaxpatterns.org/Java_Ajax_Frameworks

[2] http://wicket.sourceforge.net/Introduction.html

side), security issues (the user controls the browser)… In the worst case the JavaScript library generates the HTML (HTML code goes out of developer control and design control).

Many hard core Java web programmers find the JavaScript centric programming a risk and a nightmare.

1.2.2 Server centric fully based in custom tags with JavaScript/HTML generation

A typical server centric approach is to generate the JavaScript code to perform the DHTML behavior, but generating the HTML and JavaScript code too usually using custom tags. JavaScript and HTML are fully managed and controlled by the framework and very hard to customize. This extreme is usually seen in frameworks trying to mimic a desktop application, they are so sophisticated that they do not seem web applications (this may be a compliment and a criticism too).

Many frameworks are competing in this area, competing to offer "filthy rich GUIs"[3] in the web space:

- The winner => the user experience, "wow is cool".
- The loser => the developer, abandoning almost absolutely the control to the framework.
- Unsolved problem => customization. Most of these amazing widgets/GUIs are very hard to customize beyond the predefined layout. If layout and behavior match your needs ok, if not you have a serious and perhaps unsurpassed problem.
- The definitive loser => *the customer* and his "Oh cool, but please change this thing as…", the developer answer "I'm sorry I can't do it my framework doesn't do that". Cool != does the job.

A desktop application appearance can be fully customized because is pixel based but usually is a hard work, this is not necessary because a typical CRUD desktop application has not very much "special effects".

The web space is fully different, web developers love to fully control the layout and appearance of their web pages; fully control of the HTML code beyond the color change, and fortunately HTML language (and CSS) is not very hard and we have visual tools. Furthermore, advanced web developers like to add some cool visual effects with JavaScript (and CSS). If the HTML/JavaScript pair is not controlled by the developer is very hard to do this stuff.

Conclusion: a `<tree>` tag may be a dream and a nightmare: too intrusive, too legacy, too closed, too vendor lock in.

1.2.3 View code and view control code tightly coupled

There are tons of template processors including of course JSP, they all share the same idea: special instructions mixed with HTML code: `<c:if…>` `<%if…>` `#if` … different flavours but they are basically the same.

What is the problem? Some kind of view control is unavoidable, isn't it? Yes of course, but there is no reuse, if the view changes, the view logic must be repeated:

Example (JSTL):

```
    <c:forEach var="item" items="${sessionScope.cart.items}">
```

[3] Variant of the "Filthy Rich Clients" a famous book of Romain Guy and Chet Haase devoted to Swing.

```
        <td align="right" bgcolor="#ffffff">
            ${item.quantity}
        </td>
    </c:forEach>

    ...

    <c:forEach var="item" items="${sessionScope.cart.items}">
      <p>
        <b>Quantity:</b> ${item.quantity}
      </p>
    </c:forEach>
```

What is unnecessarily repeated? Answer: the `<c:forEach var="item"…>` statement.

With this approach there is no much "generic view programming", a clean separation between view code and algorithms applied to the view. Something like this:

```
    <c:pattern name="tableCellQuantity">
        <td align="right" bgcolor="#ffffff">
            ${item.quantity}
        </td>
    </c:pattern>

    <c:pattern name="paragQuantity">
      <p>
        <b>Quantity:</b> ${item.quantity}
      </p>
    </c:pattern>

    ...
    <c:forEach var="item" items="${sessionScope.cart.items}"
        appliedTo="tableCellQuantity" />
    ...
    <c:forEach var="item" items="${sessionScope.cart.items}"
        appliedTo="paragQuantity"/>
```

ItsNat goes far beyond this idea using Java for the view logic and no Java bindings in markup (pure HTML).

1.2.4 Too much XML and XML-Java bindings, too much declarative programming

XML metaprogramming is a modern trend and perhaps overused, may be valid with page based web sites because the navigation is relatively simple, but in the AJAX world there is no classic (page based) navigation inside the same page[4]. XML metaprogramming is used too to bind view component events to Java handlers and many other types of bindings and commands. This approach is not very productive, makes hard any reuse and is difficult to maintain (no refactoring, no compile time checkings). In fact most of typical web frameworks are banishing developers of Object Oriented Programming, because most of the web logic is being coded in XML files and in the form of framework tags and expression languages.

XML based declarative programming is a nice feature for frameworks oriented to tools because Java code has not a fixed structure, but declarative programming is not oriented to developers because is verbose, does not provide reuse (no OOP, too many duplication) and easy "batch configuration" (to apply the same configuration pattern to many elements, only the different element needs a specific configuration).

[4] Eelco Hillenius, from Wicket Team, explains very well what is wrong with XML driven navigation at http://chillenious.wordpress.com/2006/07/16/on-page-navigation/

ItsNat is not oriented to tools like other web frameworks like JSF or Struts, you only need a decent Java editor and optionally a visual HTML editor.

1.2.5 No API reuse, all is new again and again

Usually Java web frameworks reinvent all again and again, developers must learn a new API, and there is no very much reuse between a desktop and a web application sharing the same data model. Some Java web frameworks mimics the Swing API, this may work with quick ports of Swing applications, but most of the Swing API is tightly bound to the desktop programming style, based in pixels. But the web is different, the pixel based approach is wrong or unnecessarily forced and the HTML layout control is completely lost to simulate desktop components.

But in the Swing API there are classes and interfaces not bound to the desktop view/screen: data and selection models and related listeners… almost no Java web framework reuses this API except Swing mimic web frameworks (there is some exception like Wicket trees). May be these Swing models are not perfect, but they are a mature standard and improves the integration with the desktop. ItsNat uses Swing data and selection models to build the typical components.

1.2.6 AJAX introduced artificially

Most of well established component based frameworks were designed in the pre-AJAX era; they have been leveraged to AJAX artificially with hacks and as an extension.

1.3 ITSNAT: RETURNING TO THE ROOTS IN THE AJAX ERA

ItsNat is an AJAX, Component Based, Java Web Application Framework. No news.

One phrase summarizes the ItsNat approach: "**The Browser Is The Server**" (**TBITS**). The Java server code deals with the browser as if the browser object model was on the server memory, like a "W3C Java browser".

The principle behind ItsNat is very simple, the HTML DOM tree is replicated at the server side, exactly is the opposite, the pure W3C HTML DOM tree in server side is replicated in the client browser.

This technique is not fully new, old frameworks like Cocoon and Barracuda used DOM in some way before with no very much success… but we live in the AJAX time, everything has changed.

The incremental page modification fits perfectly well with the "DOM in the server" technique, if the server DOM changes the change is propagated to the client.

In summary: **ItsNat simulates a Universal W3C Java Browser in the server**. ItsNat can be seen as a FireFox or WebKit in the server using the client browser as the GUI.

1.3.1 AJAX from scratch

AJAX is changing how the web is developed. Classic frameworks are designed around the page navigation. ItsNat is designed with AJAX *from scratch,* furthermore it does not have very much page navigation facilities because they are going to be outdated or over engineered[5]. An AJAX

[5] One interesting feature provided by ItsNat for classic navigation is "referrerer" based in AJAX.

centric web may be like a desktop application, typical desktop applications have only one frame; at the same way your web application may have only one page! The "Feature Showcase" included on ItsNat distribution is an example of this approach.

1.3.2 Pure W3C DOM

ItsNat maintains a Java W3C DOM document in the server matching the browser DOM document, any change performed on the server DOM is propagated to the client as JavaScript DOM instructions thanks to W3C DOM Mutation Events. JavaScript generated by the framework is adapted to the usual inconsistencies between the specific browser and the W3C standard. If the DOM modification is a response of an AJAX event the generated JavaScript code is small if the DOM modification is small.

Client DOM and server will be in sync driven this synchronization by the server. This synchronization is mandatory for the "dynamic parts" (DOM subtrees willing to be changed on the server), static subtrees (in a server point of view) may change in the client with no problem allowing any kind of DHTML effects (including client DOM modifications) used only in the client. These static and dynamic HTML zones need to be declared.

No strange programming artefacts, only pure Java W3C DOM!

1.3.3 The view: no JSP, no XML, no custom tags, only pure HTML with no logic

Of course an ItsNat generated web page is NOT fully coded with Java DOM. Any page starts with a pure normal HTML page, the template. This page has no custom tags (only a few custom optional attributes and two custom optional tags, comment and include, processed and removed in the server). This page works as a template, where some parts are static and other parts will be changed by the developer using the Java DOM API in the server. Of course this pure HTML file can be designed with your favorite HTML WYSIWYG editor.

When a user invokes an ItsNat page, the framework loads a template and is converted to DOM, and then the developer has the opportunity to adapt the original page to the desired HTML output using Java W3C APIs. The final DOM tree is serialized and sent to the browser as a normal HTML/XHTML page in this load phase.

In summary ItsNat templates contain pure view information, this is a radical (extreme) separation of view and logic (view logic), because developers change the initial view using DOM in Java and this Java code is absolutely separated from the view. These changes are "pushed" to the view with the desired order instead of the typical "pull" and top/down execution model of most of web based frameworks, because templates are "executable" files and work like "imperative programming languages" (JSP as the prominent example)[6].

1.3.4 Event system: pure W3C DOM Events & AJAX

User events are sent to the server using AJAX and received as W3C DOM Events[7].

The developer can bind an `org.w3c.dom.events.EventListener` Java object to any server Java DOM element calling a special ItsNat method very similar to `org.w3c.dom.events.EventTarget.addEventListener()`. This listener is registered as a *remote listener*, ready to receive events from the browser. For instance: if an event listener

[6] The "MVC push template" concept and technique was popularized by Terrence Parr, the author of StringTemplate. http://www.cs.usfca.edu/~parrt/papers/mvc.templates.pdf

[7] http://www.w3.org/TR/DOM-Level-2-Events/

is registered as a mouse "click" listener of a specific Java DOM element, when the symmetric element in the browser is clicked, the browser event is sent to the server as a W3C DOM Event object, with the `currentTarget` property set as the Java DOM element "listened". The approach is similar to W3C Rex specification[8] but no XML is used (a big performance penalty) and using a custom XPath technique. Again the browser events are perceived as "in the server" like a real Java browser.

ItsNat uses an internal and custom XPath (is not a real XPath implementation) to bind browser and server DOM elements. To increase performance (path resolution takes time), two automatic DOM element registries (caches), in the server and in the client, are used behind the scenes where any explicitly used element has a unique (internally generated) id.

1.3.5 Multiple remote views!

If the original DOM tree is on the server, the browser DOM is basically a "clone" of the server tree and if browser events are processed on the server and DOM modifications are sent to the client again… nothing prevents having more than one browser window representing the same server DOM document/page!

In ItsNat a new browser window can be bound to an existing DOM document (page) working as a remote viewer of the page of another window/browser/user; the new window loads the current DOM server state.

Of course ItsNat provides a security system to avoid of deny this amazing kind of "spyware" (remote view is not active by default and authorization is granted per "remote view" request). Current ItsNat implementation does not permit to send events from a "cloned" view (only the originator window can as a normal web application), two or more windows can not interact with the same server DOM tree.

Furthermore ItsNat offers some kind of remote control: remote viewer refresh is controlled by an optional server listener; this Java listener can do anything including to change the server DOM tree.

Applications… remote supervision, help desk, "legal" spying, monitoring suspicious behavior, teaching, web based distributed shows etc.

1.3.6 No XML, no declarative programming, only pure Java and Java IoC

ItsNat configuration is fully based in Java, of course some configuration parameters can be put in a user defined XML, or using Spring etc. In fact old Java `.properties` configuration files are used in ItsNat examples to bind the physical HTML template file paths with logical names used by ItsNat, of course this technique is an example and is optional, ItsNat does not mandate to use any configuration file.

ItsNat uses the IoC (Inversion Of Control) paradigm frequently with the old fashion style: programmatic registering of Java object listeners, the framework calls these listeners when appropriate (when loading a page, when receiving an client event, when a component is created…). In fact there is no logging system, is user election when log and how log, the framework listeners are the join points with the well defined lifecycles of the framework.

1.3.7 Custom JavaScript coded in the server

[8] http://www.w3.org/TR/rex/

When user code changes the DOM tree, this change is automatically converted to JavaScript and sent to the client at the end of the event cycle (and in the load cycle if the page load mode is set as "slow"). Any developer can add custom JavaScript code in server code in any time of the server cycle to be sent to client, for instance, to call a DHTML JavaScript method, to submit a form programmatically etc.

1.3.8 "Smart" DOM memory caching to save memory

The DOM approach is memory intensive, this is the main complaint about this technique (memory is cheaper and bigger than before but this complaint is valid anyway).

ItsNat uses an automatic customizable DOM caching technique: ItsNat detects if a DOM subtree is not going to change, then the subtree is serialized as text and removed from the tree and registered the text in a cache registry; a text mark is leaved in the tree to remind the removed/cached subtree position. When this part is going to be sent to the client the mark is replaced with the cached text (on load time with the serialized document or into an `innerHTML` string). Cached subtrees saved as text are saved once per HTML template, all user pages share the same cached serialized subtrees. Of course a developer can declare a markup zone as no cacheable if ItsNat thinks it is static and is not with a special attribute.

ItsNat has a feature called HTML/XML fragments; a fragment is piece of markup to be included in a page/document, a fragment is registered and managed much like a normal page. ItsNat automatically caches fragment subtrees, if a fragment is inserted in a document the cached subtrees are shared (fragment subtrees cached are in memory as text once).

Conclusion: big (static) parts of HTML pages can be cached in memory as text in a per template basis, the memory consumed is not different to any other template based framework including JSP (HTML template code is in memory as string literals).

1.3.9 Test the view in the server!

As the DOM tree is in the server you can test your markup in the server with Java code.

ItsNat can fire W3C DOM Events and send them to the browser simulating user actions, this technique is called "server-sent events"[9]. These user actions usually are processed by the server again, the test code can check whether the desired behavior is accomplished checking the server DOM tree (testing the view in the server) or new/removed/updated data.

Furthermore W3C DOM Events can be sent optionally to the server DOM directly with no browser interaction!

1.3.10 Pattern based view manipulation using DOM utilities

A joke is a good way to explain this feature; a psychologist interviews us with a "HTML Rorschach's test":

Q) What do you see in this HTML fragment?

```
<div>
    <p><b>Tiger</b></p>
    <p><b>Lion</b></p>
    <p><b>Giraffe</b></p>
</div>
```

[9] This technique is being standardized in W3C HTML 5, W3C version needs a HTML 5 compliant browser, ItsNat approach works in any supported browser.

A) A list of animals

Q) Good. How would you add a new animal like "Zebra"

A) Very easy:

```
        …
        <p><b>Zebra</b></p>
    </div>
```

Q) Fine. What do you see in this fragment?

```
    <select>
        <option>Tiger</option>
        <option>Lion</option>
        <option>Giraffe</option>
    </select>
```

A) Again the same list of animals, using a standard `<select>` based list.

Q) OK. And how to add "Zebra"?

A) Obvious:

```
        …
        <option>Zebra</option>
    </select>
```

Q) What is shared?

A) Both are two lists of animals (data model), both are two ways to show the same list, both share the same structure or pattern:

```
    <parentList>
        <listItem><optElem1>…<optElemN>Item Content
                  </optElemN>…</optElem1></listItem>
        ...
    </parentList>
```

Q) What are the differences?

A) Tag names are different, one list uses `<b>` as a decorator or you can see `<p><b>` as the elements used as the item pattern, in `<select>` only the `<option>` element is the pattern.

Q) Construct the same list with the following structure/pattern:

```
    <ul>
        <li><b><i>Content</i></b></li>
    </ul>
```

A) Simple

```
    <ul>
        <li><b><i>Tiger</i></b></li>
        <li><b><i>Lion</i></b></li>
        <li><b><i>Giraffe</i></b></li>
        <li><b><i>Zebra</i></b></li>
    </ul>
```

Q) More difficult, what is this?

```
    <div>
        <span>Root</span>
        <ul>
            <li>
                <span>Child 1</span>
                <ul>
                    <li>
                        <span>Child 1.1</span>
                        <ul/>
                    </li>
                </ul>
            </li>
            <li>
                <span>Child 2</span>
                <ul />
            </li>
        </ul>
    </div>
```

A) Course: a tree

Q) Add the "Child 1.2" new node

A) A child's game:

```
               …
                <ul>
                    <li>
                        <span>Child 1.1</span>
                        <ul/>
                    </li>
                    <li>
                        <span>Child 1.2</span>
                        <ul/>
                    </li>
                </ul>
               …
```

Q) More difficult again, what is this?

```
    <div>
        <span>Root</span>
        <table style="margin-left:10px">
            <tbody>
                <tr>
                    <td>
                        <span>Child 1</span>
                        <table style="margin-left:10px">
                            <tbody>
                                <tr>
                                    <td>
                                        <span>Child 1.1</span>
                                        <table style="margin-left:10px">
                                              <tbody/>
                                        </table>
                                    </td>
                                </tr>
                            </tbody>
                        </table>
                    </td>
                </tr>
                <tr>
                    <td>
                        <span>Child 2</span>
                        <table style="margin-left:10px"><tbody/><table>
                    </td>
                </tr>
            </tbody>
        </table>
    </div>
```

A) The same tree (same content) with a different layout. I'm boring.

Q) Finally add "Child 1.2"

A) Here it is. I'm sleeping, something more?

```
...
 <tr>
     <td>
         <span>Child 1.1</span>
         <table style="margin-left:10px">
               <tbody/>
         </table>
     </td>
 </tr>
 <tr>
     <td>
         <span>Child 1.2</span>
         <table style="margin-left:10px">
               <tbody/>
         </table>
     </td>
 </tr>
...
```

Q) No. You showed me you know to distinguish that the concrete view is orthogonal to the concrete data model and several views can share the same structure and can be manipulated using the same pattern based technique.

This is how ItsNat utilities work; ItsNat has pattern based element lists, tables and trees, these utilities (classes) are agnostic with the concrete DOM elements declared in the markup using the pattern based technique (a list must contain an item as pattern, a table a cell, a tree a tree node).

For instance:

```
    ItsNatDocument itsNatDoc = ...;
    Document doc = itsNatDoc.getDocument();
    Element parentElem = doc.getElementById("listParentId");
    ElementGroupManager factory = itsNatDoc.getElementGroupManager();
    ElementList elemList = factory.createElementList(parentElem,true);
    elemList.addElement("Tiger");
    elemList.addElement("Lion");
    ...
```

Where `listParentId` is the `id` attribute of the parent element of the list, any previous list is valid (`<div>`, `<select>`, `<ul>`…). The `ElementList` implementation uses a default renderer, this renderer writes the string into the deepest element of the item list pattern (below `<b>` and `<i>` if both exists).

1.3.11 Layering versus replacing. Minimalist API

ItsNat is constructed over the old Servlet Specification, over the W3C DOM Level 2 standard, over HTML 4.x/XHTML 1.x/XML 1.x (every standard supported by Xerces). ItsNat uses layering to extend the functionality when necessary but there is no full replacement, every need already covered by a well established API/standard is not rewritten. For instance ItsNat covers with layers the Servlet infrastructure (e.g. `ItsNatServlet`, `ItsNatServletRequest`…), but the original objects can be got when needed, for instance to get the parameters sent to an ItsNat page.

Only very small bunch ItsNat interfaces and W3C DOM APIs are needed to develop complex AJAX based web sites. Most of ItsNat APIs (DOM utilities, components etc) are optional.

1.3.12 COMET without special servers

ItsNat provides an amazingly simple but powerful COMET[10] solution to push code from server to client without an explicit client request using server push, a kind of "Reverse AJAX". ItsNat COMET implementation works with any servlet engine.

1.3.13 A non-intrusive component system

ItsNat has two levels: **Core** and **Component** levels; the Component level is constructed over the Core but Core has no dependency with Components, in fact developers can construct ItsNat based applications without components.

ItsNat components are the "typical" classes encapsulating and managing the state of a visual element alongside a data model (and usually a selection model) and processing user events.

But the ItsNat approach deeply differs of the typical approach. Again, another "ItsNat-Rorschach's test":

Q) What do you see in this HTML fragment?

```
<input type="button" value="Click me!" />
```

A) A button

Q) And this?

```
<p>Click me!</p>
```

A) A button again!

Q) Really?

A) Yes, only changes the appearance

Q) And this fragment?

```
Select the color:
 <input type="radio" name="colors" value="Red" />
 <input type="radio" name="colors" value="Green" />
 <input type="radio" name="colors" value="Blue" />
```

A) Three radio buttons to select a color

Q) And this?

```
Select the color:
 <p>Red</p>
 <p>Green</p>
 <p>Blue</p>
```

A) The same radio buttons to select a color

[10] http://en.wikipedia.org/wiki/Comet_(programming)

Q) Really?

A) Yes, only changes the appearance!

Q) And this fragment?

```
<select>
    <option>Option 1</option>
    <option>Option 2</option>
</select>
```

A) A single selection list box

Q) And this?

```
<div>
    <p>Option 1</p>
    <p>Option 2</p>
</div>
```

A) Of course the same list box! May be with single selection, is unspecified. Something more?

Q) No, it's absolutely clear for me, you have an ItsNat mind!!

ItsNat components go far beyond the typical form control-Java component: every markup element can be a component!!

ItsNat has buttons, lists, tables and trees, more components will be added, any developer can add new components to the framework as plug-ins.

ItsNat components bind the HTML view with an event model, a data model and a selection model (lists, tables and trees). The framework component system uses the view pattern approach; the developer is free to design the component markup how he or she likes (using the pattern technique), and the concrete data model. ItsNat syncs data model and selection model changes with the view automatically, routes client events to the registered component event listener in the server and syncs form control changes (text introduced in a text box etc) with matched components and related server side DOM elements (for instance, the text box component automatically updates the `value` attribute of the `HTMLInputElement` server object when the browser `<input type="text">` control changes).

Furthermore some components associated to form controls can be configured as "markup driven", in this mode form controls can be fully controlled by DOM, the component API is not needed.

ItsNat has a pluggable user defined structure (layout) system when the used user's markup structure is not supported by default by the framework (remember, structure != concrete tag names). **Developer freedom** is one key feature behind ItsNat architecture.

1.3.14 Component system reusing Swing when possible

Why reinvent the wheel if the wheel is already circular, standard, mature and popular? Swing has many classes/interfaces independent of the desktop UI, like data models, selection models and related listeners; furthermore `CellEditor` and `CellEditorListener` are UI independent! and of course used by ItsNat.

ItsNat is strongly inspired in the component architecture of Swing, for instance there is an `ItsNatTree` and an `ItsNatTreeUI` (and of course uses the Swing's tree data and selection models including event listeners). But ItsNat does NOT try to fit the web UI (based in markup with built-in layout rules) with the desktop UI (based in pixels), no ItsNat method gets/sets the (x,y) pixel position of a component!

1.3.15 User defined components, a child's game …

Creating a custom component is so easy as to implement `ItsNatComponent` and optionally register a `CreateItsNatComponentListener` listener.

Furthermore, what is a component? In ItsNat a component is an object binding markup with a data model with some kind of behavior depending on events. But if we remove the "data model" part, an even listener bound to a concrete DOM element could be considered a "component" (not in ItsNat sense), so there is no "panel" component in ItsNat, every DOM element can be considered a panel.

1.3.16 Beyond (X)HTML: pure SVG and SVG embedded in XHTML

May be you now that FireFox 1.5, Safari 3 and Opera 9 (and the new QtWebKit) support pure SVG documents, the most interesting thing is SVG support includes AJAX in these browsers!!

ItsNat treats SVG documents as first class citizens with the same features as X/HTML documents. SVG documents have server state, receive events, have fragments, referrers, COMET, timers, remote control… and components! For instance: a "circle list", a pie chart seen as a list, tables and trees with graphic elements… of course they all include data models, selection models, custom structures, renderers…

Furthermore, FireFox 1.5+, Safari 3 and Opera 9 and QtWebKit support SVG elements embedded inline in XHTML[11]. In ItsNat server based SVG elements in XHTML can receive events, can be attached to components and so on as any other XHTML element.

1.3.17 Beyond (X)HTML: XML

ItsNat can generate XML documents (for instance RDF) with no server state (the server processes the document only in the load phase). ItsNat DOM utilities like lists, tables and trees can be used to create the resulting XML using pattern based techniques, simplifying very much the typical DOM manipulation. XML templates can be used too including node caching (an XML template can have "static" parts). XML documents do not have AJAX events but it does not prevent that components can be used too!

1.3.18 Non-AJAX and JavaScript disabled modes

ItsNat is very strongly based on AJAX, anyway we are conscious AJAX is a relatively new technology and many developers, companies, clients, users etc are not ready to enter in this new era. Notwithstanding direct DOM manipulation at the server, smart cache to gain memory size, DOM cache for quick node path resolution, DOM utilities (renderers, lists, tables, trees) and Swing-like components[12] are worth enough to work with ItsNat in spite of AJAX is missing.

Furthermore, JavaScript can be fully disabled, this is the lowest downgraded mode of ItsNat. This mode is basically the same as non-AJAX mode but no JavaScript code is sent to the client, this feature allows ItsNat to serve pages to clients with JavaScript disabled.

1.3.19 One Web: AJAX everywhere including in your mobile browser

[11] XHTML supports elements with other namespaces, but only nodes with known namespaces are rendered, SVG in XHTML is rendered on browsers with native SVG support.

[12] ItsNat components can be used in a non-AJAX mode, of course with no events, only the load phase.

Besides supported desktop browsers like FireFox 1+, Internet Explorer 6+, Safari 3+ and Opera 9+, ItsNat supports many mobile browsers: Opera Mini, Opera Mobile, Minimo, NetFront, WebKit browsers (iPhone/iTouch, S60WebKit, Android, Iris, QtWebKit) and IE Mobile!! All of them including AJAX[13]!!

1.3.20 Why "ItsNat" name? And what ItsNat is not

ItsNat means "It's Natural", because it is a natural way to develop Java web applications: Java centric, pure HTML, pure W3C DOM APIs, nodes and events, non intrusive, HTML layout highly controlled by the developer, Swing inspired/reused, no new templating language, no strange artifacts like custom tags…

ItsNat "natural" approach does not try to replace the Java developer work and HTML developer art; ItsNat is a "conservative" technology because it tries not to limit your imagination[14]. Current implementation does not provide obscure, sophisticated, overloaded, black boxed, absolute position based and highly intrusive UI controls. For instance do your mission critical applications need resizable cells in tables? Most of them do not. These features are cool but they usually require tons of JavaScript code and behavior is not usually the same cross-browser.

Bad news, ItsNat is not for newcomers, is not for developers looking for a drag & drop framework tied to GUI tools, some serious previous knowledge is necessary like W3C DOM and Swing basics if components are used, and of course the framework itself has a learning curve. ItsNat has many interfaces because is highly structured and highly customizable and fully oriented to Java developers not to declarative oriented GUI tools.

The good news: the learning curve is very very flat because only some basic ItsNat interfaces and the W3C DOM Core API are enough to develop complex AJAX applications.

ItsNat components like buttons, editable labels, text boxes, lists, tables and trees can be used "out of the box" and easily bound to your markup code, but some work is up to you, for instance how to decorate a selected element (to change background color, position, resize…), how to expand/collapse a tree node (display or not display) etc. Most of the time this code will be in the server using pure Java and DOM and when you find your own "style" this code can be reused again and again because is pure Java, pure DOM usually "tag agnostic". Do not worry about this, ItsNat provides "already made" examples like the "Feature showcase"; you can reuse and modify these examples with no limitation.

ItsNat is a highly customizable framework, we can call it like a *metaframework*: any minor piece of markup can be modified, components are open and customizable with custom layouts, custom data and selection models (by default ItsNat uses Swing default models), custom renderers, custom layout structures and so on. In fact a hook allows plugging new user defined components. Furthermore, the framework can be fully extended or replaced because ItsNat architecture is based on interfaces extensively (in fact, `ItsNat` class is almost the unique public class, and this class is abstract[15]).

ItsNat is focused to developers plenty to fear from AJAX and highly intrusive frameworks, fear to lose the control of their work.

---

[13] Exact versions are listed later.

[14] Do Frameworks and APIs Limit Developers' Imagination?
http://www.artima.com/lejava/articles/javaone_2007_chris_maki.html

[15] There are other classes but they are very simple and isolated.

2. CONSIDERATIONS

2.1 DOCUMENT SCOPE

This manual makes an extensive documentation of ItsNat features but must be complemented with the API documentation in javadoc format.

2.2 DOCUMENT CONVENTIONS

A Verdana font is used to describe the ItsNat architecture.

`A Courier New font is used to Java, C/C++ and XML code fragments.`

2.3 REQUIRED DEVELOPER TECHNICAL SKILLS

ItsNat is based on Java 1.4, W3C DOM Level 2 Core[16]/HTML[17]/Events[18], and Servlet 2.2. A medium knowledge of these API is supposed, especially DOM and Swing.

2.4 TECHNICAL REQUIREMENTS, LIMITATIONS AND DEPENDENCIES

ItsNat compiles and runs with Java Standard Edition 1.4 as the minimum configuration, and may compile and run with any upper version without problem; ItsNat has been tested to run with Sun's JDK 1.5 with and without recompilation. To compile ItsNat with Sun JDK 1.5 use javac flags: –source 1.4 –target 1.4.

ItsNat uses the Servlet specification, no advanced API is used, virtually any Servlet 2.2 container may be valid (any specification before 2.2 is deprecated by Sun). ItsNat has been tested with Tomcat 5.5, GlassFish v2, Sun's Java SE 1.4 and 1.5 developer kits.

ItsNat supports and has been tested with the following desktop browsers:

- FireFox 1+ (tested 1.0, 1.5 and 2.0)
- Microsoft Internet Explorer 6 and 7
- Safari 3+ (preferred 3.1 and upper)
- Opera 9+ (tested 9.26)

[16] http://www.w3.org/TR/DOM-Level-2-Core/core.html

[17] http://www.w3.org/TR/DOM-Level-2-HTML/

[18] http://www.w3.org/TR/DOM-Level-2-Events/

- QtWebKit (Qt 4.4, tested Windows and QtJambi)[19]

Previous versions may work (Microsoft Internet Explorer 5 is highly improbable, 5.5 is more probable). Opera is supported in spite of some problems (`unload` event is not ever fired[20], `beforeunload` is not supported, pages are not loaded when using back/forward…). When FireFox is cited we mean any Gecko based (the web engine of FireFox) browser for instance Mozilla family browsers. FireFox, Safari 3 and Opera 9 are considered "W3C browsers", MSIE 6/7 are not "W3C browsers" (their support is poor, for instance, DOM events, namespaces, XHTML etc).

Mobile browsers supported (including AJAX):

- IE Mobile 6 of Windows Mobile 6 (WM 6.1 works too)
- Opera Mini 4
- Opera Mobile 8.6
- Minimo 0.2
- NetFront 3.5
- iPhone/iTouch/Aspen Simulator (iPhone SDK)
- S60WebKit S60 3rd of Nokia Phones (tested SDK FP2)
- Android (tested m5-rc15)
- Iris 1.0.8
- QtWebKit for Qt/Embedded Linux 4.4 is not tested but should work.

Previous versions may work but not tested. Of course mobile browsers have some limitations and some ItsNat features are not supported (all browsers listed support AJAX).

Web browser can be classified in two categories:

1. Standards support: MSIE (MSIE desktop and IE Mobile)[21] and W3C based (other).
2. Device: desktop and mobile.

External dependencies:

- Xerces for Java 2.6.2 as minimum[22]: `xercesImpl.jar` and `xml-apis.jar` files must be present.
- NekoHTML 1.9.7[23]: `nekohtml.jar` must be present.

---

[19] User agent must contain the word "Qt" or "demobrowser" to be recognized by ItsNat as a Qt browser.

[20] http://www.quirksmode.org/bugreports/archives/2004/11/load_and_unload.html

[21] MSIE browsers support some parts of W3C standards and some key parts as events are very different.

[22] This version is the same as the included in Java SE 1.5 (Java 5) http://xerces.apache.org/xerces-j/

[23] http://nekohtml.sourceforge.net

- Xalan Java 2.7.0[24]: only the `serializer.jar` file must be present. Previous versions may work but not tested.

These external products are open source.

Basically all of these browsers

2.5 LICENSE

ItsNat is dual licensed to third parties:

- To develop open source web applications: Affero General Public License Version 3 (AGPLv3)[25]. AGPL license is almost identical to GPL, is a GPL specialization for web (or network based) projects; AGPL mandates that a web application based on AGPL software must be released as source code to the users.

- To develop closed source web applications: a commercial license is needed.

2.6 COPYRIGHTS

ItsNat intellectual property and exploitation rights are owned by Innowhere Software Services S.L., a Spanish company, and Jose María Arranz Santamaría as the author of ItsNat source code, documentation and examples. Innowhere Software Services S.L. grants third party licenses of ItsNat.

The “Feature Showcase” source code (not including the source code of the ItsNat framework provided) and any example in this manual can be used including derivatives without any restriction or fee except you can not claim the original code is owned by you.

[24] http://xml.apache.org/xalan-j/

[25] http://www.fsf.org/licensing/licenses/agpl-3.0.html

3. INSTALATION

3.1 ITSNAT DISTRIBUTION

Decompress the ItsNat distribution .zip file. ItsNat distribution is a Netbeans 6.1 web project, this web application is the "Feature Showcase", an ItsNat based web application with source code showing the ItsNat main features and components.

The "Feature Showcase" is a JVM 1.4 web application and contains the framework in source code form for debugging. If installed the last JDK 1.4[26] and Tomcat 5.5[27] and configured in NetBeans we can get a pure JVM 1.4 environment to run the "Feature Showcase" out of the box with Netbeans 6.1 and upper[28]. Use File/Open Project to load the web project.

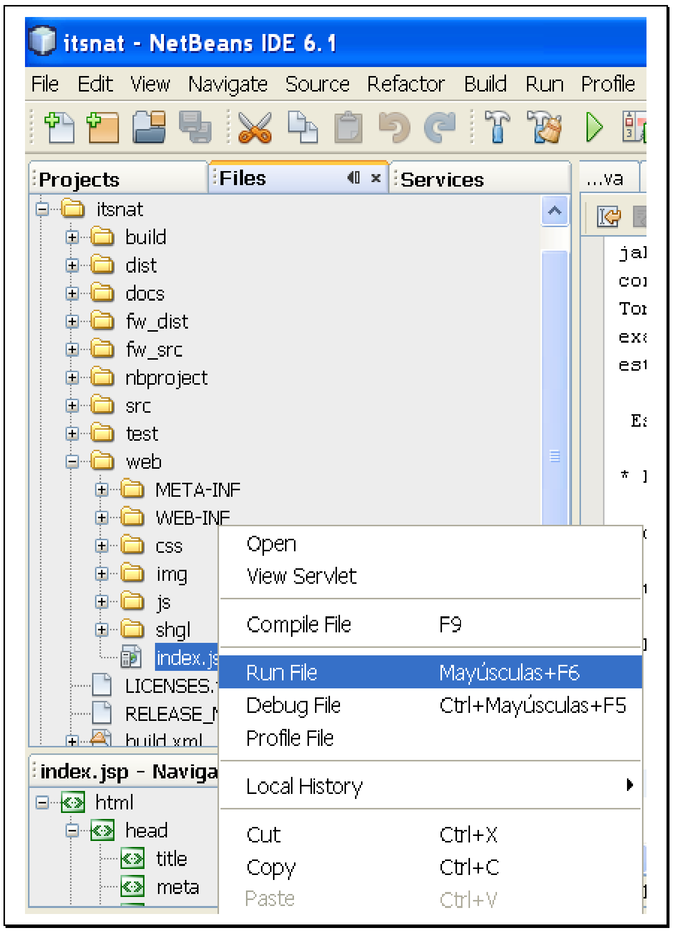
JVM 1.4 is not mandatory for web applications based on ItsNat, in fact the framework itself can be compiled with JDK 1.5 (-source 1.4 and –target 1.4 are necessary flags).

To execute the "Feature Showcase" example, start the web application (run or debug) the `index.jsp` file as shown in the figure:

[26] Select the project in NetBeans, right button, Properties option, Categories: Libraries, Java Platform combo, select the default (will be 1.5 or upper) or your desired JDK, if not present install the desired JDK and click the button "Manage Platforms…" to add the new JDK to NetBeans.

[27] Download and install Tomcat 5.5 (install the JVM 1.4 compatibility pack too), select Services/Servers, button right and select "Add Server …", follow the instructions (is recommended JVM 1.4 to run Tomcat 5.5 if you want a pure JVM 1.4 project).

[28] Project may work with previous NetBeans versions but not tested. Porting the Feature Showcase project to other NetBeans versions (or Eclipse or any other Java IDE) is a very easy task creating a new web project.



This web application is very useful to show how a complex ItsNat application can be developed. It is ready to debug ItsNat source code too because the framework is included in the web application in source code form (`fw_src` directory). This is not the recommended way in production of course, but is very useful to understand how ItsNat works exactly. The source code root directory of the examples is below WEB-INF, of course this is not a usual position too (source code is not usually included in a web application .war file); this source code is distributed with the application because must be accessible by web and showed to the user as documentation.

The `fw_dist/lib` contains the `ItsNat.jar` containing the framework binaries and a separated jar containing the source code. The `js` subdirectory contains four JavaScript files used by the framework.

If you want to execute the "Feature Showcase" example as a "production ready" Java web application, pick the `itsnat.war` archive inside the `dist` directory, this archive contains all is needed to deploy and run in any Servlet container.

To recompile the framework and examples you must change the JDK used by NetBeans selecting the appropriated JDK in *Project Properties/Libraries/Java Platform.* ItsNat can be compiled with JDK 1.5 but –source 1.4 and –target 1.4 are needed.

3.2 WHAT DOES ANY NEW JAVA WEB APPLICATION NEED?

Of course a Java web application developed using ItsNat does not need the ItsNat source code. The `fw_dist/lib/ItsNat.jar` file contains Java binaries only.

Any ItsNat based Java web application need in the standard `WEB-INF/lib` directory the following libraries/files (and where can be found in the ItsNat distribution), these files are located in `fw_dist/lib` directory:

- `ItsNat.jar`: the framework in bytecode form
- `nekohtml.jar`: NekoHTML parser
- `serializer.jar`: standard Apache XML serialization
- `xercesImpl.jar`: Xerces framework
- `xml-apis.jar`: Java standard and W3C XML/DOM APIS

And the following JavaScript files:

```
itsnat.js, itsnat_ajax.js, itsnat_msie.js, itsnat_w3c.js,
itsnat_msie_mobile.js
```

They all are located in `fw_dist/js` and must be put in a public directory of the web application (usually the `js` directory) in the target web application (these files can be added manually to HTML pages or delegate to ItsNat, in the latter case, ItsNat will be informed about the location of the JavaScript standard files).

JVM 1.4 is not mandatory for ItsNat based web applications, upper versions can be used with no problem.

4. ITSNAT ARCHITECTURE

4.1 PURE INTERFACE/IMPLEMENTATION PATTERN

There is almost only public class: `ItsNat`, this class is abstract, the concrete implementation is got calling the static method `ItsNat.get()`, the singleton returned works as a factory: new ItsNat objects implementing public interfaces like `ItsNatServletContext` and `ItsNatServlet` can be created, these objects are used as factory too and so on.

A pure interface/implementation technique allows to program "by contract", where the interface is the contract, only the interface is documented. This technique frees the developer to publish internal methods and classes, the user finds a "clean", solid and stable interface to deal with the implementation, and framework developers can hide internal and "ever changing" implementation artifacts easily. By using only interfaces the internal implementation can be changed "behind the scenes" automatically without external code modification, in fact the ItsNat implementation can be fully switched.

4.2 LAYERED ARCHITECTURE

ItsNat is constructed on top of the Servlet classes/interfaces using a layer & composition approach. An example: an `ItsNatServlet` object contains a field to the real "covered" `Servlet` object (1->1), `ItsNatServlet` interface has a method, `ItsNatServlet.getServlet()`, to return the "real" servlet object if the user need it. This way ItsNat extends the `Servlet` architecture but is not fully replaced, the user can and must use the "old stuff" when ItsNat does not offer something new.

ItsNat is built upon Java W3C DOM too, for instance, an `ItsNatDocument` covers an `org.w3c.dom.Document` object (and of course the covered Document can be got). The "ItsNat" prefix is used to easily distinguish the ItsNat layer and the original data type.

4.3 COMPONENT SYSTEM "INSPIRED" IN SWING

The architecture of ItsNat classes/interfaces (only the interfaces are public) is very similar to Swing. An example:

Swing	ItsNat interface
JTable	**ItsNatTable**
TableCellEditor	**ItsNatTableCellEditor**
TableCellRenderer	**ItsNatTableCellRenderer**
TableUI	**ItsNatTableUI**

The "ItsNat" prefix is used to distinguish ItsNat components from Swing.

ItsNat components use Swing data models, listener models and related listeners.

4.4 MODULES/PACKAGES

ItsNat has two main modules/packages[29]

- Core (`org.itsnat.core`)

  Is the "core" part of the framework, offers the basic infrastructure to develop AJAX based Java web applications.

- Components (`org.itsnat.comp`)

  Contains the component system, it relies on the "core" part, but "core" has no dependency with "components".

4.4.1 Core

The core package contains the fundamental interfaces, this package provides utilities to wrap the servlet system, to register page templates, to control the page lifecycle, to create AJAX listeners etc.

Sub packages:

- `org.itsnat.core.event`: defines event and listener classes and interfaces associated to the page lifecycle and AJAX events.
- `org.itsnat.core.html`: interfaces related to HTML documents and fragments.
- `org.itsnat.core.http`: interfaces related to HTTP servlets.
- `org.itsnat.core.script`: contains utilities to generate JavaScript code to send from server (Java) to client.
- `org.itsnat.core.domutil`: contains utilities to manipulate DOM elements using the pattern approach (lists, tables and trees).

4.4.2 Components

`org.itsnat.comp` package contains generic interfaces of components, they may be applied to HTML components, SVG components etc.

Sub packages:

- `org.itsnat.comp.ui`: contains generic interfaces to control the view of components (DOM layout of the component).
- `org.itsnat.comp.html`: contains interfaces of HTML based components, HTML components are bound to concrete HTML elements.

[29] The package org.w3c.dom contains some public JDK 1.5 DOM classes to support ItsNat compilation with JDK 1.5, they are not part of ItsNat

- `org.itsnat.comp.html.ui`: contains HTML interfaces to control the view of HTML components.
- `org.itsnat.comp.free`: contains interfaces of "free" based components, free components are not bound to concrete tags.

5. DEVELOPMENT LIFECYCLE

5.1 A CORE BASED WEB APPLICATION

We are going to create a simple AJAX based web application using the "core" part of the framework (without components).

5.1.1 Create a new servlet

ItsNat does not provide a framework servlet, the main reason is because the `init(ServletConfig)` method must be used to setup the `ItsNatHttpServlet` object and register used templates. ItsNat provides an abstract servlet: `HttpServletWrapper`, the source code is very simple and useful to understand how the layering starts:

```
public class HttpServletWrapper extends HttpServlet
{
    protected ItsNatHttpServlet itsNatServlet;

    /**
     * Creates a new instance of HttpServletWrapper
     */
    public HttpServletWrapper()
    {
    }

    public ItsNatHttpServlet getItsNatHttpServlet()
    {
        return itsNatServlet;
    }

    /**
     * Processes requests for both HTTP <code>GET</code> and <code>POST</code>
methods.
     *
     * @param request itsNatServlet request
     * @param response itsNatServlet response
     */
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
        throws ServletException, IOException
    {
        itsNatServlet.processRequest(request,response);
    }

    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);

        this.itsNatServlet = (ItsNatHttpServlet)ItsNat.get().createItsNatServlet(this);
    }

    // Other typical servlet methods (doGet, doPost,getServletInfo) go here
   ...
}
```

When the servlet is first loaded an `ItsNatHttpServlet` layer object is created wrapping the real servlet object.

As you can see in the line:

```
        itsNatServlet.processRequest(request,response);
```

Any request received by this servlet is redirected to ItsNat servlet layer.

The easiest way to create the application servlet is to inherit from `HttpServletWrapper`

```
public class servlet extends HttpServletWrapper
{
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
       ...
```

No special configuration is required in the `web.xml` archive to register the servlet (use the typical default code generated by your IDE).

A typical ItsNat application only needs one servlet, anyway multiple ItsNat based servlets may be deployed and they may cooperate because the same `ItsNatSession` and `ItsNatServletContext` objects are shared.

5.1.2 Configuring global behavior

The `init()` method is the appropriate place to setup global behavior:

```
        super.init(config);

        ItsNatHttpServlet itsNatServlet = getItsNatHttpServlet();
        ItsNatServletConfig itsNatConfig = itsNatServlet.getItsNatServletConfig();

        itsNatConfig.setDebugMode(false);
        itsNatConfig. setClientErrorMode(
                        ClientErrorMode.SHOW_SERVER_AND_CLIENT_ERRORS);
        itsNatConfig.setLoadScriptInline(true);
        itsNatConfig.setFastLoadMode(true);
        itsNatConfig.setDefaultSyncMode(SyncMode.ASYNC_HOLD);
        itsNatConfig.setAJAXTimeout(-1);
        itsNatConfig.setOnLoadCacheStaticNodes("text/html", true);
        itsNatConfig.setOnLoadCacheStaticNodes("text/xml", false);
        itsNatConfig.setNodeCacheEnabled(true);
        itsNatConfig.setAddFrameworkScriptFiles(true);
        itsNatConfig.setFrameworkScriptFilesBasePath("js");
        itsNatConfig.setDefaultEncoding("UTF-8");
        itsNatConfig.setUseGZip(UseGZip.SCRIPT);
        itsNatConfig.setDefaultDateFormat(
                        DateFormat.getDateInstance(DateFormat.DEFAULT,Locale.US));
        itsNatConfig.setDefaultNumberFormat(NumberFormat.getInstance(Locale.US));
        itsNatConfig.setEventDispatcherMaxWait(0);
        itsNatConfig.setAJAXEnabled(true);
        itsNatConfig.setScriptingEnabled(true);
        itsNatConfig.setUsePatternMarkupToRender(false);
        itsNatConfig.setAutoCleanEventListeners(true);

        ItsNatServletContext itsNatCtx = itsNatConfig.getItsNatServletContext();
        itsNatCtx.setMaxOpenDocumentsBySession(-1);
```

Most of these options can be avoided because the default values are the same, but they are included to show very important ItsNat features. All of these features can be declared per page too.

5.1.2.1 Debug mode

```
        itsNatConfig.setDebugMode(false);
```

Sets the debug mode as false (default value), the debug mode affects to JavaScript mainly, if debug is true JavaScript errors are not caught, this is very useful in development time because JavaScript debuggers stop in this situation. If debug is false errors are caught and shown to the user but the framework tries to ignore this error and the program could continue "normally". This "tolerant mode" is very much like a Java desktop program deals with exceptions.

5.1.2.2 Client error mode

```
        itsNatConfig.setClientErrorMode(
                        ClientErrorMode.SHOW_SERVER_AND_CLIENT_ERRORS);
```

Specifies the browser catches and shows server (server exceptions) and client (JavaScript) errors to the user (using an `alert`). By default is `SHOW_SERVER_AND_CLIENT_ERRORS`.

5.1.2.3 Initial script inline/loaded

```
        itsNatConfig.setLoadScriptInline(true);
```

When the document/page is first served to the browser initial JavaScript generated code can be send "inline" into a `<script>` element of can be loaded with a special URL. If this feature is set to true the code is sent "inline", this mode is useful to see the initial JavaScript code easily.

5.1.2.4 Fast/slow load mode

```
        itsNatConfig.setFastLoadMode(true);
```

Sets the load mode as "fast" (is the default value), this is the recommended value. Only in special scenarios this setting must be set to false. "Fast" and "slow" modes will be discussed further.

5.1.2.5 Default AJAX sync mode

```
        itsNatConfig.setDefaultSyncMode(SyncMode.ASYNC_HOLD);
```

Sets the default AJAX synchronization mode as "asynchronous-hold" (this is the default value), in this mode browsers events are automatically enqueued as a FIFO and sent sequentially and asynchronously to the server and, when the last event sent returns the next event is sent. This mode provides an almost synchronous event system without blocking the browser.

5.1.2.6 Default AJAX timeout

```
itsNatConfig.setAJAXTimeout(-1);
```

Sets the default timeout of asynchronous AJAX events. This is the time an asynchronous AJAX request will wait before abort. In synchronous mode is ignored (a synchronous request cannot be aborted). A negative value means no timeout. By default is -1.

5.1.2.7 On load static (X)HTML caching to save memory size

```
itsNatConfig.setOnLoadCacheStaticNodes("text/html", true);
```

Enables the "memory cache" in HTML pages (or XHTML if served as HTML), this is the default value. When the template (X)HTML page is first loaded, static DOM subtrees are serialized as text and replaced with a special text mark to save memory.

5.1.2.8 On load static XML caching to save memory size

```
itsNatConfig.setOnLoadCacheStaticNodes("text/xml", false);
```

Disables the "memory cache" to any XML page served with the `text/xml` MIME (the default value). XML generated pages are usually "content based", and fully generated with no static parts.

5.1.2.9 Node cache for speed

```
itsNatConfig.setNodeCacheEnabled(true);
```

Enables the "speed cache" (default value). When the speed cache is enabled, DOM elements are saved in a server and client registry when suitable using a global ID per element, this ID is used to communicate the DOM element identity between server and browser replacing the time consuming task of resolving localization paths in the DOM tree.

5.1.2.10 Automatic load of framework JavaScript files

```
itsNatConfig.setAddFrameworkScriptFiles(true);
```

Tells the framework to add automatically the `<script>` elements to load the JavaScript files of the framework at the bottom of the page (the default value). If false the user must to include these files (usually at the `<head>` section).

```
itsNatConfig.setFrameworkScriptFilesBasePath("js");
```

Tells the framework where the JavaScript framework files are located (a relative path to the web application public folder) to include automatically.

5.1.2.11 Default encoding

```
itsNatConfig.setDefaultEncoding("UTF-8");
```

Defines the default encoding (UTF-8 is the default value).

5.1.2.12 Use GZIP encoding if available

```
itsNatConfig.setUseGZip(UseGZip.SCRIPT);
```

If the browser accepts gzip encoding then JavaScript code (sent as response of an AJAX event) is automatically compressed. Markup may be compressed to:

```
itsNatConfig.setUseGZip(UseGZip.SCRIPT | UseGZip.MARKUP);
```

By default ItsNat uses gzip to compress JavaScript if the browser accepts this encoding.

5.1.2.13 Default date format

```
itsNatConfig.setDefaultDateFormat(
                    DateFormat.getDateInstance(DateFormat.DEFAULT,Locale.US));
```

Defines the default date format, this format is used in components like `ItsNatFormattedTextField` when dealing with dates. The default date format uses the platform locale.

5.1.2.14 Default number format

```
itsNatConfig.setDefaultNumberFormat(NumberFormat.getInstance(Locale.US));
```

Defines the default number format, this format is used in components like `ItsNatFormattedTextField` when dealing with numbers. The default number format uses the platform locale.

5.1.2.15 Default dispatched max wait

```
itsNatConfig.setEventDispatcherMaxWait(0);
```

Defines the max wait an "event dispatcher" thread will wait until a server fired event is processed. An event dispatcher thread is launched using the method `ItsNatDocument.`**`startEventDispatcherThread`**`(Runnable)`. By default is 0 (unlimited).

5.1.2.16 Use AJAX

```
itsNatConfig.setAJAXEnabled(true);
```

Defines whether AJAX is enabled. By default is true.

5.1.2.17 Use JavaScript

```
itsNatConfig.setScriptingEnabled(true);
```

Defines whether JavaScript is enabled (ItsNat sends JavaScript code to clients). By default is true.

5.1.2.18 Use the original markup (pattern) to render

```
itsNatConfig. setUsePatternMarkupToRender(false);
```

If set to true the original markup of a component (the pattern) is ever used to render a new value, for instance a list item. By default is false.

5.1.2.19 Auto Clean Event Listeners

```
itsNatConfig.setAutoCleanEventListeners(true);
```

If set to true the framework automatically removes all event listeners (DOM and User event types) associated to a DOM element and child nodes when this element is removed from the tree. This feature is a kind of "garbage collector" of removed DOM nodes and helps to avoid server and client memory leaks. With this feature set to true components with internal event listeners can be garbage collected when the associated element is removed without calling `ItsNatComponent.`**`dispose`**`()`.

When a DOM element is removed from the tree on the server any event listener still defined is not valid because ItsNat only synchronizes client and server nodes *in the tree*, if the same server DOM element is again inserted in the tree the client node counterpart inserted *is a new node*. When a server DOM element is removed from the tree the client counterpart is "lost" by the framework. There are some exceptions: the `ItsNatDocument.`**`removeEventListener`** and `ItsNatDocument.`**`removeUserEventListener`** methods can remove a listener associated to a DOM element removed from the tree (of course previously was in the tree), this listener is removed too in the client.

5.1.2.20 Max Open Documents By Session

```
ItsNatServletContext itsNatCtx = itsNatConfig.getItsNatServletContext();
itsNatCtx.setMaxOpenDocumentsBySession(-1);
```

Defines the max number of open documents can hold a user server session. This feature is very useful to limit the server memory used by web bots (crawlers), unsupported browsers and browsers with JavaScript disabled traversing pages/documents designed for AJAX.

A negative value is the default and means no limit.

Note this configuration value is set in servlet context level and not in servlet (config) level because an ItsNat session instance is shared by all ItsNat servlets of the same web application.

5.1.3 Designing the page template

ItsNat supports HTML and XHTML files, in our example we are developing a XHTML file like the following:

```
<!-- <?xml version="1.0" encoding="UTF-8"?> -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
```

```
        <title>ItsNat Core Example</title>
    </head>
    <body>
        <h3>ItsNat Core Example</h3>

        <div itsnat:nocache="true" xmlns:itsnat="http://itsnat.org/itsnat">
            <div id="clickableId1">Clickable Elem 1</div>
            <br />
            <div id="clickableId2">Clickable Elem 2</div>
        </div>
    </body>
</html>
```

The standard XML header (<?xml…?>) is optional.

Note the `isnat:nocache` attribute (and the XML mandatory `itsnat` namespace declaration), this attribute, set to true, tells the framework to avoid caching the `<div>` content. `<head>` and `<h3>` elements will be cached automatically, for instance, at the server DOM, `<head>` contains a text node with a cache mark as the only child node, `<h3>` is not touched actually because the framework detects there is no saving (only contains and very small text). The `nocache` attribute is needed because this page is cached by default.

This file will work as a template, and can not be accessible directly with a public URL, so will be saved below the standard `WEB-INF` folder, for instance:

```
<WebAppRoot>/WEB-INF/pages/manual/core_example.xhtml
```

5.1.4 Registering the page template

Now we need to bind the template with ItsNat with the following instructions (again inside `init()` method):

```
String pathPrefix = getServletContext().getRealPath("/");
pathPrefix += "WEB-INF/pages/manual/";

ItsNatHttpServlet itsNatServlet = getItsNatHttpServlet();
DocumentTemplate docTemplate;
docTemplate = itsNatServlet.registerDocumentTemplate("manual.core.example",
            "text/html",pathPrefix + "core_example.xhtml");
```

ItsNat identifies the template with the specified name, `manual.core.example`, this name uses a Java-like format; this format is not mandatory but is highly recommended.

In this example no special configuration technique or framework is used and a hard coded file name is used in the code, if you think this approach is not elegant use the configuration technique you like more (`.properties`, custom XML, Spring …), the easiest way is to use `.properties` archives using the template names as keys.

Why a XHTML file is registered with a text/html MIME? Most of supported browsers accept `application/xhtml+xml` header but Microsoft Internet Explorer (MSIE) 6 does not, this is the most compatible declaration and is the main reason why the MIME type must be explicitly declared when registering.

5.1.5 Testing the template with a link or URL

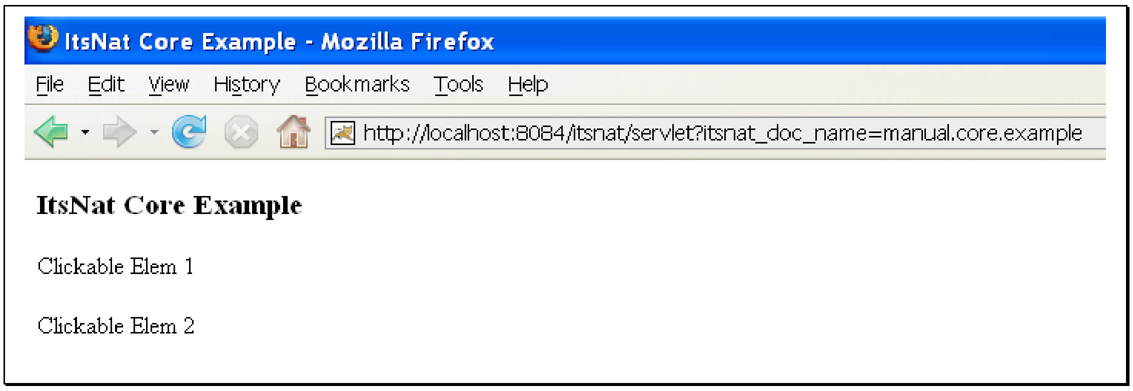
Inside any static html, JSP etc add the following relative link:

```
        <a href="servlet?itsnat_doc_name=manual.core.example">Core Example</a>
```

or type the following in your browser:

```
http://<host>:<port>/<yourapp>/servlet?itsnat_doc_name=manual.core.example
```

A page like this will be loaded in your browser:


ItsNat Core Example - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8084/itsnat/servlet?itsnat_doc_name=manual.core.example

ItsNat Core Example

Clickable Elem 1

Clickable Elem 2

This page is the template page with no "user defined" processing. If you inspect the source code the page is not exactly the original template, some non-intrusive JavaScript code was added automatically at the end of the page, this JavaScript, mostly the "unload" event listener, controls the page lifecycle notifying when the page is unloaded.

5.1.6 Adding behavior

We need to intercept any request to the `manual.core.example` page, to achieve this we need to register a "load listener" to the `DocumentTemplate` object of the page:

```
        docTemplate.addItsNatServletRequestListener(new CoreExampleLoadListener());
```

The `CoreExampleLoadListener` source code:

```
package org.itsnat.manual;

import org.itsnat.core.html.ItsNatHTMLDocument;
import org.itsnat.core.ItsNatServletRequest;
import org.itsnat.core.ItsNatServletRequestListener;
import org.itsnat.core.ItsNatServletResponse;

public class CoreExampleLoadListener implements ItsNatServletRequestListener
{
    public CoreExampleLoadListener()
    {
    }

    public void processRequest(ItsNatServletRequest request,
                               ItsNatServletResponse response)
    {
        ItsNatHTMLDocument itsNatDoc =
                (ItsNatHTMLDocument)request.getItsNatDocument();
        new CoreExampleProcessor(itsNatDoc);
    }
}
```

The `processRequest` method mimics `HttpServlet.doGet/doPost` methods, but using ItsNat object layers (in fact `request` and `response` are `ItsNatHttpServletRequest` and `ItsNatHttpServletResponse` objects). This method is called every time the page is requested to load by the client; the same page can be loaded several times.

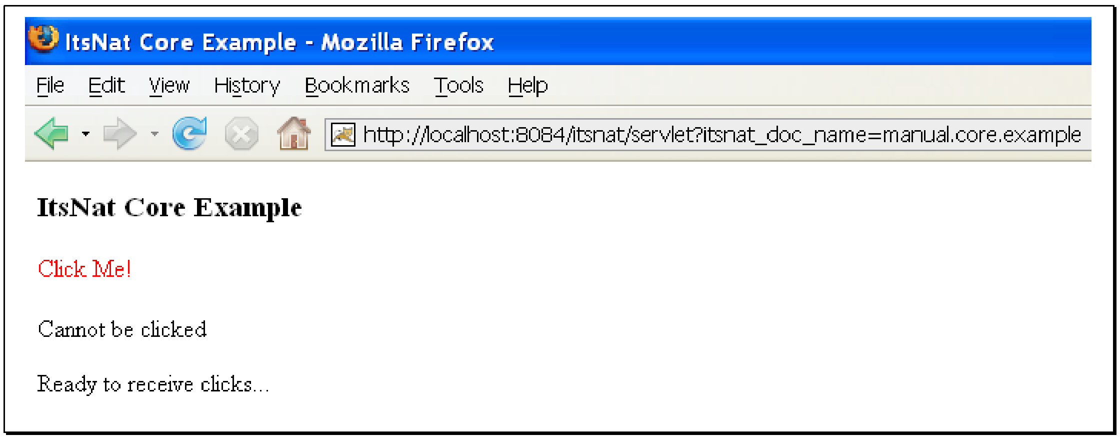
The `ItsNatHTMLDocument` object is the ItsNat object layer covering the DOM `HTMLDocument` object. The `HTMLDocument` instance is a template clone; every loaded page has a different `ItsNatHTMLDocument/HTMLDocument` object pair, then any modification made on an `HTMLDocument` while loading only affects to the concrete loading page. An `ItsNatHTMLDocument/HTMLDocument` object pair lives during the lifecycle of the page and keeps the page state, any subsequent AJAX based event is targeted to a concrete document object. `ItsNatHttpServletRequest` and `ItsNatHttpServletResponse` are unique per request and must not be saved beyond the request (load or AJAX-event request).

To isolate the document load processing and save any desired per loaded page state, a `CoreExampleProcessor` auxiliary object is created; this object does not need to be saved (registered etc).

```
package org.itsnat.manual;

import org.itsnat.core.html.ItsNatHTMLDocument;
import org.w3c.dom.Element;
import org.w3c.dom.Text;
import org.w3c.dom.html.HTMLDocument;

public class CoreExampleProcessor
{
    protected ItsNatHTMLDocument itsNatDoc;
    protected Element clickElem1;
    protected Element clickElem2;

    /** Creates a new instance of CoreExampleProcessor */
    public CoreExampleProcessor(ItsNatHTMLDocument itsNatDoc)
    {
        this.itsNatDoc = itsNatDoc;
        load();
    }

    public void load()
    {
        HTMLDocument doc = itsNatDoc.getHTMLDocument();
        this.clickElem1 = doc.getElementById("clickableId1");
        this.clickElem2 = doc.getElementById("clickableId2");

        clickElem1.setAttribute("style","color:red;");
        Text text1 = (Text)clickElem1.getFirstChild();
        text1.setData("Click Me!");

        Text text2 = (Text)clickElem2.getFirstChild();
        text2.setData("Cannot be clicked");

        Element noteElem = doc.createElement("p");
        noteElem.appendChild(doc.createTextNode("Ready to receive clicks..."));
        doc.getBody().appendChild(noteElem);
    }
}
```

The load method modifies the document changing text nodes and adding a final element, using pure Java DOM because the page DOM tree page is in the server as a Java W3C DOM tree. Any change to the original template is sent to the client.

Redeploying and reloading:


ItsNat Core Example - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8084/itsnat/servlet?itsnat_doc_name=manual.core.example

ItsNat Core Example

Click Me!

Cannot be clicked

Ready to receive clicks...

If you click the "Click Me!" text nothing occurs, we must register a DOM event listener:

```
package org.itsnat.manual;

import org.itsnat.core.ItsNatDocument;
import org.itsnat.core.ItsNatServletRequest;
import org.itsnat.core.event.ItsNatEvent;
import org.itsnat.core.html.ItsNatHTMLDocument;
import org.w3c.dom.Element;
import org.w3c.dom.Text;
import org.w3c.dom.events.Event;
import org.w3c.dom.events.EventListener;
import org.w3c.dom.events.EventTarget;
import org.w3c.dom.html.HTMLDocument;

public class CoreExampleProcessor implements EventListener
{
    ...
    public void load()
    {
        ...
        itsNatDoc.addEventListener((EventTarget)clickElem1,"click",this,false);
    }

    public void handleEvent(Event evt)
    {
        EventTarget currTarget = evt.getCurrentTarget();
        if (currTarget == clickElem1)
        {
            removeClickable(clickElem1);
            setAsClickable(clickElem2);
        }
        else
        {
            setAsClickable(clickElem1);
            removeClickable(clickElem2);
        }
        ItsNatEvent itsNatEvt = (ItsNatEvent)evt;
        ItsNatServletRequest itsNatReq = itsNatEvt.getItsNatServletRequest();
        ItsNatDocument itsNatDoc = itsNatReq.getItsNatDocument();
        HTMLDocument doc = (HTMLDocument)itsNatDoc.getDocument();
        Element noteElem = doc.createElement("p");
        noteElem.appendChild(doc.createTextNode("Clicked " +
                ((Element)currTarget).getAttribute("id")));
        doc.getBody().appendChild(noteElem);
    }

    public void setAsClickable(Element elem)
    {
        elem.setAttribute("style","color:red;");
        Text text = (Text)elem.getFirstChild();
        text.setData("Click Me!");
        itsNatDoc.addEventListener((EventTarget)elem,"click",this,false);
    }

    public void removeClickable(Element elem)
    {
        elem.removeAttribute("style");
        Text text = (Text)elem.getFirstChild();
        text.setData("Cannot be clicked");
        itsNatDoc.removeEventListener((EventTarget)elem,"click",this,false);
    }
}
```

Redeploying and reloading again our web page now receives clicks and sends client event to the server using AJAX (in asynchronous-hold mode, the declared mode by default). Now both DOM elements are enabled/disabled to receive events every time the appropriate element is clicked.

The code fragment:

```
        ItsNatEvent itsNatEvt = (ItsNatEvent)evt;
        ItsNatServletRequest itsNatReq = itsNatEvt.getItsNatServletRequest();
        ItsNatDocument itsNatDoc = itsNatReq.getItsNatDocument();
        HTMLDocument doc = (HTMLDocument)itsNatDoc.getDocument();
```

Is used to show how the DOM Event object is implemented by ItsNat and can be used to obtain the `ItsNatServletRequest` object (`ItsNatHttpServletRequest` actually). Of course the returned `ItsNatDocument` is the same object as the `itsNatDoc` field.

We have finished our first ItsNat AJAX based web application!!

5.2 A COMPONENT BASED WEB APPLICATION

Now we develop an ItsNat application using components. In this example an `<input type="text">` element will be bound to a component at the server.

The development lifecycle is basically the same as the previous example.

5.2.1 Designing the template

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

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>ItsNat Components Example</title>
    </head>
    <body>
        <h3>ItsNat Components Example</h3>

        <div>
            <input type="text" id="inputId" value="" size="40" />
        </div>
    </body>
</html>
```

We can see a “big” difference with the previous “core” example: `itsnat:nocache` is not needed. Why? Because ItsNat detects there is an element “dynamic” by nature, the `<input>` element; ItsNat prevents the `<div>` element to be cached (to avoid the `<input>` removal).

This file will be saved as:

```
   <WebAppRoot>/WEB-INF/pages/manual/comp_example.xhtml
```

5.2.2 Registering the template

Method `init(ServletConfig)`:

```
    docTemplate = itsNatServlet.registerDocumentTemplate("manual.comp.example",
                        "text/html", pathPrefix + "comp_example.xhtml");
    docTemplate.addItsNatServletRequestListener(new CompExampleLoadListener());
```

Now the template is registered and a `CompExampleLoadListener` object is ready to process load request:

```
...
public class CompExampleLoadListener implements ItsNatServletRequestListener
{
    public CompExampleLoadListener()
    {
    }

    public void processRequest(ItsNatServletRequest request,
                               ItsNatServletResponse response)
    {
        ItsNatHTMLDocument itsNatDoc =
                               (ItsNatHTMLDocument)request.getItsNatDocument();
        new CompExampleProcessor(itsNatDoc);
    }
}
```

The load listener delegates to `CompExampleProcessor`, this is a first draft:

```
package org.itsnat.manual;

import org.itsnat.comp.ItsNatComponentManager;
import org.itsnat.comp.html.ItsNatHTMLInputText;
import org.itsnat.core.html.ItsNatHTMLDocument;
import org.w3c.dom.html.HTMLDocument;
```

```
public class CompExampleProcessor
{
    protected ItsNatHTMLDocument itsNatDoc;
    protected ItsNatHTMLInputText inputComp;

    public CompExampleProcessor(ItsNatHTMLDocument itsNatDoc)
    {
        this.itsNatDoc = itsNatDoc;
        load();
    }

    public void load()
    {
        ItsNatComponentManager componentMgr =
                                   itsNatDoc.getItsNatComponentManager();
        this.inputComp =
              (ItsNatHTMLInputText)componentMgr.addItsNatComponentById("inputId");
        inputComp.setText("Change this text and lost the focus");

        inputComp.focus();
        inputComp.select();
    }
}
```

The `ItsNatComponentManager` object works like a page component registry/factory, the `addItsNatComponentById` call obtains the `<input>` element calling `ItsNatDocument.getElementById`, and returns a new `ItsNatHTMLInputText` component, this type is the appropriate to the `<input type="text">` control. The component is registered in the component manager registry too, the same component/object is returned with a call like:

```
componentMgr.findItsNatComponentById("inputId");
```

The `ItsNatHTMLInputText` component has several missions:

1. Wraps the associated DOM `HTMLInputElement` object.

2. Works as a UI coordinator: modifies the DOM object when appropriate (delegating to the `ItsNatTextFieldUI` object).

3. Synchronizes a `javax.swing.text.Document` data model object with the server DOM element (if the data model is changed the DOM server object is changed automatically).

4. Receives the "change" browser DOM event, updating the data model and DOM server element with the new text.

Returning to the example, the line:

```
inputComp.setText("Change this text and lost the focus");
```

Changes the data model (a `javax.swing.text.PlainDocument` by default) with the new text, this change can be performed using the `javax.swing.text.Document` object directly. When a data model is bound to the component (a default data model is ever bound by default) the component registers an internal `DocumentListener`, this listener changes the DOM server `HTMLInputElement` element; if the server element is changed an mutation event is fired and processed by ItsNat generating JavaScript to send to the browser to change the client element too when the server process returns.

The followings lines:

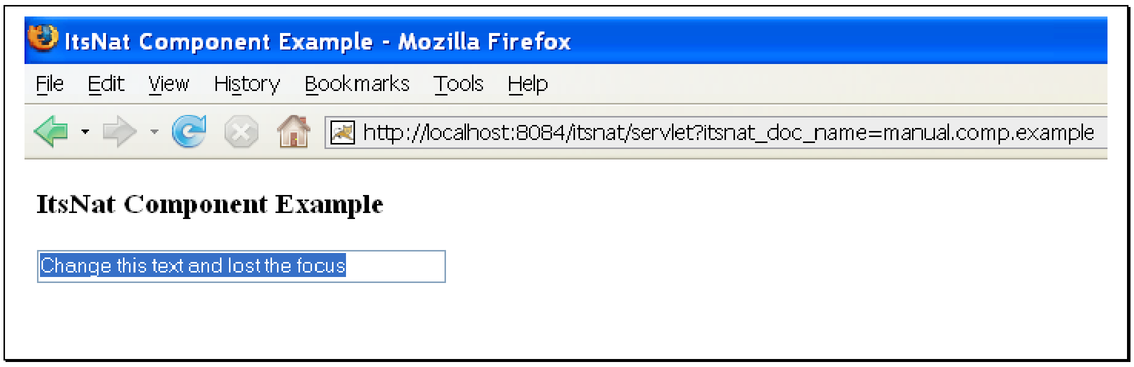
```
        inputComp.focus();
        inputComp.select();
```

Sends JavaScript code to the browser to call `focus()` and `select()` in the `<input>` DOM element[30].

This is the link to execute the example:

```
    <a href="servlet?itsnat_doc_name=manual.comp.example">Component Example</a>
```

And the visual result:


ItsNat Component Example - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8084/itsnat/servlet?itsnat_doc_name=manual.comp.example

ItsNat Component Example

Change this text and lost the focus

If the text is changed (and lost the focus) the component data model is updated.

To detect user changes we can add listeners: a "change" DOM event listener and a `DocumentListener`. In this example any text change is logged, but of course many serious tasks can be done as updating a database (as a response of a `DocumentEvent`).

The complete `CompExampleProcessor` source code:

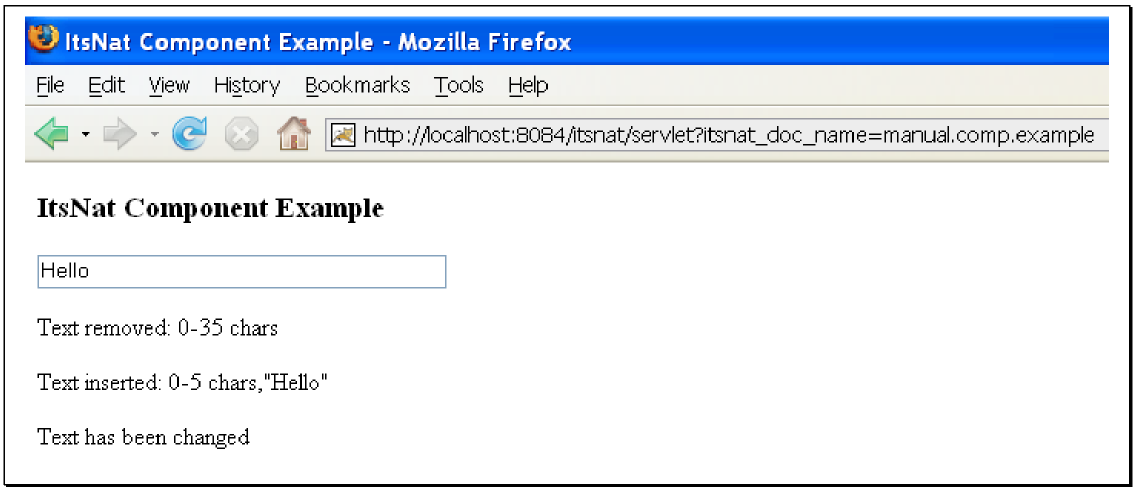
```
package org.itsnat.manual;

import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;
import javax.swing.text.BadLocationException;
import javax.swing.text.PlainDocument;
import org.itsnat.comp.ItsNatComponentManager;
import org.itsnat.comp.html.ItsNatHTMLInputText;
import org.itsnat.core.html.ItsNatHTMLDocument;
import org.w3c.dom.Element;
import org.w3c.dom.events.Event;
import org.w3c.dom.events.EventListener;
import org.w3c.dom.html.HTMLDocument;

public class CompExampleProcessor implements EventListener,DocumentListener
{
    protected ItsNatHTMLDocument itsNatDoc;
    protected ItsNatHTMLInputText inputComp;

    /** Creates a new instance of CoreExampleProcessor */
    public CompExampleProcessor(ItsNatHTMLDocument itsNatDoc)
    {
        this.itsNatDoc = itsNatDoc;
        load();
    }
```

[30] These methods are "dummy" in the Xerces implementation of `HTMLInputElement`

```
public void load()
{
    ItsNatComponentManager componentMgr =
            itsNatDoc.getItsNatComponentManager();

    this.inputComp =
    (ItsNatHTMLInputText)componentMgr.addItsNatComponentById("inputId");
    inputComp.setText("Change this text and lost the focus");

    inputComp.addEventListener("change",this);

    PlainDocument dataModel = (PlainDocument)inputComp.getDocument();
    dataModel.addDocumentListener(this);

    inputComp.focus();
    inputComp.select();
}

public void handleEvent(Event evt)
{
    log("Text has been changed");
}

public void insertUpdate(DocumentEvent e)
{
    javax.swing.text.Document docModel = e.getDocument();
    int offset = e.getOffset();
    int len = e.getLength();

    try
    {
        log("Text inserted: " + offset + "-" + len + " chars,\"" +
            docModel.getText(offset,len) + "\"");
    }
    catch(BadLocationException ex)
    {
        throw new RuntimeException(ex);
    }
}

public void removeUpdate(DocumentEvent e)
{
    int offset = e.getOffset();
    int len = e.getLength();

    log("Text removed: " + offset + "-" + len + " chars");
}

public void changedUpdate(DocumentEvent e)
{
    // A PlainDocument has no attributes
}

public void log(String msg)
{
    HTMLDocument doc = itsNatDoc.getHTMLDocument();
    Element noteElem = doc.createElement("p");
    noteElem.appendChild(doc.createTextNode(msg));
    doc.getBody().appendChild(noteElem);
```

```
        }
}
```

And finally the screenshot:


ItsNat Component Example - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
http://localhost:8084/itsnat/servlet?itsnat_doc_name=manual.comp.example

ItsNat Component Example

Hello

Text removed: 0-35 chars

Text inserted: 0-5 chars,"Hello"

Text has been changed

6. CORE MODULE FEATURES

6.1 DHTML ON THE SERVER

This is the most important feature of ItsNat, any change performed in the server DOM using normal Java W3C DOM methods automatically generates custom JavaScript to automatically change the client DOM too.

This apply to browsers like IE Mobile (Windows Mobile 6), in this browser only HTML elements are reflected in DOM (attributes, comments or text nodes are not reflected as DOM). ItsNat makes an effort to simulate that IE Mobile is a "normal" browser; from server any developer has access (including changes) to DOM elements not accessible from client side.

6.1.1 Text nodes

Text nodes are problematic in DOM because text nodes can be filtered and contiguous nodes can be automatically joined (normalized) by clients. For instance some browsers filter text nodes with spaces when the page is first loaded from markup. W3C DOM standard recommends text nodes must be normalized, that is, contiguous text nodes should be joined and this is the preferred mode of ItsNat. Before you insert a new text node be sure there is no other sibling text node, if present avoid this insertion and change the text value (calling `setData(String)`) of this already inserted text node accordingly.

Contiguous text nodes or text nodes with spaces are not an insurmountable problem for ItsNat because paths used to located nodes on server and on client are calculated ignoring text nodes. The only case text nodes are used to calc paths is when a text node is itself targeted (for instance a text node being removed or text content updated).

6.2 DOM EVENT LISTENERS

A DOM event listener is an `org.w3c.dom.events.EventListener` object associated to a Java DOM element and a specific event type at the server side, ready to be executed when the browser fires this event associated with the symmetric DOM element at client side[31].

DOM event listeners are registered using `ItsNatDocument.`**`addEventListener`** following very much the signature of the `org.w3c.dom.events.EventTarget.`**`addEventListener`** method. In fact the `useCapture` parameter is used to register the listener like W3C DOM Events; when ItsNat uses a remote event listener internally, `useCapture` is ever set to false because this mode is the most compatible with MSIE (MSIE 6 does not use the event capturing technique).

But the remote event system goes further: synchronous modes, extra parameters and custom pre-send JavaScript code can be associated too. These parameters are optional and can be used when registering a listener with `addEventListener`.

6.2.1 Synchronous modes

[31] This is why the word "remote" is used, events occur at client side.

When the browser fires an event with server side listeners, ItsNat uses an `XMLHttpRequest` object to send the event data and receive the JavaScript code. As everybody knows, browser request can be executed synchronously and asynchronously.

6.2.1.1 Synchronous

The synchronous way is the most secure and reliable, because the application has full control of the lifecycle, this is the most similar approach to the typical one-thread event dispatcher of desktop applications; the obvious problem is the browser freezes during event processing, this is not problem in a desktop application but is not usual in a web application.

Synchronous mode can be specified using the constant:

```
org.itsnat.core.SyncMode.SYNC
```

as the `syncMode` parameter.

6.2.1.2 Asynchronous

The asynchronous way is the most user friendly approach but the most insecure in a "application point of view".

Anyway the asynchronous pure mode is *almost* synchronous with ItsNat, because the framework automatically locks (synchronizes) the `ItsNatDocument` target; only one thread can (should) modify the `ItsNatDocument` and dependent objects like the DOM tree and components. ItsNat synchronizes event based threads requesting access to the same `ItsNatDocument` object, when an event is processed (for instance a user defined listener method is called), be sure, the current thread is the only running thread associated to the current document/page being processed. Any web developer must know that JavaScript execution is ever single-threaded, only one event is processed at a given time. Anyway some race conditions can occur like concurrent communications (different delays, different data sizes), concurrent races to gain control of the `ItsNatDocument` object etc.

Asynchronous mode can be specified using the constant:

```
org.itsnat.core.SyncMode.ASYNC
```

as the `syncMode` parameter.

6.2.1.3 Asynchronous-Hold

ItsNat offers a third way: the asynchronous-hold mode. In this mode there is no browser freeze (communication with server is asynchronous) and *pending events are queued* in arrival order (like a FIFO list) in the browser, waiting to the current event to be fully processed. This mode offers the best of both worlds.

The only main caveat is: queued events can not be cancelled because the browser considers these events as processed, neither server side `org.w3c.dom.events.Event` methods like `stopPropagation()` and `preventDefault()` should not be called nor reading/writing of native event JavaScript properties[32].

Asynchronous-Hold mode can be specified using the constant:

[32] Especially with MSIE because event properties are locked, an error is produced. ItsNat queues the event data when is fired (then the event object is "alive").

```
org.itsnat.core.SyncMode.ASYNC_HOLD
```

as the syncMode parameter.

6.2.2 Extra parameters

By default any received org.w3c.dom.events.Event object carries the standard W3C-DOM event properties (current target and target elements, mouse x and y positions and so on). Sometimes we need to get specific info from client, to achieve this any browser event can carry user defined extra parameters.

For instance if we need the current document title we can add a listener ready to receive events with this information:

```
ItsNatDocument itsNatDoc = ...;
Element anElem = ...;
EventListener anListener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        NormalEvent itsNatEvt = (NormalEvent)evt;
        String title = (String)itsNatEvt.getExtraParam("title");
        System.out.println("Page title: " + title);
    }
};
CustomParamTransport extraParam =
            new CustomParamTransport ("title","document.title");
itsNatDoc.addEventListener((EventTarget)anElem,"click",anListener,false,
    new ParamTransport[]{ extraParam });
```

The ClientParam object specifies a name and a JavaScript code to execute when the event is fired, this class is a specialization of the abstract class ParamTransport used to transport client to server data. In this example the name "title" is the extra parameter name, is used only to identify the parameter, and "document.title" is the JavaScript code used to get the parameter value, executed in the browser when the event is fired. CustomParamTransport inherits from ParamTransport, this class is the base of the classes oriented to transport data from the client.

6.2.3 Custom pre-send JavaScript code

Custom user-defined JavaScript code can be executed every time the event is fired (parameter preSendCode).

Example:

```
ItsNatDocument itsNatDoc = ...;
EventTarget anElem = ...;
EventListener anListener = ...;
String code = "    alert('Fired a ' + evt.type + ' event');\n";
itsNatDoc.addEventListener(anElem,"click",anListener,false,code);
```

This is a fragment of the JavaScript code generated[33]:

```
var listener =
function (evt)
{
  function ItsNatEvtExec()
  {
    this.DOMEvent = itsnat.DOMEvent;
    this.DOMEvent(evt,2,"1178285387906_7",3,itsNatDoc);
    alert('Fired a ' + evt.type + ' event');
  }
  new ItsNatEvtExec().sendEvent();
}
```

This function is called when the event is fired and the `evt` parameter is the native event object.

6.2.4 AJAX timeout

The timeout of asynchronous AJAX events can be specified when registering a listener for instance as the last parameter in:

```
public void addEventListener(EventTarget target,String type,
    EventListener listener, boolean useCapture,int syncMode,
    ParamTransport[] extraParams,String preSendCode,long ajaxTimeout);
```

If not specified the default value of the document is used.

Use this feature with care (by default there is no timeout), an aborted request may stop a COMET process, a timer, a remote view/control process etc, and the worst thing, the client may remain unsynchronized.

6.2.5 Load and unload events

Java DOM elements (`org.w3c.dom.Element`) and `org.w3c.dom.Document` objects can be targets of remote event listeners. Load and unload event types are special because in the window object is the target. To emulate this behavior, ItsNat implements a fake window object implementing the official W3C `AbstractView` interface, this object implements `Node` (most of the methods are invalid) and `EventTarget` interfaces too; this window object can be obtained using the `ItsNatDocument` object because implements `DocumentView`.

The following example shows how the "load" and "unload" events can be listened:

```
ItsNatDocument itsNatDoc = ...;
DocumentView docView = (DocumentView)itsNatDoc;
AbstractView view = docView.getDefaultView();
EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        System.out.println(evt.getType()); // outs load/unload
    }
```

[33] This code is generated by the framework and subject to changes (except the user defined code). A change to the event parameter name ("evt") is highly improbable.

```
    };
    itsNatDoc.addEventListener((EventTarget)view,"load",listener,false);
    itsNatDoc.addEventListener((EventTarget)view,"unload",listener,false);
```

6.2.6 Key events

The W3C DOM Events Level 2 does not define a key event standard API, at the time of writing W3C DOM Events Level 3 is still a draft, this standard defines key events but the specification have been evolving.

This is the current state (desktop):

- Internet Explorer +6: has the traditional proprietary key event API, in fact MSIE has only one type of event[34].
- Gecko browsers (Mozilla,FireFox): key event API is based on an early W3C DOM 2 draft[35], this API was removed on the final Recommendation.
- Safari 3 (AppleWebkit): is compliant with an old W3C DOM 3 draft[36], this API is not the current.
- Opera 9: key event API is a mix of W3C and MSIE, with properties like `keyCode`, `altKey`, `shiftKey` and `ctrlKey`. Key events can be created programmatically in Opera with the standard call `DocumentEvent.`**`createEvent`**`(String)` with "Events" as the type, initialized with the `Event.`**`initEvent`** and "manually" adding the properties `keyCode`, `altKey`, `shiftKey` and `ctrlKey`. As MSIE the property `keyCode` is the char code on "`keypress`" events.

As FireFox is considered the facto W3C standard implementation (the market share is by far bigger than Safari or Opera), ItsNat defines an interface, `org.itsnat.core.event.KeyEvent`, to provide a standard key event API, this interface is basically the key event interface used by Gecko browsers[37]. Any client key event is converted on the server to this event API, the behaviour of a FireFox key event is simulated too if the client is not a Gecko browser. Key events can be created programmatically on the server using the `ItsNatDocument` object cast to `DocumentEvent`, and calling `DocumentEvent.`**`createEvent`**`(String)` with "`KeyEvents`" or "`KeyEvent`" type as parameter, then use `KeyEvent.`**`initKeyEvent`** method to initialize this object.

6.3 REMOTE DOM MUTATION EVENT LISTENERS

W3C DOM Events Level 2 defines mutation events, mutation events are fired when a DOM node is changed in some way (a node or attribute was inserted, removed or updated). Currently, only W3C browsers (FireFox, Safari[38] and Opera) support mutation events.

---

[34] http://msdn2.microsoft.com/en-us/library/ms535863(VS.85).aspx

[35] http://www.w3.org/TR/1999/WD-DOM-Level-2-19990923/events.html#Events-KeyEvent

[36] http://www.w3.org/TR/2003/WD-DOM-Level-3-Events-20030331/events.html#Events-KeyboardEvents-Interfaces

[37] http://lxr.mozilla.org/seamonkey/source/dom/public/idl/events/nsIDOMKeyEvent.idl

[38] Safari 3.0 has several bugs: http://lists.webkit.org/pipermail/webkit-dev/2007-July/002067.html

ItsNat is designed as a server centric tool, the browser is automatically updated when the server DOM tree changes, there is no "out of the box" support to update the server DOM tree when the browser DOM tree changes[39].

This scenario is a bit different with W3C browsers, ItsNat has a special support of "remote" mutation events, offering the tools to automatically update the server when the client DOM changes. Mutation events can be received using normal listeners and ItsNatDocument.**addEventListener** but the event info is not enough to synchronize the server DOM with the client changes.

ItsNat defines the method ItsNatDocument.**addMutationEventListener**, this method internally uses a special "remote parameter" object, NodeMutationTransport; this class has a similar mission to the CustomParamTransport class and tells to ItsNat to transport the necessary mutation data from client to server.

When a mutation event is fired in the client, all needed information is collected and sent to the mutation listener, this object synchronizes the client change with the server. The addMutationEventListener method registers the listener to track and synchronize any change performed in the submitted element and children (subtree) in the client, receiving any DOMNodeInserted, DOMNodeRemoved, DOMAttrModified or DOMCharacterDataModified event fired in the subtree.

Current ItsNatDocument.**addMutationEventListener** implementation does the following:

```
public void addMutationEventListener(EventTarget target,
    EventListener listener,boolean useCapture,int syncMode,String preSendCode)
{
     ParamTransport[] params =
             new ParamTransport[]{new NodeMutationTransport()};
     addEventListener(target,"DOMAttrModified",listener,useCapture,
             syncMode,params,preSendCode);
     addEventListener(target,"DOMNodeInserted",listener,useCapture,
             syncMode,params,preSendCode);
     addEventListener(target,"DOMNodeRemoved",listener,useCapture,
             syncMode,params,preSendCode);
    addEventListener(target,"DOMCharacterDataModified",listener,
             useCapture,syncMode,params,preSendCode);
}
```

An example:

```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();

EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        MutationEvent mutEvent = (MutationEvent)evt;

        String type = mutEvent.getType();
        if (type.equals("DOMNodeInserted"))
        {
             Element parent = (Element)mutEvent.getRelatedNode();
```

[39] Some ItsNat components like components bound to form controls have some automatic server updating support (server DOM element is updated when the form control changes).

```
                Node newNode = (Node)mutEvent.getTarget();
                System.out.println("DOMNodeInserted " + newNode + " " +
                        newNode.getNextSibling());
            }
            else if (type.equals("DOMNodeRemoved"))
            {
                Element parent = (Element)mutEvent.getRelatedNode();
                Node removedNode = (Node)mutEvent.getTarget();
                System.out.println("DOMNodeRemoved " + removedNode + " " +
                        parent);
            }
            else if (type.equals("DOMAttrModified"))
            {
                Attr attr = (Attr)mutEvent.getRelatedNode();
                Element elem = (Element)mutEvent.getTarget();
                String attrValue = elem.getAttribute(mutEvent.getAttrName());
                String changeName = null;
                int changeType = mutEvent.getAttrChange();
                switch(changeType)
                {
                    case MutationEvent.ADDITION:
                        changeName = "addition";
                        break;
                    case MutationEvent.MODIFICATION:
                        changeName = "modification";
                        break;
                    case MutationEvent.REMOVAL:
                        changeName = "removal";
                        break;
                }
                System.out.println("DOMAttrModified (" + changeName + ") " +
                        mutEvent.getAttrName() + " " + attrValue + " " + elem);
            }
            else if (type.equals("DOMCharacterDataModified"))
            {
                CharacterData charNode = (CharacterData)mutEvent.getTarget();
                System.out.println("DOMCharacterDataModified " +
                        mutEvent.getNewValue());
            }
        }
    };

    itsNatDoc.addMutationEventListener((EventTarget)doc,listener,false);
```

In this example a custom listener object is registered to listen document changes as mutation events fired by the browser, this listener is called after the server was automatically synchronized with client changes. The `Document` object is used as target, therefore any client change realized in the DOM client tree is automatically synchronized at the server DOM tree.

6.4 CROSS-PLATFORM CLIENT TO SERVER SYNCHRONIZATION

ItsNat offers some portable support to easily update the server DOM tree when a client change occurs.

ItsNat provides several event driven techniques to automatically synchronize a client DOM element changed with the server counterpart:

- Attribute synchronization
- Node inner synchronization
- Node complete (attributes and inner) synchronization
- Property synchronization

6.4.1 Attribute synchronization

The objective is to transport the current values of the required attributes to the server when an event occurs in the specified client element. This is a specialization of the "Extra parameters" feature and again a specialized `ParamTransport` is used: `NodeAttributeTransport`, this class specifies the attribute name to transport and synchronize.

Example:

```
ItsNatDocument itsNatDoc = ...;
EventTarget anElem = ...;

EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        Element elem = (Element)evt.getCurrentTarget();
        System.out.println(elem.getAttribute("style"));
    }
};

itsNatDoc.addEventListener(anElem,"click",listener,false,
    new NodeAttributeTransport("style") );
```

In this example a custom listener is registered listening `click` events on the specified element. The `NodeAttributeTransport` object commands ItsNat to automatically update the current client `style` attribute at the server element. The listener is called after this synchronization.

To transport the attribute with no synchronization, use the two parameter constructor with `sync` parameter set to false. Use the `ItsNatEvent.`**`getExtraParam`**`(String)` to get the transported attribute. For instance:

```
new NodeAttributeTransport("style",false);
```

6.4.1.1 Transporting all attributes

This feature works well with concrete attributes, the method allows a `NodeAttributeTransport` array, but how can ItsNat synchronize any attribute change? Answer: using a `NodeAllAttribTransport` object. This object mandates ItsNat to carry and synchronize all current attributes of the specified client element:

```
itsNatDoc.addEventListener(anElem,"click",listener,false,
    new NodeAllAttribTransport() );
```

ItsNat sends to the server *all declared element attributes* (names and values) from the client. This technique can detect and sync added or removed attributes in the client.

To transport the attributes with no synchronization, use the one parameter constructor with `sync` parameter set to false. Use the `ItsNatEvent.getExtraParam(String)` to get the transported attribute. For instance:

```
new NodeAllAttribTransport(false);
```

6.4.2 Node inner synchronization

The objective is to transport the current content of the specified client node to the server when an event occurs in this node. A specialized `ParamTransport` is used: `NodeInnerTransport`.

Example:

```
<a href="javascript:void(0);" id="clickableId"
   onclick="addNode(this);">Click to add a new child node</a>

<script type="text/javascript">
function addNode(link)
{
    var newElem = document.createElement("b");
    newElem.appendChild(
           document.createTextNode(" New Node " + link.childNodes.length));
    link.appendChild(newElem);
}
</script>
```

Java code:

```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
Element anElem = doc.getElementById("clickableId");

EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        Element currTarget = (Element)evt.getCurrentTarget();
        Node newNode = currTarget.getLastChild();
        Text text = (Text)newNode.getFirstChild();
        System.out.println("New node : " + newNode + " " + text.getData());
    }
};

itsNatDoc.addEventListener((EventTarget)anElem,"click",listener,false,
           new NodeInnerTransport());
```

In this example a new child node is added in the client when the user clicks the specified link. The `NodeInnerTransport` transports the node content and automatically updates the server content of the link.

6.4.3 Complete node synchronization

This case is the sum of the behaviour of `NodeAllAttribTransport` and `NodeInnerTransport`. The class `NodeCompleteTransport` is used.

6.4.4 Property synchronization

Browser DOM elements have properties, a property is not the same as an attribute though some attributes become properties like `value` in a `<input>` element, because the property can have a different value from the attribute. These properties are ever reflected as attributes in the server DOM, though Java W3C DOM API have get/set based methods to get/set properties, Xerces implementation uses attributes behind the scenes. A user interface action can change an element property like the `value` property of an `<input>` element; this property change is not manifested as an attribute change, then we can not use the ItsNat "attribute synchronization" technique to update the server.

ItsNat provides a technique very similar to attribute synchronization, in this case is "property synchronization".

Like in attribute synchronization a specialized `ParamTransport` descriptor object is used: `NodePropertyTransport`, this class specifies the property name to transport and synchronize as an attribute:

Example:

```
ItsNatDocument itsNatDoc = ...;
HTMLInputElement anElem = ...;

EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        HTMLInputElement elem = (HTMLInputElement)evt.getCurrentTarget();
        System.out.println(elem.getValue());
    }
};

itsNatDoc.addEventListener((EventTarget)anElem,"change",listener,false,
    new NodePropertyTransport("value"));
```

In this example a listener object is registered listening `change` events over the specified server `HTMLInputElement` element (e.g. a text box). The `NodePropertyTransport` object commands to ItsNat to transport the `value` property and synchronizes as the `value` attribute. The listener is called after synchronization.

An alternative `NodePropertyTransport` constructor may be used to synchronize using Java reflection:

```
// Alternative
itsNatDoc.addEventListener((EventTarget)anElem,"change",listener,false,
    new NodePropertyTransport("value",String.class));
```

With this mode the current client `value` property is synchronized at the server element calling `HTMLInputElement.`**`setValue`**`(String)` using Java reflection following the Java Beans method name pattern and data type (`String`) known through the specified `NodePropertyTransport` object.

To transport the property with no synchronization, use the any constructor with the `sync` parameter set to false. Use the `ItsNatEvent.`**`getExtraParam`**`(String)` to get the transported property. For instance:

```
new NodePropertyTransport("value",false);
```

6.5 REMOTE CONTINUE EVENT LISTENERS

Client and server are two different programming scenarios; server code is executed when a client event occurs, server code usually sends JavaScript code to modify the client state and the event lifecycle finally ends. Sometimes when processing an event we need to change the client state and continue again in the server, the typical example is when the server needs some value from the client and this value is not present in the event object. ItsNat provides a new event/listener type: continue.

A continue listener is a Java object implementing the DOM interface `org.w3c.dom.events.EventListener`, this listener is ready to receive `ContinueEvent` events, this interface inherits from `org.w3c.dom.events.Event` because is defined by ItsNat as new DOM event type (a DOM extension). This listener must be registered with the method `ItsNatDocument.`**`addContinueEventListener`**, this action tells to ItsNat to generate and send JavaScript code to the client to call again the server firing a `ContinueEvent`; this event is received by the registered `EventListener` object. When the `ContinueEvent` event is received and processed the listener is automatically unregistered and a new call to `addContinueEventListener` is needed to start a new cycle. When registering a listener, we have the option to specify with a `CustomParamTransport` object the custom JavaScript code what we need to transport data from the client. The `addContinueEventListener` method admits an optional `EventTarget` parameter; this parameter is useful if used along with `ParamTransport` objects.

Example:

```
public void handleEvent(Event evt)
{
    EventTarget currTarget = evt.getCurrentTarget();
    ItsNatEvent itsNatEvent = (ItsNatEvent)evt;
    ItsNatDocument itsNatDoc = itsNatEvent.getItsNatDocument();

    // We need the page title in this context:

    EventListener listener = new EventListener()
    {
        public void handleEvent(Event evt)
        {
            ContinueEvent contEvt = (ContinueEvent)evt;
            String title = (String)contEvt.getExtraParam("title");
            System.out.println("This is the title: " + title);
        }
    };

    ParamTransport[] extraParams =
          new ParamTransport[]{new CustomParamTransport("title","document.title")};
    itsNatDoc.addContinueEventListener(null,listener,itsNatEvent.getSyncMode(),
        extraParams,null,-1);
}
```

In this example a continue request gets the page title from the client.

A continue request is not queued in asynchronous-hold mode to ensure is the next event sent.

6.6 REMOTE USER DEFINED EVENT LISTENERS

Usually events are fired by the browser as a result of a user action over a specific markup element (a mouse click, text changed etc). User defined events allow to call the server from the client in a programmatic manner calling a JavaScript function. This event/listener type is very useful to full control when the server is called.

Another useful scenario is when ItsNat is integrated with a well established JavaScript library (usually a DHTML library); usually JavaScript libraries generate HTML code, this HTML code of course is not controlled by ItsNat and can be very hard to bind ItsNat DOM listeners to this markup. Notwithstanding these JavaScript libraries usually provide hooks to register user defined JavaScript based listeners, these listeners are the right place to call user defined ItsNat event listeners.

A user defined listener is a Java object implementing the interface `org.w3c.dom.events.EventListener`, this listener is ready to receive `UserEvent` events, this interface inherits from `org.w3c.dom.events.Event` because is defined by ItsNat as new DOM event type (a DOM extension). This listener must be registered with the method `ItsNatDocument.`**`addUserEventListener`**, using a target node and an event name, both parameters, target node and name, are necessary to call the listener (target node may be null). Then the listener is ready to receive `UserEvent` events. If several listeners were registered with the same pair node-name, they all will be called at the same time.

To fire a `UserEvent` use the following methods from JavaScript:

```
document.getItsNatDoc().createUserEvent(name)



document.getItsNatDoc().dispatchUserEvent(targetNode,evt)
```

The first method creates a user event object with the specified name (this name is used to locate associated listeners). This JavaScript object may be used to transport optional parameters calling the object method: `setExtraParam(name,value)` (an example is shown later); this optional parameter is obtained in the server as usual. The second one dispatches the specified user event to the listeners associated to the specified target node and event name.

A third method is available:

```
document.getItsNatDoc().fireUserEvent(targetNode,name)
```

This method is equal to:

```
document.getItsNatDoc().dispatchUserEvent(targetNode,

      document.getItsNatDoc().createUserEvent(name))
```

Example:

```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();

Element buttonElem = doc.getElementById("buttonId");
```

```
EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        UserEvent userEvt = (UserEvent)evt;
        String title = (String)userEvt.getExtraParam("title");
        System.out.println("Page title: " + title);
        String url = (String)userEvt.getExtraParam("url");
        System.out.println("URL: " + url);
    }
};

itsNatDoc.addUserEventListener((EventTarget)buttonElem,"myUserAction",listener);

String code = "";
code += "var itsNatDoc = document.getItsNatDoc();";
code += "var evt = itsNatDoc.createUserEvent('myUserAction');";
code += "evt.setExtraParam('title',document.title);";
code += "evt.setExtraParam('url',document.location);";
code += "itsNatDoc.dispatchUserEvent(this,evt);";
buttonElem.setAttribute("onclick",code);
```

This example registers a listener ready to receive the page title and URL, the listener name is used to construct the `onclick` JavaScript handler of an element. When this element is clicked the user event is fired transporting the document title and URL, both values are received as "extra" parameters.

`ParamTransport` objects can be used too to transport data from client to server.

6.7 TIMERS

ItsNat provides a time based event/listener system. The architecture is very similar to `java.util.Timer` adapted of course to the web. Similar to `java.util.Timer`, ItsNat defines a utility object implementing the `ItsNatTimer` interface and can be created with `ItsNatDocument`.

The `ItsNatTimer` provides all `schedule*` methods introduced in `java.util.Timer` (same functionality with different signatures). The approach is different:

- There is no server thread controlling the scheduled tasks.
- Any registered task is scheduled using the JavaScript `setTimeout` method on the client (a repetitive task will call `setTimeout` several times).
- The concrete task, an object implementing `org.w3c.dom.events.EventListener` in the server, is called when the task is scheduled receiving a `TimerEvent` event object. A `TimerEvent` inherits from `org.w3c.dom.events.Event` because is defined by ItsNat as a DOM extension.
- A `TimerHandle` object is returned when a new task is scheduled, and represents the "task scheduled" (the same `EventListener` object can be shared between several scheduled tasks). A scheduled task can be cancelled with a call to `TimerHandle.`**`cancel`**`()`.

When a scheduled task is fully completed (there is no future scheduled execution) is automatically unscheduled.

`schedule*` methods admit an optional `EventTarget` parameter; this parameter is useful if used along with `ParamTransport` parameters.

Timers (scheduled tasks) are useful to refresh the client with a fixed time interval, animations controlled by the server etc.

Code example:

```
ItsNatDocument itsNatDoc = ...;

EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        TimerEvent timerEvt = (TimerEvent)evt;
        TimerHandle handle = timerEvt.getTimerHandle();
        long firstTime = handle.getFirstTime();
        if ((new Date().getTime() - firstTime) > 10000)
        {
            handle.cancel();
            System.out.println("Timer canceled");
        }
        else System.out.println("Tick, next execution: " +
            new Date(handle.scheduledExecutionTime()));
    }
};

ItsNatTimer timer = itsNatDoc.createItsNatTimer();
TimerHandle handle = timer.schedule(null,listener,1000,2000);

System.out.println("Timer started, first time: " +
            new Date(handle.getFirstTime()) + " period: " + handle.getPeriod());
```

6.8 REMOTE ASYNCHRONOUS TASKS

A remote asynchronous task was developed with this scenario in mind:

1. A server task takes very much time
2. This task is asynchronous by nature (other tasks could be executed at the same time)
3. The `ItsNatDocument` should not be locked (the long task could be executed asynchronously)
4. The client need to be notified when the long task finishes or the long task modifies the client in same way

A Remote Asynchronous Task (RAT) satisfies these requisites. A RAT is a `Runnable` object registered with the method `ItsNatDocument.`**`addAsynchronousTask`**, this method starts a new thread executing the `Runnable` object in this thread; then immediately generates JavaScript code to fire an asynchronous event when is executed in the client, this asynchronous event is caught internally by the framework (implementation details are not public) and the execution does not return to client until the user task ends, this ensures any modification performed at the DOM tree or in general any JavaScript code scheduled to send is sent to the client as the event response.

The `addAsynchronousTask` method has a `lockDoc` parameter, this parameter tells ItsNat to lock (synchronize) the `ItsNatDocument` while executing the task. If set to false (the typical and recommended scenario if the task is long) the user code **must not use** the `ItsNatDocument` or related objects in no way without synchronization, because `ItsNatDocument` and dependent objects are not thread safe.

The following example shows a correct use of a RAT, while executing a RAT any other event can be processed by the document.

```
final ItsNatDocument itsNatDoc = ...;
Runnable task = new Runnable()
{
    public void run()
    {
        try
        {
            Thread.sleep(2000);
        }
        catch(InterruptedException ex) { }

        synchronized(itsNatDoc)  // MANDATORY, lock is false!!
        {
            itsNatDoc.addCodeToSend("alert('Asynchronous task finished!');");
        }
    }
};

itsNatDoc.addAsynchronousTask(task,false); // lock is false !!

itsNatDoc.addCodeToSend("alert('An asynchronous task has started');");
```

6.9 COMET NOTIFIER

ItsNat provides some COMET support without relying on special servers. The ItsNat COMET approach is based on AJAX: the client is ever waiting for an AJAX asynchronous event to return; when new code must be sent to the client this event returns and updates the client and automatically a new AJAX asynchronous request is sent to the server waiting to new asynchronous server changes. This technique is called "long polling"[40].

This solution does not scale very much because requires one server thread per comet notifier, and a HTTP client to server connection blocked. In spite of this technique is the most standard because better COMET solutions require custom HTTP servers. ItsNat COMET implementation keeps this thread stalled most of the time when no asynchronous server changes occur. The HTTP client/server connection blocked is the worst problem because HTTP 1.1 standard only specifies two live connections per browser with the same host[41]; use only one comet notifier to avoid a browser block!.

ItsNat COMET support is based on the interface/default implementation of `CometNotifier`, a COMET notifier object is created with the method

[40] "New Adventures in Comet: polling, long polling or Http streaming with AJAX. Which one to choose?". Jean-Francois Arcand. http://weblogs.java.net/blog/jfarcand/archive/2007/05/new_adventures.html

[41] Mozilla/FireFox and Microsoft Internet Explorer obey this limitation.

`ClientDocument.`**`createCometNotifier`**`()`. To notify the client about a server change call `CometNotifier.`**`notifyClient`**`()` in any time.

The following code is a simple but complete example of how to update the client using `CometNotifier` when a background server thread makes this change. A link is used to start the background task and COMET notifier:

```
final ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();

Element linkToStart = doc.getElementById("linkToStartId");

EventListener evtListener = new EventListener()
{
    protected CometNotifier notifier;
    protected Thread backgroundThr;
    protected boolean started = false;

    public void handleEvent(Event evt)
    {
        EventTarget currTarget = evt.getCurrentTarget();

        if (started)
        {
            stop();
            System.out.println("COMET notifier stopped");
        }

        this.started = true;

        this.notifier = itsNatDoc.getClientDocumentOwner().createCometNotifier();

        System.out.println("COMET notifier started");

        this.backgroundThr = new Thread()
        {
            public void run()
            {
                System.out.println("Background server task started");

                for( ; ; )
                {
                    try
                    {
                        Thread.sleep(2000);
                    }
                    catch(InterruptedException ex) { }

                    if (!started)
                        return; // Finishes the bg task

                    synchronized(itsNatDoc)
                    {
                        itsNatDoc.addCodeToSend("alert('Tick');");
                    }
                    notifier.notifyClient();
                }
            }
        };
```

```
            backgroundThr.start();
        }

        public void stop()
        {
            if (!started) return;

            this.started = false;

            notifier.stop();
            this.notifier = null;

            try{ backgroundThr.join(); } catch(InterruptedException ex) { }
            this.backgroundThr = null;
        }
    };

    itsNatDoc.addEventListener((EventTarget)linkToStart,"click",evtListener,false);
```

The following code fragment is the most interesting part:

```
synchronized(itsNatDoc)
{
    itsNatDoc.addCodeToSend("alert('Tick');");
}
notifier.notifyClient();
```

Synchronization is absolutely necessary because the background thread is going to use the `ItsNatDocument` (or dependent objects) and this object is not synchronized (ItsNat automatically synchronizes the document in a normal request). The `notifyClient()` call does not need to be synchronized (in spite of it is document dependent); this call tells to ItsNat to send this new code immediately as the result of an, already pending, asynchronous AJAX event, and a new asynchronous AJAX request is automatically sent to wait new changes. The method `CometNotifier.`**`stop`**`()` ends the COMET based listener.

6.10 STRING TO DOM CONVERSION

ItsNat provides a very simple way to convert a string with serialized markup to DOM using `ItsNatDocument.toDOM(String)`. This method returns a `DocumentFragment` object ready to be inserted into the document using DOM methods. Use this method with small pieces of markup because source code is ever parsed from scratch and there is no caching, use instead "MARKUP FRAGMENTS".

Example:

```
ItsNatDocument itsNatDoc = ...;

Element refElem = ...;



String code = "<b>New Markup Inserted</b><br />";
```

```
        DocumentFragment docFrag = itsNatDoc.toDOM(code);

        refElem.getParentNode().insertBefore(docFrag, refElem);
```

6.11 MARKUP FRAGMENTS

Any typical templating system has some type of include mechanism. ItsNat has an include mechanism of course, before describing how to include a markup fragment, we need to know how ItsNat manages them.

A markup fragment must be registered in the ItsNat servlet very much as a normal HTML/XML file, in fact, markup fragments support caching too: static parts of markup fragments are automatically cached to save memory, a cached subtree is saved as text once in memory and is not included as DOM in the target document saving memory.

6.11.1 HTML/XHTML fragments

An HTML/XHTML fragment is a normal HTML/XHTML file, for instance:

```
<!-- <?xml version="1.0" encoding="UTF-8"?> -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>What is Lorem Ipsum?</title>
    </head>
    <body>
        <h4>What is Lorem Ipsum?</h4>
        <p>Lorem Ipsum is simply dummy text of the printing and typesetting
           industry.
        </p>
    </body>
</html>
```

ItsNat automatically recognize two fragments: the `<head>` content and the `<body>` content, these parts are "the fragments" (`<head>` and `<body>` elements are not included).

To register an HTML file as a fragment (**`init`**`(ServletConfig)` servlet method):

```
    String pathPrefix = getServletContext().getRealPath("/");
    pathPrefix += "WEB-INF/pages/manual/core/";

    ItsNatHttpServlet itsNatServlet = getItsNatHttpServlet();

    DocFragmentTemplate fragTemplate;
    fragTemplate = itsNatServlet.registerDocFragmentTemplate(
               "manual.core.htmlFragExample","text/html",
               pathPrefix + "fragment_example.xhtml");
```

Registers the fragment (pair of fragments) with the name `manual.core.htmlFragExample`.

6.11.2 SVG fragments

Creating an SVG fragment is similar to HTML:

```
<?xml version="1.0" standalone="no"?>

<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
    <text x="25" y="150" font-family="Verdana" font-size="20" fill="green" >
    Text Included via Fragment
    </text>
</svg>
```

The fragment is the `<svg>` content, the `<svg>` element is not included itself.

To register the XML file as a fragment (continues the previous example):

```
    fragTemplate = itsNatServlet.registerDocFragment("manual.core.svgFragExample",
                "image/svg+xml",pathPrefix + "svg_fragment_example.xml");
```

Registers the SVG fragment with the name `manual.core.svgFragExample`.

6.11.3 XML fragments

An XML fragment (not HTML/XHTML or SVG) is very similar to SVG:

```
<?xml version='1.0' encoding='UTF-8' ?>
<root>
    <title>CD List</title>
    <subtitle>in XML</subtitle>
</root>
```

The fragment is the `<root>` content, the `<root>` element is not included itself, in fact the "root" name is not mandatory, use the root element you like more.

To register the XML file as a fragment (continues the previous example):

```
    fragTemplate = itsNatServlet.registerDocFragment("manual.core.xmlFragExample",
                "text/xml",pathPrefix + "xml_fragment_example.xml");
```

Registers the XML fragment with the name `manual.core.xmlFragExample`.

6.11.4 Static includes

ItsNat has two include mechanisms: static and dynamic.

Static include uses a special ItsNat-prefixed tag, `<include>`, this tag is resolved and replaced with the specified fragment when the template container is first loaded.

Example:

```
 <!-- <?xml version="1.0" encoding="UTF-8"?> -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:itsnat="http://itsnat.org/itsnat">
    <head>
        <meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
        <itsnat:include name="manual.core.htmlFragExample" />
    </head>
    <body>
        <h3>This pages includes a dummy text</h3>
        <itsnat:include name="manual.core.htmlFragExample" />
    </body>
</html>
```

This template is resolved on memory as:

```
<!-- <?xml version="1.0" encoding="UTF-8"?> -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:itsnat="http://itsnat.org/itsnat">
    <head>
        <meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
        <title>What is Lorem Ipsum?</title>
    </head>
    <body>
        <h3>This pages includes a dummy text</h3>
        <h4>What is Lorem Ipsum?</h4>
        <p>Lorem Ipsum is simply dummy text of the printing and typesetting
           industry.
        </p>
    </body>
</html>
```

ItsNat detects automatically if the `<include>` tag is inside `<head>` or `<body>` and includes the appropriate sub fragment.

In XML is very much the same, in this case there is no "sub fragments" because there is no head or body recognized.

To avoid problems with HTML editors there are an alternative to `<include>` not so intrusive, using an attribute: `itsnat:include="fragment name"`:

```
    <span itsnat:include="manual.core.htmlFragExample" />
```

This element is fully replaced by the specified fragment.

Another alternative is `itsnat:includeInside="fragment name"`:

```
    <div itsnat:includeInside="manual.core.htmlFragExample" />
```

In this case the container element is not replaced, the fragment is inserted *inside* the container node.

6.11.4.1 Comments

ItsNat only has two custom tags `<include>` and `<comment>`. Both are replaced or removed on load time. The `<comment>` element can be used to add text (and markup) to templates, this element and content is removed when the template is first loaded. For instance:

```
    <itsnat:comment><![CDATA[ Comment example
        with <tag> as text
    ]]></itsnat:comment>

    <itsnat:comment>Comment example with tags
        <p>A paragraph</p>
```

```
</itsnat:comment>
```

Another alternative not so intrusive is the attribute version: `itsnat:comment=”a comment”`. This attribute is removed when the template is first loaded.

```
<div itsnat:comment="this is a comment" />
```

6.11.5 Dynamic include

ItsNat supports including fragments programmatically, the standard way to add a fragment to a W3C DOM tree is using `org.w3c.dom.DocumentFragment` objects.

For instance:

```
ItsNatDocument itsNatDoc = ...;
Element includeParentElem = ...;

ItsNatServlet servlet = itsNatDoc.getDocumentTemplate().getItsNatServlet();

HTMLDocFragmentTemplate fragTemplate =
    (HTMLDocFragmentTemplate)servlet.getDocFragmentTemplate(
           "manual.core.htmlFragExample");
DocumentFragment docFrag = fragTemplate.loadDocumentFragmentBody(itsNatDoc);

includeParentElem.appendChild(docFrag); // docFrag is empty now
```

The method `ItsNatServlet.`**`getDocFragmentTemplate`**`(String)` returns the specified fragment template and `HTMLDocFragmentTemplate.`**`loadDocumentFragmentBody`**`(ItsNatDocument)` creates a new `DocumentFragment` using the original fragment as a template and the current `ItsNatDocument`. In a XML document use `DocFragmentTemplate.`**`loadDocumentFragment`**`(ItsNatDocument)`.

6.12 DOM UTILITIES TO TRAVERSE AND CREATE DOM ELEMENTS

The W3C DOM is a very powerful API but sometimes is not practical, for instance, typical DOM tree navigation only uses elements filtering text nodes and comments and built-in DOM methods deals with any type of node. W3C DOM Traversal and Range Specification[42] provide two iterator interfaces, almost unknown[43]: `NodeIterator` and `TreeWalker`, they are very useful but both interfaces have an unnecessary state (“current node”) and the most useful interface, `TreeWalker`, has an annoying behavior.

For instance, we want to walk through the child elements of a given root element:

```
Document doc = ...;
Element root = ...;

// Using TreeWalker
```

[42] http://www.w3.org/TR/DOM-Level-2-Traversal-Range

[43] Probably unknown by the people claiming W3C DOM API is hard to use

```
DocumentTraversal travDoc = (DocumentTraversal)doc;
TreeWalker walker =
            travDoc.createTreeWalker(root,NodeFilter.SHOW_ELEMENT,null,true);
Element child = (Element)walker.firstChild();
while(child != null)
{
    // Do something with child ...
    child = (Element)walker.nextSibling();
}
```

What is wrong with this code? Nothing, but if the `root` element is empty (no child elements, `firstChild` returns null) the `walker` object retains the `root` as the "current node"; if root is not empty the walker is set finally with the last child as the current node; this is a problem if `walker` is going to be reused.

To avoid this annoying behavior:

```
Element child = (Element)walker.firstChild();
if (child != null)
{
    while(child != null)
    {
        // Do something with child ...
        child = (Element)walker.nextSibling();
    }
    walker.parentNode();
}
```

In this case `walker` remains `root` as the current node ever.

Using the `ItsNatTreeWalker` class:

```
Element child = ItsNatTreeWalker.getFirstChildElement(root);
while(child != null)
{
    // Do something with child ...
    child = ItsNatTreeWalker.getNextSiblingElement(child);
}
```

No object creation, no filter declaration, no state. `ItsNatTreeWalker` rewrites `TreeWalker` methods with no state and with methods to traverse elements. There are several DOM node types: elements, comments, text nodes, CDATA etc, but the king is the element, ItsNat DOM utilities are oriented to manage DOM elements.

The `ItsNatDOMUtil` class offers some utility methods like methods to get/set text content of an element:

```
Element elem = ...; // <elem>Some Text</elem>
ItsNatDOMUtil.setTextContent(elem,
    ItsNatDOMUtil.getTextContent(elem) + "(updated)");
```

Another useful method is `ItsNatDOMUtil.createElement(String,String,Document)`, this method creates a new DOM element with the specified text as the only child node. For instance:

```
Document doc = ...;
Element h1 = ItsNatDOMUtil.createElement("h1","Home Page",doc);
```

6.13 FREE ELEMENT LISTS

Most of the time we need to deal with consecutive elements: iterate, add, search, and remove one or several elements of an element list. The typical element list is several consecutive elements separated by text nodes with blanks, tabulators or carriage returns, they all share the same parent element; this kind of structure is the typical of XML documents and some HTML constructs. For instance:

```
<select>
    <option>Car</option>
    <option>Airplane</option>
    <option>Ship</option>
</select>
```

Other HTML examples: rows or columns of a table.

ItsNat provides some utilities to deal with element lists to avoid the typical and verbose DOM manipulation. The first one is the `ElementListFree` interface implemented by ItsNat, this interface represents an element list as an integer indexed list ignoring non-element nodes like text nodes. Elements may have different tag names (the meaning of "free").

`ElementListFree` has the usual methods to search, add, insert and remove elements, but it goes beyond, `ElementListFree` inherits from `org.w3c.dom.NodeList` and `java.util.List` and supports `java.util.Iterator` and `java.util.ListIterator`.

An `ElementListFree` object is created using `ElementGroupManager` obtained from `ItsNatDocument`:

```
<div id="elementListId" />
```

```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
Element parent = doc.getElementById("elementListId");
ElementGroupManager factory = itsNatDoc.getElementGroupManager();
ElementListFree elemList = factory.createElementListFree(parent,true);
```

The previous `createElementListFree` call submits a parameter as true: `master`. If `master` is true the `ElementListFree` object is created in "master" mode, in this mode this `ElementListFree` object must be used to add, remove or replace child elements avoiding direct DOM manipulation, this is the fastest mode. If `master` is false, elements can be added or removed using normal DOM methods (`Node.`**`appendChild`**, `Node.`**`removeChild`** etc) too, in this mode `ElementListFre` is more flexible but slower.

Use examples:

```
Element elem;

elem = ItsNatDOMUtil.createElement("p","Item 1",doc);
elemList.addElement(elem);

elem = ItsNatDOMUtil.createElement("p","Item 2",doc);
elemList.addElement(elem);
```

These sentences add two `<p>Item X</p>` elements below the list parent (`<div>`). They both may be obtained using a zero-based index (`<div>` element was initially empty):

```
elem = elemList.getElementAt(1);
System.out.println(ItsNatDOMUtil.getTextContent(elem)); // "Item 2"
```

And replaced:

```
elem = ItsNatDOMUtil.createElement("h1","Tittle",doc);
elemList.setElementAt(0,elem); // Replaces <p>Item 1</p> => <h1>Tittle</h1>
```

Finally as an `ElementListFree` inherits from `java.util.List`, DOM elements can be iterated using standard iterator interfaces:

```
List list = elemList;
for(ListIterator it = list.listIterator(); it.hasNext(); )
{
    Element curElem = (Element)it.next();
    System.out.println(it.nextIndex() + ":" +
           ItsNatDOMUtil.getTextContent(curElem));
    it.remove();
}
```

The starting point of previous examples was an empty list, this is not a requisite, when the `ElementListFree` is just created is synchronized automatically with the "real" DOM element list to represent the current state. This works in master mode too (initial synchronization). For instance:

```
<select>
    <option>Car</option>
    <option>Airplane</option>
    <option>Ship</option>
</select>
```

If an `ElementListFree` is created to manage this already existing list, a call to the method `getLength()` will return 3, and `getElementAt(2)` will return the third DOM element (containing "Ship").

6.14 FREE ELEMENT TABLES

Free element tables are very similar to free element lists, the main different is that elements form a table. For instance (this example consciously avoids the typical `<table>` based example):

```
<div>
    <div>
        <span>Car</span><span>Road</span><span>Ford</span>
    </div>
    <div>
        <span>Airplane</span><span>Air</span><span>Boeing</span>
    </div>
    <div>
        <span>Ship</span><span>Sea</span><span>Ferry</span>
    </div>
</div>
```

ItsNat provides the `ElementTableFree` interface implemented by ItsNat too; this interface represents an element table as an integer indexed row list (and a column list per row) ignoring non-element nodes like text nodes. Elements may have different tag names (the meaning of "free").

ElementTableFree has methods to search, add and insert rows, columns and individual cells. ElementTableFree inherits from ElementListFree, list methods see the table as a row list: every item is a row DOM Element, for instance, an Iterator or ListIterator can be used to iterate/modify the table rows. The method ElementTableFree.**getCellElementListOfRow**(int row) can be used to manage the cells of a row as an ElementListFree.

An ElementTableFree object is created using ItsNatDocument and ElementGroupManager:

```
<div id="elementTableId" />


ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
Element tableParent = doc.getElementById("elementTableId");
ElementGroupManager factory = itsNatDoc.getElementGroupManager();
ElementTableFree table = factory.createElementTableFree(tableParent,true);
```

The previous createElementTableFree call submits a parameter as true: master. If master is true the ElementTableFree object is created in "master" mode, as in ElementListFree in this mode this object must be used to add, remove or replace rows and cells avoiding direct DOM manipulation, this is the fastest mode.

Use examples:

```
Element rowElem;
Element cellElem;

rowElem = doc.createElement("div");

    cellElem = doc.createElement("span");
    cellElem.appendChild(doc.createTextNode("Cell 0,0"));
    rowElem.appendChild(cellElem);

    cellElem = doc.createElement("span");
    cellElem.appendChild(doc.createTextNode("Cell 0,1"));
    rowElem.appendChild(cellElem);

table.addRow(rowElem);

rowElem = doc.createElement("div");

    cellElem = doc.createElement("span");
    cellElem.appendChild(doc.createTextNode("Cell 1,0"));
    rowElem.appendChild(cellElem);

    cellElem = doc.createElement("span");
    cellElem.appendChild(doc.createTextNode("Cell 1,1"));
    rowElem.appendChild(cellElem);

table.addRow(rowElem);
```

These sentences create the following table (the parent <div> element is initially empty and spaces are included to beautify the markup):

```
    <div id="elementTableId">
        <div>
            <span>Cell 0,0</span>
            <span>Cell 0,1</span>
        </div>
        <div>
            <span>Cell 1,0</span>
            <span>Cell 1,1</span>
        </div>
    </div>
```

The starting point of previous examples was an empty table, this is not a requisite, when the `ElementTableFree` is just created is synchronized automatically with the "real" DOM element table to represent the current state. This works in master mode too (initial synchronization).

6.15 DOM RENDERERS

DOM renderers are objects implementing the interfaces `ElementRenderer`, `ElementLabelRenderer`, `ElementTableRenderer` and `ElementTreeNodeRenderer`. They are used to convert an object value to markup (usually as a text node) using a markup pattern. ItsNat defines default renderers, they all are based on the default `ElementRenderer`, this renderer replaces the text node of the pattern with the value converted to string with `Object.toString()`. This default `ElementRenderer` renderer is used by the "DOM element group managers" like labels, lists, tables, trees and by some ItsNat components when no other custom renderer is defined.

For instance:

```
    <div id="elementId"><b><i><img src="image.png" />Text Pattern<img
src="image.png" /></i></b></div>
```

Processed with:

```
    ItsNatDocument itsNatDoc = ...;
    Document doc = itsNatDoc.getDocument();

    Date value = new Date();

    ElementGroupManager factory = itsNatDoc.getElementGroupManager();
    ElementRenderer renderer = factory.createDefaultElementRenderer();
    renderer.render(null,value,doc.getElementById("elementId"), true);
```

Renders:

```
    <div id="elementId"><b><i><img src="image.png"/>Fri Jun 08 18:43:30 CEST
2007<img src="image.png"/></i></b></div>
```

The default renderer supports many structures enclosing a text node (basically replacing the first text found traversing the subtree):

```
<tagS1>...<tagSN><tagI1/>...<tagIN/>Text<tagD1/>...<tagDN/></tagSN>...</tagS1>
```

This renderer is not valid if we want to render a date to the following pattern:

```
<table id="elementId2" border="1px" cellspacing="0" cellpadding="10px">
    <tbody>
        <tr><td>Year:</td><td>(year)</td></tr>
        <tr><td>Month:</td><td>(month)</td></tr>
        <tr><td>Day:</td><td>(day)</td></tr>
    </tbody>
</table>
```

A custom renderer can solve this problem:

```
ElementRenderer customRenderer = new ElementRenderer()
{
    public void render(Object userObj,Object value,Element elem, boolean isNew)
    {
        DateFormat format =
           DateFormat.getDateInstance(DateFormat.LONG,Locale.US);
        // Format: June 8,2007
        String date = format.format(value);
        int pos = date.indexOf(' ');
        String month = date.substring(0,pos);
        int pos2 = date.indexOf(',');
        String day = date.substring(pos + 1,pos2);
        String year = date.substring(pos2 + 1);

        HTMLTableElement table = (HTMLTableElement)elem;
        HTMLTableSectionElement tbody =
          (HTMLTableSectionElement)ItsNatTreeWalker.getFirstChildElement(table);

        HTMLTableRowElement yearRow =
          (HTMLTableRowElement)ItsNatTreeWalker.getFirstChildElement(tbody);
        HTMLTableCellElement yearCell =
          (HTMLTableCellElement)yearRow.getCells().item(1);
        ItsNatDOMUtil.setTextContent(yearCell,year);

        HTMLTableRowElement monthRow =
          (HTMLTableRowElement)ItsNatTreeWalker.getNextSiblingElement(yearRow);
        HTMLTableCellElement monthCell =
          (HTMLTableCellElement)monthRow.getCells().item(1);
        ItsNatDOMUtil.setTextContent(monthCell,month);

        HTMLTableRowElement dayRow =
          (HTMLTableRowElement)ItsNatTreeWalker.getNextSiblingElement(monthRow);
        HTMLTableCellElement dayCell =
          (HTMLTableCellElement)dayRow.getCells().item(1);
        ItsNatDOMUtil.setTextContent(dayCell,day);
    }

    public void unrender(Object userObj,Element elem)
    {
    }
};

customRenderer.render(null,value,doc.getElementById("elementId2"),true);
```

Renders:

```
<table id="elementId2" border="1" cellpadding="10" cellspacing="0">
    <tbody>
        <tr><td>Year:</td><td>2007</td></tr>
        <tr><td>Month:</td><td>June</td></tr>
        <tr><td>Day:</td><td>8</td></tr>
    </tbody>
</table>
```

Previous example does not need a renderer at all, is an example of how to define user defined renderers to be used by "DOM element group managers".

Using `ItsNatVariableResolver` the custom renderer code can be strongly simplified and markup independent. `ItsNatVariableResolver` will be studied later.

The `unrender` method may be called before the involved DOM element is removed of the DOM tree. The default renderer object does nothing.

6.16 PATTERN BASED ELEMENT LABELS

Pattern based DOM element labels are part of the "DOM element group managers" along with lists, tables and trees. A pattern based DOM element label, an instance of `ElementLabel`, leverages an `ElementRenderer` introducing the concept of DOM patterns. `ElementLabel` relies on `ElementLabelRenderer`, but default implementation relies on `ElementRenderer` default implementation. A DOM pattern is a piece of markup saved in some place and used to render the final markup going to be included in the DOM tree. When an `ElementLabel` object is created associated to a DOM element, the element content is saved as the "pattern" and optionally removed; if a Java value is rendered, then this DOM pattern is used as the pattern to render as markup.

A label may be used to render a Java value (usually converted to a string) with several markup layouts.

Two examples of layouts:

```
     <p><i><b>LABEL VALUE</b></i></p>

    <div><img src="paragraph.gif" /><b>LABEL VALUE</b></div>
```

A functional example:

```
    <p id="elementId"><b><i>Pattern</i></b></p>
```

The Java code:

```
    ItsNatDocument itsNatDoc = ...;
    Document doc = itsNatDoc.getDocument();
    ElementGroupManager factory = itsNatDoc.getElementGroupManager();
    ElementLabel label =
                factory.createElementLabel(doc.getElementById("elementId"),true,null);
    label.setLabelValue("Hello I'm Jose");
```

Renders:

```
    <p id="elementId"><b><i>Hello I'm Jose</i></b></p>
```

Previous example attaches an `ElementLabel` object to the specified DOM element, by default the original element content (the pattern) is removed (parameter `removePattern = true`) and uses the default `ElementRenderer`. When `ElementLabel.`**`setElementValue`**`(Object)` is called the pattern markup is added again to the element and the final markup is used to render the specified value using the default renderer. Following calls to `setElementValue` use the current rendered markup unless `ElementLabel.`**`isUsePatternMarkupToRender`**`()` returns true (by default is false), then the current markup content is replaced with the original pattern markup and rendered again; this feature takes more time (markup is replaced with the pattern ever before rendering) but is very interesting used along with `ItsNatVariableResolver` (seen later).

6.17 PATTERN BASED ELEMENT LISTS

Pattern based DOM element lists are part of the "DOM element group managers" along with labels, tables and trees.

ItsNat understands a pattern based DOM element list as described in `ElementListFree` but all child elements have the same tag name. Child elements usually have optional sub elements containing a text node. With a pattern based element list we can add new elements with no knowledge about the new child element being created, to achieve this ItsNat needs a child element uses as a pattern, this pattern is used to create new elements. Using this approach the Java code is markup agnostic because the specific markup is declared in the pattern.

```
<parent>
    <item><opt1>...<optN>Pattern</optN>...</opt1></item>
</parent>
```

Example:

An HTML template with:

```
<p>List with an HTML element pattern:</p>
<ul id="elementListId">
    <li><i>Element Pattern</i></li>
</ul>
```

The following Java code:

```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
Element parent = doc.getElementById("elementListId");
ElementGroupManager factory = itsNatDoc.getElementGroupManager();
ElementList elemList = factory.createElementList(parent,true);
elemList.addElement("Madrid");
elemList.addElement("Barcelona");
elemList.addElement("Sevilla");
```

will output the following HTML to the client (some spaces and line feeds have been added to embellish the code):

```
<p>List with an HTML element pattern:</p>
<ul id="elementListId">
    <li><i>Madrid</i></li>
    <li><i>Barcelona</i></li>
    <li><i>Sevilla</i></li>
</ul>
```

The previous `createElementList` call submits one parameter as true: `removePattern`; this parameter specifies whether the pattern itself is removed when the list manager is created (the list is empty but the pattern is saved out of the DOM tree to create new child elements cloning the pattern). The ElementList interface inherits from org.w3c.dom.NodeList and current implementation ever works in "master" mode.

Another example, using a table:

```
<table border="1px" cellpadding="5px">
    <tbody id="elementListId">
        <tr><td>Element Pattern</td></tr>
    </tbody>
</table>
```

using the same Java code will output:

```
<table border="1" cellpadding="5">
    <tbody id="elementListId">
        <tr><td>Madrid</td></tr>
        <tr><td>Barcelona</td></tr>
        <tr><td>Sevilla</td></tr>
    </tbody>
</table>
```

6.17.1 Using custom renderers

How an element is added or updated can be customized programmatically using a custom renderer. A custom renderer is a class implementing ElementListRenderer. For instance:

```
ElementListRenderer customRenderer = new ElementListRenderer()
{
    public void renderList(ElementList list,int index,Object value,
                    Element elem, boolean isNew)
    {
        String style;
        if (index == 0)
            style = "font-style:italic;";
        else if (index == 1)
            style = "font-weight:bold;";
        else
            style = "font-size:large;";
        elem.setAttribute("style",style);

        ItsNatDOMUtil.setTextContent(elem,value.toString());
    }

    public void unrenderList(ElementList list,int index,Element elem)
    {
    }

};
```

The renderList method is called after the new element was added or when an element is being updated, the elem parameter is the content parent of the list item. The unrenderList method is called before the list item is removed. This example sets the style property with different values: the first item with italic, the second with bold else with size 150%. The call setTextContent replace the text content with the new value.

This custom renderer must be submitted to the list manager:

```
ElementGroupManager factory = itsNatDoc.getElementGroupManager();
```

```
ElementList elemList = factory.createElementList(parent,true,
                                  null,customRenderer);
```

If applied to this list:

```
<p>List using a custom renderer:</p>
<ul id="elementListId3">
    <li>Element Pattern</li>
</ul>
```

outputs:

```
<ul id="elementListId3">
    <li style="font-style: italic;">Madrid</li>
    <li style="font-weight: bold;">Barcelona</li>
    <li style="font-size:large;">Sevilla</li>
</ul>
```

6.17.2 Using custom structures

By default the list renderer receives the item DOM element top most parent as the parent element of the item content parent (the `<li>` element in the previous example). The default list renderer uses the default renderer explained on "DOM RENDERERS", this renderer has no problem to replace a text node on the bottom of a tree. But our custom renderer does not support this. For instance, our custom renderer does not work with:

```
    <i><b>Text</b></i>
or
    <i><b><img scr="pre.png"/>Text<img scr="post.png"/></b></i>
```

We can improve our renderer but another option is to define a custom structure that returns the parent element of the text node as the “parent content”.

An example with a very complex structure:

```
<table border="1px" cellpadding="5px">
    <tbody id="elementListId4">
        <tr>
            <td>
                <table border="1px" cellpadding="5px">
                    <tbody>
                        <tr><td>Element Pattern</td></tr>
                    </tbody>
                </table>
            </td>
        </tr>
    </tbody>
</table>
```

This structure shows the following sequence until the text node:

```
<tr><td><table><tbody><tr><td>Text</td></tr></tbody></table></td></tr>
```

Our custom renderer is valid if the content parent element of the item is the last `<td>`:

```
ElementListStructure customStructure = new ElementListStructure()
{
    public Element getContentElement(ElementList list,int index,Element elem)
    {
        /*
```

```
                <tr><td><table><tbody><tr><td>Text</td>...
              */
            HTMLTableRowElement rowElem = (HTMLTableRowElement)elem;
            HTMLTableCellElement cellElem =
               (HTMLTableCellElement)ItsNatTreeWalker.getFirstChildElement(rowElem);
            HTMLTableElement tableElem =
               (HTMLTableElement)ItsNatTreeWalker.getFirstChildElement(cellElem);
            HTMLTableSectionElement tbodyElem = (HTMLTableSectionElement)
                      ItsNatTreeWalker.getFirstChildElement(tableElem);
            HTMLTableRowElement rowElem2 = (HTMLTableRowElement)
                      ItsNatTreeWalker.getFirstChildElement(tbodyElem);
            HTMLTableCellElement cellElem2 = (HTMLTableCellElement)
                      ItsNatTreeWalker.getFirstChildElement(rowElem2);
            return cellElem2;
        }
    };
```

In this example the `elem` parameter receives the top most item parent element, `<tr>`. The element returned by `getContentElement` is the element submitted to the renderer as the `elem` parameter.

This custom structure must be passed to the `ElementList`:

```
    ElementGroupManager factory = itsNatDoc.getElementGroupManager();
    ElementList elemList = factory.createElementList(parent,true,
        customStructure,customRenderer);
```

And this is the output:

```
    <table border="1" cellpadding="5">
        <tbody id="elementListId4">
        <tr>
            <td>
                <table border="1" cellpadding="5">
                    <tbody>
                    <tr><td style="font-style: italic;">Madrid</td></tr>
                    </tbody>
                </table>
            </td>
        </tr>
        <tr>
            <td>
                <table border="1" cellpadding="5">
                    <tbody>
                    <tr><td style="font-weight: bold;">Barcelona</td></tr>
                    </tbody>
                </table>
            </td>
        </tr>
        <tr>
            <td>
                <table border="1" cellpadding="5">
                    <tbody>
                    <tr><td style="font-size:large;">Sevilla</td></tr>
                    </tbody>
                </table>
            </td>
        </tr>
        </tbody>
    </table>
```

Of course this custom structure manager can be easily universalized ("tag agnostic"):

```
<elem1>...<elemN>Text</elemN>...</elem1>
```

```
    public Element getContentElement(ElementList list,int index,Element elem)
    {
        /*
            <elem1>...<elemN>Text</elemN>...</elem1>
         */
        Element parent = elem;
        Element child = elem;
        do
        {
            parent = child;
            child = ItsNatTreeWalker.getFirstChildElement(parent);
        }
        while(child != null);

        return parent;
    }
```

This method returns the `<elemN>` DOM element.

`ElemenList` objects support the "UsePatternMarkupToRender" feature, if set to true (default is false) the original markup of the content of a child element is ever used to render a new value.

6.18 PATTERN BASED ELEMENT TABLES

Pattern based element tables are conceptually symmetric to lists. Tables are managed without head, if the programmer wants to deal with the head part programmatically (`<thead>` in an HTML table), head cells can be managed as an element list.

The generic table structure is:

```
<tableParent>
    <row>
        ...
        <optRowContent>
            <cell>
                <opt1>...<optN>Pattern</optN>...</opt1>
            </cell>
            ...
        </optRowContent>
        ...

    </row>
    ...
</tableParent>
```

An example:

```
<table border="1px" cellpadding="10px">
    <tbody id="tableId">
        <tr>
            <td style="font-size:large;">A City</td>
```

```
                <td style="font-weight:bold;">A Public square</td>
                <td style="font-style:italic;">A Monument</td>
            </tr>
        </tbody>
    </table>
```

The first row is the table pattern, any new row follows the row pattern, and new columns use the first cell as pattern (in fact, only one cell is mandatory as pattern).

ElementTable is the main interface to deal with pattern based tables. The following example fills the 3 column table with data:

```
    ItsNatDocument itsNatDoc = ...;
    Document doc = itsNatDoc.getDocument();

    Element parent = doc.getElementById("tableId");
    ElementGroupManager factory = itsNatDoc.getElementGroupManager();
    ElementTable elemTable = factory.createElementTable(parent,true);
    elemTable.setColumnCount(3);
    elemTable.addRow(new String[] {"Madrid","Plaza Mayor","Palacio Real"});
    elemTable.addRow(new String[] {"Sevilla","Plaza de España","Giralda"});
    elemTable.addRow(new String[] {"Segovia","Plaza del Azoguejo",
        "Acueducto Romano"});
```

Like pattern based lists ElementTable tables ever works in master mode.

By default any table has 0 columns, but the initial row pattern is saved as is, the call setColumnCount(3) allows using the 3-column row pattern with different cell layouts, if the call was setColumnCount(4) the fourth column would be based on the first cell of the row pattern.

This is the output:

```
    <table border="1" cellpadding="10">
    <tbody id="tableId">
        <tr>
            <td style="font-size: large;">Madrid</td>
            <td style="font-weight: bold;">Plaza Mayor</td>
            <td style="font-style: italic;">Palacio Real</td>
        </tr>
        <tr>
            <td style="font-size: large;">Sevilla</td>
            <td style="font-weight: bold;">Plaza de España</td>
            <td style="font-style: italic;">Giralda</td>
        </tr>
        <tr>
            <td style="font-size: large;">Segovia</td>
            <td style="font-weight: bold;">Plaza del Azoguejo</td>
            <td style="font-style: italic;">Acueducto Romano</td>
        </tr>
    </tbody>
    </table>
```

6.18.1 Using custom renderers

Like lists, tables support custom renderers and structures.

An example using a custom renderer with the following pattern:

```
<table border="1px" cellpadding="10px">
<tbody id="tableId2">
    <tr><td>Cell pattern</td></tr>
</tbody>
</table>
```

and the custom renderer changing cell decorations (same decoration per row):

```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();

ElementTableRenderer customRenderer = new ElementTableRenderer()
{
    public void renderTable(ElementTable table,int row,int col,Object value,
            Element cellContentElem, boolean isNew)
    {
        String style;
        if (row == 0)
            style = "font-style:italic;";
        else if (row == 1)
            style = "font-weight:bold;";
        else
            style = "font-size: large;";
        cellContentElem.setAttribute("style",style);

        ItsNatDOMUtil.setTextContent(cellContentElem,value.toString());
    }
    public void unrenderTable(ElementTable table,int row,int col,
            Element cellContentElem)
    {
    }
};

Element parent = doc.getElementById("tableId2");
ElementGroupManager factory = itsNatDoc.getElementGroupManager();
ElementTable elemTable = factory.createElementTable(parent,true,
          null,customRenderer);
...
```

outputs:

```
<table border="1" cellpadding="10">
<tbody id="tableId2">
    <tr>
        <td style="font-style: italic;">Madrid</td>
        <td style="font-style: italic;">Plaza Mayor</td>
        <td style="font-style: italic;">Palacio Real</td>
    </tr>
    <tr>
        <td style="font-weight: bold;">Sevilla</td>
        <td style="font-weight: bold;">Plaza de España</td>
        <td style="font-weight: bold;">Giralda</td>
    </tr>
    <tr>
        <td style="font-size: large;">Segovia</td>
        <td style="font-size: large;">Plaza del Azoguejo</td>
        <td style="font-size: large;">Acueducto Romano</td>
    </tr>
</tbody>
</table>
```

The parameter `elem` is the cell content parent being rendered, this element is determined by the structure manager (`<td>` by default with HTML tables).

6.18.2 Using custom structures

To following example changes the default structure manager to construct an over complex (silly) table using the previous renderer:

```
<table border="1px" cellpadding="10px">
<tbody id="tableId3">
    <tr> <= row parent
        <td>
            <table border="1px" cellpadding="5px">
            <tbody>
                <tr> <= row content parent
                    <td>   <= cell parent
                        <table border="1px" cellpadding="5px">
                        <tbody>
                            <tr>
                                <td>Cell</td> <= td: cell content parent
                            </tr>
                        </tbody>
                        </table>
                    </td>
                </tr>
            </tbody>
            </table>
        </td>
    </tr>
</tbody>
</table>
```

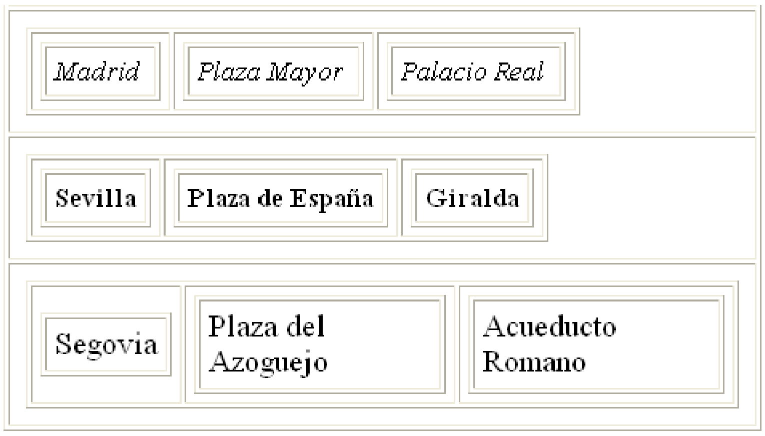
and custom structure:

```
ElementTableStructure customStructure = new ElementTableStructure()
{
    public Element getRowContentElement(ElementTable table,int row,Element elem)
    {
        HTMLTableRowElement rowElem = (HTMLTableRowElement)elem;
        HTMLTableCellElement cellElem = (HTMLTableCellElement)
                   ItsNatTreeWalker.getFirstChildElement(rowElem);
        HTMLTableElement tableElem = (HTMLTableElement)
                   ItsNatTreeWalker.getFirstChildElement(cellElem);
        HTMLTableSectionElement tbodyElem = (HTMLTableSectionElement)
                   ItsNatTreeWalker.getFirstChildElement(tableElem);
        HTMLTableRowElement rowElem2 = (HTMLTableRowElement)
                   ItsNatTreeWalker.getFirstChildElement(tbodyElem);
        return rowElem2;
    }

    public Element getCellContentElement(ElementTable table,
                   int row,int col,Element elem)
    {
        HTMLTableCellElement cellElem = (HTMLTableCellElement)elem;
        HTMLTableElement tableElem = (HTMLTableElement)
                   ItsNatTreeWalker.getFirstChildElement(cellElem);
        HTMLTableSectionElement tbodyElem = (HTMLTableSectionElement)
                   ItsNatTreeWalker.getFirstChildElement(tableElem);
        HTMLTableRowElement rowElem = (HTMLTableRowElement)
```

```
                      ItsNatTreeWalker.getFirstChildElement(tbodyElem);
            HTMLTableCellElement cellElem2 = (HTMLTableCellElement)
                      ItsNatTreeWalker.getFirstChildElement(rowElem);
            return cellElem2;
        }
    };

    Element parent = doc.getElementById("tableId3");
    ElementGroupManager factory = itsNatDoc.getElementGroupManager();
    ElementTable elemTable = factory.createElementTable(parent,true,
        customStructure,customRenderer);
```

The method `getRowContentElement` returns the DOM element parent of the row cells (row content parent), and `getCellContentElement` returns the element parent of the cell content.

The HTML ouput is very big, an image is better:



Madrid	Plaza Mayor	Palacio Real
Sevilla	Plaza de España	Giralda
Segovia	Plaza del Azoguejo	Acueducto Romano

6.18.3 Tables without <table> (free tables)

The default structure manager is tag agnostic, a `<tbody>` table fits perfectly well as a pattern based table structure "out of the box" but `<table>` related elements are not mandatory.

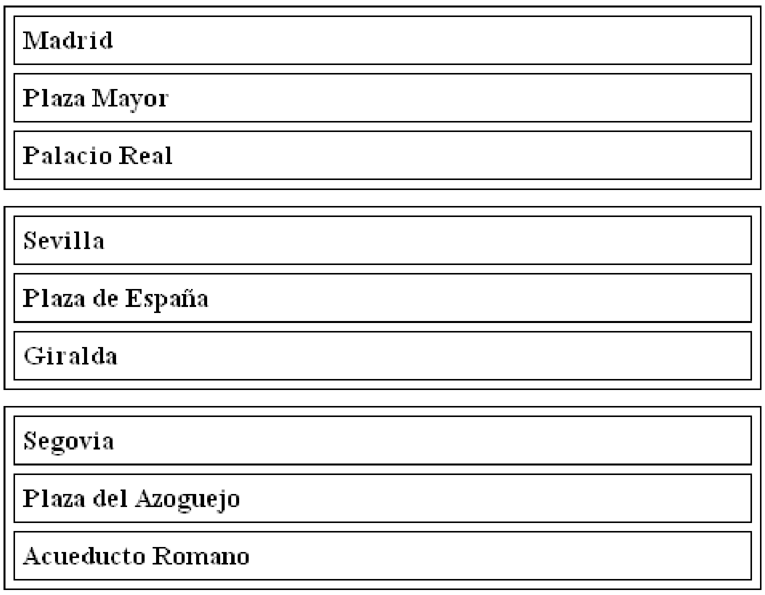
The following example shows a non-HTML table structure without no custom structure and renderer:

```
    <div id="tableId4" style="margin-top: 10px; width: 100%;">
        <div style="margin-top: 10px; border: 1px solid;">
            <div style="margin: 5px; padding: 5px; border: 1px solid;"><b>Cell
Pattern</b></div>
        </div>
    </div>


    Element parent = doc.getElementById("tableId4");
    ElementGroupManager factory = itsNatDoc.getElementGroupManager();
    ElementTable elemTable = factory.createElementTable(parent,true);
```

This is the output:

```
    <div id="tableId4" style="margin-top: 10px; width: 100%;">
        <div style="border: 1px solid ; margin-top: 10px;">
            <div style="border: 1px solid ; margin: 5px; padding: 5px;">
               <b>Madrid</b></div>
            <div style="border: 1px solid ; margin: 5px; padding: 5px;">
               <b>Plaza Mayor</b></div>
            <div style="border: 1px solid ; margin: 5px; padding: 5px;">
               <b>Palacio Real</b></div>
        </div>
        <div style="border: 1px solid ; margin-top: 10px;">
            <div style="border: 1px solid ; margin: 5px; padding: 5px;">
               <b>Sevilla</b></div>
            <div style="border: 1px solid ; margin: 5px; padding: 5px;">
               <b>Plaza de España</b></div>
            <div style="border: 1px solid ; margin: 5px; padding: 5px;">
               <b>Giralda</b></div>
        </div>
        <div style="border: 1px solid ; margin-top: 10px;">
            <div style="border: 1px solid ; margin: 5px; padding: 5px;">
               <b>Segovia</b></div>
            <div style="border: 1px solid ; margin: 5px; padding: 5px;">
               <b>Plaza del Azoguejo</b></div>
            <div style="border: 1px solid ; margin: 5px; padding: 5px;">
               <b>Acueducto Romano</b></div>
        </div>
    </div>
```

And the most interesting part, how is viewed:



6.18.4 Tables and "UsePatternMarkupToRender" feature

`ElemenTable` objects support the "UsePatternMarkupToRender" feature, if set to true (default is false) the original markup of the content of a table cell element is ever used to render a new value.

6.19 PATTERN BASED ELEMENT TREES

Again pattern based element trees are conceptually symmetric to lists and tables. HTML has not a standard `<tree>` element, but people long time ago have designed trees with `<ul>`, tables etc.

The typical tree structure is (tag names are invented):

```
<treeParent>
    <rootNode>
        <content><handle/><icon/>
                 <label><opt1>...<optN>Pattern</optN>...</opt1></label>
        </content>
        <children></children>
    </rootNode>
</treeParent>
```

Every tree has a root node (or no node, but if exists is only one), this node is usually used as the pattern; new child nodes are added below `<children>`.

For instance:

```
<ul id="treeId" style="list-style-type: none;">
    <li>
        <span><img src="img/tree/tree_node_collapse.gif"/>
              <img src="img/tree/gear.gif"/>
              <span><b>Label</b></span>
         </span>
        <ul style="list-style-type: none;" />
    </li>
</ul>
```

The ItsNat tree utility interface is `ElementTree`. `ElementTree` objects only work in master mode.

The following code creates a tree filled with Spanish cities:

```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
Element parent = doc.getElementById("treeId");
ElementGroupManager factory = itsNatDoc.getElementGroupManager();
ElementTree elemTree = factory.createElementTree(false,parent,true);

ElementTreeNode rootNode = elemTree.addRootNode("Spain");
ElementTreeNodeList rootChildren = rootNode.getChildTreeNodeList();

ElementTreeNode provincesNode =
           rootChildren.addTreeNode("Autonomous Communities");
provincesNode.getChildTreeNodeList().addTreeNode("Asturias");
provincesNode.getChildTreeNodeList().addTreeNode("Cantabria");
provincesNode.getChildTreeNodeList().addTreeNode("Castilla La Mancha");

ElementTreeNode ccaaNode = rootChildren.addTreeNode("Cities");
ccaaNode.getChildTreeNodeList().addTreeNode("Madrid");
ccaaNode.getChildTreeNodeList().addTreeNode("Barcelona");
ccaaNode.getChildTreeNodeList().addTreeNode("Sevilla");
```

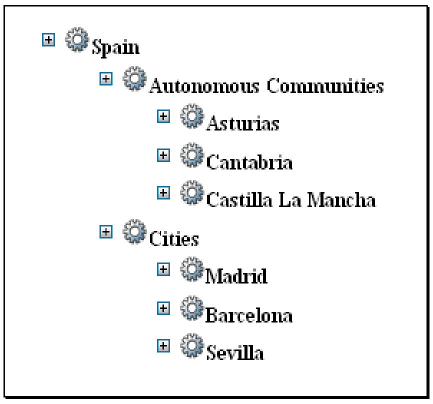
An `ElementTreeNode` object represents a tree node and `ElementTreeNodeList` the children.

This is the HTML output:

```
<ul id="treeId" style="list-style-type: none;">
```

```
        <li>
            <span><img src="img/tree/tree_node_collapse.gif">
                  <img src="img/tree/gear.gif">
                  <span><b>Spain</b></span>
             </span>
            <ul style="list-style-type: none;">
                <li>
                    <span><img src="img/tree/tree_node_collapse.gif">
                          <img src="img/tree/gear.gif">
                          <span><b>Autonomous Communities</b></span>
                    </span>
                    <ul style="list-style-type: none;">
                        <li>
                            <span><img src="img/tree/tree_node_collapse.gif">
                                  <img src="img/tree/gear.gif">
                                  <span><b>Asturias</b></span>
                            </span>
                            <ul style="list-style-type: none;"></ul>
                        </li>
                        <li>
                            <span><img src="img/tree/tree_node_collapse.gif">
                                 <img src="img/tree/gear.gif">
                                 <span><b>Cantabria</b></span>
                            </span>
                            <ul style="list-style-type: none;"></ul>
                        </li>
                        <li>
                            <span><img src="img/tree/tree_node_collapse.gif">
                                  <img src="img/tree/gear.gif">
                                  <span><b>Castilla La Mancha</b></span>
                            </span>
                            <ul style="list-style-type: none;"></ul>
                        </li>
                    </ul>
                </li>
                <li>
                    <span><img src="img/tree/tree_node_collapse.gif">
                          <img src="img/tree/gear.gif">
                          <span><b>Cities</b></span>
                    </span>
                    <ul style="list-style-type: none;">
                        <li>
                            <span><img src="img/tree/tree_node_collapse.gif">
                                 <img src="img/tree/gear.gif">
                                 <span><b>Madrid</b></span>
                            </span>
                            <ul style="list-style-type: none;"></ul>
                        </li>
                        <li>
                            <span><img src="img/tree/tree_node_collapse.gif">
                                 <img src="img/tree/gear.gif">
                                 <span><b>Barcelona</b></span>
                            </span>
                            <ul style="list-style-type: none;"></ul>
                        </li>
                        <li>
                            <span><img src="img/tree/tree_node_collapse.gif">
                                 <img src="img/tree/gear.gif">
                                 <span><b>Sevilla</b></span>
                            </span>
                            <ul style="list-style-type: none;"></ul>
                        </li>
                    </ul>
                </li>
            </ul>
        </li>
    </ul>
```

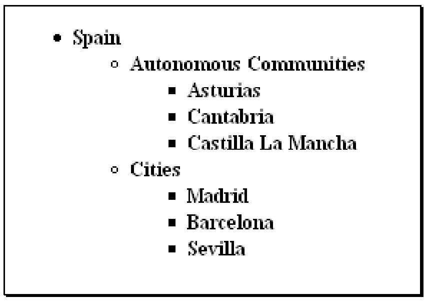
And is rendered as:



Icon and handle are optional, if missing is detected automatically:

```
    <ul id="treeId">
        <li>
            <span><b>Label</b></span>
            <ul></ul>
        </li>
    </ul>
```

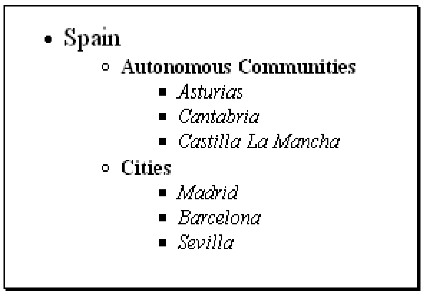
Rendered as:



Similar to the first row in tables, trees allow multiple level tree nodes used as patterns. For instance:

```
    <ul id="treeId">
        <li>
            <span style="font-size:large;">Level 1</span>
            <ul>
                <li>
                    <span style="font-weight:bold;">Level 2</span>
                    <ul>
                        <li>
                            <span style="font-style:italic;">Level 3</span>
                            <ul></ul>
                        </li>
                    </ul>
                </li>
            </ul>
        </li>
    </ul>
```

Using the same Java code to build the tree, this is the output:



- Spain
  - **Autonomous Communities**
    - *Asturias*
    - *Cantabria*
    - *Castilla La Mancha*
  - **Cities**
    - *Madrid*
    - *Barcelona*
    - *Sevilla*

6.19.1 Trees with non-removable root

ItsNat allows creating trees with non-removable root. Template declaration is slightly different, tree parent is the root node, and in Java code a root node, `ElementTreeNode`, is directly created.

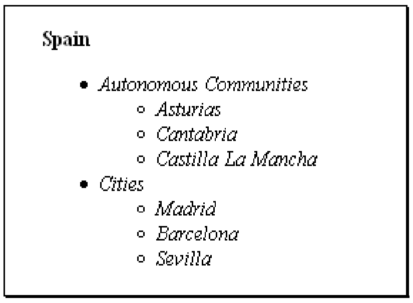
```
<div id="treeId">
    <span><b>Root Content Pattern</b></span>
    <ul>
        <li>
            <span><i>Child Content Pattern</i></span>
            <ul></ul>
        </li>
    </ul>
</div>
```

In this example an optional child pattern with `<li>` is used because root node is based on `<div>`, child nodes will be based on the child pattern.

Java code:

```
ItsNatDocument itsNatDoc =...;
Document doc = itsNatDoc.getDocument();
Element parent = doc.getElementById("treeId");
ElementGroupManager factory = itsNatDoc.getElementGroupManager();
ElementTreeNode rootNode = factory.createElementTreeNode(parent,true);
rootNode.setValue("Spain");

ElementTreeNodeList rootChildren = rootNode.getChildTreeNodeList();
...
```

And the output:



Spain

- *Autonomous Communities*
  - *Asturias*
  - *Cantabria*
  - *Castilla La Mancha*
- *Cities*
  - *Madrid*
  - *Barcelona*
  - *Sevilla*

6.19.2 Rootless

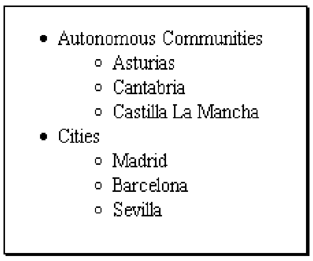
A tree rootless is a tree with no root node, the factory method return an `ElementTreeNodeList` object.

Example:

```
<ul id="treeId">
    <li>
        <span>Content Pattern</span>
        <ul></ul>
    </li>
</ul>
```

```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
Element parent = doc.getElementById("treeId");
ElementGroupManager factory = itsNatDoc.getElementGroupManager();
ElementTreeNodeList rootList =
            factory.createElementTreeNodeList(false,parent, true);
...
```

And renders:



- Autonomous Communities
  - Asturias
  - Cantabria
  - Castilla La Mancha
- Cities
  - Madrid
  - Barcelona
  - Sevilla

6.19.3 Using custom renderer and structure

ItsNat trees use custom renderers and structures too. With a custom renderer we can specify how to fill and decorate the tree node content and a custom structure to define a non-standard layout.

In this example we do not need handle and icon nodes:

```
<ul id="treeId">
    <li>
        <div>
            <span>Content Pattern</span>
            <ul></ul>
        </div>
    </li>
</ul>
```

```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
Element parent = doc.getElementById("treeId");

ElementTreeNodeRenderer customRenderer = new ElementTreeNodeRenderer()
{
    public void renderTreeNode(ElementTreeNode treeNode,Object value,
                   Element labelElem, boolean isNew)
```

```
        {
            int level = treeNode.getDeepLevel();

            String style;
            if (level == 0)
                style = "font-size:large;";
            else if (level == 1)
                style = "font-weight:bold;";
            else
                style = "font-style:italic;";

            labelElem.setAttribute("style",style);
            ItsNatDOMUtil.setTextContent(labelElem,value.toString());
        }

        public void unrenderTreeNode(ElementTreeNode treeNode,Element labelElem)
        {
        }
    };

    ElementTreeNodeStructure customStructure = new ElementTreeNodeStructure()
    {
        public Element getContentElement(ElementTreeNode treeNode,Element nodeElem)
        {
            Element child = ItsNatTreeWalker.getFirstChildElement(nodeElem);
            return ItsNatTreeWalker.getFirstChildElement(child);
        }

        public Element getHandleElement(ElementTreeNode treeNode,Element nodeElem)
        {
            return null;
        }

        public Element getIconElement(ElementTreeNode treeNode,Element nodeElem)
        {
            return null;
        }

        public Element getLabelElement(ElementTreeNode treeNode,Element nodeElem)
        {
            return getContentElement(treeNode,nodeElem);
        }

        public Element getChildListElement(ElementTreeNode treeNode,Element nodeElem)
        {
            Element contentParentElem = getContentElement(treeNode,nodeElem);
            return ItsNatTreeWalker.getNextSiblingElement(contentParentElem);
        }
    };

    ElementGroupManager factory = itsNatDoc.getElementGroupManager();
    ElementTree elemTree = factory.createElementTree(false,parent, true,
                customStructure,customRenderer);
    ...
```

6.19.4 Tree-Tables

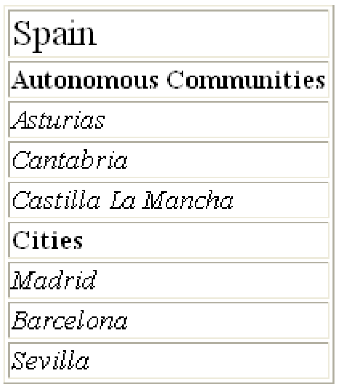
ItsNat supports “out of the box” tree tables: a tree which layout is a list following the tree order, this list can be hold by an HTML table when every node is a row. The tree is showed as a list or table but the hierarchy is internally kept, for instance when a tree node is removed child nodes are removed too. Child list parent DOM element is ever the table parent element. At Java code the parameter `treeTable` of `createElementTree` must be set to true. A `<table>` is the most direct and useful element layout descriptor:

```
<table border="1px">
    <tbody id="treeId">
        <tr>
            <td>Label</td>
        </tr>
    </tbody>
</table>
```

Using the previous custom renderer:

```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
Element parent = doc.getElementById("treeId");

ElementTreeNodeRenderer customRenderer = ...;

ElementGroupManager factory = itsNatDoc.getElementGroupManager();
ElementTree elemTree = factory.createElementTree(true,parent,true,
           null,customRenderer);
...
```

Renders the following:

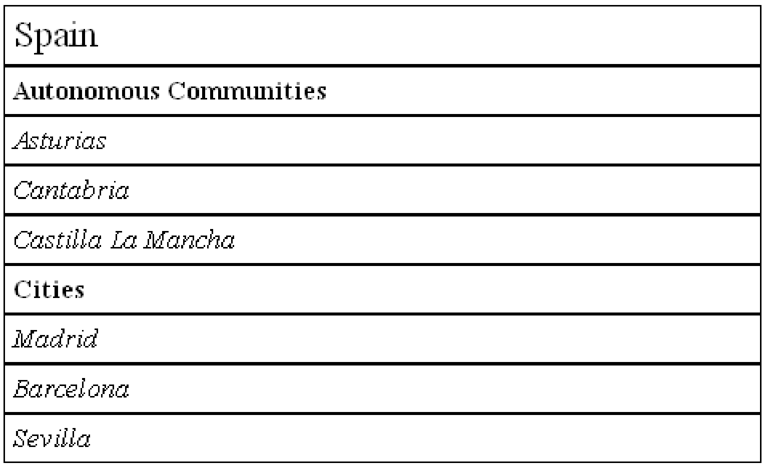


Spain
Autonomous Communities
Asturias
Cantabria
Castilla La Mancha
Cities
Madrid
Barcelona
Sevilla

Of course, tree tables can be constructed without `<table>`:

```
<style type="text/css">
    .freeTable
    {
        border: 1px solid; margin:0; padding:5px;
    }
</style>

...

<div id="treeId">
    <p class="freeTable">Label</p>
</table>
```

That renders (using the previous Java code):



Spain
Autonomous Communities
Asturias
Cantabria
Castilla La Mancha
Cities
Madrid
Barcelona
Sevilla

6.19.5 Trees and "UsePatternMarkupToRender" feature

`ElemenTreeNode` objects support the "UsePatternMarkupToRender" feature, if set to true (default is false) the original markup of the content of the label element is ever used to render a new value.

6.20 VARIABLE RESOLVER

An ItsNat strong principle is to disconnect markup from Java code, DOM methods like `Document`.**`getElementById`** or `ItsNatTreeWalker`.**`getFirstChildElement`** are very useful to make Java code tolerant to markup changes (location changes mainly); these methods are DOM `Element` centric. ItsNat provides a new utility, `ItsNatVariableResolver`, to locate/replace text marks (variables) inside text based nodes (text nodes, comments, attribute values etc) without knowing the exact position in the DOM tree.

The variable syntax follows the JSP Expression Language notation:

```
${name}
```

Where `name` is the variable name. This declaration is replaced with the variable value if resolved.

Variable resolution is not performed automatically by ItsNat, the developer must specify what DOM tree part is resolved and what is the variable repository: including request parameters, session variables etc.

Example:

```
<div id="remoteCtrl">
    ...
    <a href="servlet?itsnat_remctrl=true
           &itsnat_refresh_method=timer
           &itsnat_session_id=${sessionId}
           &itsnat_doc_id=${docId}
           &itsnat_refresh_interval=${refreshInterval}
           &itsnat_syncmode=${syncMode}" target="_blank">
        ${linkText}
    </a>
    ...
    </div>
```

This markup fragment contains a link used to launch a "remote view/control" page to monitor "this" page. This link contains several ItsNat variables, these variables need to be resolved before the link is shown.

This is the Java code used to resolve these variables:

```
final ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
ItsNatVariableResolver resolver = itsNatDoc.createItsNatVariableResolver();
ClientDocument owner = itsNatDoc.getClientDocumentOwner();
ItsNatHttpSession itsNatSession = (ItsNatHttpSession)owner. getItsNatSession();
HttpSession session = itsNatSession.getHttpSession();
session.setAttribute("sessionId",itsNatSession.getId());
itsNatDoc.setAttribute("docId",itsNatDoc.getId());
resolver.setLocalVariable("refreshInterval",new Integer(3000));
resolver.setLocalVariable("syncMode",new Integer(itsNatDoc.getDefaultSyncMode()));
resolver.setLocalVariable("linkText","Click to monitor your page");

Element div = doc.getElementById("remoteCtrl");
resolver.resolve(div);
```

The call `itsNatDoc.`**`createItsNatVariableResolver`**`()` creates a new `ItsNatVariableResolver` object with scope: document-session; name-value pairs bound to the `HttpSession` using `HttpSession.`**`setAttribute`** and to the `ItsNatDocument` using `ItsNatDocument.`**`setAttribute`** are "visible" (located) from this object. `ItsNatVariableResolver` admits local variable using `ItsNatVariableResolver.`**`setLocalVariable`**, these variables are only visible using this object.

The call `resolver.`**`resolve`**`(div)` traverses the specified DOM subtree replacing any `${}` construction inside text nodes and attribute values with the associated value if found (if not found no action is performed).

For instance, this is the HTML output sent to client after variable resolution:

```
<div id="remoteCtrl">
    ...
    <a href="servlet?itsnat_remctrl=true&itsnat_refresh_method=timer
           &itsnat_session_id=1179308955078_2&itsnat_doc_id=1179310638515_1
           &itsnat_refresh_interval=3000&itsnat_syncmode=2" target="_blank">
        Click to monitor your page
    </a>
    ...
</div>
```

An `ItsNatVariableResolver` object can have several scopes to resolve variables: another variable resolver, request, document, servlet session and servlet context:

- `ItsNatServletContext.`**`createItsNatVariableResolver`**`()`

  Creates a new resolver with the scope: locals + context attributes (name-values registered using `ServletContext.setAttribute(String,Object)`) in this order.

- `ItsNatSession.`**`createItsNatVariableResolver`**`()`

Creates a new resolver with the scope: locals + session attributes + context attributes in this order.

- `ItsNatDocument.`**`createItsNatVariableResolver`**`()`

  Creates a new resolver with the scope: locals + document attributes + session attributes + context attributes in this order.

- `ItsNatServletRequest.`**`createItsNatVariableResolver`**`()`

  Creates a new resolver with the scope: locals + request parameters/attributes + document attributes + session attributes + context attributes in this order.

- `ItsNatVariableResolver.`**`createItsNatVariableResolver`**`()`

  Creates a new resolver with the scope: locals + parent scope in this order.

Now we are ready to simplify the custom renderer seen on chapter "DOM RENDERERS" using `ItsNatVariableResolver`:

```
<table id="elementId2" border="1px" cellspacing="0" cellpadding="10px">
    <tbody>
        <tr><td>Year:</td><td>${year}</td></tr>
        <tr><td>Month:</td><td>${month}</td></tr>
        <tr><td>Day:</td><td>${day}</td></tr>
    </tbody>
</table>
```

```
ElementRenderer customRenderer = new ElementRenderer()
{
    public void render(Object userObj,Object value,Element elem, boolean isNew)
    {
        DateFormat format =
           DateFormat.getDateInstance(DateFormat.LONG,Locale.US);
        // Format: June 8,2007
        String date = format.format(value);
        int pos = date.indexOf(' ');
        String month = date.substring(0,pos);
        int pos2 = date.indexOf(',');
        String day = date.substring(pos + 1,pos2);
        String year = date.substring(pos2 + 1);

        ItsNatVariableResolver resolver = itsNatDoc.createItsNatVariableResolver();
        resolver.setLocalVariable("year",year);
        resolver.setLocalVariable("month",month);
        resolver.setLocalVariable("day",day);
        resolver.resolve(elem);
    }

    public void unrender(Object userObj,Element elem)
    {
    }
};

customRenderer.render(null,new Date(),doc.getElementById("elementId2"), true);
```

Now the custom renderer is markup independent (of course this example does not need a custom renderer class because is called directly).

Furthermore ItsNat variable resolvers support JavaBeans conventions and introspection:

```
<table id="elementId3" border="1px" cellspacing="0" cellpadding="10px">
    <tbody>
        <tr><td>First Name:</td><td>${person.firstName}</td></tr>
        <tr><td>Last Name:</td><td>${person.lastName}</td></tr>
        <tr><td>Age:</td><td>${person.age}</td></tr>
        <tr><td>Married:</td><td>${person.married}</td></tr>
    </tbody>
</table>


ItsNatDocument itsNatDoc = ...;

Object value = new PersonExtended("John","Smith",30,true);

Document doc = itsNatDoc.getDocument();
ItsNatVariableResolver resolver = itsNatDoc.createItsNatVariableResolver();
resolver.introspect("person",value);

Element elem = doc.getElementById("elementId3");
resolver.resolve(elem);
```

The class `PersonExtended`:

```
public class PersonExtended extends Person
{
    protected int age;
    protected boolean married;

    public PersonExtended(String firstName,String lastName,int age,boolean married)
    {
        super(firstName,lastName);

        this.age = age;
        this.married = married;
    }

    public int getAge()
    {
        return age;
    }

    public void setAge(int age)
    {
        this.age = age;
    }

    public boolean isMarried()
    {
        return married;
    }

    public void setMarried(boolean married)
    {
        this.married = married;
    }
}
```

The most interesting line is:

```
    resolver.introspect("person",value);
```

This call uses the specified prefix and `Introspector.getBeanInfo(Class)` to add the bean properties as local variables.

6.20.1 Variables and cache

ItsNat avoids caching a node is this contains an attribute or text node with a `${}` construction.

6.20.2 Using variable resolvers and UsePatternMarkupToRender feature

The "UsePatternMarkupToRender" feature of labels, lists, tables and tree nodes are very useful used along with variable resolvers because if this feature is set to true (by default is false), the original markup/pattern of the "label" (the markup content used to render Java values in labels, lists, tables, and tree nodes) is ever replaced over the current content before a new Java value is rendered, then if used variables these variables may be resolved using a `ItsNatVariableResolver` on the renderer.

Example:

```
    <p id="elementId"><b><i>${variable_to_resolve}</i></b></p>


    final ItsNatDocument itsNatDoc = ...;
    Document doc = itsNatDoc.getDocument();
    ElementLabelRenderer renderer = new ElementLabelRenderer()
    {
        protected ItsNatVariableResolver resolver =
                itsNatDoc.createItsNatVariableResolver();

        public void renderLabel(ElementLabel label,Object value,
                Element elem, boolean isNew)
        {
            resolver.setLocalVariable("variable_to_resolve",value);
            resolver.resolve(elem);
        }

        public void unrenderLabel(ElementLabel label,Element elem)
        {
        }
    };
    ElementGroupManager factory = itsNatDoc.getElementGroupManager();
    ElementLabel label =
           factory.createElementLabel(doc.getElementById("elementId"),true,renderer);
```

At the moment the current DOM tree is (`removePattern` parameter was true):

```
    <p id="elementId"></p>
```

In spite of the original content markup was saved as pattern internally.

```
label.setLabelValue("First Value");
```

Renders:

```
<p id="elementId"><b><i>First Value</i></b></p>
```

Because the pattern was used to render this first value. Now new calls to `setElementValue` use the current renderer markup this does not work with our renderer because we use variables, unless the "UsePatternMarkupToRender" is set to true.

```
label.setUsePatternMarkupToRender(true);
label.setLabelValue("Second Value");
```

This renders:

```
<p id="elementId"><b><i>Second Value</i></b></p>
```

And finally:

```
label.setLabelValue("Third Value");
```

Renders:

```
<p id="elementId"><b><i>Third Value</i></b></p>
```

Previous example applied to labels can be rewritten using lists, tables or tree nodes.

6.21 W3C ELEMENTCSSINLINESTYLE IMPLEMENTATION

ItsNat provides a partial Java implementation[44] of the W3C Style/CSS[45] `ElementCSSInlineStyle` interface[46] because Xerces does not implements it. This interface is very useful to break into parts a complicated style declaration and modify a concrete style property. Modification capability is not mandatory by W3C in `ElementCSSInlineStyle`, ItsNat allows modification because is the most useful aspect of this interface. ItsNat `ElementCSSInlineStyle` implementation is tolerant to collateral modification of the "style" property like using `Element.`**`setAttribute`** method.

The method `ItsNatDocument.`**`getItsNatNode`**`(Node)` is used to obtain an `ElementCSSInlineStyle` object wrapping the original Xerces `Element`.

The following example shows a link that changes its border color when clicked:

```
<a id="linkId" href="javascript:void(0)"
   style="padding:10px; border: 1px solid rgb(100,150,255);">
   Click me to change border color
</a>
```

[44] Cursor syntax is not implemented

[45] http://www.w3.org/TR/DOM-Level-2-Style/

[46] http://www.w3.org/2003/01/dom2-javadoc/org/w3c/dom/css/ElementCSSInlineStyle.html

```
public void handleEvent(Event evt)
{
    Element currTarget = (Element)evt.getCurrentTarget(); // Link

    ItsNatDocument itsNatDoc = ((ItsNatEvent)evt).getItsNatDocument();
    ElementCSSInlineStyle style =
         (ElementCSSInlineStyle)itsNatDoc.getItsNatNode(currTarget);
    CSSStyleDeclaration cssDec = (CSSStyleDeclaration)style.getStyle();

    CSSValueList border = (CSSValueList)cssDec.getPropertyCSSValue("border");
    int len = border.getLength();
    String cssText = "";
    for(int i = 0; i < len; i++)
    {
        CSSValue value = border.item(i);
        if (value.getCssValueType() == CSSValue.CSS_PRIMITIVE_VALUE)
        {
            CSSPrimitiveValue primValue = (CSSPrimitiveValue)value;
            if (primValue.getPrimitiveType() == CSSPrimitiveValue.CSS_RGBCOLOR)
            {
                RGBColor rgb = primValue.getRGBColorValue();
                System.out.println("Current border color: rgb(" +
                   rgb.getRed().getCssText() + "," +
                   rgb.getGreen().getCssText() + "," +
                   rgb.getBlue().getCssText() + ")");
            }
            else cssText += primValue.getCssText() + " ";
        }
        else cssText += value.getCssText() + " ";
    }
    cssDec.setProperty("border",cssText,null); // Removed border color

    CSS2Properties cssDec2 = (CSS2Properties)style.getStyle();
    String newColor = "rgb(255,100,150)";
    cssDec2.setBorderColor(newColor); // border-color property
    System.out.println("New border color: " + cssDec2.getBorderColor());
}
```

Previous example shows how to use some W3C CSS interfaces and finally to change the border color of the link when clicked.

6.22 REMOTE VIEW/CONTROL

Remote view/control is an IsNat "out of the box" feature, is a direct consequence of the TBITS (The Browser Is The Server) approach. ItsNat allows binding a browser window to an existing DOM document; this window becomes a remote viewer, very useful to monitor other user actions. The observer window loads the current DOM server state being observed, and may be associated to a different web (servlet) session from the observed document.

ItsNat provides a security system to avoid or deny remote view/control requests; ItsNat remote view/control system is not active by default.

Current ItsNat implementation does not permit to send events from a "cloned" view (only the originator window can, as a normal web application), two or more windows can not interact with the same server DOM tree. But ItsNat offers some kind of remote control: every remote viewer refresh event can optionally be caught by a server listener; this Java listener can do anything including changing the server DOM tree (this is because the "control" word is used in this context).

Browser correspondence is not mandatory, a page open by a browser X can be monitored by a browser Y where X and Y are supported browsers. Both client pages share the same server DOM tree but client-server synchronization tasks are isolated by ItsNat adapted to concrete target client. User JavaScript code sent to the client calling methods like `ItsNatDocument.addCodeToSend(Object)` is sent to the monitoring client too as is.

Remote view/control of a concrete "alive" document may be requested using a URL (a GET request). The chapter "VARIABLE RESOLVER" showed how to construct a URL to self-monitor a page using a timer. These are the mandatory and optional URL parameters to request a document remote view/control:

- **`itsnat_remctrl`**`=true`

  Specifies a remote view/control request, ever set to true.

- **`itsnat_refresh_method`**`=timer | comet`

  Specifies the refresh method, a timer or comet.

- **`itsnat_session_id`**`=`*`sessionToMonitorId`*

  Specifies the session id of the document target to monitor.

- **`itsnat_doc_id`**`=`*`docToMonitorId`*

  The id of the document to monitor

- **`itsnat_refresh_interval`**`=`*`milliseconds`*

  Refresh interval in milliseconds, only if specified a timer refresh.

- **`itsnat_syncmode`**`=`*`syncModeIntValue`*

  Synchronous mode of refresh events, this integer value must be one of the following: `SyncMode.ASYNC` (1), `SyncMode.ASYNC_HOLD` (2) or `SyncMode.SYNC` (3). If not specified the default value of the target document being monitored is used.

- **`itsnat_ajaxtimeout`**`=`*`AJAXTimeoutIntValue`*

  The maximum time in milliseconds any AJAX event (timer refresh or COMET updater event) will wait to the server before to abort the request (the remote control session is stopped). If not specified the default value of the target document being monitored is used. A negative value means no timeout.

For instance using a timer:

http://localhost:8080/myapp/servlet?
itsnat_remctrl=true&itsnat_refresh_method=timer&itsnat_session_id=1179315974562_3&it

snat_doc_id=1179319387515_1&itsnat_refresh_interval=3000&itsnat_syncmode=2&itsnat_ajaxtimeout=-1

The same using comet:

http://localhost:8080/myapp/servlet?itsnat_remctrl=true&itsnat_refresh_method=comet&itsnat_session_id=1179315974562_3&itsnat_doc_id=1179319387515_1&itsnat_syncmode=2&itsnat_ajaxtimeout=-1

To enable remote view/control, a supervisor request listener implementing `RemoteControlEventListener`, must be registered. This listener receives `RemoteControlEvent` event objects. A `RemoteControlEvent` event object has two subtypes: `RemoteControlTimerEvent` if refresh method is `timer` and `RemoteControlCometEvent` if specified `comet`.

For instance:

```
public class RemoteControlSupervision implements RemoteControlEventListener
{
    public RemoteControlSupervision()
    {
    }

    public void handleRequest(RemoteControlEvent event)
    {
        int phase = event.getPhase();

        if (phase == RemoteControlEvent.REQUEST)
        {
            if (event instanceof RemoteControlTimerEvent)
            {
                RemoteControlTimerEvent timerEvent =
                        (RemoteControlTimerEvent)event;
                boolean accepted = (timerEvent.getRefreshInterval() >= 3000);
                event.setAccepted(accepted);
            }
            else if (event instanceof RemoteControlCometEvent)
            {
                event.setAccepted(true);
            }
        }
        else if (phase == RemoteControlEvent.OBSERVED_INVALID)
        {
            ItsNatServletRequest request = event.getItsNatServletRequest();
            ClientDocument observer = request.getClientDocument();
            observer.addCodeToSend("alert('Observed document was destroyed');");
        }
        else if (phase == RemoteControlEvent.REFRESH)
        {
            ItsNatServletRequest request = event.getItsNatServletRequest();
            ClientDocument observer = request.getClientDocument();
            long initTime = observer.getCreationTime();
            long currentTime = System.currentTimeMillis();
            long limitMilisec = 15*60*1000;
                        // 15 minutes (to avoid a long monitoring session)
            if (currentTime - initTime > limitMilisec)
            {
                event.setAccepted(false);
                observer.addCodeToSend("alert('Remote Control Timeout');\n");
            }
        }
        // RemoteControlEvent.LOAD & UNLOAD : nothing to do
    }
}
```

This complete example shows the typical code used to control a remote view life cycle.

The remote view life cycle has the following phases:

- `RemoteControlEvent.`**`REQUEST`**

  The remote viewer requests to be attached to the specified session/document.

- `RemoteControlEvent.`**`LOAD`**

  The request was accepted and document state is going to be sent to the viewer.

- `RemoteControlEvent.`**`REFRESH`**

  The viewer sent a refresh event (timer) or the observed document notifies a change (comet).

- `RemoteControlEvent.`**`UNLOAD`**

  The user has closed the remote viewer.

- `RemoteControlEvent.`**`OBSERVED_INVALID`**

  The original document was unloaded (closed the "owner" window) but the remote viewer remains open.

By default any remote view request is not accepted, this sentence in the `RemoteControlEvent.`**`REQUEST`** phase tells ItsNat if the request is accepted:

```
event.setAccepted(accepted);
```

The remaining code is called when other phases occur, and only if the remote view/control request was accepted. This code shows how to limit the monitoring time if using a timer and how to notify the "spy" user if the original window page was closed (document destroyed). The `ClientDocument` object represents the browser window/page mirroring the DOM server tree.

To receive events targeted to a `RemoteControlEventListener` object, this object must be registered.

```
RemoteControlSupervision remCtrlSup = new RemoteControlSupervision();
```

Two options to register a remote control listener: globally, per document template or per document.

1. Globally registered: registering at servlet level

   ```
   ItsNatServlet servlet = ...;
   servlet.addRemoteControlEventListener(remCtrlSup);
   ```

2. Per document template: registering per document template

   ```
   DocumentTemplate docTemplate = ...;
   ```

```
docTemplate.addRemoteControlEventListener(remCtrlSup);
```

3. Per document

```
ItsNatDocument itsNatDoc = ...;
itsNatDoc.addRemoteControlEventListener(remCtrlSup);
```

6.22.1 Controlling other users/sessions

To control other pages loaded on other sessions we need the session and document ids. ItsNat provides some methods to navigate across the ItsNat page system.

The first one method is:

```
ItsNatServletContext.enumerateSessions(ItsNatSessionCallback)
```

This method enumerates all `ItsNatSession` sessions running on the web application, calling the callback specified per session.

The other one is:

```
ItsNatSession.getItsNatDocuments()
```

This method returns all `ItsNatDocument` objects alive at the specified session.

The following example lists all documents with the same template (same "page") as the caller:

```
<table border="1px" cellspacing="0" cellpadding="5px">
    <thead><tr><th>Session/doc Ids</th><th>User Agent</th>
               <th>Using a Timer</th><th>Using Comet</th></thead>
    <tbody id="otherSessionsId">
        <tr><td>${sessionId}/<br/>${docId}</td><td>${agentInfo}</td>
            <td><a href="servlet?itsnat_remctrl=true
                   &itsnat_refresh_method=timer
                   &itsnat_session_id=${sessionId}&itsnat_doc_id=${docId}
                   &itsnat_refresh_interval=${refreshInterval}
                   &itsnat_syncmode=${syncMode}&itsnat_ajaxtimeout=-1"
                   target="_blank">Link</a></td>

            <td><a href="servlet?itsnat_remctrl=true
                   &itsnat_refresh_method=comet
                   &itsnat_session_id=${sessionId}&itsnat_doc_id=${docId}
                   &itsnat_syncmode=${syncMode}&itsnat_ajaxtimeout=-1"
                   target="_blank">Link</a></td>
        </tr>
    </tbody>
</table>


ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
DocumentTemplate thisDocTemplate = itsNatDoc.getDocumentTemplate();

ItsNatServlet itsNatServlet = itsNatDoc.getDocumentTemplate().getItsNatServlet();
ItsNatServletContext appCtx =
                   itsNatServlet.getItsNatServletConfig().getItsNatServletContext();

final List sessionList = new LinkedList();
ItsNatSessionCallback cb = new ItsNatSessionCallback()
```

```
    {
        public boolean handleSession(ItsNatSession session)
        {
            sessionList.add(session);
            return true; // continue
        }
    };
    appCtx.enumerateSessions(cb);

    ElementGroupManager factory = itsNatDoc.getElementGroupManager();
    ElementList sessionNodeList =
                factory.createElementList(doc.getElementById("otherSessionsId"),true);

    ItsNatVariableResolver resolver = itsNatDoc.createItsNatVariableResolver();
    resolver.setLocalVariable("refreshInterval",new Integer(3000));
    resolver.setLocalVariable("syncMode",new Integer(itsNatDoc.getDefaultSyncMode()));

    for(int i = 0; i < sessionList.size(); i++)
    {
        ItsNatHttpSession otherSession = (ItsNatHttpSession)sessionList.get(i);

        ItsNatDocument[] remDocs = otherSession.getItsNatDocuments();

        for(int j = 0; j < remDocs.length; j++)
        {
            ItsNatDocument currRemDoc = remDocs[j];
            if (itsNatDoc == currRemDoc) continue;
            String id;
            synchronized(currRemDoc)
            {
                DocumentTemplate docTemplate = currRemDoc.getDocumentTemplate();
                if (docTemplate != thisDocTemplate)
                    continue;
            }

            String docId = currRemDoc.getId(); // No sync is needed
            Element sessionElem = (Element)sessionNodeList.addElement();

            ItsNatVariableResolver resolver2 = resolver.createItsNatVariableResolver();
            resolver2.setLocalVariable("sessionId",otherSession.getId());
            resolver2.setLocalVariable("agentInfo",otherSession.getUserAgent());
            resolver2.setLocalVariable("docId",docId);
            resolver2.resolve(sessionElem);
        }
    }
```

6.22.2 Comet limitations

ItsNat internally uses a comet notifier per remote view, to avoid a browser hang open only a remote viewer. See "COMET NOTIFIER" for more info.

6.23 JAVASCRIPT GENERATION UTILITIES

ItsNat provides some utilities to convert Java DOM code to JavaScript code to be sent to the client (coding JavaScript in server side). For instance, we need to generate JavaScript code to call a DOM element method or to access a property. The interface `ScriptUtil` can be used to do this, `ScriptUtil` provides some methods to convert a Java `Node` reference to a JavaScript reference, to convert a Java String to a "transportable" JavaScript string etc. An object implementing ScriptUtil can be obtained calling ItsNatDocument.**getScriptUtil**().

An example:

```
<input type="text" id="inputElemId" size="30" value="" />


ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();

HTMLInputElement inputElem =
             (HTMLInputElement)doc.getElementById("inputElemId");

ScriptUtil scriptGen = itsNatDoc.getScriptUtil();

String code;
String msg = "A Java String transported as a JavaScript string";
String inputElemJS = scriptGen.getNodeReference(inputElem);
String newValue = scriptGen.getTransportableStringLiteral(msg);
code = inputElemJS + ".value = " + newValue + ";";

itsNatDoc.addCodeToSend(code);

code = scriptGen.getSetPropertyCode(inputElem,"value",msg,true);
itsNatDoc.addCodeToSend(code);

code = scriptGen.getCallMethodCode(inputElem,"select",null,true);
itsNatDoc.addCodeToSend(code);
```

Previous Java code sets a new value to the <input> element and selects the content. This is the generated JavaScript code sent to the client[47]:

```
itsNatDoc.getNodeFromPath("1186161948265_18","1186161948265_1642","0,0").value
      = "A Java String transported as a JavaScript string";
itsNatDoc.getNodeFromPath(null,"1186161948265_1642",null).value
      = "A Java String transported as a JavaScript string";
itsNatDoc.getNodeFromPath(null,"1186161948265_1642",null).select();
```

6.24 EVENT MONITORS

Any desktop application has some kind of cursor "wait mode" when the event dispatcher thread is busy. When the browser connects to the server using AJAX (XMLHttpRequest) no visual notification is shown to the user. Typical AJAX based frameworks offer a default "wait" notification, usually an animated image, ItsNat does not offer an "out of the box" notification. ItsNat is consistent with its philosophy: "wait" notification is pluggable.

A user defined JavaScript based listener can be registered in the JavaScript based document of ItsNat. This listener must be defined using an object oriented style and is called when an event is going to be sent (before method is called) and when the response is received (after method is called).

The following example shows how to define an event monitor and register in ItsNat document using the method addEventMonitor. This monitor shows/hides a text message:

[47] This code is implementation detail and may change

```
<script type="text/javascript">
function EventMonitor()
{
    this.before = before;
    this.after = after;

    this.monitor = document.getElementById("monitorId");
    this.count = 0;

    function before(evt)
    {
        if (this.count == 0)
            this.monitor.style.display = "";

        this.count++;
    }

    function after(evt,timeout)
    {
        if (this.count == 0)
           return; // to avoid some pending events before registering

        this.count--;

        if (this.count == 0)
            this.monitor.style.display = "none";

       if (timeout) alert("AJAX Event Timeout!!");
    }
}
document.monitor = new EventMonitor();
document.getItsNatDoc().addEventMonitor(document.monitor);
</script>

<span id="monitorId" style="color:white; background:red; display:none;">
Wait a moment!</span>
```

The parameter `evt` is an internal ItsNat event object and is not documented, if this event is a DOM event you can call `evt.getEvent()` to obtain the "real" native event object. The boolean parameter `timeout` is true if AJAX request was aborted due to a timeout.

Following the example, to remove the monitor use `removeEventMonitor`:

```
<script type="text/javascript">
document.getItsNatDoc().removeEventMonitor(document.monitor);
</script>
```

And finally the method `setEnableEventMonitors` temporally enables/disables event monitoring (if disabled monitors are not notified about events, monitors keep registered). For instance to disable monitoring:

```
document.getItsNatDoc().setEnableEventMonitors(false);
```

6.25 EVENTS FIRED BY THE SERVER (SERVER-SENT EVENTS)

ItsNat converts browser events in W3C Java events, this is the normal behavior. But ItsNat supports the opposite, Java W3C events fired and sent to the client converted to browser events and dispatched to the browser using `dispatchEvent` (W3C browsers) or `fireEvent` (MSIE); these events usually are dispatched again to the server listeners completing the cycle. Furthermore, events can be dispatched directly to the server DOM tree bypassing the browser.

Server-sent events feature completes the TBITS philosophy (the server fires and receives events) and opens ItsNat to uncommon uses, for instance:

1. AJAX bookmarking: where the server simulates user actions to drive the application to the desired state.

2. Server Driven Web Testing (SDWT): where test code simulates user actions using Java with no need of external tools.

The protagonist method is `EventTarget.`**`dispatchEvent`**`(Event)`, this method is defined in Xerces implementation, the Xerces implementation performs internal dispatching to listeners registered with `EventTarget.`**`addEventListener`**`(String,EventListener,boolean)`, but this is not the desired behaviour because our events are "remote" and must travel to the browser. ItsNat reimplements the `EventTarget` interface using a wrapper object on top the original Xerces object[48], this wrapper is the same as the one implementing `ElementCSSInlineStyle` and can be got calling `ItsNatDocument.`**`getItsNatNode`**`(Node)`. This method returns an `ItsNatNode` object, this object implements `org.w3c.dom.Node` and `org.w3c.dom.events.EventTarget`, and `org.w3c.dom.Element` and `org.w3c.dom.css.ElementCSSInlineStyle` if it is an element; use this interface and `instanceof` to distinguish between a Xerces node and a wrapper.

An alternative method is `ItsNatDocument.`**`dispatchEvent`**`(EventTarget,Event)`, the target parameter may be the original Xerces object, in fact, `EventTarget.`**`dispatchEvent`**`(Event)` simply calls the document version.

There are two scenarios:

1) Events sent to the browser simulating user actions

2) Events directly dispatched to the server DOM tree

6.25.1 Events sent to the browser simulating user actions

ItsNat provides a special method:

`ClientDocument.`**`startEventDispatcherThread`**`(Runnable)`

This method executes the specified code in a new thread controlled by ItsNat. The code executed in this thread is ready to call:

```
EventTarget.dispatchEvent(Event)  or
ItsNatDocument.dispatchEvent(EventTarget,Event) or

ClientDocument.dispatchEvent(EventTarget,Event,int,long)
```

[48] Other methods like EventTarget.addEventListener(String,EventListener,boolean) are implemented ala ItsNat too; current implementation calls the appropriated ItsNatDocument.addEventListener method using "this" as the target.

many times. This "dispatcher" thread represents the "user sequential behaviour". The method `startEventDispatcherThread` **must** be called using a normal servlet-request thread.

These methods send the specified event to the browser and wait until is dispatched calling `dispatchEvent` (W3C browsers) or `fireEvent` (MSIE) on the browser; the Java call returns when the event is processed by the browser[49]. This synchronous behaviour is of course accomplished using several threads (the caller thread is stopped waiting the browser response sent in a new thread). If the thread calling any `dispatchEvent` method is a normal ItsNat servlet-request thread, the `ItsNatDocument` is already locked and then no other thread can be used to transport the browser response to the server. So dispatch methods can not be called using a servlet-request thread because this thread is going to be stopped, this is the reason of the new thread created by `startEventDispatcherThread` this new thread does not need to lock the `ItsNatDocument` to dispatch server events (in fact if the `ItsNatDocument` is locked by the caller thread this will hang indefinitely[50]). Furthermore `startEventDispatcherThread` (called by a servlet-request thread) is used to tell the browser to automatically request the server again to send any server-dispatched event to the browser.

The last piece missing is how we can create DOM events, DOM events are created with `DocumentEvent.createEvent(String type)`. Xerces implementation is poor because it does not support `MouseEvent` events; ItsNat provides an alternative DOM Level 2 implementation of `DocumentEvent`, the `ItsNatDocument` object implements this interface, a simple cast obtains the ItsNat `DocumentEvent` version.

The following example shows how to call three buttons sequentially with no user interaction (the user must start the test by clicking "`Start`").

```
<a id="linkId" href="javascript:void(0)">Start</a>
<p>
    <input id="buttonId0" type="button" value="Button 1" /> &nbsp;
    <input id="buttonId1" type="button" value="Button 2" /> &nbsp;
    <input id="buttonId2" type="button" value="Button 3" /> &nbsp;
    <input id="userButtonId" type="button" value="User Event" />
</p>


ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();

final Element linkElem = (Element)doc.getElementById("linkId");

final Element[] buttonElems = new Element[3];
for(int i = 0; i < buttonElems.length; i++)
    buttonElems[i] = doc.getElementById("buttonId" + i);

final Element userButton = doc.getElementById("userButtonId");

EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        EventTarget currTarget = evt.getCurrentTarget();
```

[49] An event is processed by the browser when dispatchEvent or fireEvent method returns, any asynchronous AJAX request sent while dispatching the event may be unfinished.

[50] Current implementation detects this scenario to avoid dead lock errors.

```
        if (currTarget == linkElem)
        {
            ItsNatEvent itsNatEvt = (ItsNatEvent)evt;
            final ItsNatDocument itsNatDoc = itsNatEvt.getItsNatDocument();
            Document doc = itsNatDoc.getDocument();
            final org.w3c.dom.events.DocumentEvent docEvent =
                    (org.w3c.dom.events.DocumentEvent)itsNatDoc;

            Runnable dispCode = new Runnable()
            {
              public void run()
              {
                for(int i = 0; i < buttonElems.length; i++)
                {
                  Element buttonElem;
                  EventTarget buttonElemEnh;
                  MouseEvent mouseEvt;
                  synchronized(itsNatDoc)
                  {
                    AbstractView view =
                           (DocumentView)itsNatDoc).getDefaultView();
                    mouseEvt =(MouseEvent)docEvent.createEvent("MouseEvents");
                    mouseEvt.initMouseEvent("click",true,true,view,0,
                       0,0,0,0,false,false,false,false,(short)0/*left button*/,null);

                    buttonElem = buttonElems[i];
                    buttonElemEnh =
                           (EventTarget)itsNatDoc.getItsNatNode(buttonElem);
                  }

                  buttonElemEnh.dispatchEvent(mouseEvt);
                  // Alternative:
               // itsNatDoc.dispatchEvent((EventTarget)buttonElem,mouseEvt);
                }

                EventTarget userButtonEnh;
                UserEvent userEvt;
                synchronized(itsNatDoc)
                {
                  userEvt =
                       (UserEvent)docEvent.createEvent("itsnat:UserEvents");
                  userEvt.initEvent("itsnat:user:myEvent",false,false);

                  userButtonEnh =
                           (EventTarget)itsNatDoc.getItsNatNode(userButton);
                }

                userButtonEnh.dispatchEvent(userEvt);
              }
            };
            ClientDocument client = itsNatDoc.getClientDocumentOwner();
            client.startEventDispatcherThread(dispCode);
        }
        else
        {
            System.out.println("Clicked: " +
                             ((Element)currTarget).getAttribute("value"));
        }
    }
};
```

```
itsNatDoc.addEventListener((EventTarget)linkElem,"click",listener,false);

for(int i = 0; i < buttonElems.length; i++)
    itsNatDoc.addEventListener((EventTarget)buttonElems[i],
            "click",listener,false);

itsNatDoc.addUserEventListener((EventTarget)userButton,"myEvent",listener);
userButton.setAttribute("onclick",
    "document.getItsNatDoc().fireUserEvent(this,'myEvent');");
```

Inside the dispatcher thread any access to the `ItsNatDocument` **must** be synchronized because this is not a normal servlet-request thread so the `ItsNatDocument` is not locked. This rule does not apply to `dispatchEvent` calls (in fact if the document is locked an exception is thrown).

The user event example is interesting because it shows how we can create ItsNat user events with standard W3C methods:

```
userEvt = (UserEvent)docEvent.createEvent("itsnat:UserEvents");
userEvt.initEvent("itsnat:user:myEvent",false,false);
```

Where:

"`itsnat:UserEvents`" or "`itsnat:UserEvent`" is the "family" event type.

"`itsnat:user:`*`eventName`*" is the concrete event type. The event name is the same used in listeners.

The "Feature Showcase" includes a basic example and a complete example of functional testing using components.

6.25.2 Events directly dispatched to the server DOM tree

If ItsNat simulates a W3C Java Browser on the server… why cannot ItsNat avoid the browser? Yes, it can.

Only the dispatch method

```
ClientDocument.dispatchEvent(EventTarget,Event,int,long)
```

ever sends the specified event to the browser, this method *must* be called by a "dispatcher thread" created with `ClientDocument.`**`startEventDispatcherThread`**`(Runnable)`, other dispatch methods, `EventTarget.`**`dispatchEvent`**`(Event)` and `ItsNatDocument.`**`dispatchEvent`**`(EventTarget,Event)` detect if the calling thread is a special dispatcher thread. If the calling thread is not a dispatcher thread then the call is redirected to:

```
ItsNatDocument.dispatchEventLocally(EventTarget,Event)
```

This method dispatches directly the specified event to the DOM server routing the event and calling the registered listeners as specified in the W3C DOM Events standard including bubbling and capturing. The code is the same as the browser version, but no special threads and document locks are needed, a normal servlet-request thread may be used and events are ever dispatched synchronously (no timeouts, synchronous modes, waits etc involved).

The "Feature Showcase" again includes a basic example of server only functional testing and a complete example using components.

6.25.3 More about testing

The main use of events fired by the server is functional testing. ItsNat is strongly based on AJAX, AJAX applications may be hard to test because AJAX is usually used asynchronously; an asynchronous browser event may be processed (`dispatchEvent` and `fireEvent` return) but the response is received later. ItsNat provides the "hold" mode to minimize the indeterminist behavior, this is sufficient to web applications (events are processed sequentially) but not for testing because the dispatch call returns before the event is processed. To deal with this problem there are several approaches[51]:

1) Using synchronous mode when testing (this is absolutely easy with ItsNat because synchronous mode can be defined globally).

2) Wait an amount of time per event fired.

3) Wait until the required change occurs.

ItsNat environment is very powerful to the third option because DOM changes (or component model changes, etc) occur on the server side, and tests are executed on the server side, this is a strong advantage over the typical external tool, furthermore, the server based DOM manipulation is in fact a form of browser indirect manipulation, for instance, key stroke simulation to write in a input text box does not work in Internet Explorer (MSIE requires real user interaction to change the control), changing the `value` attribute on the server DOM automatically updates the `value` property (and attribute) on the client (basically the same as user writing), the "Feature Showcase" functional testing examples with components show how to do this.

6.25.4 Bookmarking and search engines

Events fired by the server can be used for bookmarking an AJAX application, because user actions can be simulated to transport the application to the desired state. A user defined *permalink* system can be defined in your application, this custom url can be processed to detect the required state to be returned to the user firing from server the appropriated events simulating user actions to bring the application to that desired state. Events dispatched directly to the server are more appropriated because is the faster option because no browser interaction is required. Of course firing events is an approach, you can bring the application to the desired stated directly (modifying the server DOM tree directly etc).

The Feature Showcase has an example showing these three techniques of bookmarking.

Traversal of AJAX applications by search engines is another big problem, events dispatched directly to the server is a very interesting technique because it does not requires browser interaction (no JavaScript needed). Use the "fast load" mode if you want your AJAX application is searchable because most of search engines ignore the JavaScript code.

Again the Feature Showcase shows how to build an AJAX application "search engine" capable.

6.26 FAST AND SLOW LOADING MODES

[51] http://mguillem.wordpress.com/2007/07/24/htmlunit-re-synchronize-ajax-calls-for-simple-deterministic-test-automation/

ItsNat sends a document/page to the client using two modes: fast and slow. By default ItsNat is configured in fast mode, this is the preferred and most of the web applications can work on this mode. Fast/slow mode only affects to the loading phase, because in this phase the initial markup of the page is sent to the client as normal serialized markup, fast and slow modes tell ItsNat to send DOM template changes as serialized markup (fast) or JavaScript (slow); fast and slow are in the browser's point of view (a browser renders faster serialized markup than in JavaScript DOM form). DOM changes performed during an (AJAX) event process are *ever* sent to the client in a JavaScript form.

Fast/slow mode is set using the method `ItsNatServletConfig.`**`setFastMode`**`(boolean)` when initializing the ItsNat servlet. If you want a per template configuration use `DocumentTemplate.`**`setFastMode`**`(boolean)` when registering the template.

6.26.1 Fast mode

When loading in fast mode internal (Xerces) mutation events are disabled, any DOM server tree modification in this phase *is not* converted to JavaScript. When the loading process finishes the final DOM server tree is serialized and sent to the client as markup.

This approach is the faster in a browser's point of view, but has some problems: if a listener is bound to an element in the loading phase, this action is sent to the client as JavaScript code when the loading process finishes, an absolute location path in the tree is used to locate the element; this code is executed when the page is loaded by the browser. If the element was removed or changed its position during the loading process after the binding, the JavaScript code fails to locate the element. This is not a serious problem because elements willing to receive events usually do not change their position. For instance: a component based list automatically binds a listener to the element parent of the list, but no child element has a listener (list content is going to change frequently).

For instance, the following code does not work in fast mode (fails in the browser) because the new node used is finally removed (`addEventListener` fails):

```
ItsNatHTMLDocument itsNatDoc = ...;
HTMLDocument doc = itsNatDoc.getHTMLDocument();
HTMLTableElement tableElem = (HTMLTableElement)doc.createElement("table");
doc.getBody().appendChild(tableElem);

EventListener listener = new EventListener()
{
    public void handleEvent(Event evt) { }
};

itsNatDoc.addEventListener((EventTarget)tableElem,"click",listener,false);

itsNatDoc.removeEventListener((EventTarget)tableElem,"click",listener,false);

doc.getBody().removeChild(tableElem);
```

Note: `removeEventListener` call is not the problem, the JavaScript generated behind the scenes by this method does not use the DOM element (uses an internal listener id), this method can be called with elements already removed from the tree (`addEventListener` counterpart only can be called using a DOM element present in the tree).

6.26.2 Slow mode

When loading in slow mode mutation events are enabled, the document template is sent to the client as is, DOM tree modifications during the loading phase are sent to the client as JavaScript code. In this mode there is no problem with location changes (or removal) on elements with listeners bound because both actions are converted and sent to the client using JavaScript in sequential order. This is the slow mode because page construction is done with JavaScript and JavaScript code takes more time to load and execute. Another minor drawback is: the template is first sent to the client as is (to be modified dynamically using JavaScript), a browser view-source shows this template code; for instance a list has only one element, the pattern element.

The previous example runs ok in slow mode.

6.26.3 How to select the most appropriated mode

Use the fast mode if you can, if fails try to use a "load" listener to do the offending tree modification or listener bindings, if fails again use the slow mode.

6.27 GLOBAL LOAD PROCESSING

Optional global listeners can be registered to process any document/page load request, this listener, if defined, is called before any other request listener. These listeners are user defined `ItsNatServletRequestListener` objects.

The following example registers a global page request listener to output simple log information about the page requested.

```
ItsNatHttpServlet itsNatServlet = ...;

ItsNatServletRequestListener listener = new ItsNatServletRequestListener()
{
    public void processRequest(ItsNatServletRequest request,
                               ItsNatServletResponse response)
    {
        ItsNatDocument itsNatDoc = request.getItsNatDocument();
        if (itsNatDoc != null)
        {
            String docName = itsNatDoc.getDocumentTemplate().getName();
            System.out.println("Loading " + docName);
        }
    }
};

itsNatServlet.addItsNatServletRequestListener(listener);
```

6.27.1 No standar page loading

If the query string (URL part following "?") does not contains `itsnat_doc_name` or `itsnat_doc_id` (nor `itsnat_remctrl` or `itsnat_load_script`), ItsNat can not load a new document/page and dispatch the request to the registered listeners in the selected template. In this case only the servlet level `ItsNatServletRequestListener` listeners are dispatched, as no document is loaded a call to `ItsNatServletRequest.`**`getItsNatDocument`**`()` returns null. This case is basically the same as the standard Java servlet process (overriding `HttpServlet.doGet/doPost` etc).

One very interesting case is to process pretty URLs.

6.27.2 Pretty URLs

ItsNat supports pretty URLs, a pretty URL is a URL easy to remember, for instance:

`http://<host>:<port>/itsnat/`**`page/manual/core/prettyurl`**

is more readable (and shorter) than:

`http://<host>:<port>/itsnat/`**`servlet?itsnat_doc_name=manual.core.prettyurl`**

An example:

Adding the following to the archive `web.xml`:

```
<servlet-mapping>
    <servlet-name>servlet</servlet-name>
    <url-pattern>/page/*</url-pattern>
</servlet-mapping>
```

Any URL starting with `/itsnat/page/…` will be redirected to `servlet` and `HttpServletRequest.`**`getPathInfo`**`()` will return the URL part following the `/page` prefix Completing our global (servlet level) listener:

```
ItsNatServletRequestListener listener = new ItsNatServletRequestListener()
{
    public void processRequest(ItsNatServletRequest request,
                ItsNatServletResponse response)
    {
        ItsNatDocument itsNatDoc = request.getItsNatDocument();
        if (itsNatDoc != null)
        {
            ...
        }
        else // Pretty URL case
        {
            HttpServletRequest servRequest =
                    (HttpServletRequest)request.getServletRequest();
            String pathInfo = servRequest.getPathInfo();
            if (pathInfo == null)
                throw new RuntimeException("Unexpected URL");

            String docName = pathInfo.substring(1); // Removes '/'
            docName = docName.replace('/','.');  // => "name.name"
            request.getServletRequest().setAttribute("itsnat_doc_name",docName);

            ServletResponse servResponse = response.getServletResponse();
            request.getItsNatServlet().processRequest(servRequest,servResponse);
        }
    }
};
```

This example converts the format `/name/name` to our name format `name.name`, sets the `itsnat_doc_name` as a request attribute and forwards the request/response pair again to the ItsNat servlet. Now ItsNat knows the document name and loads and returns a new document.

ItsNat ever gets `itsnat_doc_name` value calling first `ServletRequest.getParameter(String)` if no parameter is defined then calls `ServletRequest.getAttribute(String)`, this way we can reuse the same request and response instances.

Pretty URLs change how referenced elements by the page like JavaScript files or images are resolved, for instance image paths must be absolute etc. One important problem to resolve is where to load the framework JavaScript files; use an absolute path like the following when registering the template (if framework files are automatically included):

```
DocumentTemplate docTemplate = ...;
...
docTemplate.setFrameworkScriptFilesBasePath("/itsnat/js");
```

6.28 THREADING

ItsNat is prepared to work in mutithread environments "out of the box", most of the time the developer does not need to deal with synchronization issues.

In a Java web application there are two type of threads:

1. The thread used to initialize the servlet (method `Servlet.init(ServletConfig)`)

   Use this thread to configure ItsNat and register the documents and fragments. The `Servlet.init(ServletConfig)` is ever called once.

2. The thread used to process a web request/response

   A web request has an objective: to load a new document (page) or process an AJAX event. The user code is ever called with the `ItsNatDocument` target (just loaded or the event target) previously synchronized. `ItsNatDocument` and dependent objects (child/aggregated objects) are not thread save but no synchronization is necessary because is already done by ItsNat. Events with the same document are processed secuentially but there is no order guarantee because network transport is asynchronous and AJAX events can be sent asynchronously by the browser.

   If a developer needs to access the document using a user created thread is highly recommended to synchronize the `ItsNatDocument` first, no other dependent object needs to be synchronized because these objects are ever obtained using the document as a factory.

6.29 REFERRERS

When a user clicks a link or submits a form the HTTP header sent to the target contains the property "referrer", this property contains the URL of the page source[52]. ItsNat simulates the referrer feature with a server centric approach.

6.29.1 Referrer "pull"

[52] The referrer property can be obtained with the call request.getHeader("referer") where request is the HttpServletRequest.

When a user leaves a page by clicking a link or submitting a form, the current `ItsNatDocument` is saved temporally as the "referrer" of the target page. The target `ItsNatDocument` can access the previous document, only inside the loading process, using the method `ItsNatServletRequest.`**`getItsNatDocumentReferrer`**`()`. This method returns null if the referrer feature is disabled in the source (the default state).

For instance:

```
public void processRequest(ItsNatServletRequest request,
                ItsNatServletResponse response)
{
    ItsNatDocument itsNatDoc = request.getItsNatDocument();

    ItsNatDocument itsNatDocRef = request.getItsNatDocumentReferrer();
    ...
}
```

By default the referrer feature is disabled, it can be enabled globally with `ItsNatServletConfig.`**`setReferrerEnabled`**`(boolean)` or per-template basis with `DocumentTemplate.`**`setReferrerEnabled`**`(boolean)`; if referrer is disabled the document can not be "referred". The referrer feature only works with AJAX enabled.

Referrers have interesting uses, for instance the target DOM document can copy source markup (importing first with `Document.`**`importNode`**`(Node,boolean)`), data models of components etc. This approach, to get some information from the referrer, may be named "referrer pull" The "Feature Showcase" contains a complete example, this example shows how to detect the back and forward buttons using the referrer document.

6.29.2 Referrer "push"

The referrer "pull" approach copies some data from the referrer (source) to the target, the referrer document is got when the target page is loaded, using target code.

The referrer "push" approach is the opposite, the referrer document is notified that a target document is being loaded and the referrer is going to be unloaded, this is an opportunity to copy some data from the referrer (source) to the target, but now this code is executed by the referrer *before* the target `ItsNatServletRequestListener` objects are executed, by this way the source/referrer *prepares* the target.

ItsNat supports this tecnhique providing a special `ItsNatServletRequestListener` registry in the `ItsNatDocument`. Registered request listeners calling `ItsNatDocument.`**`addReferrerItsNatServletRequestListener`**`(ItsNatServletRequestListener)` are executed with the same `ItsNatServletRequest` and `ItsNatServletResponse` objects being sent *after* to the normal target request listeners; referrer request listeners can access and modify the target `ItsNatDocument`, add request attributes etc.

For instance, the following code register a "referrer listener" to be executed when a target is being loaded (and the referrer document unloaded), this listener copies a DOM fragment from the source/referrer (this document) to the target.

```
final ItsNatDocument itsNatDoc = ...;

ItsNatServletRequestListener listener = new ItsNatServletRequestListener()
{
        public void processRequest(ItsNatServletRequest request,
                       ItsNatServletResponse response)
        {
            ItsNatDocument itsNatDocTarget = request.getItsNatDocument();

            Document doc = itsNatDoc.getDocument();
            Element listParentElem = doc.getElementById("messageListId");
            Node contentNode =
                  ItsNatDOMUtil.extractChildren(listParentElem.cloneNode(true));
            if (contentNode != null)
            {
                Document docTarget = itsNatDocTarget.getDocument();

                Element listParentElemTarget =
                       docTarget.getElementById("messageListId");
                contentNode = docTarget.importNode(contentNode,true);
                listParentElemTarget.appendChild(contentNode);
            }
        }
    };
    itsNatDoc.addReferrerItsNatServletRequestListener(listener);
```

Referrer "push" technique is not enabled by default, it can be enabled globally with `ItsNatServletConfig.`**`setReferrerPushEnabled`**`(boolean)` or per-template basis with `DocumentTemplate.`**`setReferrerPushEnabled`**`(boolean)`. If referrer push in *target* is disabled the target document can not be "pushed". The referrer feature must be enabled in the *source* document otherwise `addReferrerItsNatServletRequestListener` call throws an exception.

Referrer "push" is an alternative to "pull" but they can be mixed. The "Feature Showcase" contains again a complete example, this example does the same functionality as the "pull" version but using "push".

6.30 NON-AJAX MODE

AJAX is enabled by default, but can be disabled globally with `ItsNatServletConfig.`**`setAJAXEnabled`**`(boolean)` or in a per-template basis with `DocumentTemplate.`**`setAJAXEnabled`**`(boolean)`. If AJAX is disabled any AJAX/event related feature or method is disabled, usually they do nothing, for instance `ItsNatDocument.`**`addEventListener`** methods would may be called but nothing is done and no JavaScript is sent to client. Anyway methods like `ItsNatDocument.`**`addCodeToSend`**`(Object)` continue working as normal. DOM manipulation, fragments, DOM utilities keep working in the load phase.

The development model is the classical page to page navigation where the `ItsNatDocument` only exists in the load phase, every new request creates a new `ItsNatDocument`. The referrer feature is disabled because needs AJAX, anyway you can use `ItsNatSession/HttpSession` objects to store a previous `ItsNatDocument` to access the old DOM, copy data of components etc; be conscious this "saved" `ItsNatDocument` is invalid (is not attached anymore to a browser page) and only read-only operations should be done. The "Feature Showcase" includes a small example of a core based non-AJAX application.

6.31 JAVASCRIPT DISABLED MODE

A step further, JavaScript can be fully disabled, this is the lowest downgraded mode of ItsNat. This mode is basically the same as non-AJAX mode but no JavaScript code is sent to the client, for instance a call to `ItsNatDocument.`**`addCodeToSend`**`(Object)` throws an exception. JavaScript is enabled by default, but can be disabled globally with `ItsNatServletConfig.`**`setScriptingEnabled`**`(boolean)` or in a per-template basis with `DocumentTemplate.`**`setScriptingEnabled`**`(boolean)`.

This feature allows ItsNat to serve pages to clients with JavaScript disabled. DOM manipulation, fragments, DOM utilities keep working in the load phase but only if the page is in fast load mode (see `DocumentTemplate.`**`setFastLoadMode`**`(boolean)`). Again the "Feature Showcase" includes an example very similar to the non-AJAX example but with no client JavaScript code.

6.32 EXCEPTIONS

ItsNat only provides two unchecked exception classes: `ItsNatException` and `ItsNatDOMException`, the second one inherits from `ItsNatException`, and both are `RuntimeException` based. Every checked exception thrown internally is encapsulated inside an `ItsNatException`, if this error contains a DOM node context an `ItsNatDOMException` may be used.

6.33 SVG

6.33.1 Pure SVG documents

FireFox 1.5+, Safari 3, Opera 9 and QtWebKit support pure SVG documents natively including JavaScript, events and AJAX, so ItsNat treats SVG documents as first class citizens with the same features as X/HTML documents including components. For instance: a "circle list", a pie chart seen as a list, tables and trees with graphic elements… of course they all include data models, selection models, custom structures, renderers…

ItsNat only adds AJAX support to SVG documents from SVG templates registered with the MIME type `image/svg+xml`.

Xerces DOM does not use SVG specific DOM[53], this is not a serious limitation, normal DOM Level 2 can be used to manage SVG elements. In ItsNat Xerces does not read DTDs so there is no SVG validation, one consequence is the missing support of the "id" attribute, SVG DTD defines the "id" attribute as an XML ID, so in the server side `Document.`**`getElementById`**`(String)` ever returns null; use `ItsNatDOMUtil.`**`getElementById`**`(String,Node)` with a `Document` object as second parameter as an alternative to locate SVG elements[54].

6.33.2 SVG embedded inline in XHTML

ItsNat supports SVG elements embedded inline in XHTML[55] including events and components as any other XHTML element.

[53] http://www.w3.org/TR/SVG/svgdom.html

[54] ItsNat uses the same algorithm as Xerces (for instance to search HTML elements by id).

The following example shows how to create a SVG circle list, embedded in a XHTML document, with the following pattern:

```
    <svg:svg id="circleListId" itsnat:nocache="true" width="400" height="300"
               xmlns:svg="http://www.w3.org/2000/svg">
        <svg:circle cx="50" cy="150" r="70" fill="#0000ff" fill-opacity="0.5"/>
    </svg:svg>
```

```
    ItsNatDocument itsNatDoc = ...;
    Document doc = itsNatDoc.getDocument();

    Element listParentElem = doc.getElementById("circleListId");
    ElementGroupManager factory = itsNatDoc.getElementGroupManager();
    ElementList circleList = factory.createElementList(listParentElem,true);

    for(int i = 0; i < 5; i++)
    {
        Element circleElem = circleList.addElement();
        int cx;
        if (i > 0)
        {
            Element prevCircle = circleList.getElementAt(i - 1);
            cx = Integer.parseInt(prevCircle.getAttribute("cx"));
        }
        else cx = 30;
        cx += 50;
        circleElem.setAttribute("cx",Integer.toString(cx));
    }
```

The SVG element `<g>` is very useful as parent of groups of SVG elements managed with components or in general using pattern based techniques (DOM utilities).

6.34 XML GENERATION

ItsNat can generate non-X/HTML like RDF. ItsNat DOM utilities like lists, tables and trees can be used to create the resulting XML using pattern based techniques, simplifying very much the typical DOM manipulation. XML templates can be used too, including memory oriented caching (an XML template can have "static" parts). But XML documents do not have state, neither events nor components (ItsNat component system heavily relies on browser events).

For instance, the following XML template is designed to contain a CD list:

```
<?xml version='1.0' encoding='UTF-8' ?>

<discs>
    <cdList>
        <cd>
            <title>Tittle</title>
            <artist>Artist</artist>
            <songs>
                <song>Song Pattern</song>
            </songs>
        </cd>
```

[55] XHTML supports elements with other namespaces, but only nodes with known namespaces are rendered and receive events, SVG nodes in XHTML are recognized on browsers with native SVG support.

```
    </cdList>
</discs>
```

Registering into the servlet (`init(ServletConfig)` method):

```
    String pathPrefix = getServletContext().getRealPath("/");
    pathPrefix += "WEB-INF/pages/manual/";

    ItsNatHttpServlet itsNatServlet = getItsNatHttpServlet();

    DocumentTemplate docTemplate;
    ...
    docTemplate = itsNatServlet.registerDocumentTemplate("manual.core.xmlExample",
                       "text/xml", pathPrefix + "xml_example.xml");
    docTemplate.addItsNatServletRequestListener(new CoreXMLExampleLoadListener());
```

The listener code:

```
public class CoreXMLExampleLoadListener implements ItsNatServletRequestListener
{
    public CoreXMLExampleLoadListener()
    {
    }

    public void processRequest(ItsNatServletRequest request,
                               ItsNatServletResponse response)
    {
        ItsNatDocument itsNatDoc = request.getItsNatDocument();
        Document doc = itsNatDoc.getDocument();
        Element discsElem = doc.getDocumentElement();

        Element cdListElem =
                       ItsNatTreeWalker.getLastChildElement(discsElem);
       ElementGroupManager factory = itsNatDoc.getElementGroupManager();
        ElementList discList = factory.createElementList(cdListElem,true);

        addCD("Help","The Beatles",
               new String[] {"A Hard Day's Night","Let It Be"},discList);
        addCD("Making Movies","Dire Straits",
               new String[] {"Tunnel Of Love","Romeo & Juliet"},discList);
    }

    public void addCD(String title,String artist,String[] songs,
                      ElementList discList)
    {
        Element cdElem = discList.addElement();
        Element titleElem = ItsNatTreeWalker.getFirstChildElement(cdElem);
        ItsNatDOMUtil.setTextContent(titleElem,title);
        Element artistElem = ItsNatTreeWalker.getNextSiblingElement(titleElem);
        ItsNatDOMUtil.setTextContent(artistElem,artist);
        Element songsElem = ItsNatTreeWalker.getNextSiblingElement(artistElem);

        ItsNatDocument itsNatDoc = discList.getItsNatDocument();
       ElementGroupManager factory = itsNatDoc.getElementGroupManager();
        ElementList songList = factory.createElementList(songsElem,true);
        for(int i = 0; i < songs.length; i++)
            songList.addElement(songs[i]);
    }
}
```

Note the Java code is "tag agnostic" because new elements are created and filled using the pattern/renderer approach.

Finally this is the XML rendered and sent to client:

```
<?xml version="1.0" encoding="UTF-8"?>
<discs>
    <cdList>
        <cd>
            <title>Help</title>
            <artist>The Beatles</artist>
            <songs>
                <song>A Hard Day&apos;s Night</song>
                <song>Let It Be</song>
            </songs>
        </cd>
        <cd>
            <title>Making Movies</title>
            <artist>Dire Straits</artist>
            <songs>
                <song>Tunnel Of Love</song>
                <song>Romeo &amp; Juliet</song>
            </songs>
        </cd>
    </cdList>
</discs>
```

6.34.1 XML fragments

XML documents support XML fragment insertion, statically and dynamically. Markup fragments are explained at "STRING TO DOM CONVERSION" chapter.

The following example shows how to insert an XML fragment statically:

```
<?xml version="1.0" encoding="UTF-8"?>
<discs>
    <itsnat:include name="manual.core.xmlFragExample"
            xmlns:itsnat="http://itsnat.org/itsnat" />
    <cdList>
        <cd>
    ...
```

Renders:

```
<?xml version="1.0" encoding="UTF-8"?>
<discs>
    <title>CD List</title>
    <subtitle>in XML</subtitle>
    <cdList>
        <cd>
    ...
```

Again inserting dynamically:

```
...
Element cdListElem = ItsNatTreeWalker.getLastChildElement(discsElem);

ItsNatServlet servlet = request.getItsNatServlet();
DocFragmentTemplate fragTemplate =
            servlet.getDocFragmentTemplate("manual.core.xmlFragExample");
DocumentFragment docFrag = fragTemplate.loadDocumentFragment(itsNatDoc);
discsElem.insertBefore(docFrag,cdListElem); // docFrag is empty now
```

```
    ElementGroupManager factory = itsNatDoc.getElementGroupManager();
    ElementList discList = factory.createElementList(cdListElem,true);
    ...
```

6.35 BROWSER ISSUES

ItsNat makes a strong effor to overcome browser limitations for instance generating the appropiated JavaScript code adapted to the target browser.

Some limitations can not be surpassed, the best example is IE Mobile 6 (WM 6), IE Mobile have many limitations, for instance only links and form controls can receive links, and DHTML support is very limited (`setAttribute` and `attachEvent` are missing, text nodes and comments are not reflected in DOM etc). In spite of this IE Mobile layout can be fully controlled from server much like any other supported browser, using W3C APIs including `setAttribute`, text nodes and comments can be added/removed/deleted, multiple event listeners can be bound to links and form controls and so on.

ItsNat best support is achieved with MSIE, FireFox and Safari, FireFox is the best contender.

These are the most important problems found by ItsNat:

6.35.1 Unload events are not ever fired

ItsNat needs an unload event to destroy the server side `ItsNatDocument` object when the user leaves the page.

Opera 9 and some mobile browsers do not fire unload events in some circunstances for instance when the back or forward button is pressed[56] or when the tab is closed.

This affects to remote control: if you are monitoring a document loaded by a browser with unload notification not guaratied, you cannot detect when the user leaves the page (using back/forward buttons or closing the window) until has lapsed the max inactive time of the session. One workaround is using `ItsNatDocument.`**`getClientDocumentOwner`**`()` and `ClientDocument.`**`getLastRequestTime`**`()` methods to monitor the last access to the document, this gives you an accurate view about the real use of the page of the observed user.

To avoid orphan `ItsNatDocument` objects ItsNat automatically removes `ItsNatDocument` objects never accesed during the interval returned by `HttpSession.`**`getMaxInactiveInterval`**`();` this ensure the document is lost when is not used during the max inactive interval defined by the session. Session expiration is not necessary the user may actively using another page (of course when the servlet session expires all documents are lost). The same technique is used to clean `ClientDocument` objects representing remote views open with browsers with unload not guarantied observing another page/document.

Another way to prevent an excessive memory comsumption by orphan documents (and to avoid user abuse) is the configuration value defined by `ItsNatServletContext.getMaxOpenDocumentsBySession().` If a session have more documents than the maximum allowed, then documents in excess are invalidated and lost; the criteria is the time past the last access (more time more orphan).

[56] When we talk about “back/forward” buttons, history navigation using JavaScript (window.history.go(n) etc) is supposed too

6.35.2 Pages are not reloaded from server using back and forward buttons

This problem affects to Opera desktop and some mobile browsers (NetFront, IE Mobile, Opera Mini and Mobile and some WebKit based browsers).

Opera is fast traversing the navigation history using back/forward buttons mainly because pages are not reload using the cache. Opera ignores any HTTP header or meta tag instructing not to be cached (only are obeyed with HTTPS)[57]. This is a serious problem in ItsNat because browser pages and server side documents must be tighly synchronized and ItsNat ever tries to avoid caching including with back/forward buttons because back/forward buttons are considered special "links" (referrers can be used to detect the previous page as in normal navigation).

To minimize this problem, ItsNat configures Opera to execute the load event when the page is open using back/forward buttons with the following JavaScript commands:

```
window.opera.setOverrideHistoryNavigationMode("compatible");
window.history.navigationMode = "compatible";
```

This way ItsNat can detect a new load event in a non existing document (if was unloaded) or in a page already loaded (load event is received twice, the second one says to ItsNat the page is loaded from cache). In this case ItsNat has detected the client page was not loaded from server and force to the page to reload sending the following JavaScript sentence:

```
window.location.reload(true);
```

In mobile browsers there is no way to detect when the page is cached until the user touch something and an event is sent to the server, if the server document is alive is processed as usual because server and client are in sync (unload event was not received when user leaved the page), if the server document was lost then the same JavaScript sentence is sent to the client to reload again the page from server.

The problem of this JavaScript sentence is a new entry added to the window history, this affects slightly to behavior of the referrer feature when back/forward buttons are used.

If you have problems with referrers using Opera and mobile browsers you can use alternative techniques to gain access to the previous page[58].

6.35.3 Form auto-fill and user actions while the page is loading

Form auto-fill is a feature of Safari and Opera very useful for users and a nightmare for web developers.

Form auto-fill is problematic for ItsNat because breaks the rule of client as a slave of server, this feature can fill a form before the loading script generated by ItsNat is executed (Safari) or after the loading process (Opera) without change events. In summary: the server state may differ from client state.

ItsNat is conscious of this problem and "disables" form auto-fill in Safari and Opera, basically reverts the state of form controls to the server state. This action guaranties form

[57] http://my.opera.com/yngve/blog/2007/02/27/introducing-cache-contexts-or-why-the

[58] See "Degraded Modes" examples of the "Feature Showcase" of how to use the session to save the previous page to access it from the new page.

synchronization between server and client, client state is the same of the server state after loading.

There is another case of server and client un-sync in forms, when the user changes form controls when the page is loading, before the initial script generated by ItsNat is executed. The initial script usually binds ItsNat form components to form controls to keep in sync the server when the user changes the state of a form control, but any change before the initial script is executed is not controlled by ItsNat.

Again ItsNat fixes this problem, ItsNat reverts any user change to form controls on loading time.

In summary: form controls are guarantied to be in sync with server on loading time. If form controls are controlled by ItsNat components then bidirectional sync is guarantied, because user actions in form controls are propagated to the server and DOM server is updated accordingly using the corresponding attributes (`checked`, `selected` and `value`).

6.35.4 XMLHttpRequest only asynchronous in some browsers

Some WebKit based mobile browsers like Android, S60WebKit and Iris do not support synchronous XHR requests because this feature was implemented in WebKit later in non-Mac OS X versions of WebKit[59].

For these browsers ItsNat ignores the sync mode and uses the async-hold. This mode ensures sequential processing of client events keeping the order of firing (is not the same as sync mode but is similar).

[59] Future versions of these browsers will support synchronous requests but the initial versions supported by ItsNat do not.

7. COMPONENTS

7.1 OVERVIEW

The Components module defines a component system very similar to any event based desktop component system like Swing. The Component module is based on Core, all Core features can be used in a component based application.

ItsNat components are the "typical" classes encapsulating and managing the state of a visual element alongside a data model and processing user events. To avoid expose implementation details, component objects are accessed using interfaces, almost no ItsNat implementation class is public.

As seen in the prologue ItsNat components go far beyond the typical form control-Java component: every markup element can be a component.

ItsNat has buttons, lists, tables and trees, more components will be added in a future. Any developer can add new components to the framework as plug-ins.

ItsNat components bind the HTML view with an event model, a data model and a selection model (lists, tables and trees). The framework component system uses the view pattern approach: any developer is free to design the component markup how he likes (using the pattern technique), and bind to a concrete data model.

ItsNat syncs data model and selection model changes with the view automatically, routes client events to the registered component event listener at the server and syncs client form control changes (text introduced in a text box etc) with matched components and related server side DOM elements (for instance, a text component automatically updates the `value` attribute of the `HTMLInputElement` server object and the data model when the browser `<input>` control changes).

ItsNat components define a renderer-structure system again, this system is very similar to Core DOM utilities; in fact by default renderers and structures are based on Core renderers/structures behind the scenes, many concepts seen in Core are applied again.

ItsNat reuses many Swing classes and interfaces like data models, selection models and related listeners. ItsNat component architecture is strongly inspired in Swing, for instance there is an `ItsNatTree`, an `ItsNatTreeUI`, an `ItsNatTreeCellEditor` and an `ItsNatTreeCellRenderer`. ItsNat uses Swing when possible, but it does not try to fit the web UI (based in markup) with the desktop UI (based in pixels), for instance, no ItsNat method gets/sets the (x,y) pixel position of a component.

Any component is associated to a DOM node, usually an element. Usually this element is bound to the document tree but this is not mandatory. If a component was created associated to a DOM node, usually this node can not change by other (reattachment).

ItsNat defines many component interfaces, many of them are empty or almost empty; two reasons:

1. Identification/classification: a simple `instanceof` may be used to identify a component object.

2. Future methods: new methods may be added precisely avoiding the typical "this method does nothing if this object is …).

7.2 CLASSIFICATIONS

7.2.1 By data model

Buttons, text based components, lists, tables, trees, custom/user defined.

7.2.2 By fixed based or free DOM elements

- Fixed components: components wrapping typical HTML elements like anchors of form elements: `<input>`, `<select>`, `<button>`, `<textarea>` etc
- Free components: using any DOM element. They are not bound to a fixed namespace like HTML, for instance, a list can be a list of SVG circle elements.

7.3 NAME CONVENTIONS

All component interfaces start with the `ItsNat` prefix.

7.3.1 HTML components

Most of the HTML components follow the pattern:

`ItsNat` + DOM interface name without "Element"

For instance, the associated component interface of a `<button>` element is:

`ItsNat` + `HTMLButton`~~`Element`~~ = `ItsNatHTMLButton`

Parent interfaces do not follow this pattern (they are not bound to a specific HTML element).

The following table shows all HTML elements with associated components and corresponding ItsNat interfaces:

HTML element	W3C Java DOM Interface	Interface
<a [itsnat:compType="buttonLabel"]>	**HTMLAnchorElement**	**ItsNatHTMLAnchor** **ItsNatHTMLAnchorLabel**
<button [itsnat:compType="buttonLabel"]>	**HTMLButtonElement**	**ItsNatHTMLButton** **ItsNatHTMLButtonLabel**
<form>	**HTMLFormElement**	**ItsNatHTMLForm**

<input>	**HTMLInputElement**	**ItsNatHTMLInput** **ItsNatHTMLInput*X* (X=value of type attribute)** **ItsNatHTMLInputText** **ItsNatHTMLInputTextFormatted**
<label>	**HTMLLabelElement**	**ItsNatHTMLLabel**
<select>	**HTMLSelectElement**	**ItsNatHTMLSelect** **ItsNatHTMLSelectComboBox** **ItsNatHTMLSelectMul**
<table>	**HTMLTableElement**	**ItsNatHTMLTable**
<textarea>	**HTMLTextAreaElement**	**ItsNatHTMLTextArea**

Most of the HTML interfaces are final (no more interfaces are inherited).

7.3.2 Free components

Free components follow the pattern: `ItsNatFree` + *ComponentType*

For instance: `ItsNatFreeCheckBox`, `ItsNatFreeComboBox`, `ItsNatFreeInclude` etc.

Most of the free interfaces are final (no more interfaces are inherited).

7.3.3 Non-HTML and non-Free base interfaces

Follow the pattern: `ItsNat` + *ComponentType*

The component type name usually follows a left/generic to right/specific pattern: *MainCompType* + *SubCompType* … For instance `ItsNatButton` is the base of `ItsNatButtonToggle` (this one is a specialization of a generic button).

No non-HTML, non-free component interface is final (this not apply to UI interfaces).

7.3.4 Other

Three ItsNat interfaces are directly associated to W3C DOM Core interfaces:

`Node` => `ItsNatComponent` (method `getNode()`)

`Element` => `ItsNatElementComponent` (method `getElement()`)

`HTMLElement` => `ItsNatHTMLElementComponent` (method `getHTMLElement()`)

They are defined mainly to classification purposes.

7.4 LIFE CYCLE

A special object implementing `ItsNatComponentManager` (and `ItsNatHTMLComponentManager`) is used as a component factory and registry. This object is got from `ItsNatDocument` as a singleton (a specialization object).

7.4.1 Creation

A component is created using `ItsNatComponentManager`, two modes:

7.4.1.1 Explicit creation with optional registration

The following HTML:

```
<a id="linkId" href="javascript:void(0)">Click me</a>
```

and Java code:

```
ItsNatDocument itsNatDoc = ...;
ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

ItsNatHTMLAnchor linkComp =
            (ItsNatHTMLAnchor)componentMgr.createItsNatComponentById("linkId");
```

creates and associates an anchor component to the HTML anchor. The component manager knows what kind of component to create because an `ItsNatHTMLAnchor` is the default component associated to an HTML anchor.

The following instruction:

```
componentMgr.addItsNatComponent(linkComp);
```

adds the component to the registry, creation and registration can be done with only one instruction:

```
linkComp = (ItsNatHTMLAnchor)componentMgr.addItsNatComponentById("linkId");
```

The component registry "holds" the component to avoid garbage collection. To get the registered component associated to a concrete node:

```
linkComp = (ItsNatHTMLAnchor)componentMgr.findItsNatComponentById("linkId");
```

With "free" components we need to specify what kind of component we want to create:

```
<div id="buttonId">Free Button, Click Me</div>


ItsNatFreeButtonNormal buttonComp =
    (ItsNatFreeButtonNormal)componentMgr.createItsNatComponentById(
            "buttonId","freeButtonNormal",null);
```

The `freeButtonNormal` name specifies the component type to associate to the <div> element (a button).

7.4.1.2 Automatic creation from markup

We can avoid an explicit creation of components defining all needed configuration data in the markup (itsnat namespace was previously declared):

```
<div id="buttonId" itsnat:compType="freeButtonNormal">
    Free Button, Click Me</div>

ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

componentMgr.buildItsNatComponents(doc.getDocumentElement());

ItsNatFreeButtonNormal buttonComp =
    (ItsNatFreeButtonNormal)componentMgr.findItsNatComponentById("buttonId");
```

The method `buildItsNatComponents` traverse the DOM subtree creating and adding components automatically depending on the markup. The `compType` is a standard attribute to specify the component type usually of a free component.

7.4.2 Destruction

Components must be destroyed explicitly when they are not going to be used anymore to free allocated resources and to unregister event listeners. Explicit destruction is not needed if the user leaves the page/document (associated `ItsNatDocument` is automatically destroyed along with components. When do you need to destroy a component? Typically when you are going to remove and lost the associated element; this occurs when developing a highly dynamic web page (with frequent partial page substitutions).

7.4.2.1 Explicit destruction

```
buttonComp.dispose();
```

Component destruction automatically unregisters it from `ItsNatComponentManager`.

7.4.2.2 Automatic destruction

```
componentMgr.removeItsNatComponents(doc.getDocumentElement(),true);
```

The previous call unregisters and destroys (with a `dispose` call) the components registered found while traversing the DOM tree.

7.5 DOM EVENTS

ItsNat components automatically receive specific DOM events from the client when the associated element to the component is involved (e.g. clicked). For instance a button component receives `click` events, a text box `change` events, a table receive `click` events and so on. When a component receives an event usually does something useful: a button tells to the data button model that it was clicked, a text box updates the data model with the new text and a table selects the clicked cell using a selection model; when a (Swing) data or selection model is changed fires standard (Swing) events.

Swing data and selection models have specific event/listeners; you can register listeners very much like you would register inside a Swing application. Any component has a default data model and some of them a selection model, these default models can be changed by the user.

Besides Swing event listeners, you can add listeners to be notified when a DOM event is received by the component, this listener is executed *after* the standard processing if any:

```
ItsNatFreeButtonNormal buttonComp = ...;

EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        System.out.println("Event " + evt.getType());
    }

};
buttonComp.addEventListener("click",listener);
```

Sometimes a component supports several "incompatible" event types. For instance a button can receive `click` events or `mousedown` and `mouseup` events, by default only the `click` event type is enabled, sometimes we might need to replace the standard behavior of `click` with `mousedown` and `mouseup`, to do this use `disable/enableEventListener` methods:

```
buttonComp.disableEventListener("click");
buttonComp.enableEventListener("mousedown");
buttonComp.enableEventListener("mouseup");
```

7.6 BUTTONS

Any DOM element can be a button component, because any DOM element can receive a mouse event. Of course form based buttons are supported "out of the box".

All ItsNat standard buttons implement the `ItsNatButton` interface.

ItsNat buttons use `javax.swing.ButtonModel` data model and listen DOM `click` events by default. When a user clicks a DOM element declared as a button component, the `click` event is received by the server object and simulates a complete pressing cycle with the data model. This is the "internal" code:

```
javax.swing.ButtonModel dataModel = ...;
Event evt = ...;

String type = evt.getType();

if (type.equals("click"))
{
    dataModel.setArmed(true);
    dataModel.setPressed(true);
    dataModel.setPressed(false);
    dataModel.setArmed(false);
}
```

The ButtonModel object fires a ChangeEvent for any change. The data model should fire one ActionEvent per click (when button is armed and not pressed). User defined listeners registered in the data model can receive this events like a Swing application.

If a more control is wanted, click events can be replaced with mousedown/mouseup events using disable/enableEventListener methods[60]:

```
buttonComp.disableEventListener("click");
buttonComp.enableEventListener("mousedown");
buttonComp.enableEventListener("mouseup");
```

This is the "internal" code:

```
else if (type.equals("mousedown"))
{
    dataModel.setArmed(true);
    dataModel.setPressed(true);
}
else if (type.equals("mouseup"))
{
    dataModel.setPressed(false);
    dataModel.setArmed(false);
}
```

Buttons can receive mouseover/mouseout events too; these events are not enabled by default to avoid too much traffic. This is the default processing if enabled (using enableEventListener):

```
else if (type.equals("mouseover"))
{
    dataModel.setRollover(true);
}
else if (type.equals("mouseout"))
{
    dataModel.setRollover(false);
    dataModel.setArmed(false);
}
```

There are two types of buttons: normal and toggle buttons.

7.6.1 Normal Buttons

Normal buttons are buttons with no saved state (unlike toggle buttons).

All normal buttons implements ItsNatButtonNormal:

[60] A button component must not receive "click" and "mousedown/mouseup" events at the same time because is "confused" (processed as two clicks).

Markup	Interface
`<input type="button">`	`ItsNatHTMLInputButton`
`<input type="submit">`	`ItsNatHTMLInputSubmit`
`<input type="reset">`	`ItsNatHTMLInputReset`
`<input type="image">`	`ItsNatHTMLInputImage`
`<button>`	`ItsNatHTMLButton`
`<a>`	`ItsNatHTMLAnchor`
`<`*any* `[itsnat:compType="freeButtonNormal"]`[61]`>`	`ItsNatFreeButtonNormal`
`<`*any* `[itsnat:compType="freeButtonNormalLabel"]`[62]`>`	`ItsNatFreeButtonNormalLabel`

Some interfaces are empty and mainly defined to identify the component type.

All normal buttons use a `javax.swing.DefaultButtonModel` model object; this model can be changed using `setButtonModel(ButtonModel)`. The following code snippet shows the richness of listeners bound to a button component (an anchor with id "`linkId`")[63]:

```
ItsNatDocument itsNatDoc = ...;
ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

ItsNatHTMLAnchor linkComp =
            (ItsNatHTMLAnchor)componentMgr.createItsNatComponentById("linkId");

EventListener evtListener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        System.out.println("Clicked :" + evt.getCurrentTarget());
    }
};
linkComp.addEventListener("click",evtListener);

ButtonModel dataModel = linkComp.getButtonModel();

ActionListener actListener = new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("Clicked :" + e);
    }
};
dataModel.addActionListener(actListener);
```

[61] If component type attribute is not specified, it must be specified explicitly in Java.
[62] If component type attribute is not specified, it must be specified explicitly in Java.
[63] The `ButtonModel` has support of `ItemListener` listeners but these have no sense with normal buttons

```
    ChangeListener chgListener = new ChangeListener()
    {
        public void stateChanged(ChangeEvent e)
        {
            System.out.println("Button state has changed:");
            ButtonModel model = (ButtonModel)e.getSource();

            String fact = "";
            if (model.isArmed())
                fact += "armed ";
            if (model.isPressed())
                fact += "pressed ";
            if (model.isRollover())
                fact += "rollover ";
            if (model.isSelected()) // ever false with normal buttons
                fact += "selected ";

            if (!fact.equals(""))
                System.out.println(fact);
        }
    };
    dataModel.addChangeListener(chgListener);
```

Of course usually only one listener type is really useful.

7.6.2 Toggle Buttons

7.6.2.1 Form controls

Toggle buttons have state: selected and unselected. This state is managed by the javax.swing.JToggleButton.ToggleButtonModel, and may change if the button is clicked.

There are two toggle button types: check boxes and radio buttons. The main difference is radio buttons form a group where only one button can have the state "selected". All toggle button components implement ItsNatButtonToggle, check boxes implement ItsNatButtonCheckBox and radio buttons implement ItsNatButtonRadio:

Markup	Interface
<input type="checkbox">	**ItsNatHTMLInputCheckBox**
<input type="radio">	**ItsNatHTMLInputRadio**
<*any* [itsnat:compType="freeCheckBox"]>	**ItsNatFreeCheckBox**
<*any* [itsnat:compType="freeCheckBoxLabel"]>	**ItsNatFreeCheckBoxLabel**
<*any* [itsnat:compType="freeRadioButton"]>	**ItsNatFreeRadioButton**
<*any* [itsnat:compType="freeRadioButtonLabel"]>	**ItsNatFreeRadioButtonLabel**

The following code snippet shows how to bind several radio buttons; again some code is redundant to show the richness of listeners:

```
<input type="radio" name="radioName" id="inputId1" />
<input type="radio" name="radioName" id="inputId2" />


ItsNatDocument itsNatDoc = ...;
ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

final ItsNatHTMLInputRadio inputComp1 =
  (ItsNatHTMLInputRadio)componentMgr.createItsNatComponentById("inputId1");
final ItsNatHTMLInputRadio inputComp2 =
  (ItsNatHTMLInputRadio)componentMgr.createItsNatComponentById("inputId2");

EventListener evtListener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        EventTarget currentTarget = evt.getCurrentTarget();
        String button;
        if (currentTarget == inputComp1.getHTMLInputElement())
            button = "button 1";
        else
            button = "button 2";

        System.out.println("Clicked " + button);
    }
};

inputComp1.addEventListener("click",evtListener);
inputComp2.addEventListener("click",evtListener);

ItsNatButtonGroup itsNatGrp1 = inputComp1.getItsNatButtonGroup();
ItsNatButtonGroup itsNatGrp2 = inputComp2.getItsNatButtonGroup();
if ( ((itsNatGrp1 == null) || (itsNatGrp2 == null)) ||
     (itsNatGrp1.getButtonGroup() != itsNatGrp2.getButtonGroup()) )
{
    ButtonGroup group = new ButtonGroup();
    ItsNatButtonGroup htmlGroup = componentMgr.getItsNatButtonGroup(group);
    htmlGroup.addButton(inputComp1);
    htmlGroup.addButton(inputComp2);
}

ToggleButtonModel dataModel1 = inputComp1.getToggleButtonModel();
ToggleButtonModel dataModel2 = inputComp2.getToggleButtonModel();

ActionListener actListener = new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        String button;
        if (e.getSource() == inputComp1.getToggleButtonModel())
            button = "button 1";
        else
            button = "button 2";

        System.out.println("Clicked :" + button);
    }
};
```

```
    dataModel1.addActionListener(actListener);
    dataModel2.addActionListener(actListener);

    ItemListener itemListener = new ItemListener()
    {
        public void itemStateChanged(ItemEvent e)
        {
            String fact;
            int state = e.getStateChange();
            if (state == ItemEvent.SELECTED)
                fact = "selected";
            else
                fact = "deselected";
            fact += " button ";
            if (e.getItem() == inputComp1.getToggleButtonModel())
                fact += "1";
            else
                fact += "2";

            System.out.println(fact);
        }
    };
    dataModel1.addItemListener(itemListener);
    dataModel2.addItemListener(itemListener);
```

Some interesting parts:

```
    if ( ((itsNatGrp1 == null) || (itsNatGrp2 == null)) ||
          (itsNatGrp1.getButtonGroup() != itsNatGrp2.getButtonGroup()) )
    {
        ButtonGroup group = new ButtonGroup();
        ItsNatButtonGroup htmlGroup = componentMgr.getItsNatButtonGroup(group);
        htmlGroup.addButton(inputComp1);
        htmlGroup.addButton(inputComp2);
    }
```

This code is used to ensure both radio buttons are included in the same group, the standard class `javax.swing.ButtonGroup` is used, but is wrapped with `ItsNatButtonGroup` because `ButtonGroup.`**`add`**`(AbstractButton)` is not valid with ItsNat. The method `getItsNatButtonGroup(ButtonGroup)` looks for an `ItsNatButtonGroup` if no one is found one is created and registered. In our example this code is not needed because `itsNatGrp1` and `itsNatGrp2` are the same object, then the same group; because ItsNat automatically generates a group using the value of the `name` attribute of `<input>` radio buttons, if the name is the same the group is the same.

You can avoid using `ButtonGroup` absolutely with the following methods:

```
   ItsNatComponentManager.getItsNatButtonGroup(String name)

   ItsNatComponentManager.createItsNatButtonGroup()
```

The first one looks for an `ItsNatButtonGroup` with the specified name, if not found a new one is created with this name (and automatically registered). The second method creates (and registers) a new group with an auto-generated name and a new `ButtonGroup`.

ItsNat ensures that two `ItsNatButtonGroup` objects can not have the same name or the same `ButtonGroup` object.

It is highly discouraged to change the component group using `ToggleButtonModel.setGroup(ButtonGroup)` because this call cannot be detected by ItsNat (no event is generated, anyway some change detection is implemented), the component does it automatically by using `ItsNatButtonGroup`.

A new listener, `ItemListener`, is very useful with check boxes and radio buttons. This listener keeps track of selection changes; if a button is selected or unselected an `ItemEvent` is fired and sent to the registered listeners.

Any change in the button state is reflected on the `checked` attribute in server DOM, but this is not mandatory in client, in the client the *property* `checked` is keep in sync not the attribute. The `checked` attribute reflects the current state of the selection much like the component model, but avoid any direct change of the attribute using server DOM APIs (and of course in client using custom JavaScript) use instead component APIs (this rule change in "markup driven" mode explained further).

7.6.2.2 Free toggle buttons

Free Toggle Buttons, check boxes and radio buttons, work very much the same as form based buttons, but with only one difference: no select/unselect decoration is defined by default. Use `ItemListener` listeners to keep track of changes and decorate the involved DOM element as you want (changing background color etc). The following example shows how to do this with two "free" radio buttons:

```
final ItsNatFreeRadioButton buttonComp1 = ...;
final ItsNatFreeRadioButton buttonComp2 = ...;

ItemListener itemListener = new ItemListener()
{
    public void itemStateChanged(ItemEvent e)
    {
        int state = e.getStateChange();

        if (e.getItem() == buttonComp1.getToggleButtonModel())
            updateDecoration(buttonComp1,state);
        else
            updateDecoration(buttonComp2,state);
    }

    public void updateDecoration(ItsNatFreeRadioButton buttonComp,int state)
    {
        Element buttonElem = buttonComp.getElement();
        ToggleButtonModel model = buttonComp.getToggleButtonModel();
        if (state == ItemEvent.SELECTED) // or with: (model.isSelected())
            buttonElem.setAttribute("style","background: rgb(253,147,173);");
        else
            buttonElem.removeAttribute("style");
    }
};
```

Can `<a>` or `<button>` or `<input type="button|image">` elements be toggle buttons?

Yes, you must specify the component type using the `itsnat:compType` attribute or passing this type name to the factory method as a parameter. Of course these elements have no state based decoration like check boxes or radio button, is up to you to change the visual appearance to show the current state (selected/unselected).

7.6.3 Buttons with Label

Most of the ItsNat buttons can optionally support a label. Some button types such as `ItsNatHTMLInputButton`, `ItsNatHTMLInputReset` and `ItsNatHTMLInputSubmit` have this label built-in because the `<input>` element shows visually the `value` attribute. Other button types as `ItsNatHTMLButtonLabel`, `ItsNatHTMLAnchorLabel`, `ItsNatFreeRadioButtonLabel`, `ItsNatFreeRadioButtonLabel` and `ItsNatFreeCheckBoxLabel` support this label using an `ElementRenderer`. All of them implement `ItsNatButtonLabel`, using this interface the label value can be set (and rendered) easily calling `ItsNatButtonLabel.`**`setLabelValue`**`(Object)`.

7.7 TEXT BASED COMPONENTS

Text based components are inline text boxes (text fields) and text areas. Text boxes are designed to be modified "in-place", text areas are like text boxes with support of multiple lines. Typical text based form components are supported "out of the box" like `<input type="text|password|hidden|file">` and `<textarea>`.

All of these components implement the `ItsNatTextComponent` interface, text fields implement `ItsNatTextField`:

Markup	Interface
`<input type="text">`	**`ItsNatHTMLInputText`**
`<input type="password">`	**`ItsNatHTMLInputPassword`**
`<input type="file">`	**`ItsNatHTMLInputFile`**
`<input type="hidden">`	**`ItsNatHTMLInputHidden`**
`<input type="text" [itsnat:compType="formatted TextField"]>`	**`ItsNatHTMLInputTextFormatted`**
`<textarea>`	**`ItsNatHTMLTextArea`**

Text based components are backed with `javax.swing.text.Document` as the data model, by default `javax.swing.text.PlainDocument` is used. The `change` event automatically updates the DOM `value` attribute and data model text at server side with client changes. Text changes performed in data model are automatically sent to the client updating the DOM `value` attribute and property (updating the control). With text components you do not need to manipulate the DOM value attribute directly at server side, use the data model or `getText/setText` component methods instead.

The following example shows how to use an `<input type="text">` based component:

```
<input type="text" id="inputId" value="" size="40" />

ItsNatDocument itsNatDoc = ...;
ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();
```

```
final ItsNatHTMLInputText inputComp =
       (ItsNatHTMLInputText)componentMgr.createItsNatComponentById("inputId");
inputComp.setText("Change this text and lost the focus");

EventListener evtListener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        System.out.println("Text changed: " +
            inputComp.getHTMLInputElement().getValue());
        // Alternative: inputComp.getText();
    }
};
inputComp.addEventListener("change",evtListener);

DocumentListener docListener = new DocumentListener()
{
    public void insertUpdate(DocumentEvent e)
    {
        javax.swing.text.Document docModel = e.getDocument();
        int offset = e.getOffset();
        int len = e.getLength();

        try
        {
            System.out.println("Inserted, pos " + offset + "," + len +
                    " chars,\"" + docModel.getText(offset,len) + "\"");
        }
        catch(BadLocationException ex)
        {
            throw new RuntimeException(ex);
        }
    }

    public void removeUpdate(DocumentEvent e)
    {
        javax.swing.text.Document docModel = e.getDocument();
        int offset = e.getOffset();
        int len = e.getLength();

        System.out.println("Removed, pos " + offset + "," + len + " chars");
    }

    public void changedUpdate(DocumentEvent e)
    {
        // A PlainDocument has no attributes
    }
};
PlainDocument dataModel = (PlainDocument)inputComp.getDocument();
dataModel.addDocumentListener(docListener);

inputComp.focus();
inputComp.select();
```

When the form control content changes, the new text is sent to the server contained in a change event, this new text is set to the data model as a complete deletion of the current text and an insert of the new text; then two `DocumentEvent` will be fired, a "remove" event and a "insert" event, the first is not fired if data model contains no text and the last one is not fired if the new text is an empty string.

Final two lines show us two "familiar" methods, `focus()` and `select()`. Alongside `blur()` and `click()`, these methods are present in DOM `HTMLInputElement` interface, but in Xerces they are dummy methods. They all are repeated in the `ItsNatHTMLInput` interface; if called the same JavaScript call is sent to the client and executed in the client when the event returns (or load phase ends).

Any change in the form control state is reflected on the `value` attribute in server DOM, but this is not mandatory in client, in the client the *property* `value` is keep in sync not the attribute. The `value` attribute reflects the current state of the control much like the component model, but avoid any direct change of the attribute using server DOM APIs (and of course in client using custom JavaScript) use instead component APIs (this rule change in "markup driven" mode explained further).

7.7.1 Input File component

Behavior of an `<input type="file">` based component is different to the text version, because there are some security issues. JavaScript modification of the value property of this element is ignored in MSIE and throws a security error in FireFox, only the user can define the value using the visual control. Consequence: text value must not be defined from server side using the data model or with `ItsNatTextComponent.`**`setText`**`(String)`. Anyway any change to the control value is automatically sent to the server, component works in a "passive only" mode.

Current implementation does not submit the specified file to the server; nothing prevents you of sending the file inside a normal form, using an `<iframe>` can avoid leaving the page.

7.7.2 Input Hidden component

Input hidden elements do not fire `change` events because there is no user (visual) control of the `value` property. In this case the server side component can define the hidden value in the client DOM element using the data model or with the method `ItsNatTextComponent.`**`setText`**`(String)`. In this case the component works in an "active only" mode.

7.7.3 Use of keyup and keydown events

Text components do special processing with `keyup` events if enabled. By default `keyup` and `keydown` events are disabled to avoid traffic and only the `change` event updates the server, this event is fired when the control loses the focus. If enabled `keyup`, every key stroke is sent to the server and the DOM element (`value` attribute) and data model are updated incrementally; in fact when the `change` event is fired no update is performed. There is no special processing with `keydown` events, no incremental updating is done because this event can be cancelled, and only when the event is fully processed the new character is included in the control. This is not the case of `keyup` events, the new key is included in the control *before* the `keyup` event is fired.

Conclusion: enable `keyup` events to incrementally update the server side DOM element and data model; enable `keydown` events and attach a listener if you want to monitor the complete life cycle of every stroke or to avoid undesired characters (cancelling the event with

`Event.preventDefault()`[64]). Another way to remove undesired characters is to update the data model removing the "offending" character when data model is changed during a `keyup` event processing.

7.7.4 Use of keypress events

By default `keypress` events are not enabled, no special processed is done if enabled because `keypress` has similar behaviour to `keydown` (if cancelled the new key is not included in the control). Enable `keypress` events and attach a listener if you want to monitor key strokes or cancel undesired characters.

7.7.5 Input Text Formatted component

Input Text Formatted component type is highly inspired in `javax.swing.JFormattedTextField` class. This component is a specialization of a normal text field to support formatted input and output and specific data types as values beyond strings. To declare a `<input type="text">` as a text field formatted use the attribute `itsnat:compType="formattedTextField"` or pass this type name to a factory method.

For instance:

```
<input type="text" id="inputId" value="" size="40" />
```

```
ItsNatDocument itsNatDoc = ...;
ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

ItsNatHTMLInputTextFormatted inputComp =
    (ItsNatHTMLInputTextFormatted)componentMgr.createItsNatComponentById(
            "inputId","formattedTextField",null);
try{ inputComp.setValue(new Date()); }
catch(PropertyVetoException ex) { throw new RuntimeException(ex); }

PropertyChangeListener propListener = new PropertyChangeListener()
{
    public void propertyChange(PropertyChangeEvent evt)
    {
        Date value = (Date)evt.getNewValue();
        System.out.println("Value changed to: " + value);
    }
};
inputComp.addPropertyChangeListener("value",propListener);

VetoableChangeListener vetoListener = new VetoableChangeListener()
{
    public void vetoableChange(PropertyChangeEvent evt)
                    throws PropertyVetoException
    {
        Date newDate = (Date)evt.getNewValue();
        if (newDate.compareTo(new Date()) > 0)
            throw new PropertyVetoException("Future date is not allowed",evt);
    }
};
inputComp.addVetoableChangeListener(vetoListener);
```

[64] An event can be cancelled in SYNC mode only

In this example a `java.util.Date` object is submitted as a value, `ItsNatHTMLInputTextFormatted` has default formatters to `java.util.Date` and `java.lang.Number` objects.

This is how the text box is shown:

Jun 8, 2007

If the text is changed with a non valid date (bad format), this new value is rejected and the last previous value is restored. If a valid date is introduced but is a future date, is rejected too because our `VetoableChangeListener` rejects any future date. Only a text with a valid date in the past is accepted, the listener `PropertyChangeListener`, listening `value` property changes, is notified and the method `ItsNatFormattedTextField.`**`getValue`**`()` returns the new `java.util.Date` object if called.

An `ItsNatFormattedTextField` component (currently `ItsNatHTMLInputTextFormatted` is the only one) offers several ways or levels to customize the formatting (input and output) work:

1) Default `java.util.Date` and `java.lang.Number` formatters per `ItsNatDocument`.

   These formatters are shared and used by default by all `ItsNatHTMLInputTextFormatted` components of the same document. They are especially useful if you only need to specify the localization of dates or decimal numbers.

   Change the default formatters with methods:

   `ItsNatDocument.`**`setDefaultDateFormat`**`(java.text.DateFormat)`

   `ItsNatDocument.`**`setDefaultNumberFormat`**`(java.text.NumberFormat)`

2) Specifying a formatter per component instance with the method:

   `ItsNatFormattedTextField.`**`setFormat`**`(java.text.Format)`

   The component will use `Format.`**`parseObject`**`(Object)` and `Format.`**`format`**`(String)` to convert from `Object` to `String` and vice versa. This formatter must be compatible with the values used.

3) Specifying an `ItsNatFormattedTextField.ItsNatFormatter` object registered with:

   `ItsNatFormattedTextField.`**`setItsNatFormatter`**`(ItsNatFormatter)`

   This interface has two methods to parse and convert values and strings:

   `Object `**`stringToValue`**`(String text) throws ParseException;`

   `String `**`valueToString`**`(Object value) throws ParseException;`

4) Specifying an `ItsNatFormattedTextField.ItsNatFormatterFactory` registered with:

   `ItsNatFormattedTextField.`**`setItsNatFormatterFactory`**`(`

ItsNatFormatterFactory)

This factory returns a custom `ItsNatFormatter` depending on the state of the component with the method:

`ItsNatFormatter` **`getItsNatFormatter`**`(ItsNatFormattedTextField)`

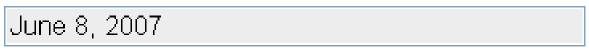
5) Specifying an `ItsNatFormatterFactoryDefault` factory

ItsNat provides a default `ItsNatFormatterFactory` implementation, this implementation implements `ItsNatFormatterFactoryDefault` and is inspired in `javax.swing.text.DefaultFormatterFactory`. This default factory provides different formatters depending on the state of the component: is the component is going to be edited returns the edition formatter, if is not editing returns the display formatter. By default `ItsNatHTMLInputTextFormatted` has `focus` and `blur` event types enabled, if the control gets the focus the component updates the control value using the formatter returned by the factory (edition formatter), when the control lost the focus (modified or not) the component updates again the control with the string formatted with the formatter returned by the factory (display formatter).

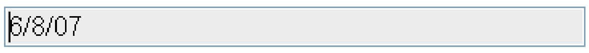
The following example shows how we can add a default factory with different display and edition formatters:

```
...
ItsNatHTMLInputTextFormatted inputComp = ...;

ItsNatFormatterFactoryDefault factory =
    (ItsNatFormatterFactoryDefault)inputComp.createDefaultItsNatFormatterFactory();

ItsNatFormatter dispFormatter =
    inputComp.createItsNatFormatter(
            DateFormat.getDateInstance(DateFormat.LONG,Locale.US));
factory.setDisplayFormatter(dispFormatter);
ItsNatFormatter editFormatter =
    inputComp.createItsNatFormatter(
            DateFormat.getDateInstance(DateFormat.SHORT,Locale.US));
factory.setEditFormatter(editFormatter);

inputComp.setItsNatFormatterFactory(factory);

try{ inputComp.setValue(new Date()); }
catch(PropertyVetoException ex) { throw new RuntimeException(ex); }
...
```

When the control has not the focus (display mode):


June 8, 2007

And when the control acquires the focus (edition mode):


6/8/07

7.7.6 Text Area components

Text area components are programmed in the same way as text boxes, of course text content may have end of lines.

7.8 LABELS

ItsNat leverages the "pattern based label" approach to a component level: binding a markup layout pattern and a data model (a Java object/value) again using pluggable renderers. Furthermore the Java value can be edited "in-place" with some type of activation (usually mouse clicks).

ItsNat labels have a default renderer implementing `ItsNatLabelRenderer`, this renderer is basically the same as `ElementRenderer` (converts the Java value to String and replaces the first text node found with this string).

Following the ItsNat style of work, label markup is defined in the template, the parent DOM element is bound to the ItsNat component, when you set a value to the label, the component calls the renderer to write the value to the markup.

Label components implement the interface `ItsNatLabel`:

Markup	Interface
<label>	**ItsNatHTMLLabel**
<*any* [itsnat:compType="freeLabel"]>	**ItsNatFreeLabel**

Label data model is the object value itself. The default renderer converts the value to string calling `Object.`**`toString`**`()` and this string replaces the appropriate text node.

Only one event type has a default process, `dblclick`, this event activates the "in-place" editor. This activation is started calling:

`ItsNatLabelEditor.`**`getLabelEditorComponent`**`(ItsNatLabel,Object,Element)`

The default editor opens a text field over the label[65], when the control loses the focus is removed and the new value is set to as the label value and markup is updated with the value. The default editor can be customized with the control to be used when editing with the method:

`ItsNatHTMLComponentManager.`**`createDefaultItsNatLabelEditor`**`(ItsNatComponent)`

The following example shows how we can use a list box to edit a label with a number:

```
<label id="labelId"><img src="image.png" /><b>NUMBER</b></label>


ItsNatHTMLDocument itsNatDoc = ...;
ItsNatHTMLComponentManager componentMgr =
            itsNatDoc.getItsNatHTMLComponentManager();
```

[65] In fact label markup content is replaced with the text box.

```
    ItsNatHTMLLabel label =
                (ItsNatHTMLLabel)componentMgr.createItsNatComponentById("labelId");
    try { label.setValue(new Integer(3)); } // Initial value
    catch(PropertyVetoException ex) { throw new RuntimeException(ex); }

    ItsNatHTMLSelectComboBox editorComp =
                       componentMgr.createItsNatHTMLSelectComboBox(null,null);
    DefaultComboBoxModel model =
                       (DefaultComboBoxModel)editorComp.getComboBoxModel();
    for(int i=0; i < 5; i++) model.addElement(new Integer(i));

    ItsNatLabelEditor editor = componentMgr.createDefaultItsNatLabelEditor(editorComp);
    label.setItsNatLabelEditor(editor);

    EventListener evtListener = new EventListener()
    {
        public void handleEvent(Event evt)
        {
            System.out.println("Edition starts...");
        }
    };
    label.addEventListener("dblclick",evtListener);

    PropertyChangeListener propListener = new PropertyChangeListener()
    {
        public void propertyChange(PropertyChangeEvent evt)
        {
            System.out.println("Changed label, old: " + evt.getOldValue() +
               ", new: " + evt.getNewValue());
        }
    };
    label.addPropertyChangeListener("value",propListener);
```

This example changes `NUMBER` text node with the number selected in the combo box when the label is edited (with a double click). Note the `PropertyChangeListener`, when the label value changes a `PropertyChangeEvent` event is fired with the property `value`.

The default editor is customizable and supports: `ItsNatHTMLInputText` (formatted and unformatted), `ItsNatHTMLSelectComboBox`, `ItsNatHTMLInputCheckBox` and `ItsNatHTMLTextArea` as the control used.

The default event used to activate the editor (`dblclick`) can be changed using:

```
ItsNatLabel.setEditorActivatorEvent(String eventType)
```

To disable in-place editing, disable the event activator or set the editor to null:

```
label.setItsNatLabelEditor(null);
```

Edition can be started programmatically with the method:

```
ItsNatLabel.startEditing()
```

7.8.1 User defined editors

You can develop your own `ItsNatLabelEditor` implementing this interface; this interface is very similar to Swing's `TableCellEditor` or `TreeCellEditor`.

```
public interface ItsNatLabelEditor extends CellEditor
{
    public ItsNatComponent getLabelEditorComponent(
                ItsNatLabel label, Object value,Element labelElem);
}
```

`ItsNatLabelEditor` inherits from `javax.swing.CellEditor`; Swing has an abstract class implementing this interface providing us almost all we need: `javax.swing.AbstractCellEditor`.

A real world example is a good motivation: a label shows basic information of a `Person`, first name and last, we want to edit the `Person` data "in-place" with a double click over the label.

`Person` class source code:

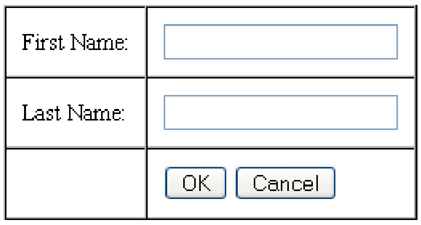
```
public class Person
{
    protected String firstName;
    protected String lastName;

    public Person(String firstName,String lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName()
    {
        return firstName;
    }

    public void setFirstName(String firstName)
    {
        this.firstName = firstName;
    }

    public String getLastName()
    {
        return lastName;
    }

    public void setLastName(String lastName)
    {
        this.lastName = lastName;
    }

    public String toString()
    {
        return firstName + " " + lastName;
    }
}
```

We want to edit `Person` data with the following markup included in an HTML fragment:

```
<!-- <?xml version="1.0" encoding="UTF-8"?> -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
    <head>
        <title>Title</title>
    </head>
    <body>

    <table id="personEditorId" border="1px" cellspacing="0" cellpadding="10px">
        <tbody>
            <tr><td>First Name:</td>
                <td><input id="firstNameId" type="text" size="20"/></td></tr>
            <tr><td>Last Name:</td>
                <td><input id="lastNameId" type="text" size="20"/></td></tr>
            <tr><td>&nbsp;</td>
                <td><input id="okPersonId" type="button" value="OK" />&nbsp;
                <input id="cancelPersonId" type="button" value="Cancel" />
                </td></tr>
        </tbody>
    </table>

    </body>
</html>
```

This is how is rendered:



We register our person editor markup fragment with the name: `feashow.components.shared.personEditor`

The next step is to define a user defined ItsNat component bound to the previous markup, this component will be returned by the method `ItsNatLabelEditor.`**`getLabelEditorComponent`**`(ItsNatLabel,Object,Element)`. A user defined component must implement `ItsNatComponent`:

```
public class PersonEditorComponent extends MyCustomComponentBase
{
    protected ItsNatHTMLInputText firstNameComp;
    protected ItsNatHTMLInputText lastNameComp;
    protected ItsNatHTMLInputButton okComp;
    protected ItsNatHTMLInputButton cancelComp;
    protected Person person;

    public PersonEditorComponent(Person person,Element parentElement,
                ItsNatComponentManager compMgr)
    {
        super(parentElement,compMgr);

        this.person = person;

        this.firstNameComp =
           (ItsNatHTMLInputText)compMgr.createItsNatComponentById("firstNameId");
        this.lastNameComp =
           (ItsNatHTMLInputText)compMgr.createItsNatComponentById("lastNameId");
        this.okComp =
            (ItsNatHTMLInputButton)compMgr.createItsNatComponentById("okPersonId");
        this.cancelComp =
          (ItsNatHTMLInputButton)compMgr.createItsNatComponentById("cancelPersonId");

        firstNameComp.setText(person.getFirstName());
        lastNameComp.setText(person.getLastName());
    }

    public void dispose()
    {
        firstNameComp.dispose();
        lastNameComp.dispose();
        okComp.dispose();
        cancelComp.dispose();
    }

    public Person getPerson()
    {
        return person;
    }

    public void updatePerson()
    {
        person.setFirstName(firstNameComp.getText());
        person.setLastName(lastNameComp.getText());
    }

    public ItsNatHTMLInputButton getOKButton()
    {
        return okComp;
    }

    public ItsNatHTMLInputButton getCancelButton()
    {
        return cancelComp;
    }
}
```

The base class MyCustomComponentBase is an almost empty class implementing required methods with dummy content. These methods are not needed with this kind of "compound" component (a component grouping other components):

```
public abstract class MyCustomComponentBase implements ItsNatComponent
{
    protected Element parentElement;
    protected ItsNatComponentManager compMgr;

    public MyCustomComponentBase(Element parentElement,
                 ItsNatComponentManager compMgr)
    {
        this.parentElement = parentElement;
        this.compMgr = compMgr;
    }

    public Node getNode()
    {
        return parentElement;
    }
    public void setNode(Node node)
    {
        throw new RuntimeException("REATTACHMENT IS NOT SUPPORTED");
    }

    public ItsNatDocument getItsNatDocument()
    {
        return compMgr.getItsNatDocument();
    }

    public ItsNatComponentManager getItsNatComponentManager()
    {
        return compMgr;
    }

    public void addEventListener(String type, EventListener listener)
    {
        throw new RuntimeException("NOT IMPLEMENTED");
    }

    // Remaining methods throw a "NOT IMPLEMENTED" runtime exception
    // ...
}
```

Now we can create our custom label editor:

```
public class PersonCustomLabelEditor extends AbstractCellEditor
                implements ItsNatLabelEditor,EventListener
{
    protected PersonEditorComponent editorComp;

    public PersonCustomLabelEditor()
    {
    }

    public ItsNatComponent getLabelEditorComponent(ItsNatLabel label, Object value,
                                    Element labelElem)
    {
        ItsNatComponentManager compMgr = label.getItsNatComponentManager();
        ItsNatDocument itsNatDoc = compMgr.getItsNatDocument();
        Document doc = itsNatDoc.getDocument();

        ItsNatServlet servlet = itsNatDoc.getDocumentTemplate().getItsNatServlet();
        DocFragmentTemplate docFragTemplate = servlet.getDocFragmentTemplate(
                    "feashow.components.shared.personEditor");
        DocumentFragment editorFrag =
                docFragTemplate.loadDocumentFragment(itsNatDoc);

        labelElem.appendChild(editorFrag);

        Element editorElem = doc.getElementById("personEditorId");

        this.editorComp =
                new PersonEditorComponent((Person)value,editorElem,compMgr);

        editorComp.getOKButton().addEventListener("click",this);
        editorComp.getCancelButton().addEventListener("click",this);

        return editorComp;
    }
    public Object getCellEditorValue()
    {
        return editorComp.getPerson();
    }

    public void handleEvent(Event evt)
    {
        if (evt.getCurrentTarget() == editorComp.getOKButton().getHTMLInputElement())
        {
            editorComp.updatePerson();

            editorComp.dispose(); // Before the DOM subtree is removed from the
tree by the editor manager
            stopCellEditing();
        }
        else
        {
            editorComp.dispose(); // "
            cancelCellEditing();
        }

        this.editorComp = null;
    }
}
```

When `getLabelEditorComponent` method is called by the label component, the label DOM element parent is received as the `labelElem` parameter. Before the call the label DOM subtree was removed to be filled with the editor markup; this markup code is obtained from the HTML fragment registered before (the person editor markup). An instance of `PersonEditorComponent` is created and returned; current implementation does noting with this object and a null can be returned.

If the “OK” or “Cancel” button is pressed the edition process finishes by calling the standard methods `stopCellEditing` or `cancelCellEditing`, these methods notify the internal editor manager in the label this fact, then the editor manager removes the editor markup and restores the original label markup content, then calls `getCellEditorValue()` to get the new data value (if called `stopCellEditing`) this new value is set to the label, the label automatically updates the view by calling the label renderer.

Now we are ready to use the new label editor to be used with a free label showing and editing a `Person` object:

```
    ItsNatHTMLDocument itsNatDoc = ...;
    ItsNatHTMLComponentManager componentMgr =
                itsNatDoc.getItsNatHTMLComponentManager();

    ItsNatFreeLabel comp = (ItsNatFreeLabel)componentMgr.createItsNatComponentById(
                "labelId","freeLabel",null);
    try { comp.setValue(new Person("Jose M.","Arranz")); }
    catch(PropertyVetoException ex) { throw new RuntimeException(ex); }

    ItsNatLabelEditor editor = new PersonCustomLabelEditor();
    comp.setItsNatLabelEditor(editor);
    ...
```

7.8.2 User defined renderers

A label layout can change beyond a single line. Nothing prevents you from defining custom renderers implementing `ItsNatLabelRenderer`. An `ItsNatLabelRenderer` renderer is very similar to an `ElemenLabelRenderer`:

```
    <div id="labelId">
        <table>
           <tbody>
              <tr><td>First Name:</td><td id="firstNameId">First Name</td></tr>
              <tr><td>Last Name:</td><td id="lastNameId">Last Name</td></tr>
           </tbody>
        </table>
    </div>

    ...
    ItsNatFreeLabel comp = (ItsNatFreeLabel)componentMgr.createItsNatComponentById(
               "labelId","freeLabel",null);

    ItsNatLabelRenderer renderer = new PersonCustomLabelRenderer();
    comp.setItsNatLabelRenderer(renderer);

    try { comp.setValue(new Person("Jose M.","Arranz")); }
    catch(PropertyVetoException ex) { throw new RuntimeException(ex); }

    ...
```

And finally:

```
public class PersonCustomLabelRenderer implements ItsNatLabelRenderer
{
    public PersonCustomLabelRenderer()
    {
    }

    public void renderLabel(ItsNatLabel label, Object value,
                Element labelElem, boolean isNew)
    {
        Person person = (Person)value;

        ItsNatDocument itsNatDoc = label.getItsNatDocument();
        Document doc = itsNatDoc.getDocument();
        Element firstNameElem = doc.getElementById("firstNameId");
        ItsNatDOMUtil.setTextContent(firstNameElem,person.getFirstName());
        Element lastNameElem = doc.getElementById("lastNameId");
        ItsNatDOMUtil.setTextContent(lastNameElem,person.getLastName());
    }
}
```

Current implementation does nothing with the returned component and can be null.

7.8.3 Labels with non-HTML elements

`ItsNatFreeLabel` implementation is not (X)HTML namespace dependent, and can be used with other namespaces, for instance, to render (and may be edit) a SVG element inside a XHTML document. FireFox, Safari 3, Opera 9, and QtWebKit support SVG elements embedded in a XHTML document and these elements can receive click events.

7.9 LISTS

ItsNat leverages the "pattern based list" approach to a component level: binding a markup layout pattern, a data model (an object list) and a selection model ready to listen and process events, again using pluggable renderers and structural layouts like pattern based lists in the ItsNat Core module.

List components implement the interface `ItsNatList` with two sub interfaces, `ItsNatComboBox` and `ItsNatListMultSel`. `ItsNatComboBox` and `ItsNatListMultSel` follow the same philosophy as `javax.swing.JComboBox` and `javax.swing.JList`. They both use `javax.swing.ListModel` as the data model, `ItsNatComboBox` based components use `javax.swing.ComboBoxModel` (`javax.swing.DefaultComboBoxModel` by default), this model includes the concept of "item selected", `ItsNatListMultSel` components use `javax.swing.ListSelectionModel` as the selection model (and `javax.swing.DefaultListModel` by default).

Markup	Interface
<select>	**ItsNatHTMLSelectComboBox**
<select multiple="multiple">	**ItsNatHTMLSelectMult**
<*any* [itsnat:compType="freeComboBox"]>	**ItsNatFreeComboBox**
<*any* [itsnat:compType="freeListMultSel"]>	**ItsNatFreeListMultSel**

7.9.1 Combo Boxes

Combo boxes are appropriate when we need to select one item and there are no duplicated items[66].

The following example shows a list of Spanish cities using a combo box:

```
<select id="compId" />


ItsNatDocument itsNatDoc = ...;
ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

ItsNatHTMLSelectComboBox comboComp =
    (ItsNatHTMLSelectComboBox)componentMgr.createItsNatComponentById("compId");

DefaultComboBoxModel dataModel =
                    (DefaultComboBoxModel)comboComp.getComboBoxModel();
dataModel.addElement("Madrid");
dataModel.addElement("Sevilla");
dataModel.addElement("Segovia");
dataModel.addElement("Barcelona");
dataModel.addElement("Oviedo");
dataModel.addElement("Valencia");
```

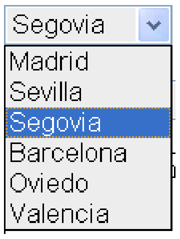
[66] Data model allows duplicated elements but the behavior of selection may be buggy because `javax.swing.ComboBoxModel` manages item selection by value (methods `setSelectedItem(Object)` and `Object getSelectedItem()`).

```
dataModel.setSelectedItem("Segovia");

EventListener evtListener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        System.out.println(evt.getCurrentTarget() + " " + evt.getType());
    }
};
comboComp.addEventListener("change",evtListener);

ListDataListener dataListener = new ListDataListener()
{
    public void intervalAdded(ListDataEvent e)
    {
        listChangedLog(e);
    }

    public void intervalRemoved(ListDataEvent e)
    {
        listChangedLog(e);
    }

    public void contentsChanged(ListDataEvent e)
    {
        listChangedLog(e);
    }

    public void listChangedLog(ListDataEvent e)
    {
        int index0 = e.getIndex0();
        int index1 = e.getIndex1();

        String action = "";
        int type = e.getType();
        switch(type)
        {
            case ListDataEvent.INTERVAL_ADDED:   action = "Added"; break;
            case ListDataEvent.INTERVAL_REMOVED: action = "Removed"; break;
            case ListDataEvent.CONTENTS_CHANGED: action = "Changed"; break;
        }

        String interval = "";
        if (index0 != -1)
            interval = " interval " + index0 + "-" + index1;

        System.out.println(action + " " + interval);
    }
};
dataModel.addListDataListener(dataListener);

ItemListener itemListener = new ItemListener()
{
    public void itemStateChanged(ItemEvent e)
    {
        String fact;
        int state = e.getStateChange();
        if (state == ItemEvent.SELECTED)
            fact = "Selected";
        else
```

```
                fact = "Deselected";

            System.out.println(fact + " " + e.getItem());
        }
    };
    comboComp.addItemListener(itemListener);
```

Of course children of `<select>` are ever `<option>` elements like `<option>Madrid</option>`, no pattern is needed.

This is how the combo box is rendered (outspread):



If the user changes the selected item a `change` event is fired and sent to the server, the component changes the selected item accordingly in the `ComboBoxModel`, the `DefaultComboBoxModel` fires an `javax.swing.event.ListDataEvent` (with `ListDataEvent.CONTENTS_CHANGED` value) and two `javax.swing.event.ItemEvent` (the first one is to notify about the element unselected, the second tell us the new element selected) received by the `javax.swing.event.ItemListener` listeners registered.

Any change to the data model (an item added or removed) is notified to the listeners and view (updated automatically using the default renderer calling the method `Object.toString()` of the item value).

Any change in selection is reflected on the `selected` attribute in server DOM, but this is not mandatory in client, in the client the *property* `selected` is keep in sync not the attribute. The `selected` attribute reflects the current state of the selection much like the component model, but avoid any direct change of the attribute using server DOM APIs (and of course in client using custom JavaScript) use instead component APIs (this rule change in "markup driven" mode explained further).

`ItsNatComboBox` supports "free combo box lists" too:

```
    <table border="1px" cellspacing="0" cellpadding="5px">
        <tbody id="compId">
            <tr><td><b>Cell/Row Pattern</b></td></tr>
        </tbody>
    </table>
```

The Java code is basically the same:

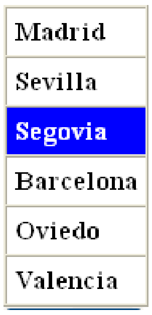
```
    ...

    ItsNatFreeComboBox comboComp =
                (ItsNatFreeComboBox)componentMgr.createItsNatComponentById(
                        "compId","freeComboBox",null);

    ...
    comboComp.addItemListener(new ComboBoxSelectionDecorator(comboComp));
    dataModel.setSelectedItem("Segovia");

    ...
    comboComp.addEventListener("click",evtListener);
```

By default the `click` event changes the selected item (the component selects the item clicked). Now there is no "automatic" selection decoration like in a combo box control and ItsNat does not impose a default decoration; our example uses a custom `ItemListener` to decorate the selected item (or to remove the decoration):

```
public class ComboBoxSelectionDecorator implements ItemListener
{
    protected ItsNatComboBox comp;

    public ComboBoxSelectionDecorator(ItsNatComboBox comp)
    {
        this.comp = comp;
    }

    public void itemStateChanged(ItemEvent e)
    {
        int state = e.getStateChange();
        int index = comp.indexOf(e.getItem());
        boolean selected = (state == ItemEvent.SELECTED);

        decorateSelection(index,selected);
    }

    public void decorateSelection(int index,boolean selected)
    {
        Element elem = comp.getItsNatListUI().getContentElementAt(index);

        if (selected)
            elem.setAttribute("style","background:rgb(0,0,255); color:white;");
        else
            elem.removeAttribute("style");
    }
}
```

The call `getContentElementAt(index)` uses the default structure manager and returns the `<td>` element; this element is the "parent" of the item content.

This object can be reused and renders:



7.9.2 Lists with multiple selected items

ItsNatListMultSel based components allow the user to select one or more objects from a list using a standard javax.swing.ListSelectionModel object (javax.swing.DefaultListSelectionModel by default) and a javax.swing.ListModel (javax.swing.DefaultListModel by default). Duplicated elements are managed with no problem because DefaultListModel and ListSelectionModel manage the data and selection list by index.

The following example shows again a list of Spanish cities using an HTML select with multiple selection:

```
<select id="compId" multiple="multiple" size="5" />
```

```
ItsNatDocument itsNatDoc = ...;
ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

ItsNatHTMLSelectMult listComp =
       (ItsNatHTMLSelectMult)componentMgr.createItsNatComponentById("compId");

DefaultListModel dataModel = (DefaultListModel)listComp.getListModel();
dataModel.addElement("Madrid");
dataModel.addElement("Sevilla");
dataModel.addElement("Segovia");
dataModel.addElement("Barcelona");
dataModel.addElement("Oviedo");
dataModel.addElement("Valencia");

ListSelectionModel selModel = listComp.getListSelectionModel();
selModel.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
selModel.setSelectionInterval(2,3);

EventListener evtListener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        System.out.println(evt.getCurrentTarget() + " " + evt.getType());
    }
};
listComp.addEventListener("change",evtListener);

ListDataListener dataListener = new ListDataListener()
{
    public void intervalAdded(ListDataEvent e)
    {
        listChangedLog(e);
    }

    public void intervalRemoved(ListDataEvent e)
    {
        listChangedLog(e);
    }

    public void contentsChanged(ListDataEvent e)
    {
        listChangedLog(e);
    }

    public void listChangedLog(ListDataEvent e)
```

```
        {
            int index0 = e.getIndex0();
            int index1 = e.getIndex1();

            String action = "";
            int type = e.getType();
            switch(type)
            {
                case ListDataEvent.INTERVAL_ADDED:   action = "Added"; break;
                case ListDataEvent.INTERVAL_REMOVED: action = "Removed"; break;
                case ListDataEvent.CONTENTS_CHANGED: action = "Changed"; break;
            }

            String interval = "";
            if (index0 != -1)
                interval = " interval " + index0 + "-" + index1;

            System.out.println(action + " " + interval);
        }
    };
    dataModel.addListDataListener(dataListener);

    ListSelectionListener selListener = new ListSelectionListener()
    {
        public void valueChanged(ListSelectionEvent e)
        {
            if (e.getValueIsAdjusting())
                return;

            int first = e.getFirstIndex();
            int last = e.getLastIndex();

            ListSelectionModel selModel = (ListSelectionModel)e.getSource();
            String fact = "";
            for(int i = first; i <= last; i++)
            {
                boolean selected = selModel.isSelectedIndex(i);
                if (selected)
                    fact += ", selected ";
                else
                    fact += ", deselected ";
                fact += i;
            }

            System.out.println("Selection changes" + fact);
        }
    };
    selModel.addListSelectionListener(selListener);
```

The number of selected items is imposed by the call:

```
selModel.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
```

All selection modes are supported. If the user violates the restriction imposed, the selection model forces the DOM select control to the correct selection; the select control always shows the server state.

When the user changes the selection in the control, a `change` event is sent to the server and the selection model is changed accordingly (correcting the control if necessary) and fires a `javax.swing.event.ListDataEvent` sent to the registered `javax.swing.event.ListSelectionListener` listeners.

As in combo boxes changes on selection is reflected on the `selected` attribute in server DOM, but this attribute must be considered read only (this rule change if the component is in "markup driven" mode explained further).

7.9.3 Free lists

`ItsNatListMultSel` supports "free element lists" too:

```
    <table border="1px" cellspacing="0" cellpadding="5px">
        <tbody id="compId">
            <tr><td><b>Cell/Row Pattern</b></td></tr>
        </tbody>
    </table>
```

The Java code:

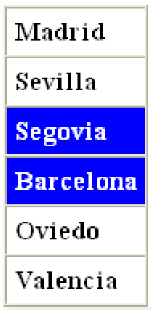
```
    ItsNatDocument itsNatDoc = ...;
    ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

    ItsNatFreeListMultSel listComp =
        (ItsNatFreeListMultSel)componentMgr.createItsNatComponentById("compId",
                "freeListMultSel",null);

    DefaultListModel dataModel = (DefaultListModel)listComp.getListModel();
    ...

    ListSelectionModel selModel = listComp.getListSelectionModel();
    selModel.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);

    selModel.addListSelectionListener(new ListSelectionDecorator(listComp));

    selModel.setSelectionInterval(2,3);

    ...
    listComp.addEventListener("click",evtListener);
    ...
```

Now the `click` event changes the selected items, ItsNat detects if the `SHIFT` and `CRTL` keys are pressed, selection behaviour is the same as Swing. ItsNat does not impose a default decoration; our example uses a custom `ListSelectionListener` to decorate the selected items (or to remove the decoration):

```
public class ListSelectionDecorator implements ListSelectionListener
{
    protected ItsNatListMultSel comp;

    public ListSelectionDecorator(ItsNatListMultSel comp)
    {
        this.comp = comp;
    }

    public void valueChanged(ListSelectionEvent e)
    {
        if (e.getValueIsAdjusting())
```

```
            return;

        int first = e.getFirstIndex();
        int last = e.getLastIndex();

        ListSelectionModel selModel = (ListSelectionModel)e.getSource();

        for(int i = first; i <= last; i++)
        {
            decorateSelection(i,selModel.isSelectedIndex(i));
        }
    }

    public void decorateSelection(int index,boolean selected)
    {
        Element elem = comp.getItsNatListUI().getContentElementAt(index);
        if (elem == null) return;

        if (selected)
            elem.setAttribute("style","background:rgb(0,0,255); color:white;");
        else
            elem.removeAttribute("style");
    }
}
```

With this reusable decorator finally renders:



7.9.4 User defined renderers and editors

List renderers and editors are conceptually the same as renderers and editors in labels. A renderer or an editor knows how a list item is rendered or edited.

7.9.4.1 User defined renderers

A user defined renderer must implement the interface: `ItsNatListCellRenderer`, specifically the method:

```
    public void renderListCell(
                ItsNatList list,
                int index,
                Object value,
                boolean isSelected,
                boolean cellHasFocus,
                Element cellElem,
                boolean isNew);
```

This method is called when a new item is added to the list or when an item value is updated. Default renderer ignores `isSelected` and `cellHasFocus` parameters[67] and uses the default `ElementRenderer` behind the scenes. The `cellElem` parameter is the DOM element parent of the content of the list item rendered.

A user defined renderer must be set into the component with:

```
ItsNatList.setItsNatListCellRenderer(ItsNatListCellRenderer)
```

The "Feature Showcase" has an example of a user defined renderer with form controls: Free List Compound. This example shows how we can use render/unrender methods to build new components when a new list item is created (`renderListCell`) and how to dispose them before the same list item is removed (`unrenderListCell`).

7.9.4.2 User defined editors

A user defined editor must implement the interface: `ItsNatListCellEditor`, specifically the method:

```
public ItsNatComponent getListCellEditorComponent(
            ItsNatList list,
            int index,
            Object value,
            boolean isSelected,
            Element cellElem);
```

The `cellElem` parameter is the DOM element parent of the content of the list item rendered. Returned value can be null.

This method is called when a list item is going to be edited "in-place", typically with a `dblclick` over the item area. Default editors are the same as label editors (same behavior).

A user defined editor must be set into the component with:

```
ItsNatList.setItsNatListCellEditor(ItsNatListCellEditor)
```

7.9.5 User defined structures

List structures are `ItsNatListStructure` objects and share the same concepts seen with `ElementListStructure` in "Core":

```
public interface ItsNatListStructure
{
    public Element getContentElement(ItsNatList list,int index,Element parentElem);
}
```

The method `getContentElement` returns the DOM element parent of the item "content". This method is called before calling the renderer to pass the returned DOM element. Default implementation uses the default `ElementListStructure` implementation behind the scenes.

[67] User defined renderers can ignore these parameters too because focus decoration of an item is not important in web and is not managed by ItsNat, and selection decoration can be managed with selection listeners, renderer is not called when an item is selected or unselected.

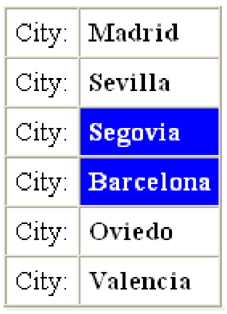
User defined structures must be provided to the component in creation time by using "artifacts" and can not be replaced.

`NameValue` is an ItsNat class provided to specify a pair name-value object used to generalize factory methods (and to provide future enhancements without API changes). An ItsNat list recognizes a structure object using the artifact with name "`useStructure`". Artifacts can be passed to the component as an array using a factory method with three parameters.

The following example creates and submits to the list as an artifact, a user defined structure returning the second cell of every row as the item content parent:

```
<table border="1px" cellspacing="0" cellpadding="5px">
    <tbody id="compId">
        <tr><td>City:</td><td><b>Name</b></td></tr>
    </tbody>
</table>

ItsNatComponentManager componentMgr = ...;

ItsNatListStructure struct = new ItsNatListStructure()
{
    public Element getContentElement(ItsNatList list, int index,
                   Element parentElem)
    {
        HTMLTableRowElement rowElem = (HTMLTableRowElement)parentElem;
        HTMLTableCellElement firstCell =
           (HTMLTableCellElement)ItsNatTreeWalker.getFirstChildElement(rowElem);
        HTMLTableCellElement secondCell =
         (HTMLTableCellElement)ItsNatTreeWalker.getNextSiblingElement(firstCell);
        return secondCell;
    }
};

NameValue[] artifacts =
                   new NameValue[] { new NameValue("useStructure",struct) };
ItsNatFreeListMultSel listComp =
    (ItsNatFreeListMultSel)componentMgr.createItsNatComponentById(
           "compId","freeListMultSel",artifacts);
...
```

Renders:



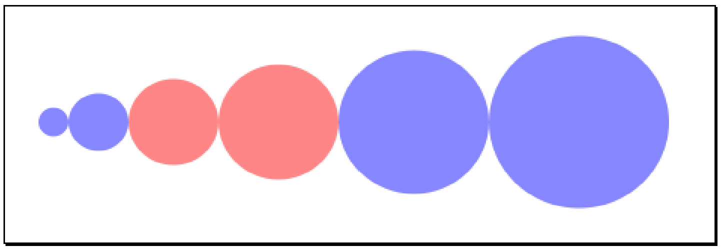
City:	**Madrid**
City:	**Sevilla**
City:	**Segovia**
City:	**Barcelona**
City:	**Oviedo**
City:	**Valencia**

7.9.6 Free lists and combos with non-HTML elements

`ItsNatFreeComboBox` and `ItsNatFreeListMultSel` are not (X)HTML namespace dependent, and they may be used with other namespaces like SVG; a mouse click can be used to select a SVG based list item. The ItsNat "Feature Showcase" shows a "live" SVG element list example, a circle list where every circle can be selected, removed, added etc.

The follow HTML code is the pattern used to render a circle list with two circles "selected":

```
<svg id="compId" itsnat:nocache="true" width="700" height="300"
           xmlns="http://www.w3.org/2000/svg">
     <circle cx="70" cy="150" r="70" fill="#0000ff" fill-opacity="0.5"/>
</svg>
```



7.10 TABLES

As lists ItsNat leverages the "pattern based table" approach to a component level: binding a markup layout pattern, a data model (an object table) and a selection model ready to listen and process events, using pluggable renderers and structural layouts like pattern based tables in the ItsNat Core module.

Table components implement the interface `ItsNatTable`. `ItsNatTable` is inspired in `javax.swing.JTable`, like `JTable` it uses `javax.swing.table.TableModel` as the data model, (`javax.swing.table.DefaultTableModel` by default). `ItsNatTable` based components use two `javax.swing.ListSelectionModel` as the selection model of rows and columns (`javax.swing.DefaultListSelectionModel` by default); they are combined as in `JTable`.

Markup	Interface
`<table>`	**`ItsNatHTMLTable`**
`<`*`any`*` [itsnat:compType="freeTable"]>`	**`ItsNatFreeTable`**

The following example shows a table of Spanish cities, public squares and monuments, using a table component:

```
<table id="compId" border="1px" cellspacing="0" cellpadding="5px">
    <thead>
        <tr style="background: rgb(220,220,220)">
            <th><b>Cell pattern</b></th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td><b>Cell pattern</b></td>
        </tr>
    </tbody>
```

```
</table>


ItsNatDocument itsNatDoc = ...;
ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

ItsNatHTMLTable tableComp =
            (ItsNatHTMLTable)componentMgr.createItsNatComponentById("compId");

DefaultTableModel dataModel = (DefaultTableModel)tableComp.getTableModel();
dataModel.addColumn("City");
dataModel.addColumn("Public square");
dataModel.addColumn("Monument");
dataModel.addRow(new String[] {"Madrid","Plaza Mayor","Palacio Real"});
dataModel.addRow(new String[] {"Sevilla","Plaza de España","Giralda"});
dataModel.addRow(new String[] {"Segovia","Plaza del Azoguejo",
           "Acueducto Romano"});

ListSelectionModel rowSelModel = tableComp.getRowSelectionModel();
ListSelectionModel columnSelModel = tableComp.getColumnSelectionModel();

rowSelModel.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
columnSelModel.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);

tableComp.setRowSelectionAllowed(true);
tableComp.setColumnSelectionAllowed(false);

rowSelModel.addListSelectionListener(new TableRowSelectionDecoration(tableComp));

rowSelModel.setSelectionInterval(1,1);

TableModelListener dataListener = new TableModelListener()
{
    public void tableChanged(TableModelEvent e)
    {
        int firstRow = e.getFirstRow();
        int lastRow = e.getLastRow();

        String action = "";
        int type = e.getType();
        switch(type)
        {
            case TableModelEvent.INSERT: action = "Added"; break;
            case TableModelEvent.DELETE: action = "Removed"; break;
            case TableModelEvent.UPDATE: action = "Changed"; break;
        }

        String interval = "";
        if (firstRow != -1)
            interval = " interval " + firstRow + "-" + lastRow;

        System.out.println(action + " " + interval);
    }
};
dataModel.addTableModelListener(dataListener);

ListSelectionListener rowSelListener = new ListSelectionListener()
{
    public void valueChanged(ListSelectionEvent e)
```

```
        {
            if (e.getValueIsAdjusting())
                return;

            ListSelectionModel rowSelModel = (ListSelectionModel)e.getSource();

            int first = e.getFirstIndex();
            int last = e.getLastIndex();

            String fact = "";
            for(int i = first; i <= last; i++)
            {
                boolean selected = rowSelModel.isSelectedIndex(i);
                if (selected)
                    fact += ", selected ";
                else
                    fact += ", deselected ";
                fact += i;
            }

            System.out.println("Selection changes" + fact);
        }
    };
    rowSelModel.addListSelectionListener(rowSelListener);
```

The previous example shows how to create a table with multiple row selection:

```
    rowSelModel.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
    columnSelModel.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);[68]

    tableComp.setRowSelectionAllowed(true);
    tableComp.setColumnSelectionAllowed(false);
```

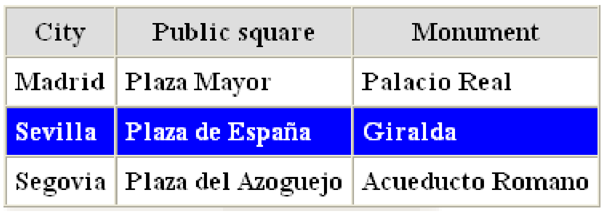
The `click` event changes the selected rows, ItsNat detects if the SHIFT and CRTL keys are pressed, selection behaviour is the same as Swing. ItsNat does not impose a default decoration; our example uses a custom `ListSelectionListener` to decorate the selected/unselected rows:

```
public class TableRowSelectionDecoration implements ListSelectionListener
{
    protected ItsNatTable comp;

    public TableRowSelectionDecoration(ItsNatTable comp)
    {
        this.comp = comp;
    }

    public void valueChanged(ListSelectionEvent e)
    {
        if (e.getValueIsAdjusting())
            return;

        int first = e.getFirstIndex();
        int last = e.getLastIndex();
```

[68] This sentence is not really needed because column selection is disabled

```
        ListSelectionModel selModel = (ListSelectionModel)e.getSource();

        for(int i = first; i <= last; i++)
        {
            decorateSelection(i,selModel.isSelectedIndex(i));
        }
    }

    public void decorateSelection(int row,boolean selected)
    {
        ItsNatTableUI tableUI = comp.getItsNatTableUI();
        int cols = comp.getTableModel().getColumnCount();
        for(int i = 0; i < cols; i++)
        {
            Element cellElem = tableUI.getCellContentElementAt(row,i);
            if (cellElem == null) return;
            decorateSelection(cellElem,selected);
        }
    }

    public void decorateSelection(Element elem,boolean selected)
    {
        if (selected)
            elem.setAttribute("style","background:rgb(0,0,255); color:white;");
        else
            elem.removeAttribute("style");
    }
}
```

Finally renders:



City	Public square	Monument
Madrid	Plaza Mayor	Palacio Real
Sevilla	Plaza de España	Giralda
Segovia	Plaza del Azoguejo	Acueducto Romano

7.10.1 Table header support

ItsNatTable supports an optional header; in a <table> element the component automatically recognizes the <thead> element, this header is managed as if it was a list. The method ItsNatTable.**getItsNatTableHeader**() returns an ItsNatTableHeader object, this interface offers some control over the header like to get/set a header renderer (there is no column header editor) and a specific selection model of the header columns seen as a list.

The header is ready to receive click events; a complete column is selected by clicking the header column (only if column selection is enabled and row selection disabled) and the selection model is updated with this selection. The header selection model may be used to do typical tasks like to sort the rows by the selected column.

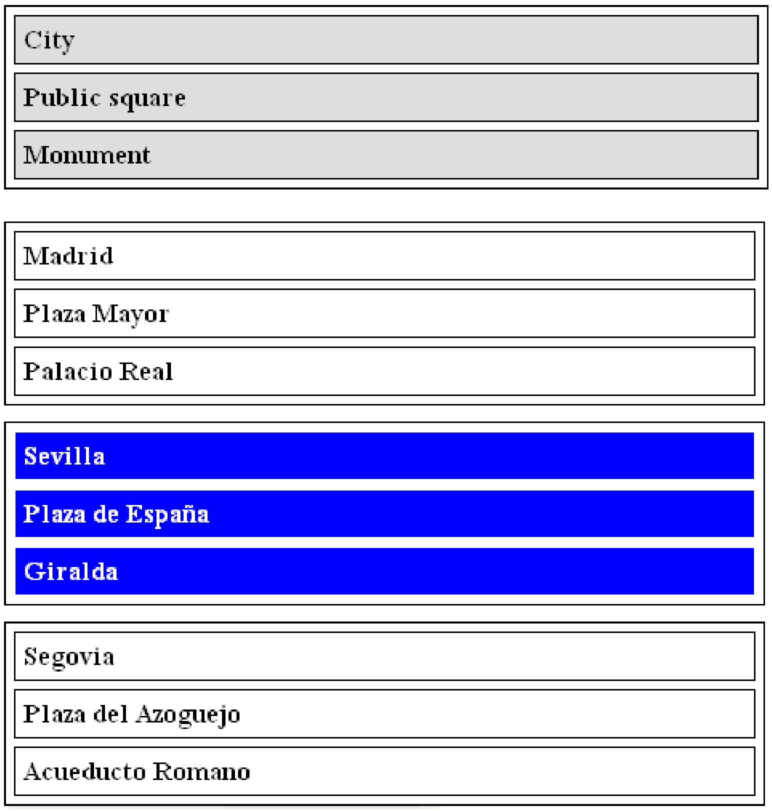
7.10.2 Free tables

ItsNatTable supports "free element tables" using ItsNatFreeTable. The following markup shows an example of a "table" without the HTML <table> element:

```
    <style type="text/css">
        .freeCell
        {
            margin: 5px;
            padding: 5px;
            border: 1px solid;
        }
    </style>

    ...

    <div id="compId">
         <div style="width: 100%; border: 1px solid;">
             <div class="freeCell" style="background: rgb(220,220,220);"><b>Cell
Pattern</b></div>
         </div>
         <div style="margin-top: 10px; width: 100%;">
             <div style="margin-top: 10px; border: 1px solid;">
                 <div class="freeCell"><b>Cell Pattern</b></div>
             </div>
         </div>
    </div>
```

The Java code may be the same as the code used with `<table>`, replacing `ItsNatHTMLTable` with `ItsNatFreeTable`[69].

The following image shows how is rendered using the default renderer:



7.10.3 User defined renderers and editors

[69] The code is identical if the interface `ItsNatTable` is used.

Table renderers and editors are conceptually the same as renderers and editors in lists and labels. A renderer or an editor knows how a table cell is rendered or edited.

7.10.3.1 User defined renderers

A user defined renderer must implement the interface: `ItsNatTableCellRenderer`, specifically the method:

```
    public void renderTableCell(
                ItsNatTable table,
                int row,
                int column,
                Object value,
                boolean isSelected,
                boolean hasFocus,
                Element cellElem,
                boolean isNew);
```

This method is called when a new cell is added to the table or when a cell value is updated. Default renderer is basically the same as lists and labels; it ignores `isSelected` and `hasFocus` parameters and uses the default `ElementRenderer` behind the scenes. The `cellElem` parameter is the DOM element parent of the content of the cell item rendered.

A user defined renderer must be set into the component with:

```
  ItsNatTable.setItsNatTableCellRenderer(ItsNatTableCellRenderer)
```

The "Feature Showcase" has an example of a user defined renderer with SVG circles: Free Table SVG.

Header renderers use the interface `ItsNatTableHeaderCellRenderer`:

```
public interface ItsNatTableHeaderCellRenderer
{
    public void renderTableHeaderCell(
                ItsNatTableHeader header,
                int column,
                Object value,
                boolean isSelected,
                boolean cellHasFocus,
                Element cellElem
                boolean isNew);
}
```

This interface is absolutely symmetric to the `ItsNatListCellRenderer` interface; the default renderer works the same as the default list renderer.

7.10.3.2 User defined editors

A user defined editor must implement the interface `ItsNatTableCellEditor`, specifically the method:

```
    public ItsNatComponent getTableCellEditorComponent(
                ItsNatTable table,
                int row,
                int column,
                Object value,
                boolean isSelected,
                Element cellElem);
```

The cellElem parameter is the DOM element parent of the content of the cell rendered. Returned value can be null.

This method is called when a table cell is going to be edited "in-place", typically with a dblclick over the cell area. Default editors are the same as list and label editors (same behavior).

A user defined editor must be set into the component with:

```
  ItsNatTable.setItsNatTableCellEditor(ItsNatTableCellEditor)
```

7.10.4 User defined structures

Table structures are ItsNatTableStructure objects and share the same concepts seen with ElementTableStructure in "Core", but now it includes header support:

```
public interface ItsNatTableStructure
{
    public Element getHeadElement(ItsNatTable table,Element tableElem);
    public Element getBodyElement(ItsNatTable table,Element tableElem);

    public Element getHeaderColumnContentElement(ItsNatTableHeader tableHeader,
                        int index,Element parentElem);
    public Element getRowContentElement(ItsNatTable table,int row,Element rowElem);
    public Element getCellContentElement(ItsNatTable table,int row,int col,
                        Element cellElem);
}
```

Default implementation uses the default ElementTableStructure implementation behind the scenes to implement getRowContentElement and getCellContentElement. The methods getHeadElement and getBodyElement returns the head and body DOM elements, default implementation supports <thead> and <tbody> out of the box[70], head and body parent element identification in free tables is supported following the <thead> and <tbody> location style.

User defined structures must be provided to the component in creation time by using artifact objects and can not be replaced.

As in lists and labels, an ItsNat table receives a structure object using an artifact with the name "useStructure", passed inside an array calling a factory method.

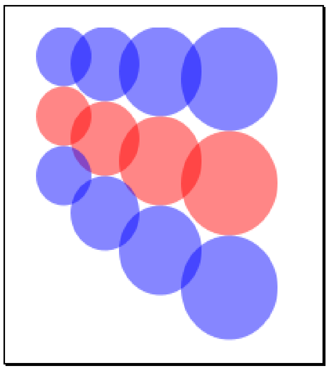
7.10.5 Free tables with non-HTML elements

ItsNatFreeTable is not (X)HTML namespace dependent, and may be used with other namespaces like SVG. The ItsNat "Feature Showcase" shows a "live" SVG element table example, a circle table where every circle row can be selected, removed, added etc.

The follow HTML code is the pattern used to render a circle table with one circle row "selected":

[70] Use ever a <tbody> element in tables, by default ItsNat adds a <tbody> element if missing

```
    <svg id="compId" itsnat:nocache="true" width="700" height="300"
                xmlns="http://www.w3.org/2000/svg">
        <g>
            <circle cx="70" cy="150" r="70" fill="#0000ff" fill-opacity="0.5" />
        </g>
    </svg>
```



7.11 TREES

Tree components leverage the "pattern based tree" approach to a component level: binding a markup layout pattern, a data model (an object tree) and a selection model ready to listen and process events, using pluggable renderers and structural layouts like pattern based trees in the ItsNat Core module.

Tree components implement the interface `ItsNatTree`. `ItsNatTree` is inspired in `javax.swing.JTree`, like `JTree` it uses `javax.swing.tree.TreeModel` as the data model, (`javax.swing.tree.DefaultTreeModel` by default[71]). `ItsNatTree` based components use `javax.swing.tree.TreeSelectionModel` as the selection model (`javax.swing.tree.DefaultTreeSelectionModel` by default), this selection model "sees" the tree as a node list (every tree node is a "row"), `ItsNatTree` implementations respect and support this approach and include the concept of "row".

There is no a standard HTML `<tree>` element, ItsNat only offers a tree implementation: the "free tree" and two variants: "normal" and "tree table".

Markup	Interface	isTreeTable()
<*any* [itsnat:compType="freeTree"]>	**ItsNatFreeTree**	**false**
<*any* [itsnat:compType="freeTree"] [itsnat:treeTable="true"] >	**ItsNatFreeTree**	**true**

The following example constructs a tree with the characters of the famous TV series Grey's Anatomy, using a tree component:

```
    <ul id="compId" style="list-style-type: none;">
        <li><span><img src="img/tree/tree_node_expanded.gif"><img src="img/tree/
tree_folder_open.gif"><span><b>Item Pattern</b></span></span>
```

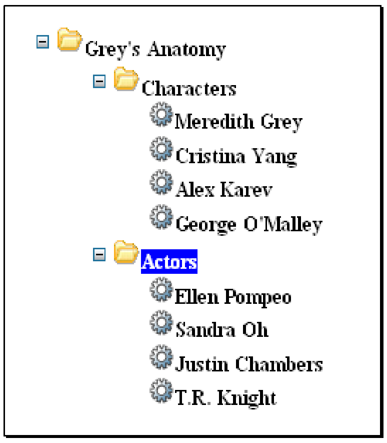
[71] The TreeModel interface does not impose TreeNode objects as tree elements, nor ItsNat, a user defined TreeModel can use any type of class as tree nodes into an ItsNatTree.

```
        <ul style="list-style-type: none;"></ul>
    </li>
</ul>


ItsNatDocument itsNatDoc = ...;
ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

ItsNatFreeTree treeComp =
(ItsNatFreeTree)componentMgr.createItsNatComponentById("compId","freeTree",null);

DefaultTreeModel dataModel = (DefaultTreeModel)treeComp.getTreeModel();
treeComp.setTreeModel(dataModel);

new FreeTreeDecorator(treeComp).bind();

DefaultMutableTreeNode parentNode;

DefaultMutableTreeNode rootNode = new DefaultMutableTreeNode("Grey's Anatomy");
dataModel.setRoot(rootNode);

    parentNode = addNode("Characters",rootNode,dataModel);

        addNode("Meredith Grey",parentNode,dataModel);
        addNode("Cristina Yang",parentNode,dataModel);
        addNode("Alex Karev",parentNode,dataModel);
        addNode("George O'Malley",parentNode,dataModel);

    parentNode = addNode("Actors",rootNode,dataModel);

        addNode("Ellen Pompeo",parentNode,dataModel);
        addNode("Sandra Oh",parentNode,dataModel);
        addNode("Justin Chambers",parentNode,dataModel);
        addNode("T.R. Knight",parentNode,dataModel);

TreeSelectionModel selModel = treeComp.getTreeSelectionModel();
selModel.setSelectionMode(TreeSelectionModel.CONTIGUOUS_TREE_SELECTION);

selModel.addSelectionPath(new TreePath(parentNode.getPath())); // Actors

TreeModelListener dataListener = new TreeModelListener()
{
    public void treeNodesChanged(TreeModelEvent e)
    {
        treeChangedLog(e);
    }

    public void treeNodesInserted(TreeModelEvent e)
    {
        treeChangedLog(e);
    }

    public void treeNodesRemoved(TreeModelEvent e)
    {
        treeChangedLog(e);
    }

    public void treeStructureChanged(TreeModelEvent e)
    {
        treeChangedLog(e);
    }

    public void treeChangedLog(TreeModelEvent e)
    {
        System.out.println(e.toString());
    }
};
dataModel.addTreeModelListener(dataListener);

TreeSelectionListener selListener = new TreeSelectionListener()
{
    public void valueChanged(TreeSelectionEvent e)
    {
        TreeSelectionModel selModel = (TreeSelectionModel)e.getSource();

        TreePath[] paths = e.getPaths();
        String fact = "";
        for(int i = 0; i < paths.length; i++)
        {
            TreePath path = paths[i];
            boolean selected = selModel.isPathSelected(path);
            if (selected)
                fact += ", selected ";
            else
                fact += ", deselected ";
            fact += path.getLastPathComponent();
        }

        System.out.println("Selection changes" + fact);
    }
};
selModel.addTreeSelectionListener(selListener);

TreeWillExpandListener willExpandListener = new TreeWillExpandListener()
{
    public void treeWillExpand(TreeExpansionEvent event)
                 throws ExpandVetoException
    {
        // Will expand
    }

    public void treeWillCollapse(TreeExpansionEvent event)
                 throws ExpandVetoException
    {
        // Will collapse
    }
};
treeComp.addTreeWillExpandListener(willExpandListener);
```

And the utility method:

```
    public static DefaultMutableTreeNode addNode(Object userObject,
               DefaultMutableTreeNode parentNode,DefaultTreeModel dataModel)
    {
        DefaultMutableTreeNode childNode = new DefaultMutableTreeNode(userObject);
        int count = dataModel.getChildCount(parentNode);
        dataModel.insertNodeInto(childNode,parentNode,count);
        return childNode;
    }
```

This is how is rendered:



Trees are more complex than tables or lists if you want features like expansible/collapsible nodes or automatic folder/leaf look and feel. `ItsNatTree` implementation does not impose these features because they are very open, is up to you if you want these features because they can implemented using normal listeners. To help you with this task, the ItsNat "Feature Showcase" includes a class `FreeTreeDecorator` providing these features; our example has expansible/collapsible nodes and automatic folder/leaf decoration because uses this class.

A tree node can have a handle, an icon and a label. ItsNat offers "structural" support of theses elements, but no concrete visual rendering is done by default.

The `FreeTreeDecoration` class is a `TreeModelListener` because we need to detect when a new subtree is inserted or removed[72]. A new node/subtree has many visual implications: handlers and icons of the new subtree and the parent of the new node must be updated accordingly. If the node/subtree is removed only the parent must be updated. A custom renderer is useful to render the label but we have no information if the node is just inserted or updated.

This is how `FreeTreeDecorator` starts:

```
public class FreeTreeDecorator implements TreeModelListener, TreeSelectionListener,
                                          TreeExpansionListener
{
    protected ItsNatFreeTree comp;

    public FreeTreeDecorator(ItsNatFreeTree comp)
    {
        this.comp = comp;
    }

    public void bind()
    {
        TreeModel dataModel = comp.getTreeModel();
        dataModel.addTreeModelListener(this);
        /* Added before to call setTreeModel again because it must be called
        last (the last registered is the first called, the component register a
        listener to add/remove DOM elements) */
        comp.setTreeModel(dataModel);
```

[72] A node may be the parent of a subtree

]

```
        /* Resets the internal listeners, the internal TreeModelListener
        listener is called first */
        comp.addTreeExpansionListener(this);

        TreeSelectionModel selModel = comp.getTreeSelectionModel();
        selModel.addTreeSelectionListener(this);
    }

    ...
}
```

This class of course is not "definitive", you can adapt to your taste, change images and so on.

Until now all data model listeners did not do any visual manipulation, only in trees we need to change the DOM using this type of listener. We have a problem: the component internally has a registered data model listener, this listener adds, updates and removes the markup when a data model item is added, updated or removed[73]. If we need to update the visual appearance of a just inserted node, the node markup must be already inserted, wherefore our data model listener must be called *after* the internal component data model listener.

If you add a listener to a `DefaultTreeModel` object this listener will be called *the first*[74] , this is not what we want. To solve this problem ItsNat components offer a "hack": if a data model is set again into the component, the components think it is a (false) new data model and removes and add again the internal listener, this listener now is the last and will be called *first*. This explains the following code fragment of `FreeTreeDecorator`:

```
    public void bind()
    {
        TreeModel dataModel = comp.getTreeModel();
        dataModel.addTreeModelListener(this);
        comp.setTreeModel(dataModel);
        ...
```

The next line adds a new type of listener:

```
        comp.addTreeExpansionListener(this);
```

The `TreeExpansionListener` is called when a node is going to be expanded or collapsed. The `ItsNatTree` component keeps track of the expanded/collapsed state of tree nodes; when a node handler is clicked, the component automatically swaps the node state and calls the listener; if a node icon is clicked the node is expanded (and listener is notified). The `FreeTreeDecorator` is a `TreeExpansionListener` listener, when a node is expanded/collapsed the decorator updates the handler icon and the subtree visibility.

We can control when a node can be expanded or collapsed using a `TreeWillExpandListener` and throwing an `ExpandVetoException` when appropriate. The following implementation avoids the collapse of any node:

```
    public void treeWillExpand(TreeExpansionEvent event)
                    throws ExpandVetoException
    {
        // Will expand
```

[73] This behavior is applied in lists, tables and trees.

[74] This behavior is the same as in DefaultListModel, DefaultTableModel etc

```
    }

    public void treeWillCollapse(TreeExpansionEvent event)
                    throws ExpandVetoException
    {
        throw new ExpandVetoException(event);
    }
```

The `FreeTreeDecorator` is a `TreeSelectionListener` implementation, to decorate the node label when is selected/unselected. Must be registered into the `TreeSelectionModel`:

```
        TreeSelectionModel selModel = comp.getTreeSelectionModel();
        selModel.addTreeSelectionListener(this);
```

A node is selected (or unselected) when is clicked; ItsNat detects if the `SHIFT` and `CRTL` keys are pressed, selection behaviour is the same as Swing. ItsNat trees support all selection modes.

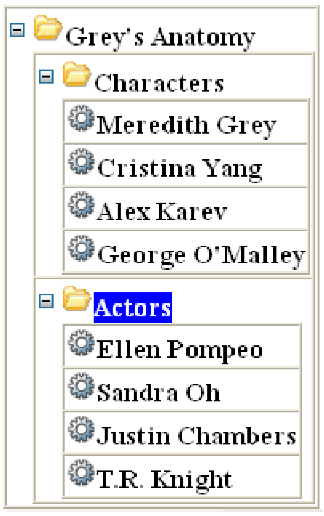
7.11.1 Trees with <table> based nodes

ItsNat trees support "out the box" `<table>` based nodes like the following template:

```
    <table border="1px" cellspacing="0">
        <tbody id="compId">
        <tr>
            <td><span><img src="img/tree/tree_node_expanded.gif">
                      <img src="img/tree/tree_folder_open.gif">
                      <span><b>Item Pattern</b></span>
                 </span>
                 <table style="margin-left:15px;" border="1px" cellspacing="0">
                     <tbody/>
                 </table>
            </td>
        </tr>
        </tbody>
    </table>
```

Using the same Java code of the previous example renders[75]:

[75] Table elements have borders to show the table based layout, remove this border in a production version if you do not like it.



7.11.2 Rootless

Tree components can be rootless, the data model may have a root node but this node is not shown (it has not markup). Rootless trees are specified using:

1. Using the markup attribute `itsnat:`**`rootless`**`="true"`

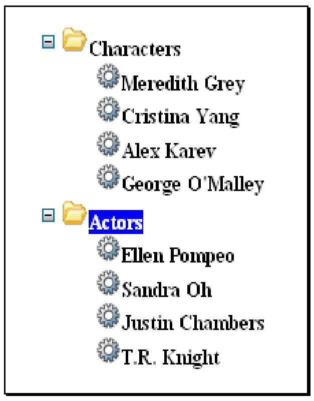
2. Using the artifact "`rootless`" with value "`true`".

```
    ...
    NameValue[] artifacts = new NameValue[]{new NameValue("rootless","true")};
    ItsNatFreeTree treeComp = (ItsNatFreeTree)compMgr.createItsNatComponentById(
                        "compId","freeTree",artifacts);
    ...
```

This code with the following markup:

```
    <ul id="compId" style="list-style-type: none;">
        <li><span><img src="img/tree/tree_node_expanded.gif"><img src="img/tree/
tree_folder_open.gif"><span><b>Item Pattern</b></span></span>
            <ul style="list-style-type: none;"></ul>
        </li>
    </ul>
```

Renders:



7.11.3 Tree-Tables

Tree table components are the component version of Core tree tables. In a tree table each node is a row. Tree tables are specified:

1. Using the markup attribute `itsnat:`**`treeTable`**`="true"`

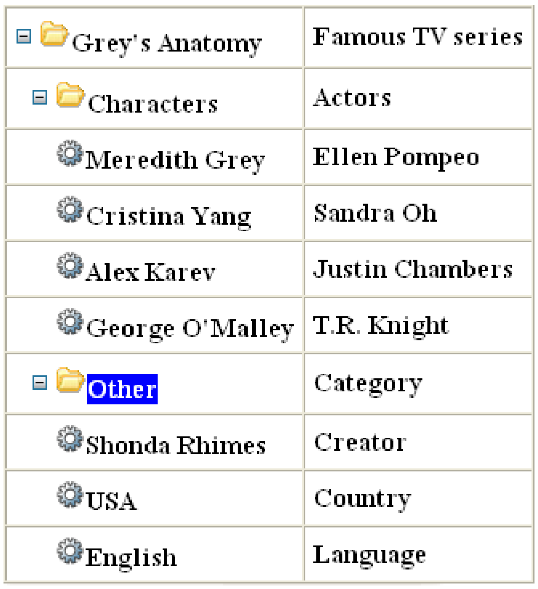
2. Using the artifact "`treeTable`" with value "`true`".

```
...
NameValue[] artifacts = new NameValue[]{new NameValue("treeTable","true")};
ItsNatFreeTree treeComp = (ItsNatFreeTree)compMgr.createItsNatComponentById(
                   "compId","freeTree",artifacts);
...
```

The "Feature Showcase" has an interesting example using a tree table. This example uses the following pattern:

```
<table border="1px" cellspacing="0" cellpadding="5px">
    <tbody id="compId">
        <tr>
            <td>
                <img src="img/tree/tree_node_expanded.gif" />
                <img src="img/tree/tree_folder_close.gif" />
                <span><b>Label</b></span>
            </td>
            <td><b>Other content</b></td>
        </tr>
    </tbody>
</table>
```

This template renders something like this (using a custom renderer):

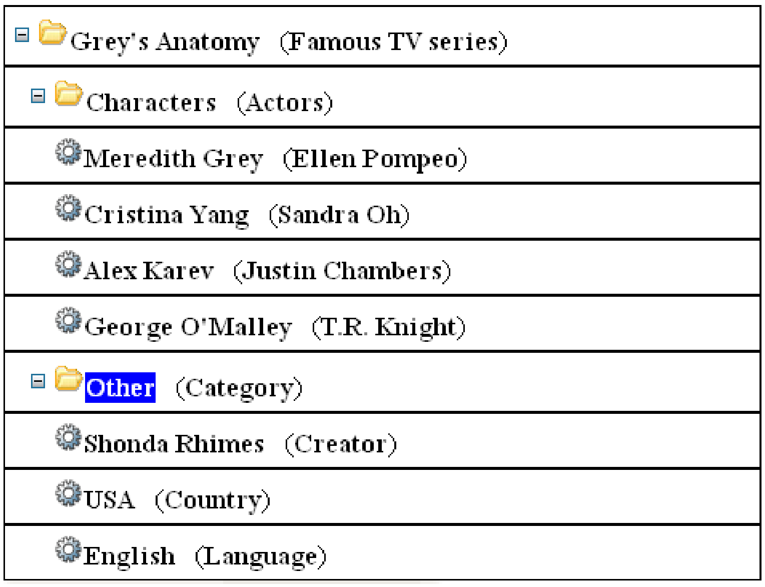


Grey's Anatomy	Famous TV series
Characters	Actors
Meredith Grey	Ellen Pompeo
Cristina Yang	Sandra Oh
Alex Karev	Justin Chambers
George O'Malley	T.R. Knight
Other	Category
Shonda Rhimes	Creator
USA	Country
English	Language

Again tree tables can be designed without `<table>` (using the same Java code):

```
<div id="compId">
    <p style="border: 1px solid; margin:0; padding:5px;">
        <img src="img/tree/tree_node_expanded.gif" />
        <img src="img/tree/tree_folder_close.gif" />
        <span><b>Label</b></span>
        &nbsp; (<span><b>Other</b></span>)
    </p>
</div>
```

Renders:


⊟ Grey's Anatomy (Famous TV series)
⊟ Characters (Actors)
Meredith Grey (Ellen Pompeo)
Cristina Yang (Sandra Oh)
Alex Karev (Justin Chambers)
George O'Malley (T.R. Knight)
⊟ Other (Category)
Shonda Rhimes (Creator)
USA (Country)
English (Language)

7.11.4 User defined renderers and editors

Tree renderers and editors are conceptually the same as renderers and editors in tables, lists and labels. A renderer or an editor knows how a tree node is rendered or edited, usually is used to render the label part of the node.

7.11.4.1 User defined renderers

A user defined renderer must implement the interface `ItsNatTreeCellRenderer`, specifically the method:

```
public void renderTreeCell(
            ItsNatTree tree,
            Object value,
            int row,
            boolean isSelected,
            boolean isExpanded,
            boolean isLeaf,
            boolean hasFocus,
            Element treeNodeLabelElem
            boolean isNew);
```

This method is called when a new node is added to the tree or when a node value is updated. Default renderer is basically the same as tables, lists and labels; it ignores `isSelected`, `isExpanded`, `isLeaf` and `hasFocus` parameters and uses the default `ElementRenderer` behind the scenes to render the label part. The `treeNodeLabelElem` parameter is the DOM element parent of the label element rendered.

A user defined renderer must be set into the component with:

```
ItsNatTree.setItsNatTreeCellRenderer(ItsNatTreeCellRenderer)
```

7.11.4.2 User defined editors

A user defined editor must implement the interface `ItsNatTreeCellEditor`, specifically the method:

```
    public ItsNatComponent getTreeCellEditorComponent(
                ItsNatTree tree,
                int row,
                Object value,
                boolean isSelected,
                boolean expanded,
                boolean leaf,
                Element labelElem);
```

The `labelElem` parameter is the DOM element of the node label part. Returned value can be null.

This method is called when a node label is going to be edited "in-place", typically with a `dblclick` over the cell area. Default editors are the same as table, list and label editors (same behavior).

A user defined editor must be set into the component with:

```
  ItsNatTree.setItsNatTreeCellEditor(ItsNatTreeCellEditor)
```

7.11.5 User defined structures

Tree structures are `ItsNatTreeStructure` objects and share the same concepts seen with `ElementTreeNodeStructure` in "Core":

```
public interface ItsNatTreeStructure
{
    public Element getContentElement(ItsNatTree tree,int row,Element nodeParent);
    public Element getHandleElement(ItsNatTree tree,int row,Element nodeParent);
    public Element getIconElement(ItsNatTree tree,int row,Element nodeParent);
    public Element getLabelElement(ItsNatTree tree,int row,Element nodeParent);
    public Element getChildListElement(ItsNatTree tree,int row,Element nodeParent);
}
```

Default implementation uses the default `ElementTreeNodeStructure` implementation. User defined structures must be provided to the component in creation time by using artifact objects and can not be replaced.

As in tables, lists and labels, an ItsNat tree receives a structure object using an artifact with the name "`useStructure`", passed inside an array calling a factory method.

7.11.6 Direct factory method

ItsNat provides a direct method to specify tree-table, rootless and structure avoiding artifacts:

```
  ItsNatFreeTree createItsNatFreeTree(Element elem,boolean treeTable,boolean rootless,
                ItsNatTreeStructure structure,NameValue[] artifacts);
```

7.11.7 Free trees with non-HTML elements

`ItsNatFreeTree` is not (X)HTML namespace dependent, and may be used with other namespaces like SVG.

7.12 FORMS

An HTML `<form>` element can be bound to a server side `ItsNatHTMLForm` component. This component does not do very much because there is no view management. But a `<form>` element has two key events: `submit` and `reset`. An `ItsNatHTMLForm` component can be used to bind listeners to listen `submit` and `reset` events; a submit event can be cancelled on server side, but this event must be sent in synchronous mode (see "Asynchronous-Hold" chapter).

There is another use: the original DOM `HTMLFormElement` specification has two methods, `submit()` and `reset()`, these methods are dummy in Xerces. `ItsNatHTMLForm` implements these methods as remote methods, if called these calls are sent to the client as JavaScript.

The following example shows how we can cancel a submit event and call the reset method from server side:

```
<form id="formId">
    <input type="text" value="Any Text" />
    <input type="submit" value="Submit" />
    <input type="reset" value="Reset" />
</form>
<br />
<a href="javascript:void(0)" id="linkId">Call reset from server</a>

ItsNatDocument itsNatDoc = ...;
ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

final ItsNatHTMLForm formComp =
           (ItsNatHTMLForm)componentMgr.createItsNatComponentById("formId");

formComp.setEventListenerParams("submit",false,SyncMode.SYNC,null,null,-1);
formComp.setEventListenerParams("reset",false,SyncMode.SYNC,null,null,-1);

EventListener evtListener = new EventListener()
{
   public void handleEvent(Event evt)
    {
        System.out.println(evt.getCurrentTarget() + " " + evt.getType());

        EventTarget currentTarget = evt.getCurrentTarget();
        if (currentTarget == formComp.getHTMLFormElement())
        {
            if (evt.getType().equals("submit"))
            {
                System.out.println("Submit canceled");
                evt.preventDefault();
                 // Cancels the submission, only works in SYNC mode
            }
            // reset is not cancellable
        }
        else // Link
        {
            formComp.reset(); // submit() method is defined too
        }
    }
};
formComp.addEventListener("submit",evtListener);
formComp.addEventListener("reset",evtListener);

ItsNatHTMLAnchor linkComp =
    (ItsNatHTMLAnchor)componentMgr.createItsNatComponentById("linkId");
linkComp.addEventListener("click",evtListener);
```

7.13 INCLUDES

The `ItsNatInclude` component helps to include/remove a markup fragment easily.

Because there is no HTML `<include>` element, only a "free" version is implemented using the interface `ItsNatFreeInclude`, any element can be bound to this component.

Markup	Interface
<*any* [itsnat:compType="freeInclude"]>	**ItsNatFreeInclude**

`ItsNatInclude` has two important methods:

```
    public void includeFragment(String name,boolean buildComp);
    public void removeFragment();
```

The method `includeFragment` includes the specified fragment given the name; the element content is replaced with the fragment. The fragment is removed using the method `removeFragment`.

The following example shows how to include and remove a markup fragment dynamically:

```
    <input type="button" id="buttonId" value="Include " />
    <br />
    <div id="includeId" />
    <br />


    ItsNatDocument itsNatDoc = ...;
    ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

    final ItsNatFreeInclude includeComp =
                (ItsNatFreeInclude)componentMgr.createItsNatComponentById(
                        "includeId","freeInclude",null);

    final ItsNatHTMLInput buttonComp =
                (ItsNatHTMLInput)componentMgr.createItsNatComponentById("buttonId");

    EventListener evtListener = new EventListener()
    {
        public void handleEvent(Event evt)
        {
            if (includeComp.isIncluded())
```

```
                uninclude();
            else
                include();
        }

        public void include()
        {
            includeComp.includeFragment("feashow.fragmentExample",true);
            buttonComp.getHTMLInputElement().setValue("Remove");
        }

        public void uninclude()
        {
            includeComp.removeFragment();
            buttonComp.getHTMLInputElement().setValue("Include");
        }
    };
    buttonComp.addEventListener("click",evtListener);
```

7.14 USER DEFINED COMPONENTS

How to define a user defined component is not an unknown problem, we made a custom component (an editor) in the "LABELS" chapter.

In this chapter we are going to know how to add a new custom component *to the framework*, this new component will be created like any other component.

ItsNat provides a hook/filter to intercept the normal component creation. This interception offers a possibility to create new components unknown by the framework. This interceptor implements the interface CreateItsNatComponentListener:

```
public interface CreateItsNatComponentListener
{
    public ItsNatComponent before(Node node,String componentType,
                NameValue[] artifacts,ItsNatComponentManager compMgr);
    public ItsNatComponent after(ItsNatComponent comp);
}
```

This listener must be registered into the DocumentTemplate or globally on ItsNatServlet. For instance:

```
    String pathPrefix = ...;
    ItsNatHttpServlet itsNatServlet = getItsNatHttpServlet();
    DocumentTemplate docTemplate;
    docTemplate = itsNatServlet.registerDocumentTemplate("manual.comp.example",
                "text/html",pathPrefix + "comp_example.xhtml");
    ...
    docTemplate.addCreateItsNatComponentListener(new LoginCreationItsNatComponentListener());
```

When a new component is requested to the framework using any factory method (ItsNatComponentManager.**createItsNatComponent***), the listener method before is called, if this method returns a component this component will be returned, if returned null the framework tries to create a predefined component. The method after may be used to configure the component just created before returning to the user, if returns null the component is rejected (and the factory method returns null).

Following the example:

```
public class LoginCreationItsNatComponentListener
                implements CreateItsNatComponentListener
{
    public LoginCreationItsNatComponentListener()
    {
    }

    public ItsNatComponent before(Node node, String componentType,
                NameValue[] artifacts, ItsNatComponentManager compMgr)
    {
        if (node.getNodeType() != Node.ELEMENT_NODE)
            return null;

        Element elem = (Element)node;
        if ((componentType != null) && componentType.equals("login"))
            return new LoginComponent(elem,compMgr);
        return null;
    }

    public ItsNatComponent after(ItsNatComponent comp)
    {
        return comp;
    }
}
```

Our interceptor introduces a new component type, `login`, and class: `LoginComponent`.

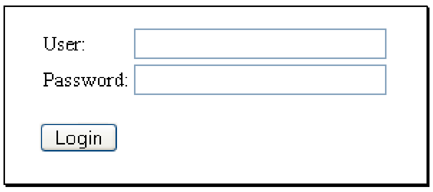
This is the `LoginComponent` source code:

```
public class LoginComponent extends MyCustomComponentBase implements ActionListener
{
    protected ItsNatHTMLInputText userComp;
    protected ItsNatHTMLInputPassword passwordComp;
    protected ItsNatHTMLInputButton validateComp;
    protected Element logElem;
    protected ValidateLoginListener validateListener;

    public LoginComponent(Element parentElem,ItsNatComponentManager compMgr)
    {
        super(parentElem,compMgr);

        this.userComp =
                (ItsNatHTMLInputText)compMgr.createItsNatComponentById("userId");
        this.passwordComp =
            (ItsNatHTMLInputPassword)compMgr.createItsNatComponentById("passwordId");
        this.validateComp =
            (ItsNatHTMLInputButton)compMgr.createItsNatComponentById("validateId");

        validateComp.getButtonModel().addActionListener(this);
    }

    public void dispose()
    {
        userComp.dispose();
        passwordComp.dispose();
        validateComp.dispose();
    }
    public ValidateLoginListener getValidateLoginListener()
    {
        return validateListener;
    }

    public void setValidateLoginListener(ValidateLoginListener listener)
    {
        this.validateListener = listener;
    }

    public void actionPerformed(ActionEvent e)
    {
        if (validateListener != null)
        {
            validateListener.validate(getUser(),getPassword());
        }
    }

    public String getUser()
    {
        return userComp.getText();
    }

    public String getPassword()
    {
        return passwordComp.getText();
    }
}
```

An example of "compatible" HTML code (`LoginComponent` does not impose a concrete layout, only requires some known components and ids) is:

```
<div id="loginCompId">
  <table>
    <tbody>
      <tr>
          <td>User:</td>
          <td><input type="text" id="userId" value="User" size="25" />
          </td>
      </tr>
      <tr>
          <td>Password:</td>
          <td><input type="password" id="passwordId"
                   value="Password" size="25" />
          </td>
      </tr>
    </tbody>
  </table>
  <br />
  <input type="button" id="validateId" value="Login" />
</div>
```

Renders:



Our custom component calls the `validate` method of the registered object implementing the interface `ValidateLoginListener`:

```
public interface ValidateLoginListener
{
    public boolean validate(String user,String password);
}
```

A complete use example:

```
    final ItsNatDocument itsNatDoc = ...;
    ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

    final LoginComponent loginComp =
                (LoginComponent)componentMgr.createItsNatComponentById(
                        "loginCompId","login",null);

    ValidateLoginListener validator = new ValidateLoginListener()
    {
        public boolean validate(String user,String password)
        {
            if (!user.equals("admin"))
            {
                System.out.println("Bad user");
                return false;
            }

            if (!password.equals("1234"))
            {
                System.out.println("Bad password");
                return false;
            }

            Element loginElem = (Element)loginComp.getNode();
            loginElem.setAttribute("style","display:none");

            Document doc = loginElem.getOwnerDocument();
            Element infoElem = doc.createElement("p");
            infoElem.appendChild(doc.createTextNode("VALID LOGIN!"));
            loginElem.getParentNode().insertBefore(infoElem,loginElem);

            return true;
        }
    };
    loginComp.setValidateLoginListener(validator);
```

The line:

```
    final LoginComponent loginComp =
                (LoginComponent)componentMgr.createItsNatComponentById(
                        "loginCompId","login",null);
```

Creates our custom component as any other predefined component.

In this example the validator only accepts "`admin`" and "`1234`" as a valid login.

7.15 AUTOMATIC COMPONENT BUILD

Almost all components used in this manual were created explicitly using `ItsNatComponentManager.`**`createItsNatComponent*`** factory methods. ItsNat provides an automatic mode, using this approach the markup specifies all data needed to create every component.

The ItsNat method `ItsNatComponentManager.`**`buildItsNatComponents`**`(Node)` creates and registers every component found while traversing the subtree below the specified node. These components can be found using methods like `ItsNatComponentManager.`**`findItsNatComponent`**`(Node)` or `ItsNatComponentManager.`**`findItsNatComponentById`**`(String)`. The method `buildItsNatComponents` uses `ItsNatComponentManager.`**`addItsNatComponent`**`(Node)`[76], this method creates and registers the appropriated component associated to the specified node; if no component can be associated it does nothing.

For instance:

```
ItsNatDocument itsNatDoc = ...;
ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

Element parentElem = itsNatDoc.getDocument().getDocumentElement();
componentMgr.buildItsNatComponents(parentElem);
```

A concrete node can be explicitly excluded from being used to create a component using `ItsNatComponentManager.`**`addExcludedNodeAsItsNatComponent`**`(Node)`.

There are three node types related with automatic component construction using `buildItsNatComponents` or `addItsNatComponent`:

1) HTML form elements

2) HTML elements with a default associated component

3) Free elements

7.15.1 HTML form elements

By default all form elements are automatically associated to components. No special ItsNat attribute is needed. There is only one exception: `<input type="text">` elements going to be associated to `ItsNatHTMLInputTextFormatted` components, the attribute `itsnat:compType` with value "`formattedTextField`" must be declared, this attribute distinguishes from the default non-formatted component. For instance:

```
<input type="text" itsnat:compType="formattedTextField"
    id="inputTextFormattedId" size="20" value="" />
```

```
ItsNatHTMLInputTextFormatted inputTextFormat =
    (ItsNatHTMLInputTextFormatted)componentMgr.addItsNatComponentById(
```

[76] ItsNatComponentManager.addItsNatComponentById(String) is basically the same using Document.getElementById to obtain the node.

```
                "inputTextFormattedId");
```

HTML form elements with default components: `<input type=”button|image|submit|reset|checkbox|radio|text|password|hidden|file”>`, `<button>`, `<textarea>`, `<select [multiple=”multiple”]>` and `<form>`.

If a concrete form element must be excluded, use the `itsnat:isComponent=”false”` attribute:

```
<input type="text" itsnat:isComponent="false" size="20" value="" />
```

7.15.2 HTML elements with a default associated component

The following elements have a default component associated: `<a>`, `<label>`, `<table>`.

These elements automatically create components only if the attribute `itsnat:isComponent=”true”` is specified. For instance:

```
<label itsnat:isComponent="true" id="labelId">Label</label>

ItsNatLabel label =
            (ItsNatLabel)componentMgr.findItsNatComponentById("labelId");
```

7.15.3 Free elements

These elements have no default component associated; we must specify the component type using the attribute `itsnat:compType`. The component type names are the same names used with the method `ItsNatComponentManager.createItsNatComponent(Node node, String compType, NameValue[] artifacts)`. For instance:

```
<span itsnat:compType="freeLabel" id="freeLabelId">Free Label</span>

ItsNatFreeLabel freeLabel =
          (ItsNatFreeLabel)componentMgr.findItsNatComponentById("freeLabelId");
```

7.15.4 Specifying a user defined structure using markup

Lists, tables and trees may have user defined structures, these structures were introduced using `NameValue` objects with `createItsNatComponent`. To specify a user defined structure using markup first we must to register it as an artifact. There are two levels:

1) Document template level

When the document template is just registered:

```
DocumentTemplate docTemplate = ...;
ItsNatListStructure customStruc = new CityListCustomStructure();
docTemplate.registerArtifact("cityCustomStruc",customStruc);
```

2) Document level

This is the preferred way, when the document is just loaded (and before calling `buildItsNatComponents`):

```
ItsNatListStructure customStruc = new CityListCustomStructure();
itsNatDoc.registerArtifact("cityCustomStruc",customStruc);
```

Now this artifact can be specified in markup using `itsnat:useStructure` attribute:

```
<table border="1px" cellspacing="0" cellpadding="5px">
    <tbody itsnat:compType="freeListMultSel"
           itsnat:useStructure="cityCustomStruc" id="listCustomStructureId">
        <tr><td>City:</td><td><b>Name</b></td></tr>
    </tbody>
</table>
```

```
ItsNatFreeListMultSel listCustomStruc =
    (ItsNatFreeListMultSel)componentMgr.findItsNatComponentById(
           "listCustomStructureId");
CityListCustomStructure structure =
    (CityListCustomStructure)listCustomStruc.getItsNatListStructure();
```

7.15.5 Other configuration parameters defined in markup or as artifacts

- `markupDriven`: boolean property recognized by `ItsNatHTMLInputButtonToggle`, `ItsNatHTMLInputTextBased` and `ItsNatHTMLSelect` and `ItsNatHTMLTextArea` based components.
- `joystickMode`: boolean property recognized by `ItsNatFreeList`, `ItsNatTable` and `ItsNatTree` based components.
- `selectionUsesKeyboard`: boolean property recognized by `ItsNatFreeListMultSel`, `ItsNatTable` and `ItsNatTree` based components.

Markup driven mode is explained in chapter "MARKUP DRIVEN MODE IN FORM BASED NODES". Joystick mode and "selection uses keyboard" modes are explained in chapter "COMPONENTS IN MOBILE DEVICES/BROWSERS".

7.15.6 Removing and disposing components automatically

The method `ItsNatComponentManager.`**`buildItsNatComponents`**`(Node)` has a counterpart method, `ItsNatComponentManager.`**`removeItsNatComponents`**`(Node node,boolean dispose)`, to unregister and optionally dispose the components created and registered below the specified node. For instance:

```
Element parentElem = itsNatDoc.getDocument().getDocumentElement();
componentMgr.removeItsNatComponents(parentElem,true);
```

7.15.7 Excluding nodes of automatic component creation

We can exclude a specific node of the automatic component creation with the method `ItsNatComponentManager.`**`addExcludedNodeAsItsNatComponent`**`(Node)`.

7.15.8 Fully automatic component creation (template level configuration)

If we want to build all components automatically when the document is first loaded, ItsNat provides the method `DocumentTemplate.`**`setAutoBuildComponents`**`(boolean)`. If set to true (is false by default) all components declared in the markup will be created and registered into the document when is first loaded.

This automatic component creation level goes beyond: when a new markup fragment is added to the document the method `ItsNatComponentManager.`**`buildItsNatComponents`**`(Node)` is called to build the contained components into the fragment. The same is applied when the markup is removed, `ItsNatComponentManager.`**`removeItsNatComponents`**`(Node node,boolean dispose)` is called with `dispose` parameter as true.

7.16 MARKUP DRIVEN MODE IN FORM BASED NODES

7.16.1 Overview

ItsNat automatically synchronizes the client DOM when the server DOM changes but not the opposite direction "out of the box". ItsNat HTML form based components like `<select>`, `<input>` and `<textarea>` based components (implementing `ItsNatHTMLSelect`, `ItsNatHTMLInput` and `ItsNatHTMLTextArea`) provide automatic synchronization of server DOM when the state of a client control changes (for instance some text was introduced by the user in a text box). Directly changing the server DOM by the developer *is not* the way to change the state of a form control in server, because in form components the server DOM (and client DOM) is a slave of the data and selection models of components, because ItsNat knows how a data model is converted to DOM (using the default or a specific renderer) but not the contrary. This is the default mode.

To achieve the maximum transparency following the philosophy of "The Browser Is The Server", ItsNat provides a *markup driven* mode in form based components. In this model the component state is driven by the server markup.

For instance when the attribute `value` of an `<input type="text">` node is modified in server the component data model is modified too containing the new string value (and of course the `value` property of the client control is updated too as always). As the server form node is backed by an ItsNat component any text introduced in the client updates the `value` attribute in server.

Another example, a `<select>` based component. Any new `<option>` added to the `<select>` element in server is equivalent to add a new list item to the data model with value the string contained into the `<option>` node, in fact if the text of an `<option>` is modified the data model matched item is modified too (this is not valid in combo boxes). In selection any change to the `selected` attribute in server modifies the selection state of the control (and in client too).

Finally similar behaviour is expected on check boxes, radio buttons and text areas.

Another characteristic of markup driven mode is the initial state, in not markup driven the initial state is mandated by the initial state of the default data and selection models (empty strings, no selection etc). In markup driven mode the initial state of the component is imposed by server markup when the component is created and associated to DOM. For instance: the initial value of the attribute `value` is the initial value of the component and the client control in text boxes; if an `<option>` element of a `<select>` contains a `selected` attribute this option/ list item is initially selected in server and client and so on.

In markup driven mode using component APIs is optional (can be used), client form controls can be managed fully with server DOM APIs and custom user data and selection models are discouraged (components may fail).

7.16.2 Components with markup driven mode feature

Markup	Components Implementing
**`<input type="checkbox	radio"]>`**	**`ItsNatHTMLInputButtonToggle`**
**`<input type="text	password	hidden"]>`**	**`ItsNatHTMLInputTextBased`**
`<select [multiple="multiple"]>`	**`ItsNatHTMLSelect`**
`<textarea>`	**`ItsNatHTMLTextArea`**

7.16.3 Setting up components in markup driven mode

In spite of components with markup driven mode feature have `setMarkupMode(boolean)` methods, is recommended to set the markup driven mode as true when the component is created, this way the initial state of the markup is respected.

Furthermore, ItsNat knows how to build automatically form based components traversing the markup using the `ItsNatComponentManager.buildItsNatComponents(Node)` method.

For instance:

```
    ItsNatDocument itsNatDoc = ...;
    Element parentElem = ...;

    ItsNatComponentManager compMgr = itsNatDoc.getItsNatComponentManager();
    compMgr.setMarkupDrivenComponents(true);
    compMgr.buildItsNatComponents(parentElem);
    compMgr.setAutoBuildComponents(true);
```

The call `setMarkupDrivenComponents(true)` defines markup driven mode as the default mode of new components, the call `buildItsNatComponents(parentElem)` traverses the desired subtree creating and registering components automatically associated to the form based nodes found, these components are created in markup driven mode because this is the default mode. Finally the call `setAutoBuildComponents(true)` configures the component manager to automatically build and register components associated to the nodes of any new subtree added to the document using normal DOM APIs, any new form based node inserted will be automatically associated to the appropriated new component in markup driven mode.

These calls are not necessary if the document template is configured in markup driven mode and to automatically build components calling `setMarkupDrivenComponents(true)` and `setAutoBuildComponents(true)` of the concrete `DocumentTemplate`.

7.17 NON-AJAX MODE AND COMPONENTES

ItsNat components are strongly based on AJAX but they can work in a non-AJAX mode, of course all event-related characteristics are disabled. In a non-AJAX mode the `ItsNatDocument` is created per request, the same is applied to components; unlike other non-AJAX component based frameworks no component data is saved on the session, if the `ItsNatDocument` is lost all dependent components are lost too. You can save the `ItsNatDocument` on the session temporary to gain access to the old page in a new page, doing this you can reuse markup and/or component data models; the following code fragment shows this approach:

```
ItsNatServletRequest itsNatRequest = ...;
ItsNatDocument itsNatDoc = itsNatRequest.getItsNatDocument();
Document doc = itsNatDoc.getDocument();

ItsNatHttpSession itsNatSession =
                    (ItsNatHttpSession)itsNatRequest.getItsNatSession();
HttpSession session = itsNatSession.getHttpSession();
ItsNatDocument itsNatDocPrev =
            (ItsNatDocument)session.getAttribute("previous_doc");
session.removeAttribute("previous_doc"); // No longer available

ItsNatHTMLSelectMult prevListComp =
    (ItsNatHTMLSelectMult)itsNatDocPrev.getItsNatComponentManager()
            .findItsNatComponentById("listId");
DefaultListModel model = (DefaultListModel)prevListComp.getListModel();
prevListComp.dispose(); // to disconnect the data model from the old markup

ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();
ItsNatHTMLSelectMult listComp =
          (ItsNatHTMLSelectMult)componentMgr.addItsNatComponentById("listId");
listComp.setListModel(model);  // Reusing the data model
...
```

The "Feature Showcase" contains a complete example using this technique.

7.18 JAVASCRIPT DISABLED MODE AND COMPONENTES

The JavaScript disabled mode is very similar to non-AJAX mode, in this case no JavaScript can be sent to the client. Components can work in this mode, of course with no events, and using similar techniques to the non-AJAX mode.

Again the "Feature Showcase" contains a complete example with JavaScript disabled and using components. Is very similar to the non-AJAX version but in this case no JavaScript is used.

7.19 XML AND COMPONENTS

XML documents (excluding XHTML) are generated by ItsNat in a one shot without events (AJAX is disabled), we know that components can be used in a non-AJAX mode, this works for XML documents too. Of course HTML based components are not available, but we have the free/tag-agnostic versions: `ItsNatFreeInclude`, `ItsNatFreeLabel`, `ItsNatFreeButtonNormal`, `ItsNatFreeCheckBox`, `ItsNatFreeRadioButton`, `ItsNatFreeComboBox`, `ItsNatFreeListMultSel`, `ItsNatFreeTable` and `ItsNatFreeTree`.

Again the "Feature Showcase" contains a small example.

7.20 COMPONENTS IN MOBILE DEVICES/BROWSERS

ItsNat supports many "mobile browsers", physical and browser capabilities of mobile devices vary very much:

- Touchscreen: some devices have a touch screen where a stylus or a finger can press any point of the screen.
- Pointer simulation: some mobile browsers of devices with or without a touchable screen simulate a pointer on screen; this pointer can be moved using the joystick to any point on the screen.
- Pure joystick navigation of live nodes: some mobile browsers can navigate with the joystick trough the live nodes of the page, usually form controls, links and nodes with a mouse listener associated.

ItsNat offers two configuration modes focused on mobile devices and browsers: "selection uses keyboard" and "joystick mode".

7.20.1 Selection uses keyboard mode

By default ItsNat supposes a complete keyboard exists, this keyboard is necessary for selection in components with multiple selection like free lists, tables and trees, because they all use the SHIFT and CTRL keys much the same as a standard `<select>` element uses them.

In mobile browsers the `<select>` element works differently, you can change the selection state of a single option with no need of the CTRL key, in fact normal behaviour is as if the CTRL key was ever pressed. This way the CTRL key is no longer needed.

In ItsNat the mode "selection uses keyboard" set to false simulates this behaviour on free lists, tables and trees, the CTRL key is "ever" pressed. This configuration is per component but can be set configured per document, template or globally as usual.

- Per component:

```
ItsNatFreeListMultSel.isSelectionUsesKeyboard()

ItsNatFreeListMultSel.setSelectionUsesKeyboard(boolean)

ItsNatTable.isSelectionUsesKeyboard()

ItsNatTable.setSelectionUsesKeyboard(boolean)

ItsNatTree.isSelectionUsesKeyboard()

ItsNatTree.setSelectionUsesKeyboard(boolean)
```

- Per document:

```
ItsNatComponentManager.isSelectionOnComponentsUsesKeyboard()

ItsNatComponentManager.setSelectionOnComponentsUsesKeyboard(boolean)
```

- Per template:

```
DocumentTemplate.isSelectionOnComponentsUsesKeyboard()

DocumentTemplate.setSelectionOnComponentsUsesKeyboard(boolean)
```

- Per servlet:

```
ItsNatServletConfig.isSelectionOnComponentsUsesKeyboard()

ItsNatServletConfig.setSelectionOnComponentsUsesKeyboard(boolean)
```

To detect if the current page is being loaded by a mobile browser, the method `ItsNatHttpSession.getUserAgent()` is very useful, the string returned is the same as the header value of "`User-Agent`" of HTTP requests.

7.20.2 Joystick mode

This mode is very useful on mobile browsers and devices where there is no stylus (or not practical) and pointer simulation. In these mobile devices the joystick is the only way to control the browser or is the most practical and the joystick only traverses "live" elements of the page (form controls, links and nodes with listeners) highlighting the desired element as the "current" (in this context pressing "enter" is equivalent to a mouse click).

Joystick mode is useful on free combos, lists, tables and trees. In these component types ItsNat only uses a mouse event listener per instance by default; this listener is associated to the parent element of the component. ItsNat determines what list, table or tree item was clicked examining the `target` property of the event object fired by the browser. This does not work in browsers with no mouse pointer emulation, because list, table and tree items are "dead" and cannot be traversed with the joystick.

In joystick mode the component registers a mouse event listener per list item, table cell, icon, handle and label of tree nodes. Exactly the component uses the "content elements"[77], elements returned by structure methods like:

```
ItsNatListStructure.getContentElement(ItsNatList,int,Element)

ItsNatTableStructure.getCellContentElement(ItsNatTable,int,int,Element)

ItsNatTableStructure.getHeaderColumnContentElement(ItsNatTableHeader,

                      int,Element)

ItsNatTreeStructure.getHandleElement(ItsNatTree,int,Element)

ItsNatTreeStructure.getIconElement(ItsNatTree,int,Element)

ItsNatTreeStructure.getLabelElement(ItsNatTree,int,Element)
```

[77] A "content element" is the effective parent element of the item markup

7.20.2.1 The case of IE Mobile

Joystick mode is unavoidable on Android and IE Mobile. IE Mobile is a browser usually distributed with devices with touch screens, why is joystick mode unavoidable? Because IE Mobile is a very limited browser, only links and some form controls can have mouse listeners and there is not "event" object, hence there is no `target` property.

In IE Mobile joystick mode set to true is necessary to know what list, table or tree item was clicked; furthermore, joystick mode is not enough, you need to return links as "content elements" because typical HTML elements of items in components (`TD`, `LI`, `SPAN`…) are not valid to receive mouse events (mouse events are not fired).

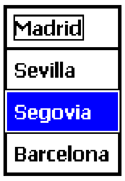
The following is an example of a free combo box component compatible with IE Mobile (it works on any other browser):

```
    <table border="1px" cellspacing="0px" cellpadding="5px">
      <tbody id="compId">
        <tr><td><a href="javascript:;"><b>Cell/Row Pattern</b></a></td></tr>
      </tbody>
    </table>
```

Java code:

```
    ItsNatDocument itsNatDoc = ...;
    ItsNatComponentManager compMgr = itsNatDoc.getItsNatComponentManager();
    Document doc = itsNatDoc.getDocument();

    Element compElem = doc.getElementById("compId");
    IEMobileFreeListStructure structure = new IEMobileFreeListStructure();
    ItsNatFreeComboBox comboComp =
               compMgr.createItsNatFreeComboBox(compElem, structure, null);
    comboComp.setJoystickMode(true);

    DefaultComboBoxModel dataModel =
               (DefaultComboBoxModel)comboComp.getComboBoxModel();
    dataModel.addElement("Madrid");
    dataModel.addElement("Sevilla");
    dataModel.addElement("Segovia");
    dataModel.addElement("Barcelona");

    comboComp.addItemListener(new
               IEMobileFreeComboBoxSelectionDecorator(comboComp));

    dataModel.setSelectedItem("Segovia");
    ...

public class IEMobileFreeListStructure implements ItsNatListStructure
{
    public IEMobileFreeListStructure()
    {
    }

    public Element getContentElement(ItsNatList list,int index,
                       Element parentElem)
    {
        HTMLTableRowElement rowElem = (HTMLTableRowElement)parentElem;
        HTMLTableCellElement cellElem =
          (HTMLTableCellElement)ItsNatTreeWalker.getFirstChildElement(rowElem);
        HTMLAnchorElement link =
          (HTMLAnchorElement)ItsNatTreeWalker.getFirstChildElement(cellElem);
        return link;
    }
}

public class IEMobileFreeComboBoxSelectionDecorator implements ItemListener
{
    protected ItsNatComboBox comp;

    public IEMobileFreeComboBoxSelectionDecorator(ItsNatComboBox comp)
    {
        this.comp = comp;
    }

    public void itemStateChanged(ItemEvent e)
    {
        int state = e.getStateChange();
        int index = comp.indexOf(e.getItem());
        boolean selected = (state == ItemEvent.SELECTED);

        decorateSelection(index,selected);
    }

    public void decorateSelection(int index,boolean selected)
    {
        ItsNatListUI compUI = comp.getItsNatListUI();
        HTMLAnchorElement link =
                       (HTMLAnchorElement)compUI.getContentElementAt(index);
        if (link == null) return;

        HTMLTableCellElement td = (HTMLTableCellElement)link.getParentNode();
        if (selected)
        {
            setAttribute(td,"style","background:rgb(0,0,255); color:white;");
            setAttribute(link,"style","color:white;");
        }
        else
        {
            td.removeAttribute("style");
            setAttribute(link,"style","color:black;");
        }
    }

    public static void setAttribute(Element elem,String name,String value)
    {
        String old = elem.getAttribute(name);
        if (old.equals(value)) return; // Avoids redundant operations

        elem.setAttribute(name,value);
    }
}
```

This is the visual result on IE Mobile:



The square around "Madrid" shows the browser cursor over the link of this item navigating with the joystick (if the enter key is pressed in this moment, "Madrid" will be selected).