ItsNat

Reference Manual

v1.3

Doc. version 1.0

Jun 6, 2013

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 doubt, the AJAX approach has revitalized 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 were definitively back, these browsers hurt seriously the DHTML promise of highly interactive web sites with almost no reload. Fortunately current browsers like Firefox, Safari 3+, Chrome or Opera are highly W3C compliant and Internet Explorer 6 is not doing very bad in the web standard space (of course v7, v8 and v9 are better) and today 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] 38 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

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

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

strange object orientation, slow, no explicit types, no refactoring tools, too many browser differences, too much code sent to the client, defensive programming on the server (no confidence in the client 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…", 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.

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

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}">
    <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.

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 checking). 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).

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

[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/

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

Client and server communication is done through AJAX based events, auxiliary SCRIPT elements can be an alternative to AJAX, your application can select the required communication method with a simple configuration option.

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

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 artifacts, 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. When writing "HTML" we mean any HTML version including HTML 5. Also you can code XHTML using HTML 5 tags and interpret it as HTML X<5 or HTML 5 because text/html MIME is used to send to client.

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 through AJAX/SCRIPT

User events are sent to the server using AJAX (`XMLHttpRequest`) or auxiliary SCRIPT elements 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 `org.w3c.dom.events.EventTarget.`**`addEventListener`**`(`

[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/

`String,EventListener,boolean)`[8], or `ItsNatDocument.`**`addEventListener`**`(…)` methods, in fact one of these methods is called by `EventTarget.`**`addEventListener`**`(…)` behind the scenes. This listener is registered as a *remote listener*, ready to receive events from the browser. For instance: if an event listener 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[9] 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 provides `org.w3c.dom.events.EventTarget.`**`removeEventListener`**`(String,EventListener,boolean)`[10] , or `ItsNatDocument.`**`removeEventListener`**`(…)` methods to unregister remote event listeners.

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) in server 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 control who access this kind of "spyware" by default the remote view/control feature is not active and authorization is granted per "remote view" request.

ItsNat implementation has two modes:

1) Read only remote views, when attached clients cannot send events to the server. In this mode ItsNat offers some kind of remote control: remote viewer refreshing is controlled by an optional server listener; this Java listener can do anything including changing the server DOM tree.

2) Full remote control: attached clients can send events to the server interacting with the same document in server.

Applications… remote supervision, help desk, cooperative applications, "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

---

[8] `ItsNatNode.isInternalMode()` must return false, the default value.

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

[10] `ItsNatNode.isInternalMode()` must return false, the default value.

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

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"). Developers can add custom JavaScript code in server code in any time of the server cycle to be sent to client, for instance, to do some animation effect.

1.3.8 "Smart" DOM memory caching and disconnected nodes 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).

Another technique to save memory in server is "disconnected nodes", you can explicitly remove nodes and server but not in client when these server nodes are no longer to be used in server.

Both techniques allow server-centric Single Page Interface and web sites mainly stateless!

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"[11]. 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>
```

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>
```

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

```
        <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, SVG 1.x or XUL (every standard supported by NekoHTML and Xerces parsers). 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 Single Page Interface web sites fully based on AJAX. 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[12] 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.

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

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

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: SVG and XUL and embedding options

1.3.16.1 SVG support

May be you now that FireFox 1.5+, many WebKit browsers (Safari 3+, Chrome 1.0, iPhone 2.1, BlackBerry JDE 5.0), Opera, Internet Explorer 9 and previous versions with Adobe SVG Viewer or Savarese Ssrc plugins 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, can 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 SVG capable browsers support SVG elements embedded inline in XHTML[13] and ItsNat adds support of SVG inline in MSIE v6,7,8 with Adobe SVG Viewer or SVGWeb. In ItsNat, server based SVG elements in XHTML can receive events, can be attached to components and so on as any other XHTML element.

1.3.16.2 XUL support

XUL[14] is a web based component system included in Gecko browsers like FireFox and Internet Explorer with Savarese Ssrc SVG/XUL plugin (this plugin is basically is FireFox registered as plugin in MSIE). ItsNat supports remote XUL applications in the same way as SVG is supported.

1.3.16.3 Embedding XHTML in SVG and XUL

Embedding XHTML elements in SVG documents[15] is a cutting edge feature of modern browsers; it is very interesting because SVG standard lacks of visual controls like text boxes, selection lists etc.

XUL allows XHTML elements embedded; in this case XHTML is not very useful because XUL can be seen as a richer alternative to XHTML, anyway it works.

ItsNat supports, in a server point of view, embedding XHTML in SVG and XUL documents, furthermore, HTML components can be created and attached to embedded XHTML elements.

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 scripting nor AJAX events but it does not prevent that components can be also used!

1.3.18 Disabled Events and Disabled JavaScript modes

ItsNat is very strongly based on AJAX (or SCRIPT elements) for events, 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

[13] 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. The same for XUL in Gecko.

[14] https://developer.mozilla.org/En/XUL

[15] http://www.w3.org/TR/SVG/extend.html

utilities (renderers, lists, tables, trees) and Swing-like components[16] 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 disabled events 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 Remote Templates

Because templating is based on pure X/HTML, SVG, XUL, XML files, any markup based file may be an ItsNat template. ItsNat supports remote templates, that is, the template may be any remote file pointed by a URL, furthermore, ItsNat provides a user defined way to load the template source, user code can freely load templates from any source and refresh them in a per-request basis, by this way (in extreme) ItsNat can be used as a front end/filter/proxy of any web application.

1.3.20 Extreme mashups

With a simple script located in the end of any web page, this alive page can be attached to an ItsNat server to be fully controlled by an ItsNat based application. Extreme mashups can be used, to enrich any web application for instance to add state and events to old applications.

1.3.21 One Web: AJAX everywhere including in your mobile browser

Besides supported desktop browsers like FireFox, Internet Explorer 6+, Safari, Opera and Google Chrome, ItsNat supports many mobile browsers: Opera Mini, Opera Mobile, WebKit browsers (iPhone/iPad/iPod Touch, S60WebKit, S40WebKit, Android) and BlackBerry. All of them including AJAX[17]!

1.3.22 Single Page Interface Stateless Mode!!!

ItsNat normal mode is stateful making use of server memory (and optionally session) to storing in server the browser state, when using multiple symmetric servers sticky sessions are required (unless session replication is enabled).

Since v1.3 ItsNat provides a new mode of working, the “stateless” mode, in stateless mode current browser state can be reconstructed in server and modificated, modifications are sent to the browser as JavaScript as DOM operations as usual, but the “page” loaded in server is not stored in server nor in session, by this way multiple symmetric servers can work together with no need of session replication (anyway user code can make use of session for storing custom data as any conventional web application).

1.3.23 Single Page Interface SEO compatible!!

Thanks to the “fast-load mode” of ItsNat, the same DOM tree is converted to markup in load time and to JavaScript as the result of an AJAX event, by this way a concrete “state” can be the initial page on load time ready to be crawled by web crawlers like Google Search.

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

[17] Concrete versions are listed later.

Single Page Interface SEO compatibility is possible in stateful and stateless modes of ItsNat!!

1.3.24 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… It's Natural because basically ItsNat simulates the behavior of a web browser in server, in few words, ItsNat is DHTML on the Server[18].

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[19]. 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 Single Page Interface 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 *meta-framework*: 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[20]).

ItsNat is focused to developers who want extensive use of AJAX and in the same time do not want to lose control of their work.

[18] http://java.dzone.com/news/itsnat-java-web-framework-inte

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

[20] 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 for source code (Java and markup code).`

2.3 LICENSE

ItsNat is Lesser General Public License Version 3 licensed to third parties (LGPL v3). LGPL v3 license allows commercial closed source derivatives based on ItsNat. Any change done to ItsNat itself must be released as source code under LGPL v3 to end users.

If LGPL v3 is not for you alternative commercial licensing exists.

2.4 COPYRIGHTS

Jose María Arranz Santamaría is the author of ItsNat source code, documentation and examples. ItsNat intellectual property and exploitation rights are owned by Jose María Arranz Santamaría. Innowhere Software, a professional service of Jose María Arranz (former 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.

2.5 REQUIRED DEVELOPER TECHNICAL SKILLS

ItsNat is based on Java 1.5, W3C DOM Level 2 Core[21]/HTML[22]/Events[23], and Servlet 2.2. A medium knowledge of these API is supposed, especially DOM and Swing. Swing knowledge is optional if ItsNat components are not used.

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

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

2.6 TECHNICAL REQUIREMENTS, LIMITATIONS AND DEPENDENCIES

ItsNat is based on Java Standard Edition (Java SE) 1.5 as the minimum configuration (source and binaries), and may compile and run with any upper version without problem (compiled and tested with Oracle's Java SE 1.6 developer kit).

ItsNat is based on the Servlet specification, no advanced API is used, virtually any Servlet container compatible with Java 1.5 may be valid. ItsNat has been tested with the following servlet containers: Tomcat 6 and 7, Google App Engine (powered by a custom version of Jetty).

2.6.1 Browsers supported

ItsNat only supports browsers with AJAX support.

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

- Google Chrome
- FireFox
- Microsoft Internet Explorer 6, 7 and 8[24]
- Microsoft Internet Explorer v9+
- Opera 12.12+
- Safari 5.1.7+

Previous versions may work.

When FireFox is cited we mean any Gecko based (the web engine of FireFox) browser for instance Mozilla family browsers. FireFox, Safari, Chrome and Opera and most of mobile browsers are considered "W3C browsers", MSIE 6/7/8 are not "W3C browsers" (their support is poor, for instance, DOM events, namespaces, XHTML etc, IE 8 is W3C DOM level 2 compliant but not including W3C DOM events), MSIE v9 is very different to previous versions and ItsNat considers internally this browser as a legitimate W3C browser.

Mobile browsers supported (minimum version, previous versions may work):

- Android 2.1+
- BlackBerry JDE 4.6+ browsers (Flip, Bold, Storm etc), since JDE 5.0 BlackBerry browser is detected as a WebKit browser
- Internet Explorer Mobile 6 (since WM 6.1.4)[25]

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

[24] By default IE 8 runs in default mode (IE8 mode). Can be changed calling HttpServletResponse.setHeader("X-UA-Compatible","IE=*value*"); or using a <meta> tag. More info here: http://msdn.microsoft.com/en-us/library/cc288325(VS.85).aspx#SetMode

- iPhone/iPad/iPod iOS 6.1+
- Opera Mini 4+ (including 5.1)
- Opera Mobile 12.10+
- S40WebKit of S40 based Nokia Phones since 6th edition
- S60WebKit of S60 5th v1.0 SDK (touchscreen)

SVG plug-ins (including AJAX):

- Adobe SVG Viewer v3[26]
- Savarese Ssrc[27]. This plugin also provides XUL for MSIE.
- SVGWeb[28] 2010-02-03 "Lurker Above"
- Batik Applet[29] v1.7. Batik is used as an applet specially tuned for ItsNat included into the ItsNat distribution folder `/web/batik` and `/fw_dist/batik_applet`.

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).

Unknown browsers (user agents unknown) are treated as bots and AJAX is automatically disabled. This saves memory because AJAX is used to destroy the document in the server when the page is closed (documents are lost too when the session ends).

Web browsers can be classified in two categories:

1. Support of standards: MSIE 6+[30] and W3C based (the rest).
2. Device type: desktop and mobile.

2.6.2 External dependencies

- Batik 1.7: `batik-dom.jar`, `batik-util.jar` and `batik-xml.jar` files must be included. Batik is used into the framework as DOM provider.

[25] Internet Explorer Mobile included in Windows Mobile 6.1.4 is based on Internet Explorer 6 and some parts (JavaScript engine) of IE 8, this browser is very different to the version included in WM 6.0/6.1, (severely limited) sometimes named "Pocket IE".

[26] http://www.adobe.com/devnet/svg/adobe-svg-viewer-download-area.html

[27] http://www.savarese.com/software/svgplugin/

[28] http://code.google.com/p/svgweb/

[29] http://xmlgraphics.apache.org/batik/

[30] MSIE browsers support some parts of W3C standards but some parts like the event system are very different.

- Xerces for Java 2.9.1[31]: `xercesImpl.jar`,`serializer.jar` and `xml-apis.jar` files must be included.
- NekoHTML 1.9.12[32]: `nekohtml.jar` must be included.
- ItsNat Batik Applet includes `core-renderer-minimal.jar`, this jar is optional and provides some support (minimal) of XHTML embedded in SVG (through `<foreignObject>`) and is part of the Flying Saucer[33] project version R8.

These external products are open source and have compatible licenses. Previous versions may work but not tested.

In previous versions ItsNat used Xerces DOM. Xerces DOM implementation has very serious threading problems including when using one thread per DOM document, this is not a problem for Batik DOM, which is thread safe if only one thread accesses a DOM document, in spite of this Batik has dependencies with Xerces (for instance Batik may use XPath) and Xerces parsers and serializers are used in ItsNat. Batik DOM Core is extended by ItsNat to provide an almost complete W3C DOM HTML Level 2 implementation.

2.6.3 Session management

There are two modes of session management in servlet containers: sticky sessions and session replication.

In sticky sessions mode any client request is ever transported to the same node and web application instance. Because there is no data transport between nodes/server instances, session data may be not serializable[34].

With session replication any data registered in the session is serialized and shared between nodes, the servlet engine takes care of serialization and the cloud environment takes care of node coordination. Usually serialization happens after any web request is completed if the container detects some change in session attributes. This is how Google App Engine (GAE) works, at this time GAE does not support sticky sessions.

ItsNat preferred mode is sticky sessions because ItsNat is server centric and focused on Single Page Interface[35] applications keeping client state in server, if session data is shared between nodes this client state must be copied to servers and some special features like remote view/control only work fine in web applications running in a single node or using sticky sessions. In spite of this session replication is a first class citizen in ItsNat and with some techniques, such as disconnected nodes, node caching provided by templates and session compression, session data transported can be reduced to minimum.

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

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

[33] https://xhtmlrenderer.dev.java.net/

[34] Serialization may be useful to provide some kind of failover

[35]http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php and http://www.uxmatters.com/mt/archives/2006/11/improving-user-workflows-with-single-page-user-interfaces.php

2.6.3.1 Sticky Sessions Mode

If `ItsNatServletContext.`**`setSessionReplicationCapable`**`(boolean)` is set to false (the default mode) no session attribute is used by ItsNat to store ItsNat session data, hence no serialization happens to user code. In this mode servlet sessions are mainly used by ItsNat to identify the client[36]. ItsNat session ids are only valid and unique in the concrete application instance. This is the preferred mode, and highly recommended in Single Page Interface applications.

In this mode, native servlet sessions are almost only used for client identification purposes, that is, the session cookie value (session id) is the base of the ItsNat security model, this does not necessarily imply using session attributes, furthermore, in a Single Page Interface application session attributes can be fully avoided by the user. In brief, ItsNat provides ItsNat specific session objects, implementing `ItsNatServletSession`, in some way these ItsNat session objects are bound to the native session objects, with 1-1 relationship. This 1-1 relationship does not imply session attributes are being used.

ItsNat generates alternative and simple session ids, these ids may be public to identify the client to other clients (this is the main reason), the kind of things other clients can do with this session id is controlled by ItsNat (native session ids must not be public to other users because they grant full control of the monitored client outside ItsNat control). Never client identification of requests is based on these ItsNat generated ids.

2.6.3.2 Session Replication Capable Mode

If the method `ItsNatServletContext.`**`setSessionReplicationCapable`**`(boolean)` is called with true, ItsNat session wrapper `ItsNatServletSession` is bound to the native session calling `HttpSession.`**`setAttribute(String,Object)`** after any request is completed to notify some change in the server state of the session in the concrete node/instance. This mode is recommended if the servlet engine is doing session replication between nodes or some kind of failover. ItsNat by default does not serialize, only `setAttribute` is called, is responsibility of servlet container serialize the `ItsNatServletSession` object.

If serialization happens user code must be serializable, otherwise an IO exception is thrown by the servlet engine. If there is one only one application instance (one node or servlet container instance) or servlet container is not doing session sharing between nodes (sticky sessions) then there is no essential functional difference between both ItsNat modes (sticky and session replication capable) in spite of session replication is enabled.

In session replication capable mode, ItsNat session public ids are SHA-1 generated values from the native session ids. Because SHA-1 is a one way algorithm the reverse process to obtain the native session id is extremely painful (brute force), and because the native session id is unique and SHA-1 generated value is also unique, the generated public session id is also unique between nodes.

In session replication some ItsNat features do no work because they were designed for single node or sticky sessions, usually they are thread-based features:

- Comet: use timers instead.

[36] ItsNat generates alternative and simple session ids, these ids may be public to identify the client (this is the reason), but security and absolute client identification of requests are not based on these ItsNat ids.

- Server-sent events using browser: this feature is heavily based on local threads.
- Asynchronous tasks: again it uses threads.
- `Iframe/object/embed/applet` child document auto-binding.
- Remote view/control: remote view/control usually involves several sessions.
- SVGWeb: automatic handling of dynamically inserted nodes does not work, use a `SVGLoad` listener instead.
- `ItsNatDocument` or related ItsNat objects cannot be explicitly serialized calling `HttpSession.`**`setAttribute`**`(String,Object)`

In session replication mode this configuration method is interesting:

`ItsNatServletContext.`**`setSessionSerializeCompressed`**`(boolean)`

When is set to true (by default is false) ItsNat compresses (using GZIP) the final serialized data of the ItsNat session when this session is being serialized (ItsNat first serializes the session as a byte buffer then writes this buffer to the stream). This action significantly reduces the amount of bytes being transported between nodes in the cloud and may provide a significative gain in performance.

Because developers most of the time develop in single instances, ItsNat provides a complementary configuration flag:

`ItsNatServletContext.`**`setSessionExplicitSerialize`**`(boolean)`

If this method is called with a true value (by default is false), ItsNat explicitly serializes the `ItsNatServletSession` object and saves the serialized byte array in session when any request is completed. In addition, when a new request arrives to the server, the `ItsNatServletSession` object is ever obtained again de-serializing the byte buffer saved in the session, by this way a new fresh session object (and dependent objects like documents bound to the session) is used in every request simulating the behavior of a cloud environment using replicated sessions like GAE. This method is independent of `ItsNatServletContext.`**`setSessionSerializeCompressed`**`(boolean)` and both can be combined, that is, sessions can be explicitly serialized and serialized data can be compressed before calling `HttpSession.`**`setAttribute`**.

2.6.4 Google Application Engine support

ItsNat works in Google App Engine (GAE) with the limitations explained before due to session replication, the only valid mode in GAE, and some more specific restrictions:

- Most of components do not work because of most of them reuse Swing classes, and Swing packages are not white listed in GAE.

  This problem is not critical because the "core" level API of ItsNat offers DOM utilities similar to components and utilities to transport data from client to server alongside AJAX events (for instance to transport the textual data of a text control).

- Features which create new threads do not work, for instance Comet (use timers instead).

To deploy an ItsNat and GAE based application you must provide an `appengine-web.xml` file like this:

```
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
    <application>yourappname</application>
    <version>1</version>
    <sessions-enabled>true</sessions-enabled>
</appengine-web-app>
```

Another requirement is you must remove the original Batik DOM jar used in ItsNat, `batik-dom.jar`, and replace it with a custom version, `batik-dom-gae.jar`, this file included in ItsNat distribution is located in `/fw_dist/gae/lib`

The file `batik-dom-gae.jar` is basically the same as the original, the only difference is the class `org.apache.batik.dom.AbstractDocument`, this class has been modified removing some problematic XPath dependencies. Google App Engine does not support `org.w3c.dom.xpath.*` classes because `org.w3c.*` content is controlled by GAE and `org.w3c.dom.xpath.*` classes are not white listed. W3C DOM XPath classes can be used in Batik DOM included in ItsNat but internally are not used and they are not loaded (unless you use them explicitly), however GAE accidentally tries to load some XPath class when document DOM objects are being serialized and this action causes an exception.

The ItsNat distribution includes some sample scripts located in `/gae` folder:

- `gae_run_local_itsnat.cmd` and `gae_run_local_itsnat.sh` execute the Feature Showcase locally using GAE SDK in Windows and unixes.

- `gae_upload_itsnat.cmd` and `gae_upload_itsnat.sh` upload the Feature Showcase to GAE. Change the application name `<application>itsnatfeatshow</application>` defined in `appengine-web.xml` to your application name registered in GAE. These scripts automatically replace `batik-dom.jar` with `batik-dom-gae.jar` and when the application is uploaded the original jar is restored.

- `_gae_shared_itsnat.cmd` and `_gae_shared_itsnat.sh` contain the environment variables used, change them for your installation.

In GAE most of your code must be serializable, serialization is problematic because it introduces the problem of versioning. When you upload a new version of your application to GAE, GAE does not free the current persistent sessions (GAE saves sessions in both datastore and memcached) and tries to reload old saved sessions in the new version of your application. If the schema has changed and the saved data is not compatible a `java.io.IOException` would be thrown. Fortunately ItsNat is tolerant to schema changes, ItsNat detects when de-serialization fails and discards the session and no explicit session cleaning is needed.

Anyway if you need to manually remove one or all sessions, use "Delete" action in "Datastore Viewer" of your GAE administration account. However this action is not enough, because session data is also saved in memcached, removing the session cookie of your browser gets rid of the session definitely. Of course the last is not practical to end users, another option is the GAE servlet for session cleanup[37].

[37]http://groups.google.com/group/google-appengine-java/browse_thread/thread/4f0d9af1c633d39a?pli=1

To detect when your application is running on GAE there are two known techniques[38].

Finally GAE do not ensure your requests are dispatched to the same node, furthermore if no request is received during several minutes GAE can automatically dispatch new requests to a new node, because the application must be loaded in the new node (only the first time) you may suffer a significative delay, this has nothing to do with session size. Sometimes if the current node being used is very busy, application loading may fail because GAE imposes a timeout per request of 30 second… the solution is to try again.

The method `ItsNatServletContext.`**`setSessionSerializeCompressed`**`(boolean)` set to true is interesting in GAE to reduce the size of data being transported between nodes. This can also help to avoid the limit of session data size imposed by GAE.

The method `ItsNatServletContext.`**`setSessionExplicitSerialize`**`(boolean)` set to true is interesting to simulate on local and single instance the behavior of GAE regarding to session replication and is also useful when session serialization takes too much time, because serialization is executed by ItsNat before calling `ItsNatSession.`**`setAttribute`**.

[38] http://radomirmladenovic.info/2009/06/15/detecting-code-execution-on-google-app-engine

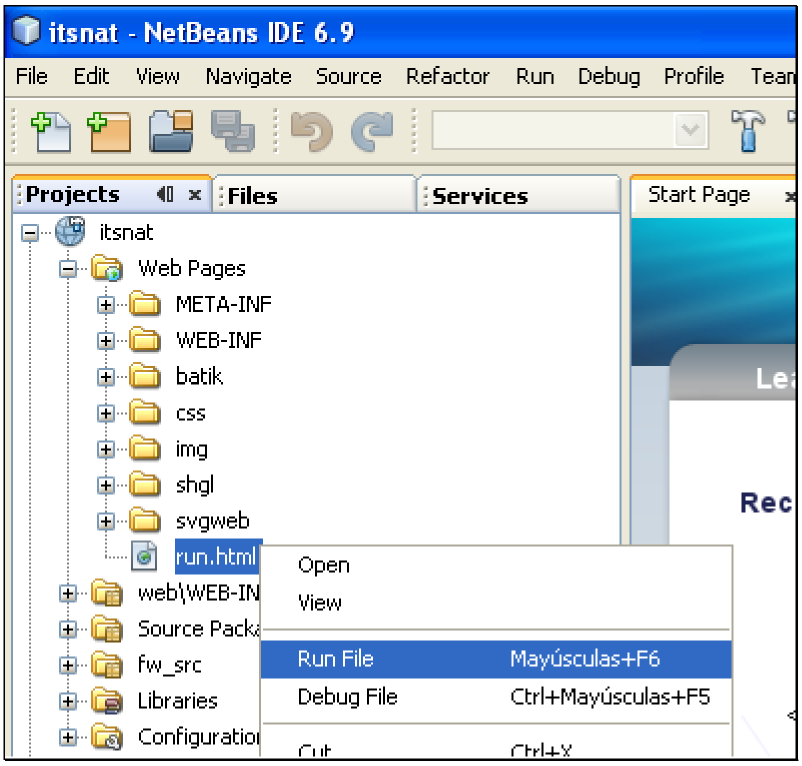
3. INSTALLATION

3.1 ITSNAT DISTRIBUTION

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

The "Feature Showcase" is a NetBeans Java 1.5 web application and contains the framework in source code form for debugging.

To quickest way to execute (run or debug) the "Feature Showcase" example with NetBeans, is through the file `run.html` as shown in the figure (this file is just a quick launcher for NetBeans):



Or start the application server and load the index page with this URL:

http://localhost:8080/itsnat

This web application is very useful to show how a complex ItsNat application can be developed. It is also ready to debug ItsNat source code 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 shown to the user as documentation.

The `fw_dist/lib` contains the `ItsNat.jar` containing the framework binaries and a separated zip file, `ItsNat_src.zip`, containing the source code.

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 and the Feature Showcase can be compiled with JDK 1.5.

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 only contains Java binaries.

Any ItsNat based Java web application need in the standard `WEB-INF/lib` directory the following libraries, these files are located in `fw_dist/lib` distribution directory:

- `ItsNat.jar`: the framework in bytecode form.
- `nekohtml.jar`: NekoHTML parser.
- `xercesImpl.jar`: Xerces framework.
- `serializer.jar`: standard Apache XML serialization got from Xerces distribution.
- `xml-apis.jar`: Java standard W3C XML/DOM API got from Xerces distribution.
- `batik-dom.jar`: Batik W3C DOM Core implementation.
- `batik-util.jar`: Required by Batik DOM.
- `batik-xml.jar`: Required by Batik DOM.

4. ITSNAT ARCHITECTURE

4.1 PURE INTERFACE/IMPLEMENTATION PATTERN

ItsNat public API is based on interfaces, only very few classes are public, the root class is `ItsNatBoot`, this class is abstract (fully implemented by ItsNat internally) and defined to get the singleton object implementing the interface `ItsNat` calling the static method `ItsNatBoot.`**`get`**`()`. The singleton object returned works as a factory: ItsNat based objects implementing public interfaces like `ItsNatServletContext` and `ItsNatServlet` can be created, these objects are used as factory of other objects and so on.

A pure interface/implementation technique allows to program "by contract", where the interface is the contract, only the interfaces are documented. This technique avoids the framework 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.

An interface based architecture avoids class inheritance as a way of extension of the framework, instead of inheritance ItsNat provides many hooks and listeners as the standard way to extend the framework, and finally the developer can optionally re-implement any default implementation of ItsNat implementing the required interface.

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. In this manner ItsNat extends the `Servlet` architecture not fully replacing it, the user can and must use the "old stuff" when ItsNat does not offer something new following the DRY[39] principle.

ItsNat is built upon Java W3C DOM too, for instance, an `ItsNatDocument` wraps an `org.w3c.dom.Document` object (and of course the covered `Document` can be got through `ItsNatDocument`). The "ItsNat" prefix is used to easily distinguish the ItsNat layer and the original data type, most of the ItsNat API starts with "ItsNat" sometimes because the interface wraps a class with the same name and sometimes because of "viral" symmetry/dependencies (for instance `ItsNatDocumentTemplate`).

4.3 COMPONENT SYSTEM "INSPIRED" IN SWING

[39] Don't Repeat Yourself

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 counterparts.

ItsNat components use as possible UI agnostic Swing elements like data models, listener models and related listeners.

4.4 MODULES/PACKAGES

ItsNat is divided in two main modules/packages[40]

- Core (`org.itsnat.core`)

  Is the "core" part of the framework, offers the basic infrastructure to develop event (AJAX or SCRIPT) based Java web applications.

- Components (`org.itsnat.comp`)

  Contains the optional component system, it relies on the "core" part, but "core" has (almost) 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 event listeners etc.

Sub packages:

- `org.itsnat.core.event`: defines event and listener classes and interfaces associated to the page lifecycle and AJAX/SCRIPT events.
- `org.itsnat.core.html`: interfaces related to HTML documents and fragments.
- `org.itsnat.core.http`: interfaces related to HTTP servlets.

[40] 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.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, XHTML, SVG, XUL or XML components. Inside this package they are sub packages usually grouping components per component type:

- buttons `(org.itsnat.comp.button),`
- labels `(org.itsnat.comp.label),`
- lists `(org.itsnat.comp.list),`
- tables `(org.itsnat.comp.table),`
- trees `(org.itsnat.comp.tree)`
- etc.

4.5 *SECURITY AND PRIVACY*

ItsNat is a server centric framework, any server centric framework is by nature securer than client centric approaches (any client centric approach needs to check any access to the server in some way security, validations etc must be replicated in server).

4.5.1 Security is inherent in The Browser Is The Server approach

In the case of ItsNat the browser is managed as a sophisticated UI terminal of the server, all the business logic and view logic is executed in the server[41]; view logic is executed in the server mainly in the form of DOM tree mutations and automatically replicated in the client.

Because the client state is accurately represented in the server (in fact is the opposite) following the approach "The Browser Is The Server", the opportunities of spoofing are very few. For instance event listeners are registered in the server, any event received by the server with no server listener listening is ignored; listeners may be seen as a gate to enter into the server, if no listener is registered in server associated to a concrete DOM node and event type, no event will be processed.

Because the view is based on a W3C DOM tree, some built-in security features are present, for instance, if a malicious user fills an input box with some JavaScript code containing an script like "`<script>some code</script>`" and this code is sent to the server, this user text is usually managed as raw text and text data is usually shown in the DOM as the content of text

[41] This is not exact, some view logic is done only in the client, usually predefined behavior with no relationship with business logic or not customizable by the user.

nodes, by default the DOM renderer "escapes" this text as "`&lt;script&gt;some code&lt;/script&gt;`", this code cannot be constructed as DOM including inside an `innerHTML` assignation[42].

4.5.2 Security and client identification

Client identity is based on the session identification of the servlet container (based on a very long cookie), in spite of ItsNat generates and uses a very simple id for sessions (or SHA-1 generated ids), ItsNat does not rely on this id to identify the client browser in the first instance. Of course this ItsNat generated id is used to identify a concrete user to other users but this id is used once the user request has been correctly identified by the servlet container using the servlet container session id. Access to other clients by using ItsNat public session ids is managed by ItsNat, controlling what the other user can do.

In ItsNat ids are not reused in the lifecycle of the web application, this applies to id sessions too. In the case of sessions ItsNat goes beyond trying to avoid reusing of ids including when the application or the servlet container is relaunched and session replication is disabled. In this case servlet containers do not ensure the session id is not reused so an additional internal number randomly generated by ItsNat is used, and is saved in server and client, if this number in the client casually matches the number of the random number in server of the newly created session, this is not a real threat because this new session was created by the user sending an event from a lost document of a lost session.

In "session replication capable" mode public session ids are SHA-1 generated from native session ids. SHA-1 is a one way algorithm and generated ids are unique because the original session id is unique between clusters.

Any other identification number (for instance document ids) is not absolute, for instance the document "cd_1" has no sense without the ItsNat session id and as said before, this session id is not the number used to identify first the user session in server.

4.5.3 Privacy

In Single Page Interface (SPI) applications there is no page navigation, only the URL used to enter into the application is saved in the browser page history, and AJAX/SCRIPT events do not generate history entries.

ItsNat use ever HTTP POST to send in AJAX events, this mode is more "secure" and "private" than GET method.

In ItsNat and SPI applications cookies are not needed, that is, cookie support is optional and can be disabled in the browser settings. In this case when the initial page of the SPI web application is loaded, ItsNat automatically adds the session id to the path used for AJAX events.

[42] ItsNat uses innerHTML when possible to increase the performance of automatically generated JS code

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)ItsNatBoot.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 an ItsNat based 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 automatically shared.

5.1.2 Configuring global behavior (global options/configuration)

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

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

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

        ItsNatServletContext itsNatCtx = itsNatConfig.getItsNatServletContext();
        itsNatCtx.setMaxOpenDocumentsBySession(10);
        itsNatCtx.setSessionSerializeCompressed(false);
        itsNatCtx.setSessionExplicitSerialize(false);

        String serverInfo = getServletContext().getServerInfo();
        boolean gaeEnabled = serverInfo.startsWith("Google App Engine");
        itsNatCtx.setSessionReplicationCapable(gaeEnabled);

        itsNatConfig.setDebugMode(true);
        itsNatConfig.setClientErrorMode(
                       ClientErrorMode.SHOW_SERVER_AND_CLIENT_ERRORS);
        itsNatConfig.setLoadScriptInline(true);
        itsNatConfig.setFastLoadMode(true);
        itsNatConfig.setCommMode(CommMode.XHR_ASYNC_HOLD);
        itsNatConfig.setEventTimeout(-1);
        itsNatConfig.setOnLoadCacheStaticNodes("text/html",true);
        itsNatConfig.setOnLoadCacheStaticNodes("text/xml",false);
        itsNatConfig.setNodeCacheEnabled(true);
        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.setEventsEnabled(true);
        itsNatConfig.setScriptingEnabled(true);
        itsNatConfig.setUsePatternMarkupToRender(false);
        itsNatConfig.setAutoCleanEventListeners(true);
        itsNatConfig.setUseXHRSyncOnUnloadEvent(true);
        itsNatConfig.setMaxOpenClientsByDocument(5);
```

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 (template) and many of them per document too.

5.1.2.1 Max Open Documents By Session

```
        ItsNatServletContext itsNatCtx = itsNatConfig.getItsNatServletContext();
        itsNatCtx.setMaxOpenDocumentsBySession(10);
```

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) with session support identified as legitimated browsers, browsers with JavaScript disabled traversing pages/documents designed for AJAX and abusive users. By default unknown browsers are treated as bots and events are automatically disabled, the document in the server is automatically destroyed after the page is loaded.

Because child ItsNat documents included by `iframe`, `object` and `embed` elements also count, this number must be greater than the max number of child documents +1 (the container page).

A negative value is the default and means no limit.

Note this configuration value is set in the servlet/application context level and not in the servlet configuration level because only one ItsNat session instance is shared by all ItsNat servlets of the same web application.

5.1.2.2 Session Replication Capable Mode

```
        itsNatCtx.setSessionReplicationCapable(gaeEnabled);
```

Configures ItsNat to work in clusters using session replication like Google App Engine (if set to true) or to work in single nodes or clusters using sticky sessions. Anyway in a single node/instance there is no significative difference between both modes.

By default `ItsNatServletContext.`**`isSessionReplicationCapable`**`(boolean)` returns false (sticky).

5.1.2.3 Session Data is Compressed When is Serialized

```
itsNatCtx.setSessionSerializeCompressed(false);
```

This method when called with a true parameter, configures ItsNat to compress the serialized data when the user's session is serialized to be shared between nodes of the cloud (or saved in some kind of persistent store for failover) to reduce the amount of transported data. This configuration option set to true only has sense if session replication is enabled and when the servlet container serializes the session.

By default `ItsNatServletContext.`**`isSessionSerializeCompressed`**`()` returns false (not compressed).

5.1.2.4 Session Data is Compressed When is Serialized

```
itsNatCtx.setSessionExplicitSerialize(false);
```

This method when called with a true parameter, configures ItsNat to explicitly serialize the user's session saving the serialized data to the native session calling `HttpSession.`**`setAttribute`**`(String,Object)` when any web request ends. When any request arrives to the server, the user's session is explicitly de-serialized from the servlet creating new fresh objects, useful to simulate how cloud environments with session replication works and to avoid serialization time limits imposed by the cloud (for instance Google App Engine). This configuration option set to true only does something if session replication is enabled.

By default `ItsNatServletContext.`**`isSessionExplicitSerialize`**`()` returns false (no explicit serialization).

5.1.2.5 Debug mode

```
itsNatConfig.setDebugMode(true);
```

Sets the debug mode as true (default value), in debug mode the framework makes more checks to ensure is correctly used. A value of true is highly recommended in development and production because in current version the performance impact is negligible.

5.1.2.6 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`.

5.1.2.7 Initial script inline/loaded

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

When the document/page is first served to the browser initial JavaScript generated code can be sent "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.

5.1.2.8 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.

5.1.2.9 Default communication mode for events

```
itsNatConfig.setCommMode(CommMode.XHR_ASYNC_HOLD);
```

Sets the default communication mode for events as "AJAX asynchronous-hold" (this is the default value), in this mode AJAX events are automatically queued as a FIFO list 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. ItsNat also supports events transported by request using SCRIPT elements including a "hold" mode.

5.1.2.10 Default timeout of asynchronous events

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

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

5.1.2.11 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. By default is true in `text/html`, `application/xhtml+xml` and `image/svg+xml` MIMEs.

5.1.2.12 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.

5.1.2.13 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.

5.1.2.14 Default encoding

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

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

5.1.2.15 Use GZIP encoding if available

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

If the browser accepts gzip encoding then JavaScript code (sent as response of an AJAX/SCRIPT 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.

5.1.2.16 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.

5.1.2.17 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.

5.1.2.18 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).

5.1.2.19 Use events

```
itsNatConfig.setEventsEnabled(true);
```

Defines whether events (transported with AJAX or SCRIPT) are enabled. By default is true.

5.1.2.20 Use JavaScript

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

Defines whether JavaScript is enabled (ItsNat sends JavaScript code to clients on load time and as answer of AJAX/SCRIPT events). By default is true.

5.1.2.21 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.

5.1.2.22 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`**`()`. The default value is true.

When a DOM node 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 node is removed from the tree the client counterpart is "lost" by the framework.

Manual removing of event listeners can be done calling `EventTarget.`**`removeEventListener`**`(…)` (in non-internal event mode), `ItsNatDocument.`**`removeEventListener`**`(…)` and `ItsNatDocument.`**`removeUserEventListener`**`(…)` methods. These methods remove listeners associated to DOM nodes currently in the DOM tree, the listeners are automatically removed in the client too.

5.1.2.23 Communication mode of internal unload events

```
itsNatConfig.setUseXHRSyncOnUnloadEvent (true);
```

When end user leaves a (stateful) page, ItsNat automatically sends an unload event to the server to notify the page is no longer loaded in client, ItsNat automatically remove the corresponding document in server freeing resources. If the default communication mode is AJAX (XMLHttpRequest or XHR) this event is sent synchronously (true by default) to ensure the unload event is sent to the server stopping the unload processing of the browser, in some browsers the unload event is not send if the AJAX request is sent asynchronously.

In some concrete circumstances (usually using COMET) and some browsers with a very limited number of open sockets per page (MSIE 6-7) this synchronous communication may hang the browser when end user leaves the page. By setting to false an asynchronous AJAX request is used. Do not worry about "orphan" documents in server, ItsNat has an aging criteria to get rid of them.

This configuration mode in no way affects to user defined event listeners registered for unload events.

5.1.2.24 Max Open Clients By Document

```
        ItsNatServletContext itsNatCtx = itsNatConfig.getItsNatServletContext();
        itsNatCtx.setMaxOpenClientsByDocument(5);
```

Defines the max number of clients attached to a document in server. This feature is very useful to limit the number of clients attached to a document in remote view/control.

A negative value is the default and means no limit.

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, in 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 to save memory, cached DOM fragments are shared between pages in form of serialized text.

This file will work as a template, and usually is not directly accessible with a public URL, so it 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();
ItsNatDocumentTemplate docTemplate;
docTemplate = itsNatServlet.registerItsNatDocumentTemplate("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 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 and the relative file path as value.

Why a XHTML file is registered with `text/html` MIME? Most of supported browsers accept `application/xhtml+xml` header but Microsoft Internet Explorer (MSIE) 6-8 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

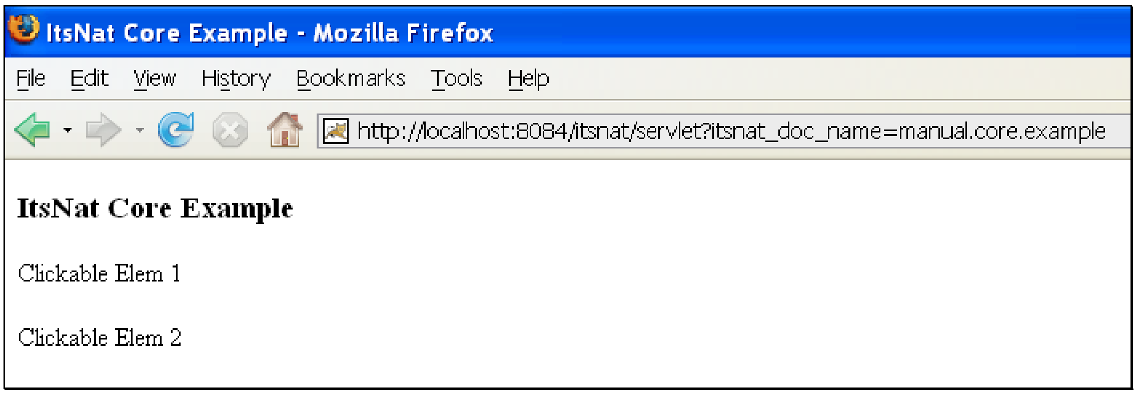
Inside any static html, JSP etc add the following relative link:

```
<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.

Page templates can be modified while running and modifications are shown when the page is reloaded, this is very useful to change the layout when nothing functional changes without redeploying the application. For instance try to change the page title and reload again.

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 `ItsNatDocumentTemplate` 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 CoreExampleDocument(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 and several browser windows can load the same page.

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 page loading.

An `ItsNatHTMLDocument/HTMLDocument` object pair lives during the lifecycle of the page and by default ItsNat keeps the page state in server, any subsequent AJAX/SCRIPT based event is targeted to the concrete document object in the server.

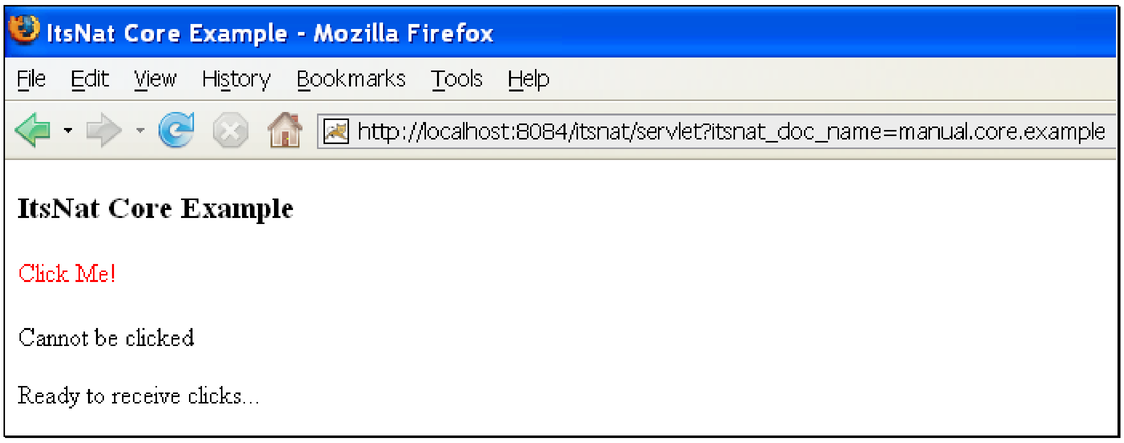
`ItsNatHttpServletRequest` and `ItsNatHttpServletResponse` are unique per request and must not be saved beyond the request (load or event request) like the standard servlet request/response counterparts wrapped by these ItsNat objects.

To isolate the document load processing and save any desired per loaded page state, a `CoreExampleDocument` 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 CoreExampleDocument
{
    protected ItsNatHTMLDocument itsNatDoc;
    protected Element clickElem1;
    protected Element clickElem2;

    public CoreExampleDocument(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 CoreExampleDocument implements EventListener
{
    ...
    public void load()
    {
        ...
        ((EventTarget)clickElem1).addEventListener("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!");
        ((EventTarget)elem).addEventListener("click",this,false);
    }

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

Redeploying and reloading again our web page now receives clicks and sends client events to the server using AJAX (because AJAX asynchronous-hold mode is 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 automatically 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.registerItsNatDocumentTemplate("manual.comp.example",
                        "text/html", pathPrefix + "comp_example.xhtml");
    docTemplate.addItsNatServletRequestListener(new CompExampleLoadListener());
```

Now the template is registered and a `CompExampleLoadListener` object and 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 CompExampleDocument(itsNatDoc);
    }
}
```

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

```
package org.itsnat.manual;

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

public class CompExampleDocument
{
    protected ItsNatHTMLDocument itsNatDoc;
    protected ItsNatHTMLInputText inputComp;
    public CompExampleDocument(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:

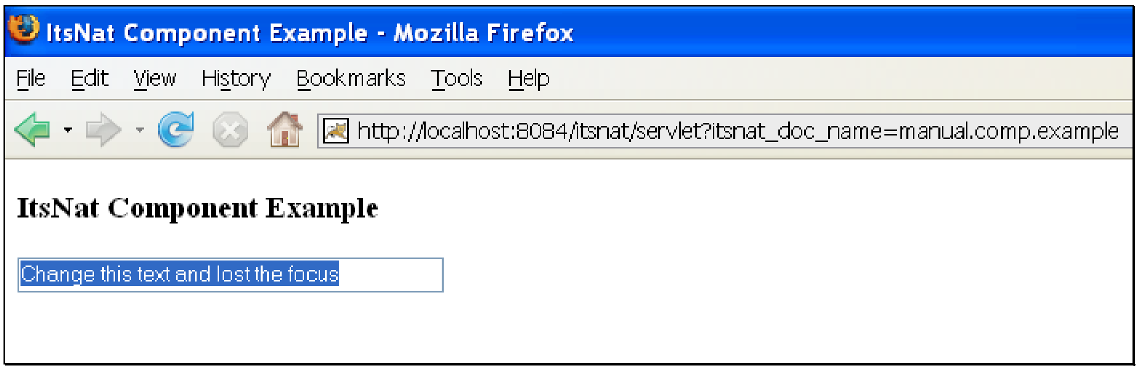
```
inputComp.focus();
inputComp.select();
```

Sends JavaScript code to the browser to call `focus()` and `select()` in the `<input>` DOM element. These method calls do the same as `HTMLInputElement.`**`focus`**`()` and **`select`**`()` methods in non-internal mode (`ItsNatNode.`**`isInternalMode`**`()` returns false).

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 the focus is lost) the component data model is automatically updated.

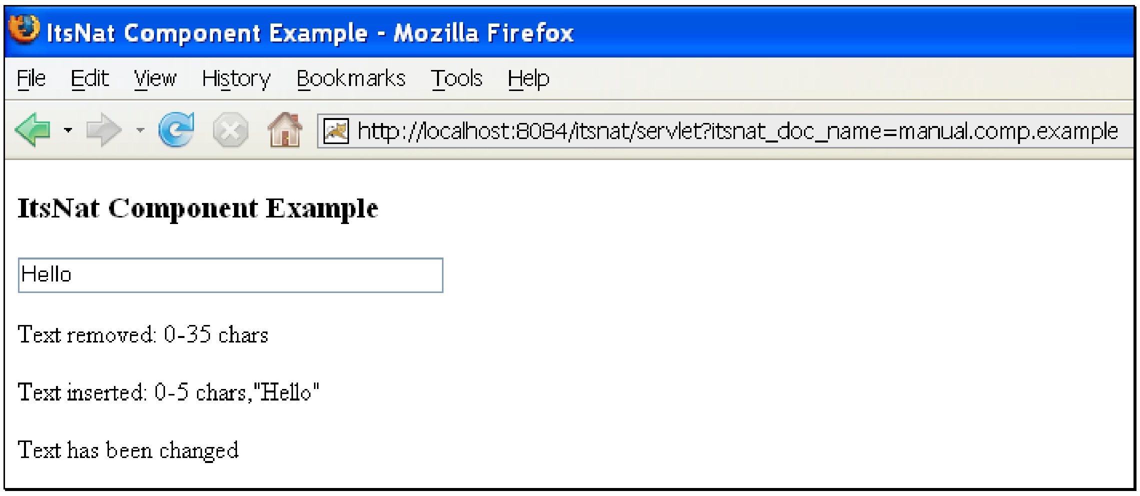
To detect user changes we can add listeners: a “change” DOM event listener and a `javax.swing.event.DocumentListener`. In this example any text change is logged, but of course many serious tasks can be done as updating a database (as a response of a `javax.swing.event.DocumentEvent`).

The complete `CompExampleDocument` 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.text.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 CompExampleDocument implements EventListener,DocumentListener
{
    protected ItsNatHTMLDocument itsNatDoc;
    protected ItsNatHTMLInputText inputComp;

    public CompExampleDocument(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.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.

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 internal paths used to locate nodes in server and in client are calculated ignoring text nodes. The only case text nodes are used to calculate 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

Because ItsNat simulates the behavior of a W3C browser in the server, this applies to DOM events, in ItsNat you can register W3C DOM event listeners associated to DOM nodes in the server, these listeners are *remote* listeners, that is, automatically JavaScript code is generated behind the scenes to add a proxy listener in the client associated to the mirror DOM node in the client, this proxy listener forwards any client event received by this node and matching the event type, to the server using AJAX or auxiliary SCRIPT elements as transport technique. The Java DOM event listener in the server is dispatched receiving a W3C DOM Event as having been "fired" in the server[43]. *Only browser events with some event listener associated in server are sent to the server*.

A DOM event listener is an `org.w3c.dom.events.EventListener` object associated to a Java DOM element and a specific event type (for instance `click`) in the server side, ready to be executed when the browser fires this event associated with the symmetric DOM element in client side.

Java server DOM event listeners are registered calling `org.w3c.dom.events.EventTarget.`**`addEventListener`**`(String,EventListener,bool`

[43] This is why the word "remote" is used, events really occur in client side.

ean)[44] or ItsNatDocument.**addEventListener(…)** methods, ItsNatDocument versions include more optional parameters.

Of course symmetric methods can be used to unregister these event listeners: org.w3c.dom.events.EventTarget.**remoteEventListener**(String,EventListener,boolean) or ItsNatDocument.**removeEventListener(…)**. Only DOM nodes into the document tree can receive remote DOM events.

ItsNat does a strong effort to provide a complete W3C DOM Events support (server point of view) in browsers with a poor support of this standard, for instance, ItsNat provides capturing (useCapture parameter as defined in W3C DOM Level 2 Events[45] standard) in browsers like Internet Explorer.

In Internet Explorer (a non-W3C browser) W3C DOM Events standard is simulated from a server point of view:

1. Listeners are executed (dispatched) in the same order they were registered in the node. This is the correct mode of W3C and is not the way as IE works.

2. Event capturing is simulated: this simulation fully works with ItsNat based event listeners in server because user defined inline handlers in client (e.g. onclick="...") are executed *before* capturing listeners defined in ancestor nodes (and this is not correct).

As Internet Explorer automatically bubbles, this browser is almost W3C DOM Events compliant from a server point of view.

In Internet Explorer Mobile 6 (WM 6) and capturing and bubbling are simulated too.

The remote event system can be tuned with optional parameters: communication 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 Communication modes for events

When the browser fires an event and some registered server side listener was registered for this event, ItsNat uses an XMLHttpRequest object or an auxiliary SCRIPT element to send the event data and receive the JavaScript code. As everybody knows, AJAX request can be executed synchronously and asynchronously, SCRIPT transport is essentially asynchronous.

6.2.1.1 AJAX synchronous mode

The AJAX synchronous mode 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 a problem in a desktop application but is not usual in a web application.

AJAX synchronous mode can be specified using the constant:

```
   org.itsnat.core.CommMode.XHR_SYNC
```

[44] ItsNatNode.isInternalMode() must return false, the default mode.

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

as the value of the `commMode` parameter registering an event listener.

6.2.1.2 Asynchronous modes

AJAX asynchronous or SCRIPT communication modes are more user friendly because there is no browser blocking but they are insecure from the "application point of view". ItsNat has two asynchronous modes, pure and hold.

The asynchronous pure modes are *almost* synchronous in 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 web request threads requesting access to the same `ItsNatDocument` object (usually events transported by AJAX or SCRIPT), only one thread per `ItsNatDocument` is allowed to be processed, by this way an ItsNat application can be considered thread-safe. The pure AJAX or SCRIPT asynchronous mode do not prevent that the second event fired by the browser arrives first to the server and gains control of the `ItsNatDocument` object before the first event.

Pure asynchronous modes can be specified using the constants:

```
org.itsnat.core.CommMode.XHR_ASYNC
org.itsnat.core.CommMode.SCRIPT
```

as the `commMode` parameter registering an event listener.

6.2.1.3 Asynchronous-Hold modes

ItsNat offers a third type mode of communication valid for AJAX and SCRIPT: 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 cannot be cancelled because the browser considers these events as processed, neither server side `org.w3c.dom.events.Event` methods like `stopPropagation()` and `preventDefault()` should be called nor reading/writing native JavaScript properties[46] of event objects.

Asynchronous-Hold modes can be specified using the constants:

```
org.itsnat.core.CommMode.XHR_ASYNC_HOLD
org.itsnat.core.CommMode.SCRIPT_HOLD
```

as the `commMode` parameter registering an event listener.

`CommMode.SCRIPT_HOLD` is highly recommended instead of `CommMode.SCRIPT` mode because requests performed by `SCRIPT` elements are really asynchronous.

6.2.2 Extra parameters

[46] 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").

By default any received `org.w3c.dom.events.Event` object carries the standard W3C-DOM event properties (current target and target nodes, 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)
        {
            ItsNatEvent itsNatEvt = (ItsNatEvent)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 an event is fired (parameter `preSendCode`).

Example:

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

This is a fragment of the JavaScript code generated[47]:

```
    var func = function (event)
    {
       alert('Fired a ' + event.getNativeEvent().type + ' event');
```

[47] This code is generated by the framework and subject to changes (except the user defined code).

```
    };
```

This JavaScript function is called when a click event is dispatched to the specified target element. The `event` parameter is a wrapper of the native event object, this native event can be obtained calling `event.`**`getNativeEvent`**`()`.

Another public JavaScript method of `event` object wrapper is **`setMustBeSent`**`(boolean)`. If the parameter is `false` this event is not sent to the server (by default is `true`, event is sent). This method is interesting because remote event listeners can be used to add/remove cross-platform event listeners executed only in client. Must be reminded ItsNat adds support of capture events to Internet Explorer 6+ (desktop and mobile), bubbling is already supported.

6.2.4 Event timeout

The timeout of asynchronous AJAX/SCRIPT 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 commMode,
    ParamTransport[] extraParams,String preSendCode,long eventTimeout);
```

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 `Document` object because it implements `DocumentView`.

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

```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
AbstractView view = ((DocumentView)doc).getDefaultView();
EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        System.out.println(evt.getType()); // outs load/unload
    }
};
((EventTarget)view).addEventListener("load",listener,false);
((EventTarget)view).addEventListener("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[48].

- Gecko browsers (Mozilla, FireFox): key event API is based on an early W3C DOM 2 draft[49], this API was removed on the final Recommendation.

- Safari 3 (Webkit browsers in general): is compliant with an old W3C DOM 3 draft[50], this API is not the current.

- Opera: 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, Chrome or Opera), ItsNat defines an interface, `org.itsnat.core.event.ItsNatKeyEvent`, to provide a standard key event API, this interface is basically the key event interface used by Gecko browsers[51]. 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 calling `DocumentEvent.`**`createEvent`**`(String)` (`ItsNatNode.isInternalMode()` must return false, the default value) or `ItsNatDocument.`**`createEvent`**`(String)` with "`KeyEvents`" or "`KeyEvent`" type as parameter, then use `ItsNatKeyEvent.`**`initKeyEvent`** method to initialize this object.

6.2.7 Global event listeners

DOM event listeners are registered per target, event type and capture mode, when a listener is registered in the server, custom JavaScript code is send to the client to register a client stub to forward client events targeting the same triad to the server.

ItsNat provides DOM global event listeners, a global event listener is an org.w3c.dom.`EventListener` object registered with the following methods:

1. `ItsNatServer.`**`addEventListener`**`(EventListener)`: per servlet registry.

---

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

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

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

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

2. `ItsNatDocumentTemplate.`**`addEventListener`**`(EventListener)`: per template registry.

3. `ItsNatDocument.`**`addEventListener`**`(EventListener)`: per document registry.

A global listener is called when *any* DOM client event is received by a document in the server which matches the condition of the registry (global listener was registered in this document, or target document is associated to the specified document template or servlet used in the registry).

A global listener is *passive*, that is when registered no JavaScript code is generated to send to the client and is designed for monitoring events received by "normal" DOM listeners because global listeners are dispatched *before* normal listeners.

For instance, global listeners may be used to monitor (and may be modify) the state of the markup and data model of an ItsNat component before an event is dispatched to the component, because some events automatically modify the markup and state of components as the normal behaviour. Methods like `Event.`**`getType`**`()` and `Event.`**`getCurrentTarget`**`()` may be used to identify the target and type of action.

6.2.8 Chain control of event listeners

ItsNat sends an event from client to server per listener registered, that is to say one server listener is dispatched per request.

This is not applied to global listeners registered in servlet, template or document; these listeners are executed before normal listeners in the same request/event processing.

Event listener chain control can be used by global listeners to control how following listeners are executed. The object used to control this flow implements `ItsNatEventListenerChain`, and is obtained calling `ItsNatEvent.getItsNatEventListenerChain()`. For instance, if `ItsNatEventListenerChain.`**`continueChain`**`()` is called, listener dispatching flow will follow this path, that is to say, next listeners will be executed before returning this method. This feature can be used to catch exceptions, to open and commit transactions etc.

The method `ItsNatEventListenerChain.`**`stop`**`()` is used to stop the normal dispatching flow, listeners following (including global) are not executed.

The following example shows how to register a global listener to automatically reload the page when some error occurs processing AJAX/SCRIPT events (usually a user defined or internal component event listener threw an exception).

```
final ItsNatDocument itsNatDoc = ...;
EventListener global = new EventListener()
{
    public void handleEvent(Event evt)
    {
        ItsNatEventListenerChain chain =
               ((ItsNatEvent)evt).getItsNatEventListenerChain();
        try
        {
            chain.continueChain();
        }
        catch(Exception ex)
        {
            itsNatDoc.addCodeToSend("alert('Unexpected Error! The page will
               reload.');");
```

```
                itsNatDoc.addCodeToSend("window.location.reload(true);");

                itsNatDoc.setInvalid();
                chain.stop();
            }
        }
};
itsNatDoc.addEventListener(global);
```

This example avoids the default error processing (exception stack shown to the user)[52].

One more complicated example is found in the "Feature Showcase" demo.

DOM event listeners in components registered calling `ItsNatComponent.`**`addEventListener`**`(…)` can be used to control the listener dispatching flow, because these listeners are dispatched sharing the same AJAX/SCRIPT request (event) and in registration order. Usually ItsNat components have a default behavior associated to a event type, a user event listener can be registered to be executed before/after the execution of the default behavior.

Event listener chain control is available on remote view/control events (`ItsNatAttachedClientEvent`) too.

6.2.9 Detection of session or page lost with global event listeners

In an ItsNat based application using events the browser page is in sync with "a page" (an `ItsNatDocument`) in server. In the following scenarios the server page associated to the page in browser is lost:

1. Servlet container or web application context was reloaded.
2. Session has expired.
3. Server page/document was invalidated explicitly in server (for instance when they are too many open pages).
4. The user has returned to the page using back/forward in a browser caching pages. ItsNat tries to disable page caching automatically reloading the page, in some browsers is not possible (usually some mobile browsers).

When the user tries to use the web page as usual (for instance clicking something) an AJAX/SCRIPT event is sent to the server, ItsNat detects there is no target document in server (there is no matching `ItsNatDocument` and `ClientDocument`).

ItsNat gives an opportunity to user code to manage this scenario dispatching this DOM event to the global event listeners registered on the `ItsNatServlet` object calling `ItsNatServlet.`**`addEventListener`**`(EventListener).`

This DOM event is special, most of the methods are useless and throw an exception if called and the most important characteristic, `ItsNatEvent.`**`getItsNatDocument`**`()` returns null (this is the way to detect a lost user page sending events). Only "`unload`" and "`beforeunload`" events are excluded and not dispatched to global listeners in servlet, because these events are sent when the user is leaving or closing the web page, in these cases ItsNat does nothing.

[52] `window.location = window.location;` is the most compatible instruction but it fails in BlackBerry and when some reference (#ref) is included in URL, an alternative version is provided.

If there is no global listener registered in `ItsNatServlet`, ItsNat by default sends the following JavaScript instruction, `window.location.reload(true)` as the event response.

The following example shows how to detect and manage a DOM event with no server page associated. This global listener must be registered the first because the call `itsNatEvt.getItsNatEventListenerChain().`**`stop`**`()` tries to avoid next global listeners are executed.

```
ItsNatHttpServlet itsNatServlet = ...;
EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        ItsNatEvent itsNatEvt = (ItsNatEvent)evt;
        if (itsNatEvt.getItsNatDocument() == null)
        {
           ItsNatServletResponse response =
               itsNatEvt.getItsNatServletResponse();
           response.addCodeToSend(
                         "alert('Session or page was lost. Reloading...');");
           response.addCodeToSend("window.location.reload(true);");
           itsNatEvt.getItsNatEventListenerChain().stop();
        }
    }
};
itsNatServlet.addEventListener(listener);
```

6.3 REMOTE VS INTERNAL DOM NODE

ItsNat extends Batik DOM to provide a transparent "The Browser Is The Server" experience to the developer using W3C DOM methods as much as possible. By default the DOM tree works in "remote" mode, that is, any action usually modifies the client state in some way:

A. Any structural change in the document tree is reflected in the client DOM.

B. The following methods work in "remote" mode:

   1. Any event listener registered calling `org.dom.events.EventTarget.`**`addEventListener`**`(…)` and unregistered calling `org.dom.events.EventTarget.`**`removeEventListener`**`(…)` is registered/unregistered in the client too:

   2. `DocumentEvent.`**`createEvent`**`(String)`[53] creates a "remote" event, the method `EventTarget.`**`dispatchEvent`**`(Event)` detects if the event submitted is remote, if so the call is redirected to `ItsNatDocument.`**`createEvent`**`(String)`.

   3. Methods like `HTMLInputElement.`**`focus`**`()` generate and send JavaScript code to do the same action in the client.

This is the default and recommended behavior, however sometimes we need non-remote behavior:

[53] The implementation of `Document` implements `DocumentEvent`.

A. To avoid automatic synchronization in the client of DOM structural changes, ItsNat provides two methods: `ItsNatDocument.`**`disableSendCode`**`()` and `ClientDocument.`**`disableSendCode`**`()`, these methods avoid sending JavaScript code to the client. Use them with extreme care because the client can result unsynchronized (in fact this is the use case).

B. Any DOM node implements `ItsNatNode`, this interface is used to extend the standard DOM API. If the node is set as "internal" calling `ItsNatNode.`**`setInternalMode`**`(boolean)` with a parameter true (`ItsNatNode.isInternalMode()` will return true, the default is false). This flag affects the behavior of DOM methods as following:

   a. `EventTarget.`**`addEventListener`**`(…)` and `EventTarget.`**`removeEventListener`**`(…)` register/unregister event listeners using the internal registry of Batik. Monitoring DOM tree changes using mutation listeners is almost the only one use case, the following example is the recommended way to do this:

```
Element elem = ...;
EventListener listener = ...;
boolean old = ((ItsNatNode)elem).isInternalMode();
((ItsNatNode)elem).setInternalMode(true);
try
{
  ((EventTarget)elem).addEventListener("DOMNodeInserted",
                               listener,false);
}
finally
{
    ((ItsNatNode)elem).setInternalMode(old);
}
```

   b. `DocumentEvent.`**`createEvent`**`(String)` now creates an internal event, the method `EventTarget.`**`dispatchEvent`**`(Event)` dispatches this event to the internal event listeners (event listeners registered in internal mode). There is no known use case.

   c. Methods like `HTMLInputElement.`**`focus`**`()` do nothing in internal mode.

6.4 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, Chrome, Safari[54], Opera etc) support mutation events.

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[55].

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

[55] 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).

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 registered with `EventTarget.`**`addEventListener`**`(…)` or `ItsNatDocument.`**`addEventListener(…)`** methods 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 commMode,String preSendCode)
{
    ParamTransport[] params =
            new ParamTransport[]{new NodeMutationTransport()};
    addEventListener(target,"DOMAttrModified",listener,useCapture,
            commMode,params,preSendCode);
    addEventListener(target,"DOMNodeInserted",listener,useCapture,
            commMode,params,preSendCode);
    addEventListener(target,"DOMNodeRemoved",listener,useCapture,
            commMode,params,preSendCode);
    addEventListener(target,"DOMCharacterDataModified",listener,
            useCapture,commMode,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();
            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.

6.5 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

6.5.1 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);
```

6.5.1.1 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);
```

6.5.2 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.

6.5.3 Complete node synchronization

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

6.5.4 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 almost ever reflected as attributes in the server DOM, though Java W3C DOM API have get/set based methods to get/set properties, Java DOM 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);
```

6.6 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 `ItsNatContinueEvent` 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 `ClientDocument.`**`addContinueEventListener`**, this action tells to ItsNat to generate and send JavaScript code to the client to call again the server firing an `ItsNatContinueEvent`; this event is received by the registered `EventListener` object. When the `ItsNatContinueEvent` 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)
{
    ItsNatEvent itsNatEvent = (ItsNatEvent)evt;
    ClientDocument clientDoc = itsNatEvent.getClientDocument();

    // We need the page title in this context:

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

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

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

6.7 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 `ItsNatUserEvent` 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 `ItsNatUserEvent` events. If several listeners were registered with the same pair node-name, they all will be called at the same time.

To fire an `ItsNatUserEvent` 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)
    {
        ItsNatUserEvent userEvt = (ItsNatUserEvent)evt;
        String title = (String)userEvt.getExtraParam("title");
        System.out.println("Page title: " + title);
```

```
            String url = (String)userEvt.getExtraParam("url");
            System.out.println("URL: " + url);
            Object[] both = userEvt. getExtraParamMultiple("both");
            System.out.println("Page title/URL: " + both[0] + "/" + both[1]);
        }
    };

    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 += "evt.setExtraParam('both',[document.title,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.

6.8 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 an `ItsNatTimerEvent` event object. An `ItsNatTimerEvent` inherits from `org.w3c.dom.events.Event` because is defined by ItsNat as a DOM extension.
- An `ItsNatTimerHandle` 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 `ItsNatTimerHandle.`**`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 = ...;
ClientDocument clientDoc = itsNatDoc.getClientDocumentOwner();

EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        ItsNatTimerEvent timerEvt = (ItsNatTimerEvent)evt;
        ItsNatTimerHandle handle = timerEvt.getItsNatTimerHandle();
        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 = clientDoc.createItsNatTimer();
ItsNatTimerHandle handle = timer.schedule(null,listener,1000,2000);

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

6.9 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 ClientDocument.addAsynchronousTask(Runnable,EventListener), this method starts a new thread executing the Runnable object in this thread; then immediately generates JavaScript code to fire an (asynchronous by default) event when is executed in the client. This event is caught internally by the framework 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. Optionally an EventListener can be registered and dispatched when the event (implementing ItsNatAsyncTaskEvent) is going to notify the client.

An alternative 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 in the asynchronous task **must not use** the ItsNatDocument or related objects 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 = ...;
ClientDocument clientDoc = itsNatDoc.getClientDocumentOwner();
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!');");
        }
    }
};

EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        itsNatDoc.addCodeToSend("alert('Finished really!');");
    }
};
clientDoc.addAsynchronousTask(task,listener);
    // by default lockDoc is false!!

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

6.10 COMET

ItsNat provides some COMET support without relying on special servers. The ItsNat COMET approach is based on AJAX or SCRIPT events: the client is ever waiting for an AJAX/SCRIPT pure asynchronous (hold modes are not allowed) event locked in server; when new JavaScript code with changes in server must be sent to the client, this event returns and updates the client and automatically a new AJAX/SCRIPT asynchronous request is sent to the server waiting for new asynchronous server changes. This technique is called "long polling"[56].

This solution requires one server thread per comet notifier, and a HTTP client to server connection blocked. This may sound a limitation for scaling to many concurrent COMET connections but is important to take into account that these threads are stalled (sleeping) when no asynchronous server changes occur, and modern operating systems can handle thousand of threads with no very much problem[57]. This technique (synchronous long polling) is the most standard and no specific NIO or Servlet 3.0 servlet container is required (for instance a NIO based like GlassFish), in fact recent studies are trying to debunk the popular belief that non-blocking IO (NIO) is more scalable than classic blocking IO[58] (this belief was true before Linux NPTL).

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; some browsers like MSIE 6 and 7 and FireFox <=v2 obey this limitation, other browsers like Chrome, Safari 3, FireFox 3 and Opera have a higher number of concurrent connections per host. Anyway use only one comet notifier to avoid your browser freezes!

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

final CometNotifier notifier =
            itsNatDoc.getClientDocumentOwner().createCometNotifier();
EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        itsNatDoc.addCodeToSend("alert('Tick From Event');");
    }
};
notifier.addEventListener(listener);

Thread backgroundThr = new Thread()
{
    public void run()
    {
        System.out.println("Background server task started");
        long t1 = System.currentTimeMillis();
        long t2 = t1;
        do
        {
```

[56] "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

[57] http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library

[58] http://blog.uncommons.org/2008/09/03/avoid-nio-get-better-throughput/

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

                if (!notifier.isStopped())
                {
                    synchronized(itsNatDoc)
                    {
                        itsNatDoc.addCodeToSend("alert('Tick From Thread');");
                    }
                    notifier.notifyClient();
                }
                t2 = System.currentTimeMillis();
            }
            while( (t2 - t1) < 10*60*1000 ); // Max 10 minutes

            notifier.stop();

            System.out.println("Background server task finished");
        }
    };

    backgroundThr.start();
```

There are two ways to modify the document:

1. In any time synchronizing the document

```
                    synchronized(itsNatDoc)
                    {
                        itsNatDoc.addCodeToSend("alert('Tick From Thread');");
                    }
                    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/SCRIPT event, and a new asynchronous AJAX/SCRIPT request is automatically sent to wait new changes. The method `CometNotifier.`**`stop`**`()` ends the COMET based listener.

2. Using an event listener registered on the `CometNotifier`.

```
    EventListener listener = new EventListener()
    {
        public void handleEvent(Event evt)
        {
            itsNatDoc.addCodeToSend("alert('Tick From Event');");
        }
    };
    notifier.addEventListener(listener);
```

When the client is going to be notified (a web request is going to return to the browser with any pending modification) an especial DOM event (implementing `ItsNatCometEvent`) is dispatched to the registered listeners, in this context the document is synchronized. Most of methods of this especial DOM event are useless.

6.11 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 (see "MARKUP FRAGMENTS" chapter).

Example:

```
        ItsNatDocument itsNatDoc = ...;
        Element refElem = ...;

        String code = "<b>New Markup Inserted</b><br />";
        DocumentFragment docFrag = itsNatDoc.toDOM(code);
        refElem.getParentNode().insertBefore(docFrag, refElem);
```

Be careful with the security using this method when the string is constructed containing user data. For instance, a malicious user can add unexpected elements like `<script>`, `<link>` or `<object>`. This is not the recommended approach, insert user provided data using DOM APIs *after* calling this method.

6.12 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. The same as page templates, markup fragments can be changed on runtime, the new markup is shown next time the fragment is included.

6.12.1 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();

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

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

6.12.2 SVG and XUL 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.registerItsNatDocFragment(
                        "manual.core.svgFragExample","image/svg+xml",
                        pathPrefix + "svg_fragment_example.xml");
```

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

Similar to SVG you can insert XUL fragments, in this case use the MIME type `application/vnd.mozilla.xul+xml`. The Feature Showcase includes some examples using XUL.

6.12.3 XML fragments

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

```
<?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.registerItsNatDocFragment(
                        "manual.core.xmlFragExample","text/xml",
                        pathPrefix + "xml_fragment_example.xml");
```

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

6.12.4 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.

6.12.4.1 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" />
```

6.12.5 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.getItsNatDocumentTemplate().getItsNatServlet();

ItsNatHTMLDocFragmentTemplate fragTemplate =
    (ItsNatHTMLDocFragmentTemplate)servlet.getItsNatDocFragmentTemplate(
           "manual.core.htmlFragExample");
DocumentFragment docFrag = fragTemplate.loadDocumentFragmentBody(itsNatDoc);
```

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

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

6.13 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[59] provide two iterator interfaces, almost unknown[60]: `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
    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();
        }
```

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

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

```
        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);
```

6.14 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<Element> list = elemList;
for(ListIterator<Element> it = list.listIterator(); it.hasNext(); )
{
    Element curElem = 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").

6.15 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).

6.16 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 first and deepest text node with some text (not only spaces, tabs or line feeds) 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>
```

Spaces, tabs and line fees can be included. Previous pattern example can be defined as:

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

And again "`Text Pattern`" is replaced by the value submitted to the renderer, because this is the first text node with "some" text (not only spaces, tabs or line feeds).

The default renderer is even smarter, when no significative text is found the tree is traversed until a deep (HTML) element is found able to contain text nodes (with significative text), in this case the text node is added to the end as a child node. This case is interesting when there is no significative text usually because the markup was previously rendered with an empty text.

For instance the following pattern:

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

Renders:

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

If our text must be on the left of the image you add a simple `span` to notify the renderer this node is able to hold text nodes:

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

Renders:

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

Anyway on rendering you can ever relay on your original pattern (usually including a significative text in the place going to be replaced by the real values) when using DOM utilities and components (they all use this default rendering by default) calling configuration methods like `ElementLabel.`**`setUsePatternMarkupToRender`**`(boolean)`, or `ItsNatLabelUI.`**`setUsePatternMarkupToRender`**`(boolean)` etc with a parameter `true`.

The default 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.

6.17 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).

6.18 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>
```

6.18.1 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>
```

6.18.2 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.

6.19 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>
```

6.19.1 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).

6.19.2 Using custom structures

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

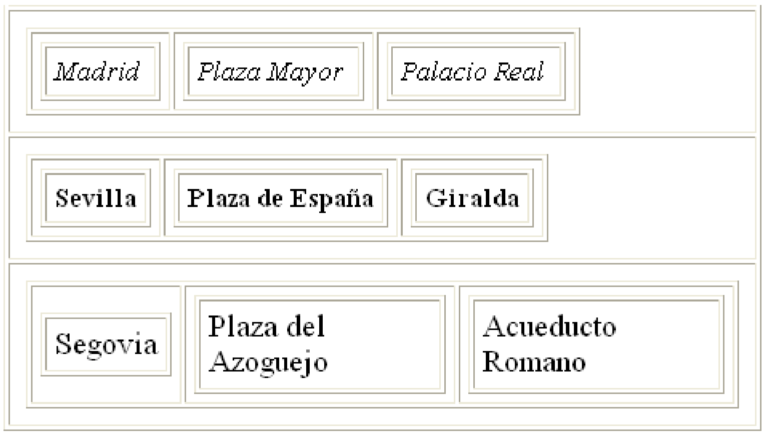
```
    <table border="1px" cellpadding="10px">
    <tbody id="tableId3">
        <tr> <= row parent
            <td>
                <table border="1px" cellpadding="5px">
                <tbody>
                    <tr> <= row content parent
                        <td>  <= cell parent
                        <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 output is very big, an image is better:



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

6.19.3 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:

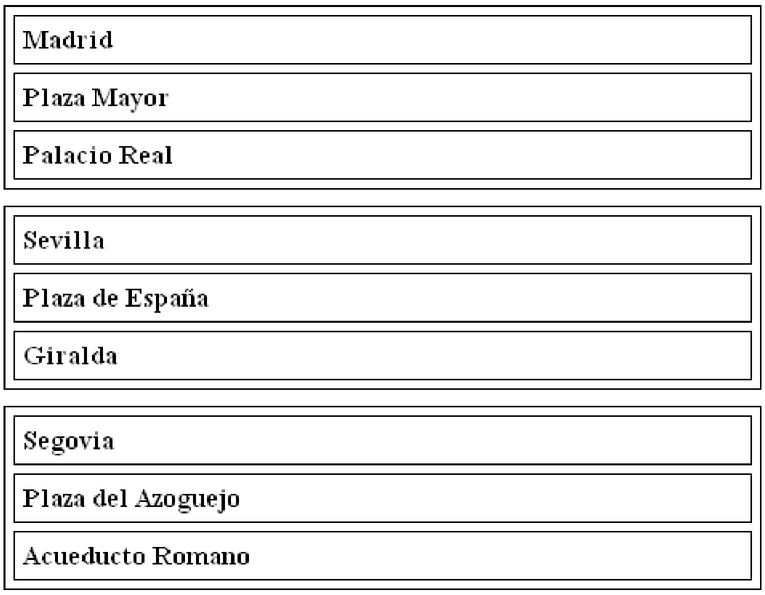
```
    <div id="tableId4" style="margin-top: 10px; width: 100%;">
        <div style="margin-top: 10px; border: 1px solid;">
            <div style="margin: 5px; padding: 5px; border: 1px solid;"><b>Cell
Pattern</b></div>
        </div>
    </div>


    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:



Madrid

Plaza Mayor

Palacio Real

Sevilla

Plaza de España

Giralda

Segovia

Plaza del Azoguejo

Acueducto Romano

6.19.4 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.

6.20 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:

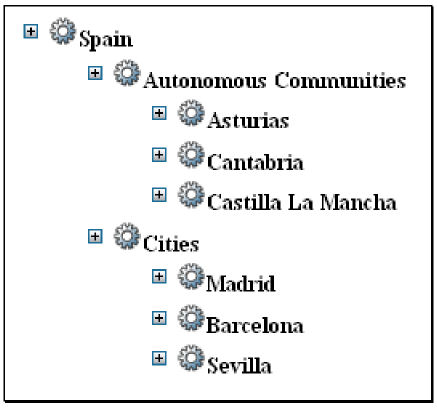
```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
Element parent = doc.getElementById("treeId");
ElementGroupManager factory = itsNatDoc.getElementGroupManager();
ElementTree elemTree = factory.createElementTree(false,parent,true);

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>
```

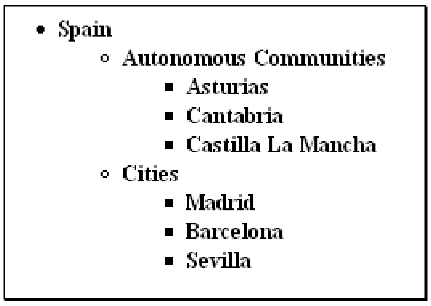
And is rendered as:


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

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

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

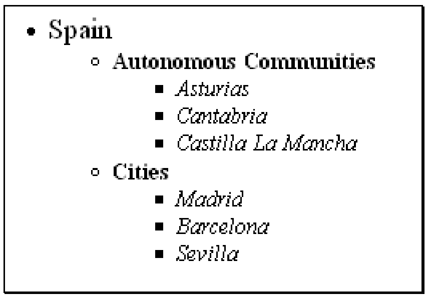
Rendered as:


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

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*

6.20.1 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.

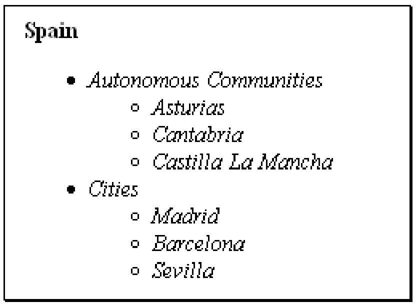
```
<div id="treeId">
    <span><b>Root Content Pattern</b></span>
    <ul>
        <li>
            <span><i>Child Content Pattern</i></span>
            <ul></ul>
        </li>
    </ul>
</div>
```

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*

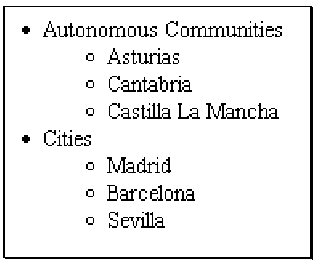
6.20.2 Rootless

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

6.20.3 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);
...
```

6.20.4 Tree-Tables

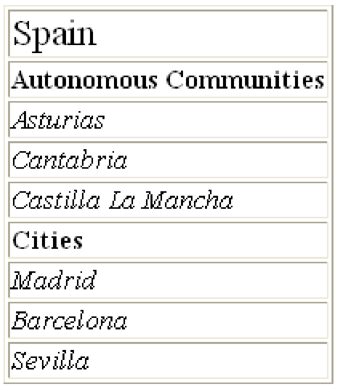
ItsNat supports “out of the box” tree tables: a tree which layout is a list following the tree order, this list can be hold by an HTML table when every node is a row. The tree is shown as a list or table but the hierarchy is internally kept, for instance when a tree node is removed child nodes are removed 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:

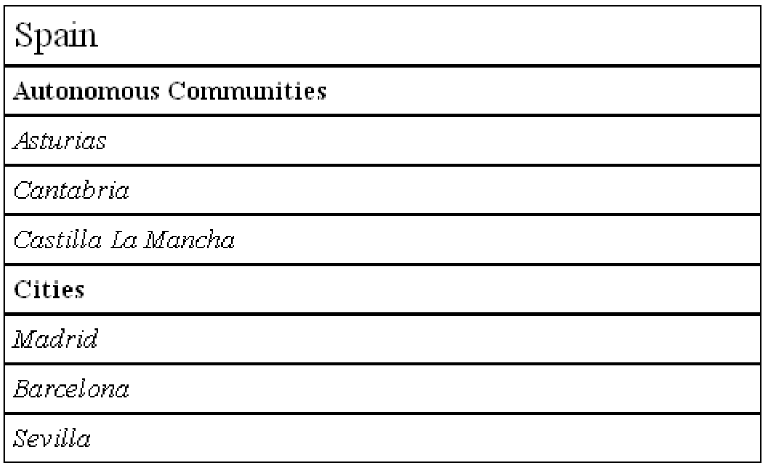


Spain
Autonomous Communities
Asturias
Cantabria
Castilla La Mancha
Cities
Madrid
Barcelona
Sevilla

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>
</div>
```

That renders (using the previous Java code):



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

6.20.5 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.

6.21 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_action=attach_client
           &itsnat_refresh_method=timer
           &itsnat_read_only=true
           &itsnat_session_id=${sessionId}
           &itsnat_doc_id=${docId}
           &itsnat_refresh_interval=${refreshInterval}
```

```
                &itsnat_comm_mode=${commMode}" target="_blank">
            ${linkText}
        </a>
        ...
    </div>
```

This markup fragment contains a link used to launch a "remote view/control" page to monitor/control "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("commMode",new Integer(CommMode.XHR_ASYNC));
    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.

Use `ItsNatDocument.`**`createItsNatVariableResolver(boolean disconnected)`** with `disconnected` set to true if only local variables defined into the variable resolver are used to resolve the variables in markup. This improves the security against attempts to "extract" undesired server data by a malicious user.

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_action=attach_client&itsnat_read_only=true
                &itsnat_refresh_method=timer&itsnat_session_id=1179308955078_2
                &itsnat_doc_id=1179310638515_1&itsnat_refresh_interval=3000
                &itsnat_comm_mode=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(true);
        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(true);
    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.

6.21.1 Variables and cache

ItsNat avoids caching a node is this contains an attribute or text node with a `${}` construction.

6.21.2 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 an `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(true);

        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.

6.22 W3C ELEMENTCSSINLINESTYLE IMPLEMENTATION

ItsNat provides a partial Java implementation[61] of the W3C Style/CSS[62] `ElementCSSInlineStyle` interface[63], any DOM `Element` implements this interface. This

[61] Cursor syntax is not implemented

[62] http://www.w3.org/TR/DOM-Level-2-Style/

[63] http://www.w3.org/2003/01/dom2-javadoc/org/w3c/dom/css/ElementCSSInlineStyle.html

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.

In a browser the CSS properties of the "style" attribute are usually reflected in the CSS properties of the style object, but the contrary is false, the attribute is not changed when a CSS property is changed through the style object. This is not the case ElementCSSInlineStyle of implementation in server, in this case the "style" attribute is the source and the target of any call to ElementCSSInlineStyle related methods. Furthermore ItsNat ElementCSSInlineStyle implementation is tolerant to collateral modification of the "style" attribute like using Element.**setAttribute**(String,String) method, any direct change is reflected by ElementCSSInlineStyle methods.

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>
```

```
public void handleEvent(Event evt)
{
    Element currTarget = (Element)evt.getCurrentTarget(); // Link

    ElementCSSInlineStyle styleElem = (ElementCSSInlineStyle)currTarget;
    CSSStyleDeclaration cssDec = styleElem.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)styleElem.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.

6.23 SAVING SERVER MEMORY: DISCONNECTED NODES

The simplest way to save server memory is node caching provided by templates (`nodecache` attribute or explicit configuration methods), if a markup zone is static (you do not need to access/modify this zone in no way by Java code) node caching works for you removing nodes in server but sending them to client when appropriated (cached nodes are shared between users in plain markup form).

Sometimes you need to render a markup zone only one time after insertion into the document, so this zone is in theory dynamic, you cannot cache because cached subtrees are never in server DOM (a cached subtree is replaced by a text node with a mark). This "temporally dynamic" subtree is occupying server memory, you cannot remove it because when removed the client DOM is also removed, it remains in memory until is replaced in some way with a different markup.

"Disconnected Nodes" feature comes to rescue, the idea is simple, remove all of child nodes of the selected parent node with no action performed in client (only elements are supported as parents). By this way, the parent node does not contain child nodes but the client part remains untouched saving memory in server. Furthermore, the client nodes below the parent node can be freely modified (including removing and insertion) by user defined JavaScript code.

Disconnecting the child nodes of an element is simple, just call the method:

`ItsNatDocument.`**`disconnectChildNodesFromClient`**`(Node)`

This method removes the child nodes returning them in the same way as the method `ItsNatDOMUtil.`**`extractChildren`**`(Node)` (null when empty, a node when containing only a child node or a `DocumentFragment`), the client DOM is not removed (only in server) and the parent node is marked so the call to method:

`ItsNatDocument.`**`isDisconnectedChildNodesFromClient`**`(Node)` returns true.

Reconnection is possible, just call to:

`ItsNatDocument.`**`reconnectChildNodesToClient`**`(Node)`

When reconnected the client DOM subtree is cleared and now client and server and again in sync, when you add new child nodes the same nodes will be added in client.

The typical use of "disconnected nodes" is to save server memory removing the child nodes of a parent element usually the parent of different DOM subtrees, to avoid the burden of remembering whether the parent node is "connected" (for instance calling `ItsNatDocument.isDisconnectedChildNodesFromClient(Node)`) before adding a new DOM subtree, when you insert a new child node into a disconnected node (that is, no child nodes), ItsNat automatically reconnect the parent node before inserting the new nodes. In summary you do not need to call `ItsNatDocument.`**`reconnectChildNodesToClient`**`(Node)` if you do not want.

The Feature Showcase contains an example of disconnection/reconnection of child nodes.

6.23.1 When to disconnect nodes

Disconnect nodes of course when you need to save server memory because these nodes no longer are going to be used in server.

Anyway sometimes we can drive our use cases to this scenario. Think you have a page state with a very long list of items, these items may have buttons or links with some kind of action with no impact on the list (for instance show a modal layer with detail data of the selected item). The typical approach of ItsNat is to add event listeners to these buttons/links or to add a single event listener to the parent element of the list, in the latter case list items are needed to check the `target` property (`Event.getTarget()`) of received events to know what item and what action element was actually clicked.

If you do not have memory issues, the straightforward approach is to keep the DOM nodes in server, but if memory is critical there is an alternative: add `onclick` handlers to the active elements of your item list (use `Element.setAttribute("onclick","your code")` ItsNat solves any browser compatibility problem for you), in the same time register a "user event listener" in server, when an active node is clicked your code in the `onclick` handler must fire and send a user event with the necessary information to know what action must be executed in server. Now you are ready to disconnect the item list.

This scenario is very typical of Single Page Interface web sites because most of actions are navigation to different states replacing DOM zones. These zones are usually static and cached but sometimes they are initially dynamic because of some rendering change (for instance internationalization of texts), in this case user events and node disconnection save tons of memory becoming your web site mainly stateless and remaining SPI and server centric.

6.23.2 Drawbacks of disconnected nodes feature

6.23.2.1 Remote view/control clients

Because disconnected DOM nodes are removed from the document in server, when you attach a new remote view/control client (a.k.a. attached client) these nodes will be missing in server and there is no way to send these missing nodes to the page of the remote view/control user showing a partially rendered initial page. This drawback only affects to the initial page, further disconnected nodes are kept in the browser of all clients already attached to the server document (owner and remote view/control clients).

The positive side of this drawback is that remote view/control clients can be used to see what DOM subtrees are missing in the server saving memory because of disconnected actions.

6.23.2.2 Loading phase in fast load mode

When you disconnect nodes when loading the page in fast load mode, the removed node/fragment only can be reinserted into the same parent node, do not perform any other action with these nodes different to reinsertion into the same parent node (and this action is very unusual because is to revert the disconnected action). To understand why this limitation happens, on loading in fast load mode the disconnected nodes are temporally kept by ItsNat engine and used to render the initial page, because page rendering happens at the end of the phase cycle, ItsNat supposes these removed nodes are in the same state as the moment when they were removed.

As you can easily understand, this limitation is minimal or none.

6.24 REMOTE VIEW/CONTROL (A.K.A. ATTACHED CLIENT)

Remote view/control is an ItsNat "out of the box" feature, is a direct consequence of the TBITS (The Browser Is The Server) approach. ItsNat allows binding a browser window (attached client) to an existing DOM document loaded by another user/window; this window becomes a remote viewer ready to monitor any action of the initial user or can be another client sending events to the server document the same as the client owner of the document. The attached client initially loads the current state of the DOM of the document in server, this client may be associated to a different web (servlet) session than the client which initiated the original document.

ItsNat provides a security system to avoid or deny remote view/control requests; ItsNat remote view/control system is not active by default.

There are two modes:

1. Read only remote views: attached clients cannot send events to the server. In this mode ItsNat offers some kind of remote control: remote viewer refreshing is controlled by an optional server listener; this Java listener can do anything including changing the server DOM tree.

2. Full remote control: attached clients can send events to the server interacting with the same server DOM tree (refresh events are also valid).

Two or more different clients with different browsers can be associated to the same document in server, ItsNat cares about the differences, client pages share the same server DOM tree but client-server synchronization tasks are managed by ItsNat and adapted to the concrete target client.

Remote view/control of a concrete "alive" document may be requested using a URL (a POST request is valid too). 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 for remote view/control:

- **`itsnat_action`**`=attach_client`

  Specifies a remote view/control request.

- **`itsnat_read_only`**`=true | false`

  Specifies if the attached client can or cannot send events to the server (read only or full remote control). This parameter is optional and by default is true (read only).

- **`itsnat_refresh_method`**`=timer | comet | none`

  Specifies the refresh method, timer or comet or none (no refresh events). This parameter is optional and by default is `none`.

- **`itsnat_session_id`**`=`*`sessionOfDocumentId`*

  Specifies the session id of the document target to attach.

- **`itsnat_doc_id`**`=`*`docToAttachId`*

  The id of the document to attach.

- **itsnat_refresh_interval**=*milliseconds*

  Refresh interval in milliseconds, only if specified a timer refresh.

- **itsnat_comm_mode**=*commModeIntValue*

  Communication mode of refresh events, this integer value must be one of the following: 1 (CommMode.XHR_SYNC), 2 (CommMode.XHR_ASYNC), 3 (CommMode.XHR_ASYNC_HOLD), 4 (CommMode.SCRIPT) or 5 (CommMode.SCRIPT_HOLD). If not specified the default value of the target document is used. In Comet mode only use pure asynchronous modes (value 2 of CommMode.XHR_ASYNC or 4 of CommMode.SCRIPT).

- **itsnat_event_timeout**=*milliseconds*

  The maximum time in milliseconds any event (timer refresh or COMET updater event) will wait to the server before aborting the request (the remote control session is stopped). If not specified the default value of the target document is used. A negative value means no timeout.

- **itsnat_wait_doc_timeout**=*milliseconds*

  The maximum time in milliseconds <iframe>, <object> or <embed> elements of a remote view/control client will wait for the target document if the client owner is going to load and automatically bind to the parent these elements, because the owner client is the responsible of initially loading the content of iframe/object/embed elements. If not specified the default value is 0, it means no wait.

For instance using a timer:

```
http://localhost:8080/myapp/myservlet?itsnat_action=attach_client&itsnat_read_on
ly=true&itsnat_refresh_method=timer&itsnat_session_id=ss_463&itsnat_doc_id=doc_3
&itsnat_refresh_interval=3000&itsnat_comm_mode=2&itsnat_event_timeout=-1
&itsnat_wait_doc_timeout=20000
```

The same using comet:

```
http://localhost:8080/myapp/myservlet?itsnat_action=attach_client&itsnat_read_on
ly=true&itsnat_refresh_method=comet&itsnat_session_id=ss_463&itsnat_doc_id=doc_3
&itsnat_comm_mode=2&itsnat_event_timeout=-1&itsnat_wait_doc_timeout=20000
```

To enable remote view/control, a supervisor request listener implementing ItsNatAttachedClientEventListener must be registered. This listener receives ItsNatAttachedClientEvent event objects. An ItsNatAttachedClientEvent event object has two subtypes: ItsNatAttachedClientTimerEvent if refresh method is timer and ItsNatAttachedClientCometEvent if specified comet, no specific interface if specified none in this mode there is no refresh events.

For instance:

```
public class RemoteControlSupervision
        implements ItsNatAttachedClientEventListener
{
    public RemoteControlSupervision()
    {
    }
    public void handleEvent(ItsNatAttachedClientEvent event)
    {
        if (event.getItsNatDocument() != null)
            processWithDocument(event);
        else
            processNotDocument(event);
    }

    public void processWithDocument(ItsNatAttachedClientEvent event)
    {
        int phase = event.getPhase();
        if (phase == ItsNatAttachedClientEvent.REQUEST)
        {
            String[] msg = new String[1];
            if (event instanceof ItsNatAttachedClientTimerEvent)
            {
                ItsNatAttachedClientTimerEvent timerEvent =
                       (ItsNatAttachedClientTimerEvent)event;
                boolean accepted = (timerEvent.getRefreshInterval() >= 3000);
                event.setAccepted(accepted);
                if (!accepted) msg[0] = "Refresh interval too short: " +
                                      timerEvent.getRefreshInterval();
            }
            else if (event instanceof ItsNatAttachedClientCometEvent)
            {
                event.setAccepted(true);
            }
            else // "none" refresh mode
            {
                event.setAccepted(true);
            }

            if (event.getWaitDocTimeout() > 30000)
            {
                event.setAccepted(false);
                msg[0] = "Too much time waiting for iframe/object/embed: " +
                               event.getWaitDocTimeout();
            }

            ItsNatServletRequest request = event.getItsNatServletRequest();
            ServletRequest servRequest = request.getServletRequest();

            if (!event.isAccepted())
            {
                ItsNatServletResponse response =
                       event.getItsNatServletResponse();
                ItsNatServlet servlet = response.getItsNatServlet();
                Map<String,String[]> newParams =
                   new HashMap<String,String[]> (servRequest.getParameterMap());
                newParams.remove("itsnat_action");
                       // Removes: itsnat_action=attach_doc
                newParams.put("itsnat_doc_name",
                   new String[]{"feashow.ext.core.misc.remCtrlReqRejected"});
                newParams.put("reason", msg); // submitted as array
                servRequest = servlet.createServletRequest(servRequest,
                                                  newParams);
                servlet.processRequest(servRequest,
                       response.getServletResponse());
            }
        }
        else if (phase == ItsNatAttachedClientEvent.REFRESH)
        {
            ClientDocument observer = event.getClientDocument();
            if (observer.getItsNatDocument().isInvalid())
            {
                observer.addCodeToSend(
                       "alert('Observed document was destroyed');");
            }
            else
            {
                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");
                }
            }
        }
        // ItsNatAttachedClientEvent.LOAD & UNLOAD : nothing to do

        if (!event.isAccepted())
            event.getItsNatEventListenerChain().stop(); // Not really necessary
    }

    public void processNotDocument(ItsNatAttachedClientEvent event)
    {
       ...
    }

}
```

This complete example shows the typical code used to control the life cycle of an attached client.

The attached client life cycle has the following phases:

- `ItsNatAttachedClientEvent.`**`REQUEST`**

  The new client requests to be attached to the specified session/document.

- `ItsNatAttachedClientEvent.`**`LOAD`**

  The request was accepted and document state in server is going to be sent to the viewer. In this phase we know the document will be returned to the client. If the event in this phase is "rejected" (calling `ItsNatAttachedClientEvent.`**`setAccepted`**`(false)`) the associated document is returned to the client but there is no further updating. Use this technique to get a snapshot of the page monitored.

- `ItsNatAttachedClientEvent.`**`REFRESH`**

  The attached client sent a refresh event (timer) or the associated document notifies a change (comet). If the document is invalid no more refresh events will be dispatched.

- `ItsNatAttachedClientEvent.`**`UNLOAD`**

The user has closed the attached client. This phase is not reached if the document was invalidated during a previous phase because ItsNat ever usually gives an opportunity to notify this fact to the end user, for instance in the last refresh.

By default any remote view/control request is not accepted, this sentence in the `ItsNatAttachedClientEvent.`**`REQUEST`** phase controls whether the request is accepted:

```
event.setAccepted(accepted);
```

If not accepted in request phase this example shows how to forward this request to return a normal error page registered on ItsNat:

```
Map<String,String[]> newParams =
  new HashMap<String,String[]>(servRequest.getParameterMap());
newParams.remove("itsnat_action");
         // Removes: itsnat_action=attach_doc
newParams.put("itsnat_doc_name",
    new String[]{"feashow.ext.core.misc.remCtrlReqRejected"});
newParams.put("reason", msg); // submitted as array
servRequest = servlet.createServletRequest(servRequest,
                                           newParams);
 servlet.processRequest(servRequest,
         response.getServletResponse());
```

This code uses a special method, `ItsNatServlet.`**`createServletRequest`**`(ServletRequest, Map)`, this method creates a new "fake" `ServletRequest` object wrapping the specified request (first parameter), the second parameter is very important because this `Map` is used as the parameter collection internally used by the new `ServletRequest` object (any other data is obtained forwarding to the wrapped request object). This feature is a workaround to add, remove and replace parameters in a `ServletRequest` object, for instance we need to remove the `itsnat_action` parameter because current value is `attach_client` and this is a failed remote/view control request[64].

Another simpler alternative:

```
ServletResponse response =
        event.getItsNatServletResponse().getServletResponse();
try
{
   Writer out = response.getWriter();
   out.write("<html><body><h1>Remote control request rejected.
        Interval too short");
   out.write("</h1></body></html>");
}
catch(IOException ex) { new RuntimeException(ex); }
```

[64] A fake request object is not needed if we just want to change the value of a parameter because internally ItsNat calls first ServletRequest.getAttribute(String) and then ServletRequest.getParameter(String) when searching for ItsNat known parameters. Because the method ServletRequest.setAttribute(String,Object) exists, setting a new attribute with the same name of the parameter going to "rewrite" solves this problem because ItsNat will look first the attribute registry. Nevertheless there is no way to remove an existing parameter, a new "fake" ServletRequest is the simplest and compatible workaround.

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 observed page was closed (document destroyed). The `ClientDocument` object represents the browser page monitoring/controlling the document in server.

To receive events targeted to an `ItsNatAttachedClientEventListener` object, this object must be registered.

```
RemoteControlSupervision remCtrlSup = new RemoteControlSupervision();
```

Three options to register a remote view/control listener: globally, per document template or per document.

1. Globally registered: registering at servlet level

   ```
   ItsNatServlet servlet = ...;
   servlet.addItsNatAttachedClientEventListener(remCtrlSup);
   ```

2. Per document template: registering per document template

   ```
   ItsNatDocumentTemplate docTemplate = ...;
   docTemplate.addItsNatAttachedClientEventListener(remCtrlSup);
   ```

3. Per document

   ```
   ItsNatDocument itsNatDoc = ...;
   itsNatDoc.addItsNatAttachedClientEventListener(remCtrlSup);
   ```

6.24.1 More about the request phase and how to ask permission to user

In the request phase the target `ItsNatDocument` can be accessed calling `getItsNatDocument()` in the remote control event. However this document object *is not synchronized*, this is a very good thing because from the thread of the remote control event request we can synchronize the target document and modify it in some way, release this lock and wait for something, for instance we can wait the monitored user to answer if he/she wants to be monitored or to accept a new client in a collaborative application.

The Feature Showcase contains a complete example of how we can request permission to the end user to monitor his/her web page. This is a code snippet of that example:

```
public boolean askToUser(ItsNatAttachedClientEvent event,String[] msg)
{
    final boolean[] ready = new boolean[] { false };
    final boolean[] answer = new boolean[] { false };

    ClientDocument observer = event.getClientDocument();
    ItsNatDocument itsNatDoc = observer.getItsNatDocument();

    RemoteControlUserRequestEventListener listener =
            new RemoteControlUserRequestEventListener(ready,answer);
    synchronized(itsNatDoc)
    {
        itsNatDoc.addEventListener(listener);
    }

    synchronized(ready)
    {
        int timeout =
```

```
                 5000 + RemoteControlTimerMgrGlobalEventListener.PERIOD;
            try { ready.wait(timeout); }
            catch(InterruptedException ex) { throw new RuntimeException(ex); }
        }

        synchronized(itsNatDoc)
        {
            // If code to ask user was not sent to the user
            // then is not sent.
            itsNatDoc.removeEventListener(listener);
        }

        boolean accepted = answer[0];
        if (!accepted)
        {
            if (itsNatDoc.isInvalid())
                msg[0] = "The user closed the page before answering.";
            else if (ready[0])
                msg[0] = "The user rejected monitoring.";
            else
                msg[0] = "Timeout (no user answer).";
        }

        return accepted;
    }
```

6.24.2 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 in 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>${agentInfo}</td>
                <td><a href="servlet?itsnat_action=attach_client
                       &itsnat_read_only=true
                       &itsnat_refresh_method=timer
                       &itsnat_session_id=${sessionId}&itsnat_doc_id=${docId}
                       &itsnat_refresh_interval=${refreshInterval}
                       &itsnat_comm_mode=${commMode}&itsnat_event_timeout=-1"
                       target="_blank">Link</a></td>
                <td><a href="servlet?itsnat_action=attach_client
                       &itsnat_read_only=true
                       &itsnat_refresh_method=comet
                       &itsnat_session_id=${sessionId}&itsnat_doc_id=${docId}
                       &itsnat_comm_mode=${commMode}&itsnat_event_timeout=-1"
                       target="_blank">Link</a></td>
            </tr>
        </tbody>
    </table>
```

```
ItsNatDocument itsNatDoc = ...;
Document doc = itsNatDoc.getDocument();
ItsNatDocumentTemplate thisDocTemplate = itsNatDoc.getItsNatDocumentTemplate();

ItsNatServlet itsNatServlet =
                        itsNatDoc.getItsNatDocumentTemplate().getItsNatServlet();
ItsNatServletContext appCtx =
                 itsNatServlet.getItsNatServletConfig().getItsNatServletContext();

final List<ItsNatSession> sessionList = new LinkedList<ItsNatSession>();
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(true);
resolver.setLocalVariable("refreshInterval",new Integer(3000));
resolver.setLocalVariable("commMode",new Integer(CommMode.XHR_ASYNC));

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)
        {
            ItsNatDocumentTemplate docTemplate =
                                  currRemDoc.getItsNatDocumentTemplate();
            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);
        }
    }
```

6.24.3 Comet limitations

ItsNat internally uses a comet notifier per remote view, to avoid a browser hang open only a remote viewer in browsers like Internet Explorer 6 or 7. See "COMET" chapter for more info.

6.24.4 Document not found detection

When ItsNat does not find a server document with the provided session and document ids, it throws by default an exception. This exception can be annoying for an end user using the remote controlling capabilities of ItsNat.

When the document is not found (there is no session with the provided id or there is no document with the provided id in that session), ItsNat dispatches an especial `ItsNatAttachedClientEvent` to the `ItsNatAttachedClientEventListener` listeners registered in "servlet level", that is to say, calling the method:

```
ItsNatServlet.addItsNatAttachedClientEventListener(ItsNatAttachedClientEventList
ener)
```

As there is no server document associated, a call to the method `ItsNatAttachedClientEvent.getItsNatDocument()` returns null.

The following class shows how we can detect this situation and redirect to a specific ItsNat based page registered with the template name `feashow.ext.core.misc.remCtrlDocNotFound`, specifically designed to show this type of error:

```
public class RemoteControlSupervision
                implements ItsNatAttachedClientEventListener
{
    public RemoteControlSupervision()
    {
    }

    public void handleEvent(ItsNatAttachedClientEvent event)
    {
        if (event.getItsNatDocument() != null)
            processWithDocument(event);
        else
            processNotDocument(event);
    }

    public void processWithDocument(ItsNatAttachedClientEvent event)
    {
        ...
    }
    public void processNotDocument(ItsNatAttachedClientEvent event)
    {
        ItsNatServletResponse response = event.getItsNatServletResponse();
        if (event.getPhase() == ItsNatAttachedClientEvent.REQUEST)
        {
            ItsNatServlet servlet = response.getItsNatServlet();
            ServletRequest servRequest =
               event.getItsNatServletRequest().getServletRequest();
            Map<String,String[]> newParams =
               new HashMap<String,String[]>(servRequest.getParameterMap());
            newParams.remove("itsnat_action");
               // Removes: itsnat_action=attach_doc
            newParams.put("itsnat_doc_name",
               new String[]{"feashow.ext.core.misc.remCtrlDocNotFound"});
            servRequest = servlet.createServletRequest(servRequest, newParams);
            servlet.processRequest(servRequest,response.getServletResponse());
        }
        else // ItsNatAttachedClientEvent.REFRESH
        {
            response.addCodeToSend("if (confirm('Session is expired. Close
Window?')) window.close();");
        }
    }
}
```

Or showing this very simple page:

```
        if (event.getPhase() == ItsNatAttachedClientEvent.REQUEST)
        {
            ServletRequest request =
                         event.getItsNatServletRequest().getServletRequest();
            String sessionId = request.getParameter("itsnat_session_id");
            String docId = request.getParameter("itsnat_doc_id");
            try
            {
              Writer out = response.getServletResponse().getWriter();
              out.write("<html><body><h1>Session/document not found with id:");
              out.write(sessionId + "/" + docId);
              out.write("</h1></body></html>");
            }
            catch(IOException ex) { new RuntimeException(ex); }
        }
```

6.24.5 Session expired or web application reload detection

When a server document is being monitored (one or more clients attached besides the client owner), the session of the owner client can expire or the web application might be reloaded, then a refresh event might be sent to the server from client to update the client with any change; in this scenario there is no vestige of the monitored page nor the remote client in the server[65]. This case is almost exclusive of attached clients using a timer when the interval is relatively high.

[65] ItsNat keeps track of the client monitoring a server document in the client session, though the observed document is invalidated the client document of the monitor is not cleared in server until a remote control event is dispatched to user code. When user code is notified no more refresh events will

In this situation ItsNat throws by default an exception. This exception can be annoying for an end user. To avoid this exception ItsNat dispatches an especial `ItsNatAttachedClientEvent` to the `ItsNatAttachedClientEventListener` listeners registered in "servlet level", this events returns null when `ItsNatAttachedClientEvent.`**`getItsNatDocument`**`()` is called.

They are two differences between this case and "document not found":

1. The phase is `REFRESH` now.

2. The client is waiting for JavaScript as result not a web page.

The following code shows how we can mix both scenarios and notify to the user when the session is lost:

```
public void handleEvent(ItsNatAttachedClientEvent event)
{
    ItsNatDocument itsNatDoc = event.getItsNatDocument();
    if (event.getItsNatDocument() != null) return;

    ItsNatServletResponse response = event.getItsNatServletResponse();
    if (event.getPhase() == ItsNatAttachedClientEvent.REQUEST)
    {
      ...
    }
    else // ItsNatAttachedClientEvent.REFRESH
    {
        response.addCodeToSend("if (confirm('Session is expired. Close
           Window?')) window.close();");
    }
```

What about `LOAD` and `UNLOAD` phases?

Remote control listeners are never called in `LOAD` and `UNLOAD` phases when the `ItsNatDocument` observed was absolutely lost.

`LOAD` case: `REQUEST` and `LOAD` phases are consecutive, if document was lost when requesting it this case is managed in `REQUEST` phase as seen before, `LOAD` phase is only reached with an "alive" document.

The `UNLOAD` phase is different, when the end user closes the browser page a `unload` notification is sent to the server, if the document was lost ItsNat silently ignores this `unload` event because nothing useful can be done (browser page is closed and there is no server data associated to this page).

6.24.6 Implications of full remote control

When there is only one client associated to the document in server many APIs work the same, that is, `ScriptUtil` as returned by `ItsNatDocument.`**`getScriptUtil`**`()` works the same as the instance returned by `ClientDocument.`**`getScriptUtil`**`()` or `ItsNatDocument.`**`addCodeToSend`**`(Object)` is the same as

be sent. The client document monitoring, if exists in server, keeps track of the document monitored though is invalidated.

`ClientDocument.`**`addCodeToSend`**`(Object)` or `ItsNatDocument.`**`addEventListener`**`(…)` and similar methods are the same as calling `ClientDocument.`**`addEventListener`**`(…)`. This is the reason this manual uses `ItsNatDocument` methods most of the time, because typical ItsNat applications do not require several clients associated to the same document (or they are usually read-only observers of the client owner being monitored).

When there is more than one client associated to the document, methods on `ItsNatDocument` work different as the same methods on `ClientDocument`. For instance, any `ItsNatDocument.`**`addEventListener`**`(…)` method registers the specified listener in ALL clients able to send events, that is, the client owner and clients with full remote control (read only parameter set to false), this implies our listener may receive events from several different clients. When you need to specify an event listener only for a concrete client use `ClientDocument.`**`addEventListener`**`(…)` methods or if custom JavaScript code must be sent only to a specific client use `ClientDocument.`**`addCodeToSend`**`(Object)`.

Similar behavior of `ItsNatComponent.`**`addEventListener`**`(String,EventListener)` and `ItsNatComponent.`**`addEventListener`**`(ClientDocument,String,EventListener)`, the former method registers a listener for all current and future clients, the latter method only registers the specified listener for the specified client provided as the first parameter.

The built-in DOM methods like `EventTarget.`**`addEventListener`**`(String, EventListener, boolean)` are equivalent to `ItsNatDocument` counterparts (in this example `ItsNatDocument.`**`addEventListener`**`(EventTarget, String, EventListener, boolean)`) because there is no way to specify the target client.

Take care with some DOM events like `load` , `DOMContentLoaded`, `beforeunload` and `unload` because they may be confusing if you register document-level listeners, client-level registration could be a better option, anyway the method `ItsNatEvent.`**`getClientDocument`**`()` returns the client which sent the event. Use of listeners related to mutation events and client to server automatic synchronization may be problematic in multi-client applications.

As said before `ItsNatDocument.`**`addEventListener`**`(…)` methods (and other similar methods like `ItsNatDocument.`**`addMutationEventListener`**`(..)`) register the specified listeners in all clients authorized to receive events. What if a new attached client with full remote control mode is added *after* the listener was registered? ItsNat automatically registers all document-level listeners in the new client.

6.25 EXTREME MASHUPS (A.K.A. ATTACHED SERVER)

From Wikipedia[66] "*mashup is a web page or application that uses or combines data or functionality from two or many more external sources to create a new service*". In spite of this definition is correct, usually the "mashup" term makes reference to the external sources not to the final aggregated result. Typical mashups take control of a piece of web page (usually a rectangle) to provide some specific service which content is delivered from a different host

[66] http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)

than the host delivering the page, other mashups modify systematically the page adding some specific behavior (for instance showing a picture of the page that points a link[67]).

ItsNat provides "extreme mashups", they are "extreme" because a client page loaded in browser (how it was delivered is not important) can be sent to the server (of course running ItsNat) and used as the initial markup template to build an ItsNatDocument (attached server document) and attaching this document to the typical user defined Java code for document processing including events. By this way the server takes FULL CONTROL of the client page.

The best way to understand how extreme mashups work is with a simple example. For instance the following HTML code was loaded by a browser, this HTML page was sent to the browser with text/html MIME (MIME is important), the source if this page is not important (could be a static file, or a dynamic page served by PHP, servlets, ASP, RoR…).

```
<!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>Extreme Mashup HTML MIME Demo</title>
</head>
<body style="background-color:white; margin:30px;">

<div itsnat:nocache="true" xmlns:itsnat="http://itsnat.org/itsnat">
    <a href="javascript:;" onclick="return false;"
         id="clickableId1">Clickable Elem 1</a>
    <br /><br />
    <a href="javascript:;" onclick="return false;"
         id="clickableId2">Clickable Elem 2</a>
</div>

<script>
    var doc_name = "extremeMashupHTMLMime";
    var proto = window.location.protocol;
    var host = window.location.host;
    var path = window.location.pathname;
    if (path.indexOf("/") != 0) path = "/" + path;
         // IE 9 bug text/html MIME only
    var address = host + path; // Can be replaced by an absolute URL

    var url = proto + "//" + address +
         "?itsnat_action=attach_server&itsnat_doc_name=" + doc_name +
         "&itsnat_method=form&itsnat_timeout=600000&timestamp=" +
         new Date().getTime();

    document.write("<script src='" + url + "'><\/script>");
</script>

</body>
</html>
```

As you can see is basically the same example shown in chapter "A CORE BASED WEB APPLICATION", in this case <div> elements have been replaced by <a>, this replacement is

[67] Snap is popular mashup service of this kind http://www.snap.com/

not mandatory is just useful to show a limitation of extreme mashups with MSIE v6. Another difference is the final `<script>` element, this element is the key of ItsNat extreme mashups.

Because the page is going to be sent to the server and the template to build an `ItsNatDocument` object, any ItsNat specific attribute or element is valid:

```
<div itsnat:nocache="true" xmlns:itsnat="http://itsnat.org/itsnat">
```

In this case the `itsnat:`**`nocache`**`=”true”` attribute prevents subtree caching in server like a normal file template.

The last `<script>` element attaches the browser’s page to the server, then the server takes control of the client page as it was loaded from ItsNat in the beginning with this page as the ItsNat template file in server. To achieve this attachment a special request to the host and servlet running ItsNat is sent through this sentence:

```
document.write("<script src='" + url + "'><\/script>");
```

The URL must point to an ItsNat servlet including these parameters:

- `itsnat_action=attach_server` : this parameter informs ItsNat this request is an “attach server” request, that is, the client page must be sent to the server creating a document in server with some behavior associated.

- `itsnat_doc_name=`*`template_name`* : the name of the template configuration in server, this template configuration must be registered in server calling `ItsNatServlet.registerItsNatDocumentTemplateAttachedServer(…)` in servlet configuration (`init` method). This parameter is mandatory.

- `itsnat_method=form | script` : specifies the technique used to send the page markup to the server. Two options are possible, `form` or `script`, the former sends the page using auxiliary and temporal `<form>` and `<iframe>` elements, the later sends the page as part of the URL of several consecutive auxiliary and temporal `<script src=url>` elements. If not specified this parameter, `form` is the default and highly recommended (*script* mode takes much more time).

- `itsnat_timeout=`*`timeout`* : the max time in milliseconds the server will wait to receive the client markup code. If the page is not sent in this time frame (the page is too big and/or network is very slow and/or the timeout is too small) the server will throw an exception and no attachment is performed. Only is useful in `form` mode (in `script` mode is ignored) and defaults to 10 minutes (600000) if no specified.

The remaining parameter:

```
timestamp=" + new Date().getTime()
```

is not mandatory for ItsNat but strongly needed to avoid URL caching, this parameter defines unique URLs. Any other technique to provide unique URLs is valid.

This final `<script>` element MUST BE THE LATEST NODE (only empty text nodes are allowed in the end), otherwise server attachment can fail.

These lines are interesting:

```
var host = window.location.host;
var path = window.location.pathname;
...
var address = host + path;
```

In this case the host and port of the ItsNat extreme mashup service is the same as the host serving the client page, this is not mandatory and you could specify a different/fixed host and port for the ItsNat service. If the host and port are different the method of communication for events must be `CommMode.SCRIPT` or `CommMode.SCRIPT_HOLD` (`SCRIPT_HOLD` is recommended) because XHR modes require the same host and port (this is a browser restriction for security), if the host and port are the same `CommMode.XHR_*` modes are preferred, the communication mode will be specified in the "template configuration" in server.

In server we must define a "template configuration" (or "attached server template") in the `init` method of the ItsNat servlet, with the template name specified in client.

```
ItsNatServlet servlet = ...;
ItsNatDocumentTemplate docTemplate =
   servlet.registerItsNatDocumentTemplateAttachedServer(
         "extremeMashupHTMLMime","text/html");
docTemplate.addItsNatServletRequestListener(
         new CoreExampleLoadListener());
docTemplate.setCommMode(CommMode.SCRIPT_HOLD);
```

There are two significative differences between an "attached server template" and a "normal" template:

1) The specific method `ItsNatServlet.`**`registerItsNatDocumentTemplateAttachedServer`**`(…)` is used.

2) No template file path is provided. This is obvious because "the template file" is in client and dynamically loaded.

Another difference (in this case is not specific of "attached server templates") is the communication mode for events, `CommMode.`**`SCRIPT_HOLD`**, this is the typical mode when your attached server template can be used from any host/port (as a real mashup provider), furthermore `CommMode.`**`SCRIPT*`** modes are almost only useful for extreme mashups because in normal ItsNat applications AJAX (`CommMode.`**`XHR*`** modes) is preferred.

In this example the specified MIME is `text/html`, this MIME MUST BE THE SAME as the MIME of the client page.

Another possible use of extreme mashups is to extend with AJAX an old application running in the same host/port (or a different port but some kind of internal/transparent forwarding to Java servlets or JSPs is possible); in this case the communication mode can one of `CommMode.`**`XHR*`** modes (`CommMode.`**`XHR_ASYNC_HOLD`** recommended).

Back to the client page:

```
<a href="javascript:;" onclick="return false;"
     id="clickableId1">Clickable Elem 1</a>
<br /><br />
<a href="javascript:;" onclick="return false;"
     id="clickableId2">Clickable Elem 2</a>
```

Why these `onclick` attributes?

In a normal AJAX based application (CommMode.**XHR*** modes) these onclick attributes are not needed, with CommMode.**SCRIPT*** modes and Microsoft Internet Explorer v6 are needed otherwise the page is blocked when clicking some link. This problem does not apply to MSIE v7 and upper and any other browser, but it applies again to MSIE v9 (preview 3 tested).

6.25.1 Pages served with XHTML MIME, SVG and XUL

Extreme mashups can work in pages served with MIME application/xhtml+xml (XHTML MIME), image/svg+xml (SVG MIME) or application/vnd.mozilla.xul+xml (XUL MIME).

In this case the sentence:

```
document.write("<script src='" + url + "'><\/script>");
```

is NOT valid (text/html MIME specific).

Use the following final script or something similar (application/xhtml+xml example) :

```
<script>
    var doc_name = "extremeMashupXHTMLMime";

    var proto = window.location.protocol;
    var host = window.location.host;
    var path = window.location.pathname;
    var address = host + path; // Can be replaced by an absolute URL
    var url = proto + "//" + address +
        "?itsnat_action=attach_server&itsnat_doc_name=" + doc_name +
"&itsnat_method=form&itsnat_timeout=600000&timestamp=" + new Date().getTime();

    var body = document.body;
    if (!body) body = document.getElementsByTagName("body")[0];
    var script = document.createElement("script");
    script.src = url;
    body.appendChild(script);
</script>
```

MSIE v6 does not recognize application/xhtml+xml MIME, therefore in theory there is no need of onclick="return false" in links, but MSIE v9 (third preview) has this problem and supports XHTML MIME. MSIE v9 and XHTML MIME does not have the bug "/ missing" in window.location.pathname.

In SVG or XUL pages replace the body variable with document.**documentElement**.

In server you must register the "attached server template" with something like:

```
   ItsNatServlet servlet = ...;
   ItsNatDocumentTemplate docTemplate =
      servlet.registerItsNatDocumentTemplateAttachedServer(
            "extremeMashupXHTMLMime","application/xhtml+xml");
   docTemplate.addItsNatServletRequestListener(
            new CoreExampleLoadListener());
   docTemplate.setCommMode(CommMode.SCRIPT_HOLD);
```

Again the MIME in "attached server template" must be the same as the client page.

There is one important limitation in extreme mashup and non-HTML MIME (not `text/html`) pages: the `DOMContentLoaded` and `load` events are never sent to the server in WebKit based browsers (Chrome, Safari etc)[68], this is not the case of FireFox and Opera. Use a custom made final "continue event" instead of these event types to detect when the page "is loaded" (server point of view) for these browsers.

6.25.2 Security concerns

Extreme mashups is a very powerful feature because with a simple script in any client page you can fully control these client pages regardless the technology used to generate these pages and from a different host. Because a document is created in ItsNat server you can add remote view/control (the original client page could be updated with timers or Comet), server-sent events (to remotely send events to the original client page simulating user actions), add mutation events (to synchronize in ItsNat server any change in client page not driven by ItsNat) and in general any kind of services provided by ItsNat.

The markup of original client pages in theory could include special tags `<itsnat:include name="…">`, this is not a serious security concern because these includes are resolved in load time converted to DOM and this included DOM do not generate JavaScript to be sent to the client and they are resolved before user Java code is executed. In spite of this, includes are not allowed in attached server mode (an exception is thrown) mainly because if allowed the new included DOM is not in the original client page and because there is no generated JavaScript to sync this new markup in client, the server DOM is not in sync with client DOM.

6.25.3 Session Replication Capable and Google App Engine support

If `ItsNatServletContext.setSessionReplicationCapable(boolean)` mode is set to true (for instance for GAE), the `form` mode of extreme mashups works in a different way compatible with Google App Engine. As consequence the load event is not dispatched to server in browsers like Internet Explorer, Opera or WebKit browsers (it works as usual in FireFox).

Avoid as possible relying on the load event in extreme mashups.

6.26 REMOTE TEMPLATES

In any typical ItsNat application, markup templates (for documents and fragments) are based on local files specified by the file path when the template is registered, notwithstanding there are more options like remote files specified by URLs or user defined sources implementing `TemplateSource`.

When a remote file is specified by its URL (as a String) ItsNat loads the remote file the first time the template is needed (to build one document or fragment) and there is no further reloading of the template (no more remote connections).

Because previous approaches may be not flexible enough (reloading or custom configuration of remote connection may be needed), ItsNat provides a mechanism to specify custom template sources implementing the interface `TemplateSource`. By this interface developers can return

[68] This could be fixed in next versions firing false `DOMContentLoaded` and load events to the server.

the required template to be used in a per request basis. This feature opens ItsNat to build sophisticated front ends/proxies/filters of any web application/site.

The following example shows how we can use ItsNat as a front end of Google Search taking on demand the pages returned by Google as template sources, modifying them on the fly including events transported with AJAX.

We are going to load as templates the initial page of Google Search and the first page of search results. Because the main page of Google Search is ever the same (not exactly true) we will fetch this page specifying the URL as a string; only one remote connection will be executed, all of documents based on this template are going to be initially the same (because the markup of the template is the same). The page with search results is different because this page is clearly based on the entered keywords by the end user, in this case default behavior of templates loaded by URLs is not valid so a `TemplateSource` implementation will be used.

The following code is the template registration, as ever, into the servlet's `init` method.

```
ItsNatHttpServlet itsNatServer = ...;
ItsNatDocumentTemplate docTemplate;

docTemplate = itsNatServer.registerItsNatDocumentTemplate(
        "remoteTemplateExample","text/html","http://www.google.com");
docTemplate.addItsNatServletRequestListener(
                new RemoteTemplateDocLoadListener());
docTemplate.setOnLoadCacheStaticNodes(false);

docTemplate = itsNatServer.registerItsNatDocumentTemplate(
        "remoteTemplateExampleResult","text/html",
        new GoogleResultTemplateSource());
docTemplate.addItsNatServletRequestListener(
        new RemoteTemplateResultDocLoadListener());
docTemplate.setOnLoadCacheStaticNodes(false);
docTemplate.setEventsEnabled(false);
```

We must disable node caching because obviously Google has no clue about ItsNat and so we are not going to find `nocache="true"` attributes and node caching is enabled by default, this is the reason of the calls:

```
docTemplate.setOnLoadCacheStaticNodes(false);
```

The Google Search's initial page will be loaded and processed on demand by `RemoteTemplateDocLoadListener`, this is the code:

```
public class RemoteTemplateDocLoadListener
                implements ItsNatServletRequestListener
{
    public RemoteTemplateDocLoadListener()
    {
    }

    public void processRequest(ItsNatServletRequest request,
                               ItsNatServletResponse response)
    {
        new RemoteTemplateDocument(request.getItsNatDocument());
    }
}
```

```
public class RemoteTemplateDocument implements EventListener
{
    protected ItsNatDocument itsNatDoc;
    protected HTMLAnchorElement link;
    protected HTMLInputElement inputSearch;

    public RemoteTemplateDocument(ItsNatDocument itsNatDoc)
    {
        this.itsNatDoc = itsNatDoc;
        load();
    }

    public void load()
    {
        Document doc = itsNatDoc.getDocument();

        NodeList list = doc.getElementsByTagName("img");
        int len = list.getLength();
        for(int i = 0; i < len; i++)
        {
            HTMLImageElement img = (HTMLImageElement)list.item(i);
            img.setSrc("http://www.google.com/" + img.getSrc());
        }

        Element form = (Element)doc.getElementsByTagName("form").item(0);
        form.setAttribute("action","");
        String onsubmit = "var elem = document.getElementById('q'); " +
             "if (elem.value == '') { alert('Empty? Sure?'); return false; }" +
             "else { alert('No, you are not going to the real Google :)');
return true; }";
        form.setAttribute("onsubmit",onsubmit);

        list = doc.getElementsByTagName("input");
        len = list.getLength();
        for(int i = 0; i < len; i++)
        {
            HTMLInputElement input = (HTMLInputElement)list.item(i);
            String name = input.getName();
            if ("q".equals(name))
            {
                input.setAttribute("id","q");
                inputSearch = input;
            }
            else
                input.removeAttribute("name"); // makes it useless
        }

        HTMLInputElement template =
               (HTMLInputElement)doc.createElement("input");
        template.setAttribute("type","hidden");
        template.setName("itsnat_doc_name");
        template.setValue("remoteTemplateExampleResult");
        form.appendChild(template);

        list = doc.getElementsByTagName("a");
        len = list.getLength();
        for(int i = 0; i < len; i++)
        {
            HTMLAnchorElement elem = (HTMLAnchorElement)list.item(i);
            elem.setHref(
               "javascript:alert('Hey!! This is not interesting! :)');");
        }
        link = (HTMLAnchorElement)doc.createElement("a");
        link.setHref("javascript:;");
        link.setAttribute("style","font-size:25px;");
        link.appendChild(doc.createTextNode(
               "ItsNat says Hello to Google... CLICK ME!"));
        form.getParentNode().insertBefore(link, form);

        ((EventTarget)link).addEventListener("click", this,false);
    }

    public void handleEvent(Event evt)
    {
        Document doc = itsNatDoc.getDocument();

        inputSearch.setValue("ItsNat");

        ((EventTarget)link).removeEventListener("click", this,false);
        Element elem = doc.createElement("div");
        elem.setAttribute("style","font-size:25px;");
        link.getParentNode().insertBefore(elem, link);
        link.getParentNode().removeChild(link);
        ((Text)link.getFirstChild()).setData("Have a Good Search!!");
        elem.appendChild(link.getFirstChild());
    }
}
```

In brief the previous code modifies the main page of Google disabling links and modifying the search form to target the second ItsNat page (the result page) instead of sending this form with the search keyword to Google, this is the reason of adding a new `<input type=hidden>` element with `name` `"itsnat_doc_name"` and as `value` the name of the second ItsNat template. To show how we can add (ItsNat) events to this page, a new link is added, when this link is clicked and AJAX event is sent to the server then the search text box is filled with the `"ItsNat"` keyword.

The second ItsNat page should receive the submitted form containing the keyword to search into the parameter `"q"`, the Google's result page of this query should be the initial DOM tree of the second page, therefore the template of the second page will be based on Google's result page, this is the reason of `GoogleResultTemplateSource` class:

```
public class GoogleResultTemplateSource implements TemplateSource
{
    public boolean isMustReload(ItsNatServletRequest request,
                                ItsNatServletResponse response)
    {
        return true;
    }

    public InputStream getInputStream(ItsNatServletRequest request,
                                      ItsNatServletResponse response)
    {
        String query = request.getServletRequest().getParameter("q");

        try
        {
            query = java.net.URLEncoder.encode(query,"UTF-8");
            URL url = new URL("http://www.google.com/search?q=" + query);
            URLConnection conn = url.openConnection();
            HttpServletRequest httpRequest =
                        (HttpServletRequest)request.getServletRequest();
            String userAgent = httpRequest.getHeader("User-Agent");
            conn.setRequestProperty("User-Agent", userAgent);
            return conn.getInputStream();
        }
        catch(Exception ex) { throw new RuntimeException(ex); }
    }
}
```

The `TemplateSource` interface represents a free-form user defined markup source to be registered as an ItsNat template.

When a document or fragment is going to be loaded based on the template which this template source was registered, the first time the method `TemplateSource.`**`getInputStream`**`(ItsNatServletRequest,ItsNatServletResponse)` is called, the request and response parameters are the original request and response objects of the document or fragment loading request.

If a document or fragment was already loaded, then any new load request asks first if the template must be reloaded calling `TemplateSource.`**`isMustReload`**`(ItsNatServletRequest,ItsNatServletResponse),` if this method returns true the method `TemplateSource.`**`getInputStream`**`(ItsNatServletRequest,ItsNatServletResponse)` is called after, if false the previous loaded template is also used for this load request.

In a very special and highly improbable case the method `TemplateSource.`**`isMustReload`**`(ItsNatServletRequest,ItsNatServletResponse)` is called twice with the same request and response object, because this may happen this function must be idempotent (same parameters imply same result).

In summary this interface could be used to ever load a new template per document load request, back to Google example, this behavior is required because different users execute different queries, this is why `isMustReload(…)` ever returns true.

The Google's result page is the template of the concrete document to be processed by the following classes:

```
public class RemoteTemplateResultDocLoadListener
                implements ItsNatServletRequestListener
{
    public RemoteTemplateResultDocLoadListener()
    {
    }

    public void processRequest(ItsNatServletRequest request,
                               ItsNatServletResponse response)
    {
        new RemoteTemplateResultDocument(request.getItsNatDocument());
    }
}

public class RemoteTemplateResultDocument
{
    protected ItsNatDocument itsNatDoc;

    public RemoteTemplateResultDocument(ItsNatDocument itsNatDoc)
    {
        this.itsNatDoc = itsNatDoc;
        load();
    }

    public void load()
    {
        Document doc = itsNatDoc.getDocument();

        HTMLInputElement inputSearch = null;
        NodeList list = doc.getElementsByTagName("input");
        int len = list.getLength();
        for(int i = 0; i < len; i++)
        {
            HTMLInputElement input = (HTMLInputElement)list.item(i);
            String type = input.getType();
            if ("text".equals(type)) { inputSearch = input; break; }
        }

        Element div = doc.createElement("div");
        div.setAttribute("style","font-size:25px;");
        div.appendChild(doc.createTextNode(
               "End of demo. Yes I know, is a clone far of perfect :)"));
        inputSearch.getParentNode().insertBefore(div, inputSearch);

        list = doc.getElementsByTagName("a");
        len = list.getLength();
        for(int i = 0; i < len; i++)
        {
            HTMLAnchorElement elem = (HTMLAnchorElement)list.item(i);
            String href = elem.getHref();
            if (!href.startsWith("http:"))
                elem.setHref("http://www.google.com" + href);
        }

        list = doc.getElementsByTagName("img");
        len = list.getLength();
        for(int i = 0; i < len; i++)
        {
            HTMLImageElement elem = (HTMLImageElement)list.item(i);
            String src = elem.getSrc();
            if (!src.startsWith("http:"))
                elem.setSrc("http://www.google.com" + src);
        }
    }
}
```

The result page shows the first result page of the query and shows a message to show this page is not the original as returned by Google.

The Feature Showcase includes this example.

6.27 JAVASCRIPT GENERATION UTILITIES

ItsNat provides some utilities to generate JavaScript code mainly to locate DOM nodes in client from DOM nodes in server code, this JavaScript must be sent to the client. 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`**`()` or `ClientDocument.`**`getScriptUtil`**`()`.

If the `ScriptUtil` object was got calling the method `ItsNatDocument.`**`getScriptUtil`**`()` the generated JavaScript code must be sent to the client calling `ItsNatDocument.`**`addCodeToSend`**`(Object)`, this JavaScript code will be received by all clients (owner and attached) of this document.

If the `ScriptUtil` object was got calling the method `ClientDocument.`**`getScriptUtil`**`()` the generated JavaScript code must be sent to the client calling `ClientDocument.`**`addCodeToSend`**`(Object)`, this JavaScript code will be received by the specified client (use ever the same client for generating and sending code).

When a DOM node is used a parameter this node must be part of the document tree (server and client) otherwise an exception is thrown, and the generated code must be sent as soon as possible to the client/s calling the appropriated method as explained before.

ItsNat generates JavaScript code referencing the specified node in client, when the `ScriptUtil` object was got calling `ItsNatDocument.`**`getScriptUtil`**`()` this reference is valid for all clients of the document (in spite of internal node caching ItsNat ensures the same internal caching id is used for all clients or no caching is used).

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' \n\t 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 an example of the generated JavaScript code sent to the client[69]:

[69] This code is implementation detail and may change

```
itsNatDoc.getNode("cn_33").value
     = "A Java String 'transported' \n\t as a \"JavaScript\" string";
itsNatDoc.getNode("cn_33").value
     = "A Java String 'transported' \n\t as a \"JavaScript\" string";
itsNatDoc.getNode("cn_33").select();
```

6.28 EVENT MONITORS IN CLIENT

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 or SCRIPT modes 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 because 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:

```
<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("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 the AJAX/SCRIPT 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);
```

6.29 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 Batik DOM implementation and extended in ItsNat, this method distinguishes if the event parameter is internal (defined by Batik) or remote (defined by ItsNat).

An event is remote when:

1. It was created calling `DocumentEvent.`**`createEvent`**`(String)` with `ItsNatNode.`**`isInternalMode`**`()` returning false (otherwise the event is "internal")

2. Was created calling `ItsNatDocument.`**`createEvent`**`(String)`.

Otherwise the event created is internal, internal events are almost only useful for monitoring mutation changes on the server DOM tree.

If the event is remote the call to `EventTarget.`**`dispatchEvent`**`(Event)` is forwarded to `ItsNatDocument.`**`dispatchEvent`**`(EventTarget,Event)`.

There are two scenarios:

1) Events sent to the browser simulating user actions

2) Events directly dispatched to the server DOM tree

6.29.1 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)
```

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[70]. 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[71]). 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 missing piece is how we can create DOM events, in W3C DOM, events are created with `DocumentEvent.createEvent(String type)`, the behavior of this method in ItsNat context was explained before. The `Document` object of Batik DOM implementation implements `DocumentEvent`.

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;
```

[70] 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.

[71] Current implementation detects this scenario to avoid dead lock errors.

```
    <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();
        if (currTarget == linkElem)
        {
            ItsNatEvent itsNatEvt = (ItsNatEvent)evt;
            final ItsNatDocument itsNatDoc = itsNatEvt.getItsNatDocument();
            Document doc = itsNatDoc.getDocument();
            final ClientDocument clientDoc = itsNatDoc.getClientDocumentOwner();

            Runnable dispCode = new Runnable()
            {
              public void run()
              {
                for(int i = 0; i < buttonElems.length; i++)
                {
                  Element buttonElem;
                  MouseEvent mouseEvt;
                  synchronized(itsNatDoc)
                  {
                    Document doc = itsNatDoc.getDocument();
                    AbstractView view =
                           (DocumentView)doc).getDefaultView();
                    mouseEvt=(MouseEvent)
                           ((DocumentEvent)doc).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];
                  }

                  ((EventTarget)buttonElem).dispatchEvent(mouseEvt);
                  // Alternatives:
              // itsNatDoc.dispatchEvent((EventTarget)buttonElem,mouseEvt);
              // clientDoc.dispatchEvent((EventTarget)buttonElem,mouseEvt);
                }


                ItsNatUserEvent userEvt;
                synchronized(itsNatDoc)
                {
                  Document doc = itsNatDoc.getDocument();
                  userEvt = (ItsNatUserEvent)
                      ((DocumentEvent)doc).createEvent("itsnat:UserEvents");
                  userEvt.initEvent("itsnat:user:myEvent",false,false);
                }

                ((EventTarget)userButton).dispatchEvent(userEvt);
              }
            };
            clientDoc.startEventDispatcherThread(dispCode);
        }
        else
        {
            System.out.println("Clicked: " +
                            ((Element)currTarget).getAttribute("value"));
        }
    }
};

((EventTarget)linkElem).addEventListener("click",listener,false);

for(int i = 0; i < buttonElems.length; i++)
    ((EventTarget)buttonElems[i]).addEventListener("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 = (ItsNatUserEvent)
                ((DocumentEvent)doc).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.

6.29.2 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 communication modes, timeouts, waits etc involved).

The "Feature Showcase" again includes a basic example of server only functional testing and a complete example using components.

6.29.3 More about testing

The main use of events fired by the server is functional testing. ItsNat is strongly based on AJAX (and SCRIPT events), AJAX applications may be hard to test because AJAX is usually used asynchronously and SCRIPT requests are ever asynchronous. In asynchronous mores the browser event may be processed (`dispatchEvent` and `fireEvent` return) but the response is received later. ItsNat provides the "hold" modes to minimize the indeterminist behavior, this is sufficient for 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[72]:

1) Using AJAX 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.

6.29.4 Bookmarking and search engines

Events fired by the server can be used for bookmarking an AJAX based application (usually Single Page Interface), because user actions can be simulated to transport the application to

[72] http://mguillem.wordpress.com/2007/07/24/htmlunit-re-synchronize-ajax-calls-for-simple-deterministic-test-automation/

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.

6.30 FAST AND SLOW LOADING MODES

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 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 `ItsNatDocumentTemplate.`**`setFastMode`**`(boolean)` when registering the template.

6.30.1 Fast mode

When loading in fast mode no JavaScript code is automatically generated when the DOM server tree is modified. When the loading process finishes the final DOM server tree is serialized and sent to the client as markup.

This approach is the fastest 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) { }
    };

    ((EventTarget)tableElem).addEventListener("click",listener,false);

    ((EventTarget)tableElem).removeEventListener("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).

6.30.2 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.

6.30.3 How to select the most appropriated mode

Fast load mode is the preferred mode and used according is rare to get into problems, if fast model fails in some DOM tree tricky modification try to use a "load" listener or listener bindings to do the problematic tree modification, anyway if everything fails use the slow mode.

6.30.4 Single Page Interface, Fast mode and SEO Compatibility

Fast load mode is the key to be able to make web sites/applications Single Page Interface and in the same time SEO compatible

If your document template is configured in fast-load mode (the default mode), when the initial page is being loaded, any DOM change performed in server is not sent as JavaScript in this load phase, after user code in server is executed final DOM state is rendered to generate the initial HTML page being sent to the client. Hence the same user code manipulating DOM generates plain HTML in load phase (in fast mode) or JavaScript if this code is executed when a client event is processed in server. By this way any client document state typically obtained after processing several client events in a Single Page Interface, is able to be reconstructed in server in page load time.

In fast-load mode we can use a parameter in the query part of the URL or in your preferred pretty URL to specify what "page", or "fundamental state" following the Single Page Interface

Manifesto terminology, is going to be loaded, executing the same code being used to insert the required HTML fragment into the content zone when processing a client event. The final DOM tree is the tree being serialized as HTML and loaded by the client.

These are the basics, more things are required to provide in SPI the same capabilities as page based web sites, such as dual (AJAX/normal) links, URL rewriting for bookmarking, support of page visit counters (in this case "state visit counters"), JavaScript disabled and support of Back/Forward buttons (history navigation). The ItsNat web site includes a tutorial on how coding a simple Single Page Interface SEO compatible web site.

6.31 GLOBAL LOAD REQUEST 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.getItsNatDocumentTemplate().getName();
            System.out.println("Loading " + docName);
        }
    }
};

itsNatServlet.addItsNatServletRequestListener(listener);
```

6.31.1 Detection of template not found with the specified document name

If the query string (URL part following “?”) contains the standard `itsnat_doc_name` parameter but no template was registered with the specified name, 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 ItsNat document is loaded a call to `ItsNatServletRequest.`**`getItsNatDocument`**`()` returns null. The global request listeners registered in the ItsNat servlet object are dispatched too as an opportunity to return a specific default page or to redirect to a concrete ItsNat page (see pretty URLs section further to know how to do is).

The following example code gratefully forwards the request to a specific error page:

```
public void processRequest(ItsNatServletRequest request,
                ItsNatServletResponse response)
    {
        ItsNatDocument itsNatDoc = request.getItsNatDocument();
        if (itsNatDoc != null)
        {
            ...
        }
        else
        {
            ServletRequest servReq = request.getServletRequest();
            String docName = servReq.getParameter("itsnat_doc_name");
            if (docName == null) docName =
                         (String)servReq.getAttribute("itsnat_doc_name");
            if (docName != null)
            {
              ItsNatServlet servlet = response.getItsNatServlet();
              ServletRequest servRequest =
                         event.getItsNatServletRequest().getServletRequest();
              Map<String,String[]> newParams =
                    new HashMap<String,String[]> (servRequest.getParameterMap());
              newParams.put("itsnat_doc_name",
                   new String[]{"feashow.docNotFound"});
              servRequest = servlet.createServletRequest(servRequest,
                         newParams);
              servlet.processRequest(servRequest,response.getServletResponse());
            }
        }
    }
```

6.31.2 Not standard page loading (custom requests)

If the query string does not contain an `itsnat_doc_name` parameter (nor `itsnat_action`) and is not an event, ItsNat has no clue about this request. This case is very similar to the previous case (template not found) and only the servlet level `ItsNatServletRequestListener` listeners are dispatched.

One very interesting case is to process pretty URLs.

6.31.3 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:

```
    public void processRequest(ItsNatServletRequest request,
        ItsNatServletResponse response)
    {
        ItsNatDocument itsNatDoc = request.getItsNatDocument();
        if (itsNatDoc != null)
        {
            ...
        }
        else
        {
            ServletRequest servReq = request.getServletRequest();
            String docName = servReq.getParameter("itsnat_doc_name");
            if (docName == null) docName =
                        (String)servReq.getAttribute("itsnat_doc_name");
            if (docName != null)
            {
                   ...
            }
            else
            {
                // Pretty URL case
                HttpServletRequest servRequest =
                         (HttpServletRequest)request.getServletRequest();
                String pathInfo = servRequest.getPathInfo();
                if (pathInfo == null)
                    throw new RuntimeException("Unexpected URL");

                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 the name format name.name (used in our examples, this format is not mandatory), sets itsnat_doc_name as a request attribute and forwards the request/response pair again to the ItsNat servlet again. 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.

6.32 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/SCRIPT 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/SCRIPT events can be sent asynchronously by 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.

6.33 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[73]. ItsNat simulates the referrer feature with a server centric approach.

6.33.1 Referrer "pull"

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();
    ...
```

[73] The referrer property can be obtained with the call request.getHeader("referer") where request is the HttpServletRequest.

```
    }
```

By default the referrer feature is disabled, it can be enabled globally with `ItsNatServletConfig.setReferrerEnabled(boolean)` or per-template basis with `ItsNatDocumentTemplate.setReferrerEnabled(boolean)`; if referrer is disabled the document can not be "referred". The referrer feature only works with events 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.

6.33.2 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 `ItsNatDocumentTemplate.`**`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".

6.34 DISABLED EVENTS MODE

Events (transported with AJAX or SCRIPT elements) are enabled by default, but can be disabled globally with `ItsNatServletConfig.`**`setEventsEnabled`**`(boolean)` or in a per-template basis with `ItsNatDocumentTemplate.`**`setEventsEnabled`**`(boolean)`. If events are disabled any event related feature or method is disabled, usually they do nothing, for instance `ItsNatDocument.`**`addEventListener`** methods may be called but nothing is done and no JavaScript is sent to client. Anyway methods like `ItsNatDocument.`**`addCodeToSend`**`(Object)`, DOM manipulation, fragments, DOM utilities keep working as usual 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 events, 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 disabled events application.

6.35 DISABLED JAVASCRIPT MODE

A step further, JavaScript can be fully disabled, this is the lowest downgraded mode of ItsNat. This mode is basically the same as disabled events 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 `ItsNatDocumentTemplate.`**`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 `ItsNatDocumentTemplate.`**`setFastLoadMode`**`(boolean)`). Again the "Feature Showcase" includes an example very similar to the disabled events example but with no client JavaScript code.

6.36 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.

6.37 SVG, XUL AND NON-XHTML NAMESPACES

6.37.1 Pure SVG documents

FireFox 1.5+, WebKit browsers (Safari 3, Chrome 1.0, iPhone 2.1), Opera, Internet Explorer v9 and previous versions with Adobe SVG Viewer, Savarese Ssrc plugin, and any browser with Java applet support and the special Batik applet provided with ItsNat, all of them support pure SVG documents including JavaScript, events and AJAX. 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`.

As you know ItsNat uses Batik in server as the DOM provider, in spite of Batik provides SVG, W3C DOM SVG[74] API of Batik *in server* is not included in ItsNat. This is not a serious limitation, normal DOM Level 2 can be used to manage SVG elements. Batik DOM supports the "id" attribute in any namespace supported by ItsNat as an identity attribute, SVG elements are searchable using `Document.`**`getElementById`**`(String)`[75].

In Internet Explorer v6,7,8 we can use Adobe SVG Viewer v3, Savarese Ssrc plugins or the ItsNat Batik applet to render SVG (of course this applet can be used with other browsers). If some SVG plugin is installed Internet Explorer delegates to the plugin to render pages served with MIME `image/svg+xml`.

There are several modes to view SVG markup in Internet Explorer v6,7,8 (v9 already supports SVG natively):

1) A complete SVG page:

   If we use a link pointing to an ItsNat URL to load a SVG based template, Internet Explorer first try to load the page when it detects a MIME type of `image/svg+xml` a second request is done again through the SVG plugin installed. Fortunately ItsNat detects the first request and returns a dummy SVG page with MIME `image/svg+xml`, your load listener is not executed.

[74] http://www.w3.org/TR/SVG/svgdom.html

[75] `ItsNatDOMUtil.getElementById(String,Node)` is designed to search elements with duplicated ids.

We cannot view a SVG page typing a URL into the browser address box, in this case Internet Explorer does not delegate to the installed SVG plugin, use instead `window.location`, a link or similar "launcher" in a X/HTML page.

2) A SVG page embedded in a X/HTML page:

   In this case the page is loaded through an `<iframe>`, `<object>`, `<embed>`, `<applet>` element. This is explained later.

3) SVG inline in a X/HTML page:

   In this case the SVG markup is rendered following a trick of ASV. Explained later.

6.37.2 SVG embedded in X/HTML using iframe, object and embed

In any browser with native support you can embed SVG pages with `<iframe>`, `<object>`, `<embed>` with no problem (we will see applets later). Any SVG page must be served with MIME type `image/svg+xml`.

Examples:

```
<object data="?itsnat_doc_name=..." type="image/svg+xml" />

<embed src="?itsnat_doc_name=..." type="image/svg+xml" />

<iframe src="?itsnat_doc_name=..." />
```

In Internet Explorer with Adobe SVG Viewer (ASV) installed we can use `<iframe>`, `<object>` and `<embed>` with some quirks:

1) Embedded with `<object>`

   In Internet Explorer `<object>` is in general problematic:

   a. The SVG page loaded by `<object>` is requested twice by the plugin.

   b. The URL must be specified with a `src` attribute *and* a parameter like `<param name="src" value="URL">`. Different contexts (load time, insertion with JavaScript with DOM or with `innerHTML`) use different URL declarations (`param` or attribute). Use also the data attribute with the same URL if you want to support browsers with native SVG and to be "auto-binding" compliant (see IFRAME/OBJECT/EMBED/APPLET AUTO-BINDING).

   c. Because URL definition is complex, we should avoid any change of the URL in a `<object>` element already inserted in the tree. In this case remove first, change the `src` attribute and parameter (and `data` attribute if auto-binding is required) and reinsert again.

   In spite of these problems, ASV embedded in `<object>` works including parent/child communication (`getSVGDocument()` is used).

2) Embedded with `<embed>`

This is the preferred mode of ASV according to Adobe. The SVG page is requested once and only the `src` attribute is needed. In spite of this element is not W3C standard (a legacy of the old Netscape Navigator) is supported by any browser with extensibility. The method `getSVGDocument()` is used for parent/child communication in client (auto-binding is possible with `<embed>`).

3) Embedded with `<iframe>`

In spite of this technique may seem problematic, is alongside `<embed>` the best technique to use ASV and being compatible with native SVG browsers:

a. The SVG page is requested twice. This is not a serious problem because in this case the first request is done by Internet Explorer, ItsNat detects and ignores this request and responses a dummy SVG page with SVG MIME, Internet Explorer receives this dummy SVG page and commands the SVG plugin to load the page. The second request is performed by the SVG plugin and processed as usual. In summary, the user code in server processes a single load request.

b. When the SVG page is loaded by an `<iframe>` the method `getSVGDocument()` is not present. This is not a problem, ItsNat automatically binds this method to the `<iframe>` element when the SVG is loaded, if the child document is "auto-binding" compliant (see IFRAME/OBJECT/EMBED/APPLET AUTO-BINDING).

Adobe SVG Viewer has some problems with scripting when `<object>` or `<embed>` is used, in load time, in some way, the window object of the parent document is used in the SVG document being loaded. Fortunately this is not a problem for ItsNat. An SVG document loaded by `<iframe>` has not this problem.

Internet Explorer can also use Savarese Ssrc SVG plugin, this plugin is basically a FireFox registered in Internet Explorer to process SVG and XUL documents. Unlike ASV, Ssrc integration in Internet Explorer is not so tight, parent and child communication in client is problematic, in spite of this Ssrc can be used with ItsNat including auto-binding. The Feature Showcase includes an interesting example of SVG loaded by MSIE and Ssrc with `<object>` or `<embed>` and how this example is compatible with browsers with native support.

6.37.3 SVG embedded inline in X/HTML and browsers with native support

ItsNat supports SVG elements embedded inline in XHTML[76] in browsers with native SVG support including events and components as any other XHTML element.

In theory the XHTML page containing SVG markup inline must be served with MIME `application/xhtml+xml`, this is not fully true. This web page[77] shows how we can render inline SVG in an X/HTML page served with `text/html`, basically the SVG markup present in load time is appropriately reinserted with JavaScript. ItsNat automatically detects any SVG (or any code with non-X/HTML namespaces) markup in the XHTML page loading with `text/html` and appropriately reinserts it again. ItsNat uses a similar technique to the web site cited before but the ItsNat approach is better because does not require any knowledge of the namespace involved, so MathML (recognized by Gecko browsers) or any other namespace recognized by

[76] XHTML supports elements with other namespaces, but only nodes with known namespaces are rendered and receive events, for instance Gecko browsers recognize MathML (also valid in ItsNat).

[77] http://intertwingly.net/blog/2006/12/05/HOWTO-Embed-MathML-and-SVG-into-HTML4

the browser can be embedded too. This is applied on load time to loaded markup, any other markup inserted by ItsNat usually as response of events is inserted with correct namespace.

Servicing X/HTML pages with `text/html` is very important to be compatible with Internet Explorer, because MSIE (including v8) does not support `application/xhtml+xml` and fortunately MSIE has some support of SVG inline with some help.

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).

6.37.4 SVG embedded inline in X/HTML, Internet Explorer and Adobe SVG

Internet Explorer can also render SVG inline with Adobe SVG Viewer (ASV) v3 plugin installed by using a trick of ASV explained in this web page[78].

Maybe you have heard this use mode of ASV is very limited, for instance, if you look for information in Internet you come to the following conclusion: SVG elements cannot be dynamically added and removed with JavaScript, almost only attributes can be changed (changing color, position etc).

In ItsNat this is not true, *you can add, remove and change inline SVG elements, text nodes and attributes*, this makes SVG inline with ASV ready to be used in Single Page Interface applications. ItsNat uses an obscure and not documented feature of ASV in this mode.

[78] http://www.schepers.cc/inlinesvg.html

However the limitations applied to events remain, only the parent element of the SVG markup fragment can receive (mouse) events, the real clicked element is unknown. Anyway we can surpass this limitation with some math and geometry.

The following example shows how we can embed inline SVG in an X/HTML document[79]. It works in MSIE 6+ with ASV plugin installed and browsers with native support (the `<object>` tag and `<?import?>` processing instruction are ignored, SVG code is rendered natively). The template must be served with MIME `text/html` to be compatible with MSIE (SVG markup is automatically reinserted by ItsNat in browsers with native support):

XHTML template:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>SVG inline in XHTML, MIME text/html and ASV</title>
</head>
<body xmlns:itsnat="http://itsnat.org/itsnat" itsnat:nocache="true"
       style="background:white;">

  <span style="display:none;">
    <object id="AdobeSVG"
       classid="clsid:78156a80-c6a1-4bbf-8e6a-3cd390eeb4e2"></object>
  </span>
  <span>
    <?import namespace="svg" urn="http://www.w3.org/2000/svg"
       implementation="#AdobeSVG"?>
  </span>

  <svg:svg id="svgId" xmlns:svg="http://www.w3.org/2000/svg" version="1.1"
       baseProfile="full" width="350px" height="170px"
       style="margin:30px; border: red 1px solid;">
    <svg:g id="circleListId">
      <svg:circle cx="60" cy="60" r="50" fill="#ff0000" stroke="#000000"
              stroke-width="2px"/>
    </svg:g>
    <svg:text id="textId" x="70" y="150" font-size="18">(Time)</svg:text>
  </svg:svg>

  <br />
  <a id="addCircleId" href="javascript:;">Add Circle</a>

</body>
</html>
```

In this markup some things are imposed by ItsNat, ASV, Internet Explorer[80] and others:

- The SVG namespace must have a prefix and must be declared only in the `<html>` element. As you can see this rule is not accomplished, because ItsNat detects this namespace declaration and automatically re-declares it in `<html>` as an attribute (load time) or adding it to `document.namespaces` in Internet Explorer.

[79] Included in the ItsNat Feature Showcase

[80] http://msdn.microsoft.com/en-us/library/dd565690%28VS.85%29.aspx

- As noted by Douglas Alan Schepers[81]: "*You can't really use any modern HTML DOCTYPEs (like the transitional XHTML one); this is not valid XHTML. Update: I found recently that I could use a truncated form of the DOCTYPE for transitional HTML 4.01, as long as I removed the URL; this still qualifies the markup as being quirky and non-standards-compliant, but it's better than nothing*".
- The `<object>` element is wrapped with `<span style="display:none">` to avoid the annoying icon of Safari 3 showing that this component is not recognized (any other browser graciously ignores it and is hidden).
- The `<?import…?>` processing instruction is wrapped with a `<span>` to avoid problems of path calculation in ItsNat[82] for elements after. Instead of the problematic `<?import…?>` you can use this simple script:

```
var ns = document.namespaces; if (ns) ns("svg").doImport("#AdobeSVG");
```

Java code:

```
public class SVGInHTMLMimeAdobeSVGLoadListener
        implements ItsNatServletRequestListener
{
    public SVGInHTMLMimeAdobeSVGLoadListener()
    {
    }

    public void processRequest(ItsNatServletRequest request,
                        ItsNatServletResponse response)
    {
        ItsNatDocument itsNatDoc = request.getItsNatDocument();
        new SVGInHTMLMimeAdobeSVGDocument(itsNatDoc);
    }
}

public class SVGInHTMLMimeAdobeSVGDocument implements EventListener
{
    protected ItsNatDocument itsNatDoc;
    protected Element svgElem;
    protected ElementList circleList;
    protected Element selectedCircle;
    protected Element textElem;
    protected Element addCircleElem;

    public SVGInHTMLMimeAdobeSVGDocument(ItsNatDocument itsNatDoc)
    {
        this.itsNatDoc = itsNatDoc;
        load();
    }

    public void load()
    {
       Document doc = itsNatDoc.getDocument();

       this.svgElem = doc.getElementById("svgId");
```

[81] http://www.schepers.cc/inlinesvg.html

[82] Processing instructions are problematic because they are present in server DOM but filtered in client (with the exception of Opera).

```
        ParamTransport[] params = new ParamTransport[]
         { new CustomParamTransport("offsetX","event.getNativeEvent().offsetX"),
           new CustomParamTransport("offsetY","event.getNativeEvent().offsetY")};
        itsNatDoc.addEventListener(((EventTarget)svgElem),"click",this,
                false,params);

        ElementGroupManager egm = itsNatDoc.getElementGroupManager();
        Element circleListElem = doc.getElementById("circleListId");
        this.circleList = egm.createElementList(circleListElem,false);

         this.addCircleElem = doc.getElementById("addCircleId");
         ((EventTarget)addCircleElem).addEventListener("click",this,false);
    }

    public void handleEvent(Event evt)
    {
        EventTarget target = evt.getTarget();
        if (target == svgElem) // MSIE or outside of circles
        {
            ItsNatEvent itsNatEvt = (ItsNatEvent)evt;
            if ("undefined".equals(itsNatEvt.getExtraParam("offsetX")))
                return; // Not MSIE and outside of circles

            String offsetXStr = (String)itsNatEvt.getExtraParam("offsetX");
            int x = Integer.parseInt(offsetXStr);
            String offsetYStr = (String)itsNatEvt.getExtraParam("offsetY");
            int y = Integer.parseInt(offsetYStr);
            itsNatDoc.addCodeToSend("alert('Clicked: ' + " + x + " + ',' + "
                + y + ");");

            for(int i = circleList.getLength() - 1; i >= 0; i--)
            {
                // The last circle has a greater SVG "z-index"
                Element circleElem = circleList.getElementAt(i);
                if (clickedCircle(x,y,circleElem))
                {
                    selectCircle(circleElem);
                    break;
                }
            }
        }
        else if ("circle".equals(((Node)target).getLocalName())) // Not MSIE
        {
            selectCircle((Element)target);
        }
        else if (target == addCircleElem)
        {
            Element lastElem = circleList.getLastElement();
            if (lastElem == null)
            {
                circleList.addElement();
            }
            else
            {
                Element newCircle = circleList.addElement();
                int cx = Integer.parseInt(lastElem.getAttribute("cx"));
                cx += 100;
                newCircle.setAttribute("cx",Integer.toString(cx));
            }
        }

         ((Text)textElem.getFirstChild()).setData(new Date().toString());
    }

    public void selectCircle(Element circleElem)
    {
        if (selectedCircle != null)
            selectedCircle.setAttribute("fill","#FF0000");

        if (circleElem != null)
            circleElem.setAttribute("fill","#00FF00");
        this.selectedCircle = circleElem;
    }

    public boolean clickedCircle(int x,int y,Element circle)
    {
        int xc = Integer.parseInt(circle.getAttribute("cx"));
        int yc = Integer.parseInt(circle.getAttribute("cy"));
        int r = Integer.parseInt(circle.getAttribute("r"));
        return (xc - r <= x)&&(x <= xc + r)&&(yc - r <= y)&&(y <= yc + r);
    }
}
```

What is doing this example?

When you click a circle MSIE fires a mouse click, the `offsetX` and `offsetY` event properties are the relative position of the mouse related to the corner of the visual area of the SVG parent element. These properties are interesting because with some math and geometry we can determinate in server what SVG element was clicked. Of course this is not needed in browsers with native SVG support, the standard `target` event property is the SVG element clicked, in these browsers `offsetX/offsetY` are not defined because they MSIE specific.

Finally with the link "Add Circle" we add a new circle using DOM code as usual.

Most of SVG examples of this Manual containing SVG code inline in XHTML documents could be slightly changed to support Internet Explorer with Adobe SVG Viewer.

6.37.5 SVG embedded inline in X/HTML processed by SVGWeb

Adobe SVG Viewer is not the only plugin able to render SVG inline in Internet Explorer. SVGWeb is a project owned by Google to provide SVG to Internet Explorer rendered by Flash (because is Flash based browsers with native support can also run SVGWeb). SVGWeb main objective is to add SVG capabilities to X/HTML pages, that is, instead of ASV or Ssrc it does not register itself as a plugin on MSIE to redirect processing of `image/svg+xml` requests.

SVGWeb adds SVG support to MSIE following two different strategies:

1. Loading external SVG files to be rendered by Flash included through `<object>` elements (SVG markup can also be provided "stringized" into the `data` attribute). SVG DOM document can be accessed through the `contentDocument` property of the `<object>` element.

2. Rendering SVG markup inline. In spite of this SVG markup is converted in an `<object>` element in MSIE and `<embed>` in non-MSIE, SVGWeb try hard to simulate that this SVG DOM is in the X/HTML DOM like it would be if MSIE could render this markup natively.

ItsNat automatically detects when SVGWeb is used and provides server side support to this technology, only the second SVGWeb strategy (SVG markup inline) is supported. Furthermore do not use the first strategy alongside ItsNat because SVGWeb adds auxiliary elements breaking the client/server symmetry required by ItsNat[83] and any JavaScript code into the SVG external documents is injected to the X/HTML page usually breaking ItsNat (SVGWeb does not provide a JavaScript engine).

Server side support added to SVGWeb by ItsNat has the same objective, SVG markup and DOM management must be the same as possible as in a client browser with native SVG support. Because ItsNat generates the appropriated JavaScript code behind the scenes, removes some requisites/limitations imposed by SVGWeb in pure client, for instance:

- No need to enclose your SVG markup inside a `<script type="image/svg+xml">` element, you just need to declare your SVG root node to be processed by SVGWeb with `itsnat:svgengine="svgweb" xmlns:itsnat="http://itsnat.org/itsnat"` attributes (as alternative `svgengine="svgweb"` is enough, without namespaces). Neither `<script type="image/svg+xml">` nor `<object>`/`<embed>` elements used by SVGWeb in client are reflected in server side DOM.

- There are not special SVGWeb methods in server, use pure DOM in server included dynamic insertion/removal of SVG root nodes, ItsNat automatically generates the appropriated JavaScript code. In fact, because `svgweb.insertBefore(...)` is provided by ItsNat in client (missing in SVGWeb) you can add new SVG root nodes in any place and in any time.

- Dynamic insertion of new SVG root nodes do not require further `SVGLoad` listeners as suggested by SVGWeb documentation, anyway this technique is supported too.

- You can add/remove/change text nodes, ItsNat ensures these actions are reflected visually.

- Comment nodes are supported in server, ItsNat automatically filters them in client.

- `<script>` elements can be dynamically inserted into SVG DOM and JavaScript code is automatically executed (SVGWeb does not support `<script>` elements on load time).

SVGWeb does not support prefixed SVG nodes like `<svg:svg xmlns:svg=”…”>`

Dynamic insertion of SVGWeb does not work in Opera, Opera support of SVGWeb seems poor.

Remote View/Control WORKS including dynamic insertion/removal of SVGWeb root nodes and any `forceflash` mode!

SVGWeb usually captures JavaScript exceptions and silently ignores them, to detect these errors use `ClientErrorMode.SHOW_SERVER_AND_CLIENT_ERRORS`, or `ClientErrorMode.NOT_CATCH_ERRORS` and some kind of JavaScript debugger like FireBug.

Finally, DO NOT mix Adobe SVG Viewer to render SVG Inline and SVGWeb in a Single Page Interface application[84]!

The following example shows how to handle the lifecycle of SVG markup and some dynamic actions.

XHTML template (registered with `text/html` MIME):

[83] The first strategy also adds auxiliary elements below <body>, in this case they all are automatically moved by ItsNat to a “hidden” DOM zone.

[84] Although the ASV <object> is removed from document, the plugin remains loaded and interfere with SVGWeb, MSIE could crash.

```
<!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>SVG in XHTML and SVGWeb Demo</title>
  <meta name="svg.render.forceflash" content="true" />
  <script src="svgweb/svg.js" data-path="svgweb"></script>
</head>
<body itsnat:nocache="true" xmlns:itsnat="http://itsnat.org/itsnat"
        style="margin:30px">

<svg id="svgId" xmlns="http://www.w3.org/2000/svg" version="1.1"
        baseProfile="full" width="350px" height="170px"
        style="margin:0px; border: red 1px solid;"
        itsnat:svgengine="svgweb" xmlns:itsnat="http://itsnat.org/itsnat">
    <g id="circleListId">
        <circle cx="60" cy="60" r="50" fill="#ff0000" stroke="#000000"
                stroke-width="2px"/>
    </g>
    <text id="textId" x="70" y="150" font-size="18">(Time)</text>
</svg>

<br /><br />
<a id="addCircleId" href="javascript:;">Add Circle</a>
<br /><br />
<a id="reinsertId" href="javascript:;">Reset (reinsertion)</a>&nbsp;
<input id="useSVGLoadId" type="checkbox" /> Use SVGLoad listener

</body>
</html>
```

The attribute declaration:

```
itsnat:svgengine="svgweb" xmlns:itsnat="http://itsnat.org/itsnat"
```

declares the SVG root node to be processed by SVGWeb. Prefix declaration is mandatory because any previous declaration (in enclosing HTML nodes) is ignored by SVGWeb and the SVGWeb parser throws an error. Because attribute and namespace declaration are too verbose, namespaces can be optionally removed:

`svgengine="svgweb"` is enough.

ItsNat automatically detects the `forceflash` declaration in URL and/or `<meta>` tag (the detection order is the same as SVGWeb). If set to false SVG markup is natively processed, ItsNat knows this fact and does not use SVGWeb to render SVG code[85]. Furthermore, SVG markup is ONLY rendered by SVGWeb if the SVG root node was declared with `svgengine="svgweb"` as seen before, including MSIE (in non-MSIE browsers SVG is rendered by SVGWeb Flash only if `forceflash` is set to true). ItsNat ensures that server code is the same for SVG rendered by SVGWeb Flash and native rendered.

Java code:

[85] SVG markup is just rendered by SVGWeb when is enclosed with <script type="image/svg+xml">, you script enclosing is done behind the scenes by SVGWeb ONLY when the SVG markup must be rendered by SVGWeb Flash.

```
public class SVGInHTMLMimeSVGWebLoadListener
        implements ItsNatServletRequestListener
{
    public SVGInHTMLMimeSVGWebLoadListener()
    {
    }

    public void processRequest(ItsNatServletRequest request,
                        ItsNatServletResponse response)
    {
        ItsNatDocument itsNatDoc = request.getItsNatDocument();
        new SVGInHTMLMimeSVGWebDocument(itsNatDoc);
    }
}

public class SVGInHTMLMimeSVGWebDocument implements EventListener
{
    protected ItsNatDocument itsNatDoc;
    protected Element svgElem;
    protected Element circleListElem;
    protected ElementList circleList;
    protected Element selectedCircle;
    protected Element textElem;
    protected Element addCircleElem;
    protected Element reinsertElem;
    protected HTMLInputElement useSVGLoadElem;

    public SVGInHTMLMimeSVGWebDocument(ItsNatDocument itsNatDoc)
    {
        this.itsNatDoc = itsNatDoc;
        load();
    }

    public void load()
    {
        Document doc = itsNatDoc.getDocument();
        EventListener listener = new EventListener()
        {
            public void handleEvent(Event evt)
            {
                loadSVG();
            }
        };
        AbstractView view = ((DocumentView)doc).getDefaultView();
        ((EventTarget)view).addEventListener("SVGLoad",listener,false);

        this.addCircleElem = doc.getElementById("addCircleId");
        ((EventTarget)addCircleElem).addEventListener("click",this,false);

        this.reinsertElem = doc.getElementById("reinsertId");
        ((EventTarget)reinsertElem).addEventListener("click",this,false);

        this.useSVGLoadElem =
               (HTMLInputElement)doc.getElementById("useSVGLoadId");
        itsNatDoc.addEventListener((EventTarget)useSVGLoadElem,"click",this,
               false,new NodePropertyTransport("checked",boolean.class));
    }

    public void loadSVG()
    {
        Document doc = itsNatDoc.getDocument();
        this.svgElem = doc.getElementById("svgId");

        this.circleListElem = doc.getElementById("circleListId");
        ElementGroupManager egm = itsNatDoc.getElementGroupManager();
        this.circleList = egm.createElementList(circleListElem,true);
        circleList.addElement();
        itsNatDoc.addEventListener(((EventTarget)circleListElem),"click",
               this,false);

        this.textElem = doc.getElementById("textId");
        ((Text)textElem.getFirstChild()).setData(new Date().toString());
    }

    public void handleEvent(Event evt)
    {
        EventTarget currTarget = evt.getCurrentTarget();
        if (currTarget == circleListElem)
        {
            EventTarget target = evt.getTarget();
            if ((target instanceof Element)  &&
                ((Element)target).getLocalName().equals("circle"))
            {
                selectCircle((Element)target);
            }
        }
        else if (currTarget == addCircleElem)
        {
            Element lastElem = circleList.getLastElement();
            if (lastElem == null)
            {
                circleList.addElement(); // Will be based on the pattern
            }
            else
            {
                Element newCircle = circleList.addElement();
                int cx = Integer.parseInt(lastElem.getAttribute("cx"));
                cx += 100;
                newCircle.setAttribute("cx",Integer.toString(cx));
            }
        }
        else if (currTarget == reinsertElem)
        {
            selectCircle(null);

            HTMLDocument doc = (HTMLDocument)itsNatDoc.getDocument();
            Element newSVGElem = (Element)svgElem.cloneNode(true);
            Node parent = doc.getBody();
            Node sibling = svgElem.getNextSibling();
            parent.removeChild(svgElem);
            parent.insertBefore(newSVGElem,sibling);
            this.svgElem = newSVGElem;

            if (useSVGLoadElem.getChecked())
            {
                EventListener listener = new EventListener()
                {
                    public void handleEvent(Event evt)
                    {
                        loadSVG();
                    }
                };
                ((EventTarget)svgElem).addEventListener("SVGLoad",
                       listener,false);

                return; // To avoid the next textElem's setData call
            }
            else
            {
                loadSVG();
            }

        }

        ((Text)textElem.getFirstChild()).setData(new Date().toString());
    }

    public void selectCircle(Element circleElem)
    {
        if (selectedCircle != null)
            selectedCircle.setAttribute("fill","#FF0000");

        if (circleElem != null)
            circleElem.setAttribute("fill","#00FF00");
        this.selectedCircle = circleElem;
    }

    public boolean clickedCircle(int x,int y,Element circle)
    {
        int xc = Integer.parseInt(circle.getAttribute("cx"));
        int yc = Integer.parseInt(circle.getAttribute("cy"));
        int r = Integer.parseInt(circle.getAttribute("r"));
        return (xc - r <= x)&&(x <= xc + r)&&(yc - r <= y)&&(y <= yc + r);
    }
}
```

This Java code requires some analysis:

```
            EventListener listener = new EventListener()
            {
                public void handleEvent(Event evt)
                {
                    loadSVG();
                }
            };
            AbstractView view = ((DocumentView)doc).getDefaultView();
            ((EventTarget)view).addEventListener("SVGLoad",listener,false);
```

When the document is loading we cannot access the SVGWeb DOM nodes in client with JavaScript because there is still no SVG DOM, we must wait to the `SVGLoad` event (DO NOT use "load", SVGWeb initialization happens *after* the load event is fired and is notified by a `SVGLoad` event). When received this event SVGWeb ensures SVG markup is already rendered and SVG DOM built. In server and `fastLoad` mode set to true in theory we can change the SVGWeb DOM with no problem (no load listener needed) because no JavaScript code is generated, but this is not valid for the attachment of event listeners to SVG nodes, event listener registration ever generates JavaScript accessing to the target node, so we need a load listener. The `SVGLoad` listener is not needed of course with native rendered (non-Flash) SVG, in this example load listener works the same for both cases.

```
                Element newSVGElem = (Element)svgElem.cloneNode(true);
                Node parent = doc.getBody();
                Node sibling = svgElem.getNextSibling();
                parent.removeChild(svgElem);
                parent.insertBefore(newSVGElem,sibling);
```

```
            this.svgElem = newSVGElem;
```

This code reinserts the SVG DOM subtree to show we can insert dynamically SVGWeb root nodes. In this case we remove FIRST the already inserted SVG node because in this example the new SVG node and the current SVG node in document have the same id attribute and SVGWeb FAILS when two or more `<svg>` elements into the client document have the same id attribute. Another alternative like changing removing the id of the new SVG node and setting again after insertion and after removing, is NOT a good idea, SVGWeb is not tolerant to id changes before the SVG is rendered (the new SVG node inserted is not yet rendered). Anyway be careful with ids in SVGWeb nodes because SVGWeb internally uses attribute ids of SVG root nodes for identification purposes (this is not a problem for ItsNat because ItsNat does not use the id attribute for node identification).

```
            if (useSVGLoadElem.getChecked())
            {
                EventListener listener = new EventListener()
                {
                    public void handleEvent(Event evt)
                    {
                        loadSVG();
                    }
                };
                ((EventTarget)svgElem).addEventListener("SVGLoad",
                       listener,false);

                return; // To avoid the next textElem's setData call
            }
            else
            {
                loadSVG();
            }
```

In theory when new SVG nodes are dynamically inserted to be rendered by SVGWeb, these nodes in client are provisional and will be replaced, after insertion into the document any change to this provisional DOM is not valid, changing attributes of the `<svg>` element is almost the only action valid (not recommended), therefore we cannot change the SVG DOM or add event listeners immediately after insertion. The SVGWeb root node automatically fires a `SVGLoad` event when rendering is done, this is the moment to change the SVG.

The previous is the approach suggested by SVGWeb documentation and valid outside ItsNat, in ItsNat `SVGLoad` listeners are not necessary, that is, you CAN modify and/or add listeners to the SVGWeb DOM immediately after insertion, ItsNat automatically queues any JavaScript generated code to be executed when rendering is done[86].

Anyway you also can the `SVGLoad` approach in server as promoted by SVGWeb. In non-Flash mode (non-MSIE browsers) a `SVGLoad` event is simulated by ItsNat fired by the native SVG root node, this avoids us to detect if SVG nodes are Flash or native, the same server code is valid in both cases. Our example uses both techniques, a checkbox selects the required one. Important: do not mix both approaches, that is, do NOT modify SVG DOM immediately after insertion AND attach a `SVGLoad` listener.

[86] Under the hood an internal SVGLoad listener is used. This technique does not work in Google App Engine, use a custom SVGLoad listener.

Finally the use of a `SVGLoad` listener in no way affects to remote view/control, both modes (with or without `SVGLoad` listener) works with no problem.

6.37.6 SVG documents loaded by the ItsNat Batik applet

ItsNat distributes an applet based on Batik SVG toolkit, this applet can be found into the folder `/web/batik` of the Feature Showcase of the ItsNat distribution. Batik provides an almost complete SVG browser based on Swing and Rhino for scripting. The Batik applet included in ItsNat fixes some problems of Batik running in an applet environment and extends it to work seamless with ItsNat in server. This applet is not signed and no special system access is needed (no user question is done to be executed).

These are the most important improvements:

- If `<applet>` or `<object>` is used to load the applet, a `<param>` tag is used to specify the URL going to be loaded. If relative the URL is resolved based on the URL of the document containing the applet. The `<param>` element has the format:

  ```
  <param name="src" value="URL" />
  ```

- The applet root class, called `ItsNatBatikAppletLauncher`) has two methods **getSrc**`()` and **setSrc**`(String url)`, the first method returns the URL currently loaded and the second method loads the SVG page specified in the URL (again is resolved based on the URL of container page) without needing a reload of the applet. In fact if the URL of the `<param>` specifying the URL to be loaded by the applet is changed *in the server* the method **setSrc**`(String url)` is automatically called by ItsNat in client loading the new page.

- An almost complete `XMLHttpRequest` is provided, including GET and POST, synchronous and asynchronous requests. Source code is based on the `XMLHttpRequest` Java class published by Paul Fremantle and Anthony Elder on IBM DeveloperWorks[87].

- Child access from parent and parent access from child document: the parent document can be accessed from the child document with `window.parent` or `window.top`. To access the child SVG document from parent, call the applet method **getSVGDocument**`()`. Anyway parent/child communication is incomplete and not perfect, and mainly defined to call public JavaScript methods of ItsNat like **fireUserEvent**`(name)`, **createUserEvent**`(name)` and **dispatchUserEvent**`(evt)`, these methods are useful to notify the parent or child document in client about something has changed in server.

- In SVG `window.location` is partially implemented: **toString**`()`, **assign**`(url)`, **reload**`([force])`, **replace**`(url)` and **href** property are defined.

- Some support of link navigation. When a link (`<a>` tag) is clicked the URL specified in `href` is loaded. There is one limitation, if the URL is the same as the currently loaded document nothing is done, to ensure the document is reloaded use `window.location.reload(true)`.

- Support of "`javascript:`" protocol in `href` of links (`<a>` tag). The JavaScript code is executed when the link is clicked.

- Paradoxically Rhino is designed for desktop (also Batik), this introduce a serious problem in applets if `importPackage` is used because resolution of JavaScript global names as Java

[87] http://www.ibm.com/developerworks/library/ws-ajax1/

class names implies network requests, most of them useless degrading seriously the performance of JavaScript. Unfortunately Batik imports several Java packages, this problem is documented here[88]. The solution is simple, in ItsNat applet only `java.lang` package is pre-declared in JavaScript environment, the drawback is your JavaScript code may be more verbose, for instance instead of `alert(Node.ELEMENT_NODE)` use `alert(Packages.org.w3c.dom.Node.ELEMENT_NODE)`[89].

ItsNat Batik applet can be added to your X/HTML page included in `<applet>`, `<object>` and `<embed>` element. The following examples work in any prominent browser including MSIE v6 to 8:

- The following example shows how to load an SVG document by the ItsNat Batik applet using an `<applet>` tag:

```
    <applet style="width:420px; height:200px;"
        code="ItsNatBatikAppletLauncher.class"
        codebase="batik"
        archive="ItsNatBatikApplet.jar,batik-awt-util.jar,batik-
bridge.jar,batik-css.jar,batik-dom.jar,batik-ext.jar,batik-gvt.jar,batik-
parser.jar,batik-svg-dom.jar,batik-script.jar,batik-swing.jar,batik-
util.jar,batik-xml.jar,batik-svggen.jar,xml-apis-dom3.jar,js.jar,core-renderer-
minimal.jar">
        <param name="src" value="?itsnat_doc_name=SomeSVGDocName" />
        Your browser doesn't seem to support Java applets.
    </applet>
```

If the containing page was loaded using a URL like:

`http://<host&port>/`*`someapp`*`/`*`anitsnatservlet`*`?itsnat_doc_name=someDoc`

The SVG page going to be loaded is:

`http://<host&port>/`*`someapp`*`/`*`anitsnatservlet`*`?itsnat_doc_name=someSVGDocName`

- In an `<object>` the URL is specified using a `<param>` element like in `<applet>`:

```
    <object type="application/x-java-applet"
          style="width: 420px; height: 200px;">
        <param name="code" value="ItsNatBatikAppletLauncher.class" />
        <param name="codebase" value="batik" />
        <param name="archive" value="ItsNatBatikApplet.jar,batik-awt-
util.jar,batik-bridge.jar,batik-css.jar,batik-dom.jar,batik-ext.jar,batik-
gvt.jar,batik-parser.jar,batik-svg-dom.jar,batik-script.jar,batik-
swing.jar,batik-util.jar,batik-xml.jar,batik-svggen.jar,xml-apis-
dom3.jar,js.jar,core-renderer-minimal.jar" />
        <param name="src" value="?itsnat_doc_name=SomeSVGDocName" />
    </object>
```

[88] http://www.nabble.com/FW%3A-Strange-applet-delay-revisited-to21494010.html

[89] Fortunately in a pure server centric web application with ItsNat, rarely you are going to need such DOM code. Anyway with some browser sniffing and hand made code like "`Node = Packages.org.w3c.dom.Node;`", this code "`alert(Node.ELEMENT_NODE);`" works again.

- The third option is with an `<embed>`:

```
    <embed type="application/x-java-applet" style="width: 420px; height: 200px;"
           code="ItsNatBatikAppletLauncher.class"
           codebase="batik"
           archive="ItsNatBatikApplet.jar,batik-awt-util.jar,batik-
bridge.jar,batik-css.jar,batik-dom.jar,batik-ext.jar,batik-gvt.jar,batik-
parser.jar,batik-svg-dom.jar,batik-script.jar,batik-swing.jar,batik-
util.jar,batik-xml.jar,batik-svggen.jar,xml-apis-dom3.jar,js.jar,core-renderer-
minimal.jar"
           src="?itsnat_doc_name=SomeSVGDocName"
           pluginspage="http://java.sun.com/javase/downloads/index.jsp" />
```

The Feature Showcase contains examples of SVG documents loaded by ItsNat Batik applet with `<applet>`, `<object>` and `<embed>`.

6.37.7 Pure XUL

In the same way as SVG, remote XUL documents are supported in ItsNat.

ItsNat adds event support (sent by AJAX or SCRIPT) to XUL documents when XUL templates are registered with the MIME type `application/vnd.mozilla.xul+xml`.

XUL is supported in Internet Explorer through Savarese Ssrc plugin.

6.37.8 Embedding inline XHTML in SVG and XUL

Embedding XHTML in SVG is a new capability in modern browsers and is not fully working (usually with many visual bugs). The best browser to test this feature is FireFox 3.5 and upper (may work in previous versions but not tested). XHTML in SVG is interesting to add input controls to SVG documents like text boxes or selection lists.

Gecko browsers (and Savarese Ssrc) allow XHTML elements to be embedded in XUL documents. This feature is not very useful because XUL can be seen as a richer alternative to XHTML, anyway it works.

ItsNat supports, in a server point of view, embedding XHTML in SVG and XUL documents, furthermore, HTML components can be created and attached to embedded XHTML elements because in a SVG or XUL document you can ever cast the `ItsNatComponentManager` instance to `ItsNatHTMLComponentManager`.

The Feature Showcase includes an example of embedding XHTML in SVG using the special `<foreignObject>` element[90].

6.37.9 Non-XHTML markup in X/HTML and text/html MIME

As said before for SVG, ItsNat automatically reinserts by JavaScript any detected non-XHTML markup in X/HTML served with text/html MIME. Elements with non-XHTML namespaces (including declaration) must not be cached because ItsNat in server needs some processing to

[90] In XUL you do not need a special container element, using the correct XHTML namespace is enough.

add auxiliary attributes useful to correctly reinsert non-XHTML markup, these auxiliary attributes are automatically removed in client.

Automatic reinsertion also applies to single attributes, for instance WAI-ARIA attributes are supported with namespaces[91]:

```
<div xmlns:aaa="http://www.w3.org/2005/07/aaa">
    <h1 id="theWinner" aaa:live="polite"></h1>
</div>
```

If automatic reinsertion of elements and attributes is problematic you can disable it with the special ItsNat attribute `itsnat:ignorens="true"`, this attribute instructs to ItsNat to ignore non-XHTML namespaces from the element this attribute was declared. For instance:

```
<span xmlns:fb="http://www.facebook.com/2008/fbml" itsnat:ignorens="true">
  <fb:login-button></fb:login-button>
</span>
```

This example uses Facebook's XFBL markup[92], because Facebook script can manage markup with namespaces in HTML (or XHTML and text/html MIME) reinsertion is not needed (Facebook script also works with valid namespaces therefore `ignorens` can also be removed).

The attribute `itsnat:ignorens="true"` is interpreted by ItsNat in server and *in client*, (most of ItsNat attributes are only useful in server), in server namespaces are W3C conformant but in client and text/html MIME are not real namespaces therefore in this case `itsnat` prefix is mandatory as is, prefix name cannot be different for `ignorens`.

6.38 IFRAME/OBJECT/EMBED/APPLET AUTO-BINDING

In ItsNat every client document (web page) is a document in server, the same is for child documents loaded by `<iframe>`, `<object>`, `<embed>` or `<applet>`. In client you can access the document object contained in an `iframe/object/embed/applet` element using JavaScript, `elem.contentDocument` in W3C browsers and `elem.contentWindow.document` in MSIE, if the contained web page was loaded from the same port and domain following the "same origin policy"[93]; this is the usual way, anyway accessing to the child document may be plugin dependent (if any is used), for instance, in MSIE loading child documents processed by Adobe SVG Viewer the preferred technique is calling the `getSVGDocument()` method, this method is also required in ItsNat Batik applet.

In the same way you can access the child document in server calling `HTMLIFrameElement.`**`getContentDocument`**`()` for `<iframe>`, `HTMLObjectElement.`**`getContentDocument`**`()` for `<object>`, `ItsNatHTMLEmbedElement.`**`getContentDocument`**`()` for `<embed>` and `ItsNatHTMLAppletElement.`**`getContentDocument`**`()` for `<applet>`. The returned object (if non-null) grants you access to the child document, and in the child `ItsNatDocument` the parent document is obtained calling `ItsNatDocument.`**`getContainerNode`**`()` (returns null if

[91] http://www.w3.org/TR/2008/WD-wai-aria-20080204/#impl_namespace

[92] http://wiki.developers.facebook.com/index.php/XFBML

[93] http://en.wikipedia.org/wiki/Same_origin_policy

this document is stand alone or not bound to the parent document). This feature is extremely useful in Single Page Interface applications with intensive use of SVG when inline SVG is problematic like in Internet Explorer (if the SVG markup is inline only one document exists).

The document loaded by an `iframe/object/embed/applet` is automatically bound to the parent in server if the:

- `src` attribute in `<iframe>` and `<embed>` or
- `data` attribute in `<object>` (not an applet) or
- the URL defined in `<param name="src" value="URL">` contained in `<object>` or `<applet>` loading the ItsNat Batik applet

matches in server the following patterns:

1. Relative URL:

   ?**itsnat_doc_name=*someDocTemplateName***&*otherparams…*#*optionalref*

2. Absolute URL:

   SamePathOfParentURL?**itsnat_doc_name=*someDocTemplateName***&*other…*#*optionalref*

For instance:

```
<iframe src="?itsnat_doc_name=iframeExample"></iframe>
<iframe src="?itsnat_doc_name=iframeExample&count=1&max=20#input"></iframe>
<iframe
src="http://192.168.2.100:8080/itsnat/servlet?itsnat_doc_name=iframeExample&coun
t=1&max=20#input"></iframe>
```

The latest matches the auto-binding rule only if the parent document was requested with the following URL:

```
http://192.168.2.100:8080/itsnat/servlet?...
```

As you can see the ItsNat servlet which loaded the container page is used to load the `iframe` page, this ensures the same origin policy[94]. If the URL attribute does not match the specified formats then the child document is loaded as usual but `HTMLIFrameElement.`**`getContentDocument`**`()`, `HTMLObjectElement.`**`getContentDocument`**`()`, `ItsNatHTMLEmbedElement.`**`getContentDocument`**`()`, `ItsNatHTMLAppletElement.`**`getContentDocument`**`()` and `ItsNatDocument.`**`getContainerNode`**`()` return null (parent and document are not bound in server).

In spite of `<object>` is discouraged with Adobe SVG Viewer (ASV), ItsNat can detect in this element if `src` attribute and `<param name="src" value="URL">` are defined, in this case the URL value of both parameters must be *exactly* the same as the `data` attribute and the

[94] In this case the "same origin policy" applied in ItsNat is the "same ItsNat servlet" policy slightly more restrictive case.

data attribute must be present (in spite of is ignored by ASV). For instance, the following declaration works in W3C browsers and MSIE with ASV:

```
<object data="?itsnat_doc_name=svgExample"
        src="?itsnat_doc_name=svgExample" type="image/svg+xml">
    <param name="src" value="?itsnat_doc_name=svgExample" />
</object>
```

6.38.1 The case of <embed> elements

<embed> elements are supported the same as <iframe> and <object>, this tag is a legacy of the old Netscape Navigator. In spite of this element is deprecated by W3C, is supported in any browser with some support of plugins. There is no standard way to access the child document loaded by <embed> in client, usually the technique is the same as <object>. For instance Adobe SVG Viewer define the method getSVGDocument(), like in <object>. In server ItsNat defines a specific interface, ItsNatHTMLEmbedElement, implemented by <embed> objects in server DOM, this interface defines the method ItsNatHTMLEmbedElement.**getContentDocument**() to be used the same as standard methods like HTMLIFrameElement.**getContentDocument**() and HTMLObjectElement.**getContentDocument**().

Example:

```
<embed src="?itsnat_doc_name=svgExample" type="image/svg+xml" />
```

6.38.2 The case of <applet> elements

Similar to <embed>, there is no standard way to access the child document loaded by an <applet> in client. In the same way ItsNat defines a specific interface, ItsNatHTMLAppletElement and the method ItsNatHTMLAppletElement.**getContentDocument**(). In ItsNat Batik applet getSVGDocument() grants access to the child document following the technique offered by Adobe SVG Viewer or some browsers with native SVG support.

The Feature Showcase has examples of auto-binding SVG documents loaded by ItsNat Batik applet using <object>, <embed> and <applet> tags.

6.38.3 Example

The following example shows how we can communicate parent document and the child document contained in an iframe. In summary an HTML button inside an iframe is "sent" to the parent when clicked, and the button in the parent is "sent" to the iframe again when clicked[95].

The markup in the template going to be the container (parent) page:

```
...
<div id="iframeParentPutHereId"></div>
<iframe id="iframeBoundId" src="?itsnat_doc_name=iframeExample"></iframe>
...
```

[95] This example is got from the "Feature Showcase"

Java code executed by the parent in load time:

```
    public void load()
    {
        ItsNatDocument itsNatDoc = ...;
        Document doc = itsNatDoc.getDocument();
        this.iframe = (HTMLIFrameElement)doc.getElementById("iframeBoundId");

        itsNatDoc.addUserEventListener(null,"update",this);
    }
```

The remaining code:

```
    public void handleEvent(Event evt)
    {
        ItsNatDocument itsNatDoc = ((ItsNatEvent)evt).getItsNatDocument();
        if (evt instanceof ItsNatUserEvent) // Button received
        {
            prepareButtonToSend(itsNatDoc);
        }
        else // Button clicked
        {
            sendToIFrame(itsNatDoc);
        }
    }

    public void prepareButtonToSend(ItsNatDocument itsNatDoc)
    {
        Document doc = itsNatDoc.getDocument();
        Element button = doc.getElementById("buttonId");

        Text text = (Text)button.getFirstChild();
        text.setData("Send to IFrame");

        ((EventTarget)button).addEventListener("click",this,false);
    }

    public void sendToIFrame(ItsNatDocument itsNatDoc)
    {
        Element button = removeButton(itsNatDoc);

        Document iframeDoc;
        try
        {
            iframeDoc = iframe.getContentDocument();
        }
        catch(NoSuchMethodError ex)
        {
            // Cause: Xerces compatibility package of Tomcat 5.5
            // misses this standard DOM method in HTMLIFrameElement interface
            // Don't worry, our required method is there.
            try
            {
                Method method =
                    iframe.getClass().getMethod("getContentDocument",null);
                iframeDoc = (Document)method.invoke(iframe,null);
            }
            catch(Exception ex2) { throw new RuntimeException(ex2); }
        }
        if (iframeDoc == null)
        {
            itsNatDoc.addCodeToSend("alert('Not loaded yet');");
            return;
        }

        ItsNatDocument iframeItsNatDoc =
               ((ItsNatNode)iframeDoc).getItsNatDocument();
               // This method is multithread

        synchronized(iframeItsNatDoc) // NEEDED!!!
        {
            Element buttonInParent = (Element)iframeDoc.importNode(button,true);
            Element contElem = iframeDoc.getElementById("iframeChildPutHereId");
            contElem.appendChild(buttonInParent);

            // Notify child document
            String ref = itsNatDoc.getScriptUtil().getNodeReference(iframe);
            StringBuffer code = new StringBuffer();
            code.append("var elem = " + ref + ";");
            code.append("var doc = (typeof elem.contentDocument !=
\"undefined\") ? elem.contentDocument : elem.contentWindow.document;"); //
contentWindow in MSIE
            code.append("doc.getItsNatDoc().fireUserEvent(null,'update');");
            itsNatDoc.addCodeToSend(code.toString());
        }
    }

    public Element removeButton(ItsNatDocument itsNatDoc)
    {
        Document doc = itsNatDoc.getDocument();
        Element button = doc.getElementById("buttonId");
        // if (button == null) return null;
        ((EventTarget)button).removeEventListener("click",this,false);
        button.getParentNode().removeChild(button);
        return button;
    }
```

To understand the previous code we need to publish the `iframe` code:

`Iframe` template markup (`iframeExample`):

```
    <h4>IFRAME</h4>

    <div id="iframeChildPutHereId">
        <button id="buttonId">Send to Parent</button>
    </div>
```

Load listener and document processor:

```
public class IFrameAutoBindingLoadListener
                implements ItsNatServletRequestListener
{
    public IFrameAutoBindingLoadListener()
    {
    }

    public void processRequest(ItsNatServletRequest request,
                ItsNatServletResponse response)
    {
        ItsNatDocument itsNatDoc = request.getItsNatDocument();
        new IFrameAutoBindingDocument(itsNatDoc);
    }
}

public class IFrameAutoBindingDocument implements EventListener
{
    protected ItsNatDocument itsNatDoc;

    public IFrameAutoBindingDocument(ItsNatDocument itsNatDoc)
    {
        this.itsNatDoc = itsNatDoc;
        load();
    }

    public void load()
    {
        itsNatDoc.addUserEventListener(null,"update",this);

        prepareButtonToSend();
    }

    public void handleEvent(Event evt)
    {
        if (evt instanceof ItsNatUserEvent) // Button received
        {
            prepareButtonToSend();
        }
        else // Button clicked
        {
            sendToParent();
        }
    }

    public void prepareButtonToSend()
    {
        Document doc = itsNatDoc.getDocument();
        Element button = doc.getElementById("buttonId");

        Text text = (Text)button.getFirstChild();
        text.setData("Send to Parent");

        ((EventTarget)button).addEventListener("click",this,false);
    }

    public void sendToParent()
    {
        // Synchronization not needed, parent document is ever synchronized
        Element iframeElem = (Element)itsNatDoc.getContainerNode();
        if (iframeElem == null)
        {
            itsNatDoc.addCodeToSend("alert('Disconnected from parent!');");
            return;
        }
        ItsNatDocument parentItsNatDoc =
                               ((ItsNatNode)iframeElem).getItsNatDocument();

        Document doc = itsNatDoc.getDocument();
        Element button = doc.getElementById("buttonId");
        ((EventTarget)button).removeEventListener("click",this,false);
        button.getParentNode().removeChild(button);
        Document parentDoc = parentItsNatDoc.getDocument();
        Element buttonInParent = (Element)parentDoc.importNode(button,true);
        Element contElem = parentDoc.getElementById("iframeParentPutHereId");
        contElem.appendChild(buttonInParent);

        // Notify the parent document
        StringBuffer code = new StringBuffer();
        code.append("if (window.parent == window) alert('NOT SUPPORTED');");
         code.append("else
window.parent.document.getItsNatDoc().fireUserEvent(null,'update');");
        itsNatDoc.addCodeToSend(code.toString());
    }
}
```

As you can see the button is initially inside the `iframe`, when the button is clicked the button is removed from document because is going to be sent to the parent, the parent `iframe` node is obtained with the call:

```
   Element iframeElem = (Element)itsNatDoc.getContainerNode();
```

The returned element is the parent `iframe` element. Through this element we can obtain the parent ItsNat document:

```
   ItsNatDocument parentItsNatDoc =
            ((ItsNatNode)iframeElem).getItsNatDocument();
```

In this case the thread calling both methods is a web request (an AJAX/SCRIPT event) from the client `iframe`, we know the ItsNat document of the `iframe` page is synchronized, is the parent `ItsNatDocument` synchronized? YES. If an ItsNat document is automatically bound to the parent (was loaded by using an `iframe` following the "same origin policy" implicit in the `src` attribute), in *every* subsequent AJAX/SCRIPT request the parent is automatically synchronized with the thread used for the request (the exact order is the parent first, then the child document). In this way we do not need to synchronize the parent document before any access to parent content. Parent synchronization avoids dead locks when a web request in parent is going to access the child iframe because this request will be hold until all requests in children end.

Because the parent document is different to the child document, the button must be imported to the parent node before inserting in parent:

```
        Document parentDoc = parentItsNatDoc.getDocument();
        Element buttonInParent = (Element)parentDoc.importNode(button,true);
        Element contElem = parentDoc.getElementById("iframeParentPutHereId");
        contElem.appendChild(buttonInParent);
```

When this request ends the iframe in client automatically is updated, the button disappears. What about the parent? The parent web page is not automatically updated because is a different web page (in spite of the DOM tree in server has changed, JavaScript code was generated but is not sent to the client).

We need to fire an event in the parent to notify the parent code in server, this is the reason of the following code, this code fires a user defined event in the parent:

```
        // Notify the parent document
        StringBuffer code = new StringBuffer();
        code.append("if (window.parent == window) alert('NOT SUPPORTED');");
        code.append("else
window.parent.document.getItsNatDoc().fireUserEvent(null,'update');");
        itsNatDoc.addCodeToSend(code.toString());
```

Note we are accessing the parent document from the client; this is only possible in "same origin".

Then a user event with name "`update`" is sent to the server from parent, in parent code we have registered an event listener for this event (otherwise it would be ignored):

```
        itsNatDoc.addUserEventListener(null,"update",this);
```

And the handler:

```
    public void handleEvent(Event evt)
    {
        ItsNatDocument itsNatDoc = ((ItsNatEvent)evt).getItsNatDocument();
        if (evt instanceof ItsNatUserEvent) // Button received
        {
            prepareButtonToSend(itsNatDoc);
        }
         ...
```

The method `prepareButtonToSend()` adds an event listener to the received button, when clicked again, this case in parent, the button is sent again to the `iframe` calling `sendToParent()`.

```
        else // Button clicked
        {
            sendToParent();
        }
```

This method is similar to the `iframe` version (some reusing is possible), in this case we need to get the document of the `iframe` calling `HTMLIFrameElement.`**`getContentDocument`**`()`:

```
        Document iframeDoc;
        try
        {
            iframeDoc = iframe.getContentDocument();
        }
        catch(NoSuchMethodError ex)
        {
           ...
        }

        ItsNatDocument iframeItsNatDoc =
               ((ItsNatNode)iframeDoc).getItsNatDocument();
               // This method is multithread
        ...

        synchronized(iframeItsNatDoc) // NEEDED!!!
        {
            Element buttonInParent = (Element)iframeDoc.importNode(button,true);
            Element contElem = iframeDoc.getElementById("iframeChildPutHereId");
            contElem.appendChild(buttonInParent);
        ...
```

In this case the current thread is a web request from parent, the parent document is already synchronized, is the child document synchronized? NO. Before accessing the child document we need to synchronize the `iframe` document to avoid any concurrent access. Dead locks are not possible because the parent document is synchronized in any web request received by child documents.

Finally the parent uses the same technique to notify the child client the button is already in the DOM tree in server. In this case a client element reference generated in server is used[96]:

```
            // Notify child document
            String ref = itsNatDoc.getScriptUtil().getNodeReference(iframe);
            StringBuffer code = new StringBuffer();
            code.append("var elem = " + ref + ";");
            code.append("var doc = (typeof elem.contentDocument !=
\"undefined\") ? elem.contentDocument : elem.contentWindow.document;"); //
contentWindow in MSIE
            code.append("doc.getItsNatDoc().fireUserEvent(null,'update');");
            itsNatDoc.addCodeToSend(code.toString());
```

In this example only DOM is changed directly by parent and client, you can use the `ItsNatUserData` interface of `Document` objects (any DOM node implements `ItsNatNode` and `ItsNatNode` inherits from `ItsNatUserData`) and `ItsNatDocument` objects (almost any ItsNat object implements `ItsNatUserData`) to pass user defined objects between parent and children code (for instance to access the `IFrameAutoBindingDocument` instance from parent code).

The Feature Showcase includes a similar example of a SVG document loaded by an `iframe` and automatically bound to the parent.

6.38.4 Dynamic insertion and URL change

The previous example showed an `iframe` defined in markup and loaded in the load phase of the parent document. This is not mandatory, auto-binding of `iframe/object/embed/applet` elements works with dynamically inserted elements in any time, furthermore, if the attribute `src` (`iframe` and `embed`) or `data` (object) or `value` (`<param name="src">` if applets in `<object>` or `<embed>`) is changed in server and the new URL matches the expected format, the new child document will be bound to the parent in server and client.

If an `iframe/object/embed/applet` element, which document is bound to the parent, is removed from the document tree, the method **`getContentDocument`**`()` of `HTMLIFrameElement`, `HTMLObjectElement`, `ItsNatHTMLEmbedElement` or `ItsNatHTMLAppletElement` returns null.

In Internet Explorer with some SVG plugin installed (ASV or Ssrc), if you are going to change with JavaScript the URL of an `<object>` *already in tree*, the `data` attribute must be the latest change after changing `src` attribute and `param`, otherwise auto-binding fails. Reinsertion is the alternative: remove first the element, change the URL attributes (`src`, `data` and `value` in `<param>`) with the same URL value, any order is valid, and reinsert again.

6.38.5 Auto-binding in SVG and XUL documents

[96] Of course "var elem = document.getElementById('iframeBoundId');" does the same in this case.

XHTML `<iframe>`, `<object>` and `<embed>` elements can be inserted in SVG (inside `<foreignObject>`) and XUL, auto-binding works in these contexts. Auto-binding do not work in XUL native iframes, use instead XHTML iframes.

6.38.6 Auto-binding and Remote View/Control

Auto-binding of `<iframe>`, `<object>`, `<embed>` and `<applet>` elements also works in a page attached to a document for remote view/control. In this case the child document bound to the parent in server, is remote viewed/controlled by the client doing remote view/control. That is, if a user is using a page with `iframe/object/embed/applets` automatically bound, another user (or another client window in general) is monitoring this page, documents contained by iframe/object/embed/applets are automatically monitored too. If the user (first page) monitored (the "owner") changes the state of the document loaded by an `iframe/object/embed/applet`, the state is changed in the document of the client `iframe/object/embed/applet` of the user monitoring.

In this case the parameter `itsnat_wait_doc_timeout` is necessary, this integer parameter must be provided in the initial request for monitoring the parent page and is the time in milliseconds the monitoring client waits until a dynamically created `iframe/object/embed/applet` loads the required document in server, when the `iframe/object/embed/applet` document is finally loaded, this document is attached for monitoring by the `iframe/object/embed/applet` of the remote view/control client (if the time spent is lower than the specified timeout). This parameter is not useful to monitor the parent document itself, if a page being monitored has not `iframe/object/embed/applets` bound to the parent, in this case the parameter `itsnat_wait_doc_timeout` is ignored.

In the Feature Showcase almost all examples using `iframe/object/embed/applets` can be remote viewed (only fail in documents with state not saved in server, for instance, examples with disabled events).

6.39 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 also be used, 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
<?xml version='1.0' encoding='UTF-8' ?>

<discs>
    <cdList>
        <cd>
            <title>Tittle</title>
            <artist>Artist</artist>
            <songs>
                <song>Song Pattern</song>
            </songs>
        </cd>
    </cdList>
</discs>
```

Registering into the servlet (`init(ServletConfig)` method):

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

    ItsNatHttpServlet itsNatServlet = getItsNatHttpServlet();

    ItsNatDocumentTemplate docTemplate;
    ...
    docTemplate = itsNatServlet.registerItsNatDocumentTemplate(
                      "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>
```

6.39.1 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();
ItsNatDocFragmentTemplate fragTemplate =
           servlet.getItsNatDocFragmentTemplate("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);
    ...
```

6.40 BROWSER ISSUES

ItsNat best support is achieved with MSIE, FireFox, Safari and Chrome desktop browsers. Opera works fine except referrers and page navigation in general (pure AJAX applications with one only page are recommended).

These are the most important problems found by ItsNat:

6.40.1 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 and some mobile browsers do not fire unload events in some circunstances for instance when the back or forward button is pressed[97] 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).

6.40.2 Pages are not reloaded from server using back and forward buttons

This problem affects to Opera desktop and some mobile browsers (Opera Mini and Mobile, and some WebKit based browsers).

[97] When we talk about "back/forward" buttons, history navigation using JavaScript (`window.history.go(n)` etc) is supposed too

Opera is fast traversing the navigation history using back/forward buttons mainly because pages are not reloaded using the cache. Opera ignores any HTTP header or meta tag instructing not to be cached (only are obeyed with HTTPS)[98]. This is an important 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 (including Opera Mini and Opera Mobile) 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 (or similar):

```
window.location.reload(true);
```

In some mobile browsers there is no way to detect when the page is cached until the user touches 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 some mobile browsers you can use alternative techniques to gain access to the previous page[99].

6.40.3 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 synchronization between server and client, client state is the same of the server state after loading.

---

[98] http://my.opera.com/yngve/blog/2007/02/27/introducing-cache-contexts-or-why-the

[99] 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.

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`).

6.40.4 XMLHttpRequest only asynchronous in some browsers

Some WebKit based mobile browsers like S60WebKit and S40WebKit do not support synchronous XHR requests because this feature was implemented in WebKit later in non-Mac OS X versions of WebKit[100].

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).

6.40.5 Old modes and Quirks Mode in Internet Explorer v9

Internet Explorer v9 is a true W3C browser, it is a very different browser than previous versions, ItsNat detects this version and uses the new engine sharing code with other W3C browsers and almost no sharing with the old MSIE code. This is why namespaces, native SVG, event capturing and so on also works in MSIE 9 in ItsNat applications.

Nevertheless Internet Explorer v9 tries to be compatible with the past:

- The end user can manually change how the engine works (IE 8, IE 7 and IE 5 modes), this affects to the JavaScript API trying to mimic the MSIE specific JavaScript API of previous versions (goodbye to W3C DOM Events). Old modes can also be set with special `<meta>` elements in the page header[101]. The problem is that browser's user agent does not change and browser detection by ItsNat is only based on user agent sniffing in server on load time. In summary, ItsNat only supports IE 9 mode (the default mode).

- Quirks mode: the browser is automatically set in quirks mode when there is no DOCTYPE or is old or not recognized[102]. In previous versions quirks mode only affected to visual rendering, in MSIE v9 also affects to JavaScript API, in quirks mode new W3C supported APIs are ignored like W3C DOM Events. Using quirks mode is a very bad idea and is not supported in ItsNat and MSIE v9.

---

[100] Future versions of these browsers will support synchronous requests but the initial versions supported by ItsNat do not.

[101] http://msdn.microsoft.com/en-us/library/cc288325%28VS.85%29.aspx

[102] http://hsivonen.iki.fi/doctype/

6.40.6 Other issues

ItsNat makes a strong effort to overcome browser limitations for instance generating the appropiated JavaScript code adapted to the target browser.

This is not an exhaustive list, you should test your ItsNat web applications against the target browsers, and some knowledge of browser limitations is necessary, anyway you will be surprised how many issues and limitations of some browsers are automatically fixed by ItsNat.

6.41 EXTERNAL JAVASCRIPT LIBRARIES AND BROWSER EXTENSIONS

ItsNat is a server centric framework with full control of the client from server, because the client is a copy of the server state and client DOM must be in sync with DOM in server.

In a first insight ItsNat does not seem very tolerant to JavaScript frameworks, browser extensions or in general any user JavaScript code or external technology modifying the client DOM. This is partially true, if these changes are controlled, ItsNat can coexist and collaborate with non-ItsNat JavaScript code doing DOM modifications only in client, the result is an interesting new kind of hybrid Single Page Interface Server-Centric and Client-Centric web applications/sites with the best of both worlds.

6.41.1 Tolerance to DOM modifications only in client in some locations

As of version 1.1 ItsNat adds tolerance (elements ignored) to:

1. Nodes inserted before, after or between `<head>` and `<body>` elements.

2. Nodes inserted into and at the end of `<head>` and `<body>` elements.

3. Nodes inserted into and at the end of the root element of non-X/HTML documents (SVG, XUL…).

Many JavaScript libraries and browser extensions (for instance the popular tool FireBug) add auxiliary nodes at these positions. In previous versions of ItsNat, when new nodes were added by ItsNat after "intrusive" nodes added at these locations, ItsNat failed accessing them for any subsequent task (including removing) because intrusive nodes broke node path calculus. Now thanks to automatic node caching ItsNat can ignore intrusive nodes in these locations.

This tolerance is useful for some ItsNat components like `ItsNatModalLayer` or `ItsNatHTMLIFrame`/`HTMLIFrameFileUpload`, both automatically adds to the end of `<body>` (or root element for non-X/HTML) the modal layer element and the `<iframe>` element if these element are not already into the document. To change this default behavior you can insert before these element where you want avoiding working in the end, in summary with some care you do not need to add with ItsNat new nodes to end of markup containers, anyway if you do so ItsNat tries to avoid problems with intrusive nodes.

ItsNat tolerance to intrusive nodes at the end of `<head>`, `<body>` or non-X/HTML root elements may not work with text nodes with content (including some text) added to the end of these elements, avoid this practice[103].

6.41.2 Tolerance to DOM modifications only in client in other locations

ItsNat and JavaScript libraries can coexist if the DOM subtree in client being managed by JavaScript frameworks is not being used in server, for instance, this subtree is static in server or not being used after an initial setup with ItsNat. Be careful with your external JavaScript code, you must to know what is exactly doing to avoid collisions with ItsNat (in summary the client DOM is not the expected by ItsNat).

If the client-centric markup zone is a cached zone in server (marked as `nocache=”true”`) there is no problem with client JavaScript because these nodes are not in server, in fact ItsNat can be useful to render the initial state of the zone going to be managed by non-ItsNat JavaScript code.

If the DOM has been used in server before delegating the client code, this zone is no longer in sync with client so you can remove only in server to save memory calling to the method `ItsNatDocument.`**`disconnectChildNodesFromClient`**`(Node)`.

Some JavaScript libraries like Google Analytics use `document.write(…)` on load time, a good thing of this technique of adding markup is the markup is added to the end of the already loaded markup of the page. To avoid problems with this type of code generation, just add a simple <span> or similar wrapping the `<script>` element. For instance (Google Analytics):

```
<span>
 <script type="text/javascript">
  var gaJsHost = (("https:" == document.location.protocol) ? "https://ssl." :
"http://www.");
  document.write(unescape("%3Cscript src='" + gaJsHost + "google-
analytics.com/ga.js' type='text/javascript'%3E%3C/script%3E"));
 </script>
</span>
```

6.41.3 Tolerance to modifications of attributes

Changes of opacity, size and visibility only need to change the `style` and `class` attributes, some prominent JavaScript libraries has built in beautiful animations playing only with these attributes. ItsNat has no problem with these attributes if they are changed only in client, you must know they are no longer in sync with server; to sync again just change the attribute value to a different value in server and the client will be updated (if you only re-set the same attribute value in server, there is no action because nothing changes in server). The same for any other attribute.

6.41.4 Conditional comments in MSIE

Internet Explorer has a sort of conditional macros using comments[104], for instance:

[103] Inserting text nodes with content as direct child nodes of `<body>` is not a good practice in web design, there many container elements for them (`<p>`, `<div>`, `<span>`…).

```
<!--[if IE 6]>
Special instructions for IE 6 here<br />
<![endif]-->
```

This kind of MSIE extension to HTML standard can be very harmful in ItsNat because comments are no processed in server. Notwithstanding with some care we can use these macros in ItsNat.

To calculate node locations, ItsNat "counts" sibling nodes including comments and ignoring text nodes, if the comment is replaced as a single element the count is not broken.

For instance avoid this practice (in `<head>`):

```
<!--[if lt IE 7]>
<link rel="stylesheet" href="ie6.css" type="text/css" />
<![endif]-->
```

Because in IE 7 or upper the comment is replaced by nothing (the count is broken).

Instead use something like:

```
<!--[if lt IE 7]>
<link rel="stylesheet" href="ie6.css" type="text/css" />
<link rel="stylesheet" href="dummy.css" type="text/css" />
<![endif]-->
<!--[if gte IE 7]>
<link rel="stylesheet" href="dummy.css" type="text/css" />
<link rel="stylesheet" href="dummy.css" type="text/css" />
<![endif]-->
```

Two comments before processing in MSIE, one comment is ever removed and the other is ever replaced by two `<link>`, total two significative nodes ignoring text nodes.

[104] http://msdn.microsoft.com/en-us/library/ms537512%28v=vs.85%29.aspx

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/she 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 `<input>` control in browser 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`, there are some differences for instance they are all interfaces, implementations are not public, public methods of `ItsNat*UI` interfaces are mainly for querying the visual (DOM) state of the component, DOM management is usually driven by data models like in Swing. ItsNat reuses 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) the 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 be replaced by another node (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 (normal and toggle), text based components, lists (multiple selection and combo boxes), tables, trees, custom/user defined etc.

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 or a list of XUL `<label>` 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[105]:

```
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

[105] A button component must not receive click and mousedown/mouseup events at the same time because is "confused" (processed as two clicks).

Normal buttons are buttons with no saved state (unlike toggle buttons).

All normal buttons implements `ItsNatButtonNormal`:

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"][106]>	**ItsNatFreeButtonNormal**
<*any* [itsnat:compType="freeButtonNormalLabel"][107]>	**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`")[108]:

```
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();
```

[106] If component type attribute is not specified, it must be specified explicitly in Java.

[107] If component type attribute is not specified, it must be specified explicitly in Java.

[108] The `ButtonModel` has support of `ItemListener` listeners but these have no sense with normal buttons because these buttons are not "selected".

```
    ActionListener actListener = new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            System.out.println("Clicked :" + e);
        }
    };
    dataModel.addActionListener(actListener);

    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="formattedTextField"]>`	**`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 `org.w3c.dom.events.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 do the same as DOM `HTMLInputElement` counterparts in non-internal mode (`ItsNatNode.`**`isInternalMode`**`()` returns false). When 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 kept 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**()[109]). 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);
```

[109] An event can be cancelled in SYNC mode only

```
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);
```

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)
```

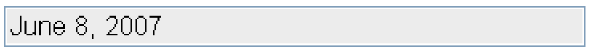
5) Specifying an `ItsNatFormatterFactoryDefault` factory

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[110], 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)`

[110] In fact label markup content is replaced with the text box.

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();

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()
```

In spite of edition can be used with ItsNatHTMLLabel components, edition is disabled by default because in-place edition changes the normal behavior of a <label> element. If you do not need the built-in functionality of <label> (the associated form control is clicked if the label is clicked) and you need in-place edition, use another HTML element (div, p etc) and ItsNatFreeLabel.

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.

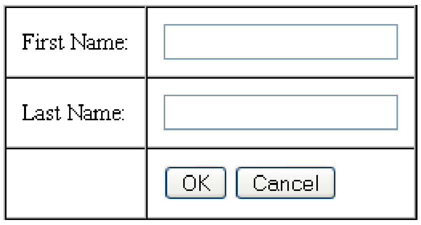
Person class source code:

```
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.getItsNatDocumentTemplate().getItsNatServlet();
        ItsNatDocFragmentTemplate docFragTemplate =
                        servlet.getItsNatDocFragmentTemplate(
                                "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 maybe to edit) a SVG (or XUL) element inside a XHTML document.

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[111].

[111] 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()`).

The following example shows a list of Spanish cities using a combo box:

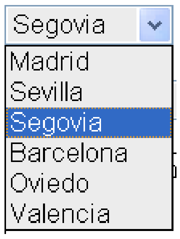
```
<select id="compId" />
```

```
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");

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);
```

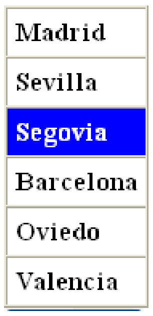
By default the click event changes the selected item (the component selects the item clicked). Now there is no "automatic" selection decoration like in a combo box control and ItsNat does not impose a default decoration; our example uses a custom ItemListener to decorate the selected item (or to remove 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>
```

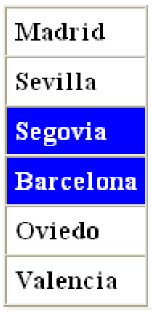
The Java code:

```
    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:


Madrid
Sevilla
Segovia
Barcelona
Oviedo
Valencia

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[112] 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:

[112] 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.

```
    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.

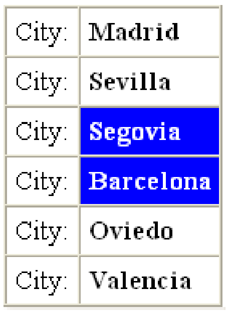
User defined structures must be provided to the component in creation time by using "artifacts" and can not be replaced.

Artifacts can be declared using `NameValue` objects, this class is 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

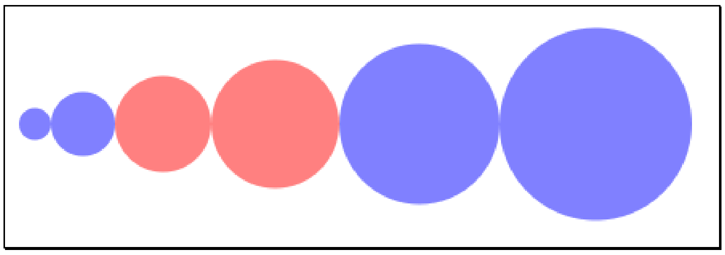
Another alternative to define the structure is using a special attribute in markup. This will be explained on "Specifying a user defined structure using markup and artifacts"

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 or XUL; a mouse click can be used to select a SVG or XUL based item list.

The ItsNat "Feature Showcase" shows a "live" SVG element list example, a circle list where every circle can be selected, removed, added etc. For instance, the following 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);[113]

    tableComp.setRowSelectionAllowed(true);
    tableComp.setColumnSelectionAllowed(false);
```

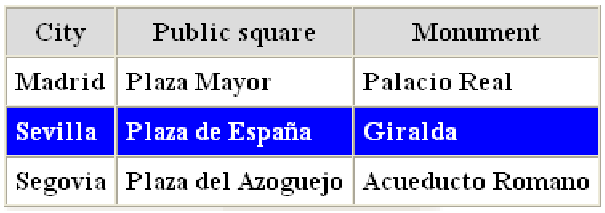
The `click` event changes the selected rows, ItsNat detects if the SHIFT and CRTL keys are pressed, selection behaviour is the same as Swing. ItsNat does not impose a default decoration; our example uses a custom `ListSelectionListener` to decorate the selected/unselected rows:

```
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();

        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);
```

[113] This sentence is not really needed because column selection is disabled

```
            }
        }

        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

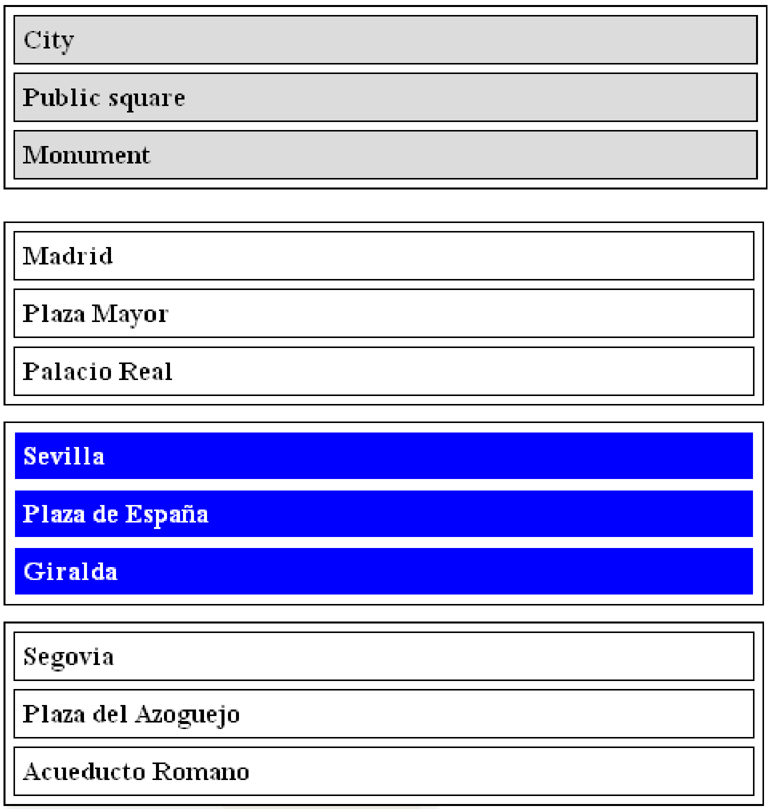
`ItsNatTable` supports "free element tables" using `ItsNatFreeTable`. The following markup shows an example of a "table" without the HTML `<table>` element:

```
    <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`[114].

The following image shows how is rendered using the default renderer:



The default structure of the table component automatically recognizes non-HTML head-less tables, for instance:

```
    <div id="compId">
        <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 first and only child element of the root element is identified as a row pattern.

7.10.3 User defined renderers and editors

[114] 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[115], 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.

Another alternative to define the structure is using a special attribute in markup. This will be explained on "Specifying a user defined structure using markup and artifacts"

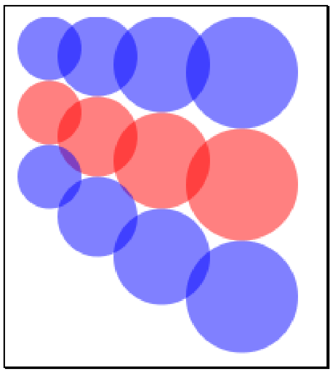
7.10.5 Free tables with non-HTML elements

[115] Use ever a `<tbody>` element in tables, by default ItsNat adds a `<tbody>` element if missing

`ItsNatFreeTable` is not (X)HTML namespace dependent, and may be used with other namespaces like SVG or XUL.

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 following HTML code is the pattern used to render a circle table with one circle row "selected":

```
<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[116]). `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"]	**ItsNatFreeTree**	**true**

[116] 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.

[itsnat:treeTable="true"] >		

The following example constructs a tree with the characters of the famous TV series Grey's Anatomy, using a tree component:

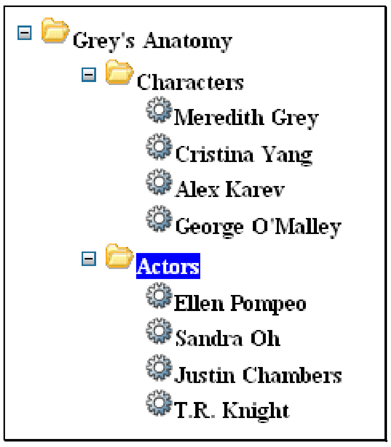
```
    <ul id="compId" style="list-style-type: none;">
        <li><span><img src="img/tree/tree_node_expanded.gif"><img
src="img/tree/tree_folder_open.gif"><span><b>Item Pattern</b></span></span>
            <ul style="list-style-type: none;"></ul>
        </li>
    </ul>
```

```
    ItsNatDocument itsNatDoc = ...;
    ItsNatComponentManager componentMgr = itsNatDoc.getItsNatComponentManager();

    ItsNatFreeTree treeComp =
    (ItsNatFreeTree)componentMgr.createItsNatComponentById("compId","freeTree",null);

    new FreeTreeDecorator(treeComp).bind();

    DefaultTreeModel dataModel = (DefaultTreeModel)treeComp.getTreeModel();

    DefaultMutableTreeNode rootNode = new DefaultMutableTreeNode("Grey's Anatomy");
    dataModel.setRoot(rootNode);

    DefaultMutableTreeNode parentNode;

        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:


Grey's Anatomy
Characters
Meredith Grey
Cristina Yang
Alex Karev
George O'Malley
Actors
Ellen Pompeo
Sandra Oh
Justin Chambers
T.R. Knight

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[117]. 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)
```

[117] A node may be the parent of a subtree

```
    {
        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);
        /* 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[118]. 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*[119] , 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);
```

[118] This behavior is applied in lists, tables and trees.

[119] This behavior is the same as in DefaultListModel, DefaultTableModel etc

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
}

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);
```

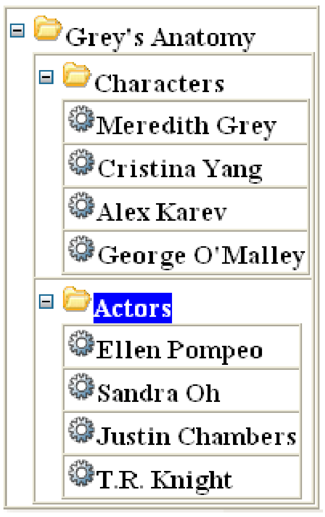
A node is selected (or unselected) when is clicked; ItsNat detects if the SHIFT and CRTL keys are pressed, selection behaviour is the same as Swing. ItsNat trees support all selection modes.

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[120]:



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 normal tree factory method, parameter `rootless`:

```
ItsNatComponentManager.createItsNatFreeTree(Element element,
    boolean treeTable,boolean rootless,
    ItsNatTreeStructure structure,NameValue[] artefacts)
```

2. Using the artifact "`rootless`" with value "`true`".

```
...
NameValue[] artifacts = new NameValue[]{new NameValue("rootless","true")};
ItsNatFreeTree treeComp = (ItsNatFreeTree)compMgr.createItsNatComponentById(
                   "compId","freeTree",artifacts);
...
```

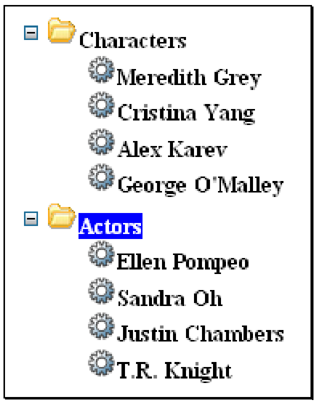
3. Using the markup attribute `itsnat:`**`rootless`**`="true"`

The previous 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:

[120] Table elements have borders to show the table based layout, remove this border in a production version if you do not like it.



7.11.3 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 normal tree factory method, parameter `treeTable`:

```
ItsNatComponentManager.createItsNatFreeTree(Element element,
   boolean treeTable,boolean rootless,
   ItsNatTreeStructure structure,NameValue[] artefacts)
```

2. Using the artifact "`treeTable`" with value "`true`".

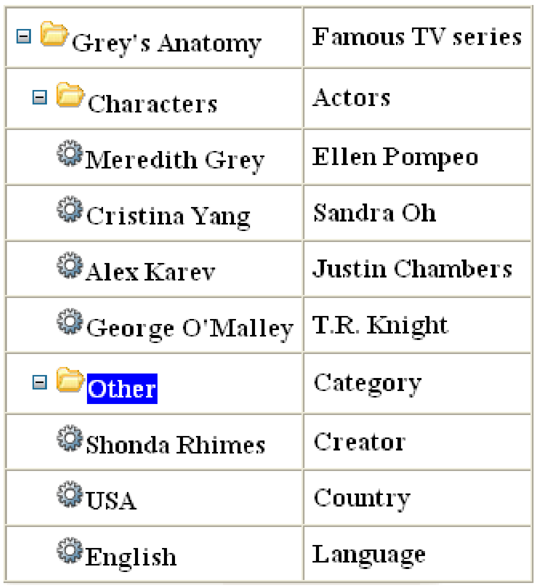
```
...
NameValue[] artifacts = new NameValue[]{new NameValue("treeTable","true")};
ItsNatFreeTree treeComp = (ItsNatFreeTree)compMgr.createItsNatComponentById(
                    "compId","freeTree",artifacts);
...
```

3. Using the markup attribute `itsnat:treeTable="true"`

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):

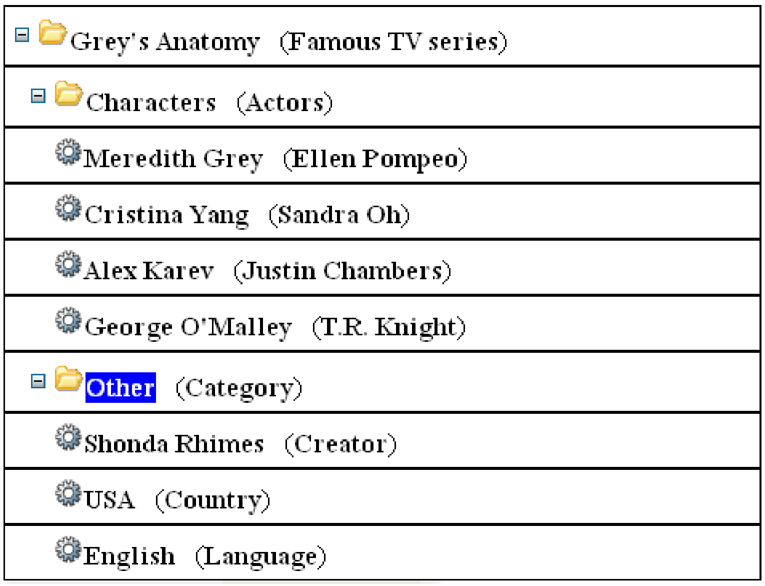


⊟ Grey's Anatomy	Famous TV series
⊟ Characters	Actors
Meredith Grey	Ellen Pompeo
Cristina Yang	Sandra Oh
Alex Karev	Justin Chambers
George O'Malley	T.R. Knight
⊟ Other	Category
Shonda Rhimes	Creator
USA	Country
English	Language

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 `NameValue` objects (artifacts) 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.

Another alternative to define the structure is using a special attribute in markup. This will be explained on "Specifying a user defined structure using markup and artifacts".

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 or XUL[121].

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 AJAX synchronous mode (see "Asynchronous-Hold Modes" chapter).

There is another use, `ItsNatHTMLForm` implements `submit()` and `reset()` as remote methods, if called these calls are sent to the client as JavaScript. These methods do the same

[121] This component do not fit well with the built-in <tree> control of XUL

as DOM `HTMLInputElement` counterparts in non-internal mode (`ItsNatNode.`**`isInternalMode`**`()` returns false).

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,CommMode.XHR_SYNC,null,null,-1);
formComp.setEventListenerParams("reset",false,CommMode.XHR_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 IFRAMES FOR FILE UPLOADING

In the AJAX world submitting an `<input type="file">` element inside a form with target a hidden `<iframe>` is the most popular trick to upload files, of course this technique is also used in ItsNat.

In ItsNat the content of the `<iframe>` can optionally be used to show some information about the uploading process, hence this element is the centre of this task and must be in the document during uploading (and file uploading may be a very long task), otherwise if the `<iframe>` is removed before the uploading task takes place, the uploading process could be not started. Therefore the component `ItsNatHTMLIFrame` attached to the provided `<iframe>` will be the originator of file uploading tasks.

We also need an `<input type="file">` element in the document, in this element the end user provides the file to upload, in spite of this element also makes file uploading possible, this element is auxiliary because is just being used when the form is submitted, immediately after the uploading process starts this element can be removed. The same for the `<form>` element, in this case this element is created temporally and removed by ItsNat when uploading starts.

In summary, developers must to provide an `<input type="file">` and an `<iframe>` for file uploads. The `<iframe>` element must have a `name` attribute in load time and before insertion and cannot change, otherwise file uploading fails (this is imposed by browsers and have to do with security); this `name` attribute is not needed for `<input type="file">` because if not defined ItsNat defines it temporally when the form is submitted.

The following markup is the minimal required to upload files:

```
    <input type="file" /><iframe name="fileUpload" src="about:blank" />
```

A real world example requires more work, the following code shows how to upload files provided by end users and notifies the uploading progression by using an ItsNat AJAX (or SCRIPT) timer:

```
    <input id="fileUploadInputId" type="file" /><br />
    <iframe name="fileUpload" src="about:blank" id="fileUploadIFrameId" /><br />
    <button id="fileUploadButtonId">Upload File</button><br />
    <p>Progress: <b id="progressId">0</b>%</p>
```

Java code:

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

ItsNatComponentManager compMgr = itsNatDoc.getItsNatComponentManager();
final ItsNatHTMLInputFile input =
        (ItsNatHTMLInputFile)compMgr.createItsNatComponentById("fileUploadInputId");
final ItsNatHTMLIFrame iframe =
        (ItsNatHTMLIFrame)compMgr.createItsNatComponentById("fileUploadIFrameId");
final Element progressElem = doc.getElementById("progressId");

ItsNatHTMLButton button =
        (ItsNatHTMLButton)compMgr.createItsNatComponentById("fileUploadButtonId");
```

```
EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        ClientDocument clientDoc = ((ItsNatEvent)evt).getClientDocument();
        ItsNatTimer timer = clientDoc.createItsNatTimer();
        EventListener timerListener = new EventListener() {
            public void handleEvent(Event evt) { }
                 // Nothing to do, this timer just update the client
                 // with the current state of progressElem
        };
        final ItsNatTimerHandle timerHnd = timer.schedule(null,timerListener,0,1000);

        final HTMLIFrameFileUpload iframeUpload =
                 iframe.getHTMLIFrameFileUpload(clientDoc,input.getHTMLInputElement());

        ItsNatServletRequestListener listener = new ItsNatServletRequestListener()
        {
            public void processRequest(ItsNatServletRequest request,
                                            ItsNatServletResponse response)
            {
                FileUploadRequest fileUpReq =
                          iframeUpload.processFileUploadRequest(request, response);

                try
                {
                    ServletResponse servRes = response.getServletResponse();
                    Writer out = servRes.getWriter();
                    out.write("<html><head /><body>");
                    out.write("<p>Content Type: \"" + fileUpReq.getContentType() +
                              "\"</p>");
                    out.write("<p>Field Name: \"" + fileUpReq.getFieldName() + "\"</p>");
                    out.write("<p>File Name: \"" + fileUpReq.getFileName() + "\"</p>");
                    out.write("<p>File Size: " + fileUpReq.getFileSize() + "</p>");
                    out.write("</body></html>");

                    long fileSize = fileUpReq.getFileSize();
                    if (fileSize == 0) return;

                    byte[] buffer = new byte[10*1024];
                    InputStream fileUp = fileUpReq.getFileUploadInputStream();
                    long count = 0;
                    int read = 0;
                    do
                    {
                        if (iframeUpload.isDisposed()) return;

                        try { Thread.sleep(50); }catch(InterruptedException ex){ }
                        count += read;

                        long per = (count * 100) / fileSize;
                        synchronized(itsNatDoc)
                        {
                            Text text = (Text)progressElem.getFirstChild();
                            text.setData(String.valueOf(per));
                        }

                        read = fileUp.read(buffer);
                    }
                    while (read != -1);
                }
                catch(IOException ex)
                {
                    throw new RuntimeException(ex);
                }
                finally
                {
                    synchronized(itsNatDoc)
                    {
                        timerHnd.cancel();
                    }
                }
            }
        };
        iframeUpload.addItsNatServletRequestListener(listener);
    }
};
button.addEventListener("click", listener);
```

In this example file uploading is started when the end user clicks the "Upload File" button.

This event starts an ItsNat timer (`ItsNatTimer` object), this timer touches the server to update the client to be in sync with server, in this case the element with id "`progressId`", showing the percentage already uploaded is going to be changed every 10Kb (as you know every change is queued in server until an ItsNat event returns to the client updating the client DOM).

The next sentence initiates the file upload processing (`iframe` reference is an `ItsNatHTMLIFrame` component):

```
        final HTMLIFrameFileUpload iframeUpload =
                 iframe.getHTMLIFrameFileUpload(clientDoc,input.getHTMLInputElement());
```

This call returns an `HTMLIFrameFileUpload` object which generates the required JavaScript code to send the file specified by user in client to the server.

Because this code is going to be sent to the client at the end of the request, we have to add the listener going to receive the form submission, remember that this form submits with target our `iframe`, therefore the answer can be a normal HTML page (the default) going to be loaded by the `iframe`.

```
        ItsNatServletRequestListener listener = new ItsNatServletRequestListener()
        {
            public void processRequest(ItsNatServletRequest request,
                                       ItsNatServletResponse response)
            {
              ...
            }
        };
        iframeUpload.addItsNatServletRequestListener(listener);
```

The file upload request *is not* the typical load process of an `ItsNatDocument`, there is no new `ItsNatDocument`, in fact `ItsNatServletRequest.`**`getItsNatDocument`**`()` returns the parent document, this request is basically the same as a normal raw servlet request, is up to the developer to return the appropriated HTML page or none if the `iframe` is hidden. Our example returns a page with informative data about the file uploaded.

```
                      ServletResponse servRes = response.getServletResponse();
                      Writer out = servRes.getWriter();
                      out.write("<html><head /><body>");
                      out.write("<p>Content Type: \"" + fileUpReq.getContentType() +
                                "\"</p>");
                      ...
                      out.write("</body></html>");
```

Also no request processing is done by default to read the file submitted, you can manually process the request parsing headers and file data with custom code or delegating to well established frameworks like Apache Commons FileUpload[122].

ItsNat can optionally do this stuff for you calling `HTMLIFrameFileUpload.`**`processFileUploadRequest`**`(ItsNatServletRequest, ItsNatServletResponse)`, this method returns a `FileUploadRequest` object which parsers the request and exposes the file being uploaded as an `InputStream` (calling `FileUploadRequest.`**`getFileUploadInputStream`**`()`).

```
FileUploadRequest fileUpReq =
      iframeUpload.processFileUploadRequest(request, response);
```

Because file uploading may be a very long task and takes place in a different "page" (an `iframe`) the thread of the web request uploading the file is not synchronized with the parent `ItsNatDocument`, any access to the `ItsNatDocument` or dependent objects must be synchronized before (for instance to cancel the timer). This explains why we synchronize the code changing the "`progressId`" element and timer cancelling (when file uploading finishes):

```
        synchronized(itsNatDoc)
        {
            Text text = (Text)progressElem.getFirstChild();
            text.setData(String.valueOf(per));
        }
…
    synchronized(itsNatDoc)
    {
        timerHnd.cancel();
    }
```

When the uploading process finishes the `HTMLIFrameFileUpload` object is automatically disposed and cannot be reused.

Of course the `<iframe>` element may be hidden and dynamically inserted when the file upload is requested, and removed when file uploading finishes.

7.15 MODAL LAYERS

ItsNat strongly promotes one single web page applications, applications with no reload similar to the typical desktop application based on a single window. In desktop applications usually the main window contains two types of child windows:

- Non-modal windows: they are usually stacked and opaque windows, they do not prevent windows behind to be clicked outside of the area occupied by the window on top.

- Modal windows: they are stacked and opaque too, the main difference is they avoid clicking on the windows behind including outside of the area occupied by the modal window. In some way a modal window can be seen as a non-modal window on top of a *modal layer*, this modal layer fully covers the main window area and captures any event;

[122] http://commons.apache.org/fileupload/

these events can not be received by windows "behind" the modal layer. This modal layer can have any background color and can be transparent, semitransparent or opaque.

ItsNat simulates modal layers using a specific component, which interface is `ItsNatModalLayer`.

In ItsNat a modal layer is an absolute positioned element (by default a `<div>`) with a background color or pattern opaque, translucent or transparent with a z-index greater than the elements being covered and covering the complete area of the web page.

When an `ItsNatModalLayer` component is created, the specified element to be used as modal layer (if not specified a `<div>` is created) is added to the DOM tree as a direct child in the end of the visual root element of the document (`<body>` in X/HTML documents).

Automatically ItsNat adjusts the width and height of this element to the extension of the document, and automatically readjusts them in a timely fashion when the document extension changes (the content has changed or the user has resized the browser window).

The objective is to prevent that elements with a lower z-index can be clicked (accessed in general).

Modal layers can be customized with the CSS properties opacity and background passed as parameters of the factory method:

```
ItsNatComponentManager.createItsNatModalLayer(Element element,
                 boolean cleanBelow,int zIndex,float opacity,
                 String background,NameValue[] artifacts)
```

ItsNat knows about the opacity support of the target browser (for instance in MSIE a filter is used), in browsers with no opacity (many mobile browsers) an opacity value <0.5 means transparent else opaque.

Because of opacity, a modal layer element should not contain child elements, because opacity is applied to any contained node too (for instance if opacity is 0 the modal layer is fully transparent including child nodes).

To create a "modal window" adds to the document a new absolute positioned element with a z-index equal or greater than the z-index of the modal layer (an equal value is recommended if you want automatically generated z-index values). Added to the end of the `<body>` element (as a direct child) is the recommended practice. This new absolute positioned element is the non-modal window on top of the modal layer described before.

Modal layers work in remote view/control too, modal layer extension and refresh is calculated and done in a per browser basis, and a new remote view/control view can be added with no problem when a modal window is already shown on the monitored client.

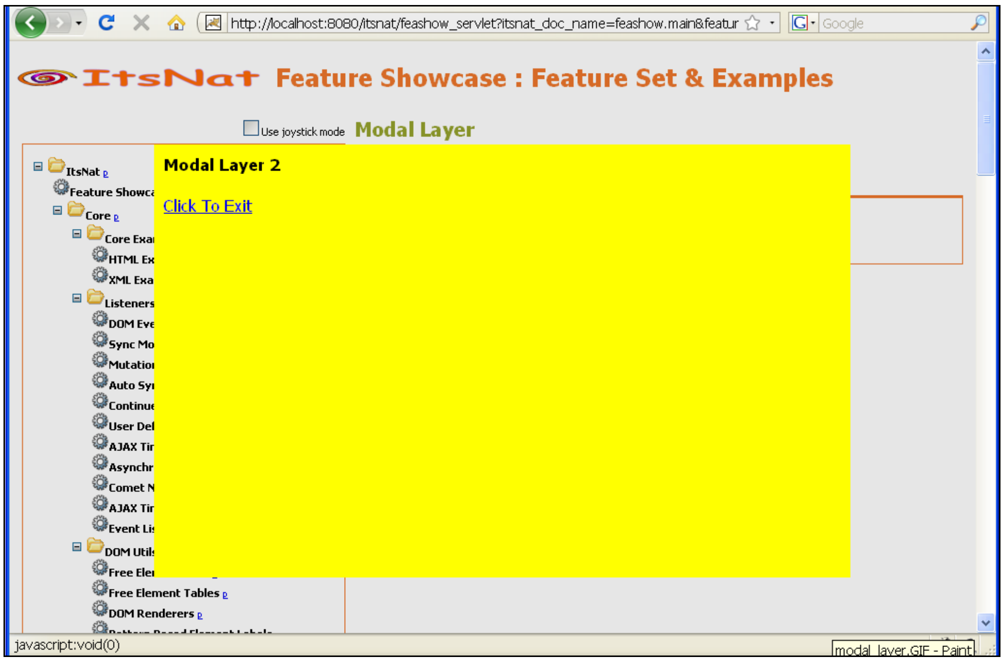
Finally the component automatically removes the modal layer element from the document when is disposed.

The following example shows how to create a semitransparent modal layer on top of the current content and how to add a new element with some content on top of the modal layer:

```
final ItsNatDocument itsNatDoc = ...;
float opacity = (float)0.1; // Semitransparent
String background = "black";

ItsNatComponentManager compMgr = itsNatDoc.getItsNatComponentManager();
```

```
    final ItsNatModalLayer modalLayer =
        compMgr.createItsNatModalLayer(null,false,opacity,background,null);
    int zIndex = modalLayer.getZIndex();

    // Note:  right:X%; alongside width and left fools BlackBerry
    String code = "<p style='position:absolute; z-index:" + zIndex + ";
background:yellow; width:70%; height:70%; left:15%; top:15%; padding:10px;'>" +
                  "<b>Modal Layer 2</b><br /><br />" +
                  "<a href='javascript:;'>Click To Exit</a>" +
                  "</p>";

    final HTMLDocument doc = (HTMLDocument)itsNatDoc.getDocument();
    DocumentFragment frag = itsNatDoc.toDOM(code);
    final Element elem = (Element)frag.getFirstChild();
    doc.getBody().appendChild(elem);

    NodeList links = elem.getElementsByTagName("a");
    final HTMLAnchorElement linkExit = (HTMLAnchorElement)links.item(0);

    EventListener listenerExit = new EventListener()
    {
        public void handleEvent(Event evt)
        {
            ((EventTarget)linkExit).removeEventListener("click",this,false);
            doc.getBody().removeChild(elem);
            modalLayer.dispose();
        }
    };

    ((EventTarget)linkExit).addEventListener("click",listenerExit,false);
```

Visual result:


http://localhost:8080/itsnat/feashow_servlet?itsnat_doc_name=feashow.main&featur
ItsNat Feature Showcase : Feature Set & Examples
Use joystick mode
Modal Layer
Modal Layer 2
Click To Exit
javascript:void(0)

The Feature Showcase contains an example of two stacked modal windows.

7.15.1 Browser issues

Some browsers have problems with form controls ignoring the z-index property, therefore can be accessed (for instance clicked) behind a modal layer. For instance `<select>` elements in Internet Explorer 6[123].

ItsNat knows this fact and selectively hides the problematic form controls according to the concrete browser. ItsNat internally and automatically stacks modal layers, this way ItsNat knows what form elements were hidden when a new modal layer was added to be show again when the modal layer is disposed.

Another important problem is browsers with buggy or no support of absolute positioning like Opera Mini 4.x. For instance in Opera Mini all elements with associated listeners can be clicked ignoring the z-index rule. In these browsers the "clean layer" technique is ever used. This is not a problem in remote view/control, a client monitoring a Opera Mini sees modal windows as usual (not using the "clean layer" technique) and a Opera Mini monitoring any other browser sees the modal windows using the "clean layer" technique. The clean layer technique used in these browsers supports stacked modal layers.

7.15.2 Detection of unexpected events received by hidden elements

In spite of ItsNat makes a strong effort to avoid user events to be dispatched to elements "behind" modal layers, is not fully achieved, in fact no web framework has resolved this problem. For instance, desktop browsers usually work fine related to z-index support, opacity and so on, nevertheless they all have a hole, you can navigate to any hidden form element or link using the TAB key, and "click" this element pressing the ENTER key.

In any Single Page Interface application this may be a serious issue because it can break the normal flow of the "state machine" of the web application, that is, the application can be "broken".

ItsNat offers the possibility of detecting when an event is being dispatched to server listeners of elements "hidden" by the modal layer, the developer is free to inform the end user of this situation (for instance forcing or inviting to reload the web application) or, if possible, the event could be stopped avoiding the default behavior (calling `ItsNatEventListenerChain.`**`stop`**`()`). The latter solution is NOT recommended because the client-server synchronization can be broken; for instance, if a "hidden" <select> control is "clicked", the client state may be changed, if this form control is bound to the server using the associated ItsNat component, a change event will be sent to the server, if this event is stopped before arriving to the server select component, the server state will be different to the client state.

The following example shows how to instruct the end user to reload the web application when an unexpected event is received:

```
final ItsNatDocument itsNatDoc = ...;
ItsNatComponentManager compMgr = itsNatDoc.getItsNatComponentManager();
final ItsNatModalLayer modalLayer =
    compMgr.createItsNatModalLayer(null,false,0.1f,"black",null);

EventListener listener = new EventListener()
{
    public void handleEvent(Event evt)
    {
        StringBuffer code = new StringBuffer();
```

[123] MSIE 6 is not the only browser, BlackBerry, Android… have some problematic form controls.

```
            code.append("if (confirm(
                'Received an unexpected event by a hidden element. Reload?')) ");
            code.append("  window.location.reload(true);");
            itsNatDoc.addCodeToSend(code.toString());
        }
    };
    modalLayer.addUnexpectedEventListener(listener);
```

This kind of event listener must be registered to *every* modal layer because only elements "hidden" by each modal layer are detected, that is, elements not hidden by the previous modal layer are the elements hidden by the next modal layer stacked.

7.15.3 Modal layers in SVG

Modal layers work in pure SVG documents, behaviour is very similar to X/HTML, the main differences are:

1. Z-index is ignored in SVG because z-order in SVG follows the rendering order, this order coincides with document order[124].
2. The visual root element is `<svg>`.
3. A `<rect>` element is the default element used as modal layer.
4. Opacity parameter is used as the value of `fill-opacity` property.
5. Background parameter is the value of `fill` property.

The Feature Showcase contains an example of using `ItsNatModalLayer` components in SVG.

7.15.4 Modal Layers in XUL

By default the component uses a `<panel>` element as layer, this element must be used as the container of any user markup shown "on top" of the layer. If `popuphidden` event is enabled calling `ItsNatComponent.enableEventListener(String)` or adding an event listener listening to the component for this event (calling `ItsNatComponent.addEventListener(String,EventListener,boolean)`), the component will be automatically disposed when an event is received (in XUL the `popuphidden` event closes the panel, by this way the component is "closed" in server too).

In XUL documents z-index is ignored because XUL ignores the z-index CSS property.

In spite of ItsNat supports stacked modal layers only one modal layer is recommended because the only the first `<panel>` is modal (any click outside the panel closes the panel), the `<panel>` area is not locked when a new `<panel>` is shown on top. An alternative is to use the "clean below" technique for the second and successive layers.

The Feature Showcase has a complete example of modal layers using XUL.

7.15.5 Clean Below mode (Clean Layers)

[124] See http://wiki.svg.org/Rendering_Order

ItsNat makes a strong effort to avoid clicking on elements behind modal layers, though there is a breach, some elements can be accessed using the tab key and clicked using the enter key. This is usually acceptable, in fact, no JavaScript framework seems to get rid of this problem.

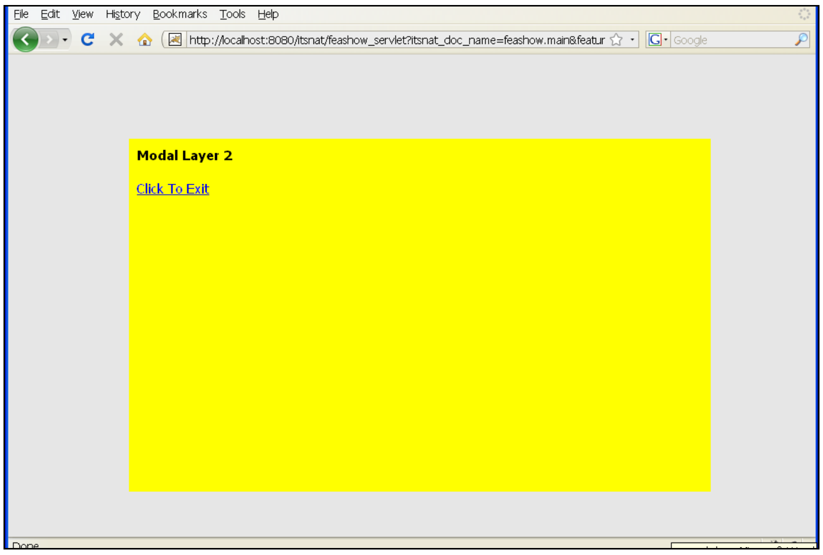
Anyway to avoid this problem we can use the *clean below* mode. The "clean below" technique, or clean layer, is simple, any direct child node of `<body>` is hidden setting `display` CSS property to `none` but not the modal layer just inserted. Normal visualization is restored when the modal layer is disposed.

To enable clean below set the parameter `cleanBelow` as true in the factory method:

```
ItsNatComponentManager.createItsNatModalLayer(Element element,
                        boolean cleanBelow,int zIndex,float opacity,
                        String background,NameValue[] artifacts)
```

Again ItsNat supports stacked modal layers in "clean below" mode, in fact, mixed modes are supported.

The following image shows the previous example in clean below mode:


File Edit View History Bookmarks Tools Help
http://localhost:8080/itsnat/feashow_servlet?itsnat_doc_name=feashow.main&featur
Google
Done

Modal Layer 2

Click To Exit

7.15.6 Artifacts and attributes

Using `NameValue` objects and/or attributes in markup we can use methods like `ItsNatComponentManager.`**`createItsNatComponentById`**`(String)` to create modal layers. Available parameters (artifact or attribute names and expected values):

- `cleanBelow` : a `boolean` value. False by default.
- `zIndex` : an integer value. By default the number of previous modal layers +1.
- `opacity` : a float value between 0 and 1. By default is 1.
- `background` : a string. By default is "black".

More information in "AUTOMATIC COMPONENT BUILD" chapter.

7.16 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 (or configure the standard component before returning). 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 `ItsNatComponentManager`, `ItsNatDocumentTemplate` or globally on `ItsNatServlet`. For instance:

```
    String pathPrefix = ...;
    ItsNatHttpServlet itsNatServlet = getItsNatHttpServlet();
    ItsNatDocumentTemplate docTemplate;
    docTemplate = itsNatServlet.registerItsNatDocumentTemplate("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).

The `before` and `after` methods of registered listeners are called too when a specific standard factory method is called (for instance `ItsNatComponentManager.`**`createItsNatFreeLabel`**`(Element,NameValue[])`), this give an opportunity to change or configure the standard component before the factory method returns. As some specific factory methods support a null node as first parameter, the method `before` of user defined creation listeners must be tolerant to null nodes.

Following the example:

```
public class LoginCreationItsNatComponentListener
                implements CreateItsNatComponentListener
{
    public LoginCreationItsNatComponentListener()
    {
    }

    public ItsNatComponent before(Node node, String componentType,
                NameValue[] artifacts, ItsNatComponentManager compMgr)
    {
        if (node == null) return null;

        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`.

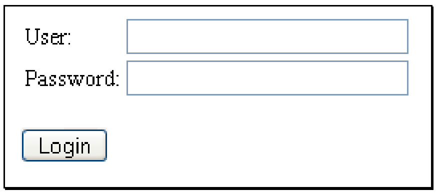
This is the `LoginComponent` source code:

```
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" component only accepts "`admin`" and "`1234`" as a valid login.

7.17 AUTOMATIC COMPONENT BUILD

Almost all components used in this manual were created explicitly using `ItsNatComponentManager.`**`createItsNatComponent`**`*` factory methods, depending on the node being used or the parameter `compType`, ItsNat creates the appropriated component. Specific factory methods, for instance `ItsNatComponentManager.`**`createItsNatFreeCheckBox`**`(Element, NameValue)`, can be used too to create a specific component.

Furthermore ItsNat provides an automatic mode based on markup, 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 `ItsNatComponentManager.`**`buildItsNatComponents`**`(Node)` uses `ItsNatComponentManager.`**`addItsNatComponent`**`(Node)`[125], 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 to automatic component construction using `buildItsNatComponents` or `addItsNatComponent`:

1) HTML form elements

2) HTML elements with a default associated component

3) Free elements

7.17.1 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="" />
```

[125] ItsNatComponentManager.addItsNatComponentById(String) is basically the same using Document.getElementById(String) to obtain the node.

```
    ItsNatHTMLInputTextFormatted inputTextFormat =
        (ItsNatHTMLInputTextFormatted)componentMgr.addItsNatComponentById(
                "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="" />
```

7.17.2 HTML elements with a default associated component

The following elements have a default component associated: `<a>`, `<label>`, `<table>`, `<iframe>`.

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");
```

7.17.3 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");
```

7.17.4 Specifying a user defined structure using markup and artifacts

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:

```
    ItsNatDocumentTemplate 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();
```

7.17.5 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".

7.17.6 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);
```

7.17.7 Excluding nodes of automatic component creation

We can exclude a specific node of the automatic component creation with the method ItsNatComponentManager.**addExcludedNodeAsItsNatComponent**(Node).

7.17.8 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 ItsNatDocumentTemplate.**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.

7.18 MARKUP DRIVEN MODE IN FORM BASED NODES

7.18.1 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).

7.18.2 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`**

7.18.3 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 `ItsNatDocumentTemplate`.

7.19 DISABLED EVENTS MODE AND COMPONENTES

ItsNat components are strongly based on events (transported with AJAX or SCRIPT elements) but they can work in disabled events mode, of course all event-related characteristics are disabled. In disabled events mode the `ItsNatDocument` is created per request, the same is applied to components; unlike other component based frameworks not using AJAX, 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.

7.20 DISABLED JAVASCRIPT MODE AND COMPONENTES

The disabled JavaScript mode is very similar to disabled events 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 disabled events mode.

Again the "Feature Showcase" contains a complete example with JavaScript disabled and using components. Is very similar to the disabled events version but in this case no JavaScript is used.

7.21 XML AND COMPONENTS

XML documents (excluding XHTML) are generated by ItsNat in a one shot without events (events are disabled), we know that components can be used in a disabled events mode, this also works for XML documents. 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.

7.22 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".

7.22.1 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:

```
ItsNatDocumentTemplate.isSelectionOnComponentsUsesKeyboard()
ItsNatDocumentTemplate.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.

7.22.2 Joystick mode

This mode is very useful on mobile browsers and devices where there is no touchable screen (or not practical) and/or 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"[126], elements returned by structure methods like:

[126] A "content element" is the effective parent element of the item markup

```
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)
```

8. STATELESS MODE

8.1 INTRODUCTION

ItsNat normal mode is stateful making use of server memory (and optionally session) to storing in server the browser state, when using multiple symmetric servers sticky sessions are required to send (AJAX/script) events ever to same server, unless session replication is enabled, in this case client state in server is replicated to servers in cluster.

Since v1.3 ItsNat provides a new working mode, the “stateless” mode. In stateless mode NO client state is kept in server, memory or session, when processing an (AJAX/script) client event the current browser state (ItsNatDocument) is fully reconstructed in server and modificated accordingly and modifications are sent to the browser as JavaScript as DOM operations as usual, the “page” loaded in server is not stored in server memory nor in session.

In stateless mode, AJAX/script requests are executed manually sending “stateless events” in some way similar (but different) to “custom events” in the stateful mode of ItsNat, the main difference is in server, there is no document in server memory with the same state in client ready to receive events, later we are going to see how we can build the document (or a part of) with the same state of the client, a phase of the stateless event processing, this new document instance will receive the stateless event to perform modifications in server to be automatically propagated as JavaScript code to the client following the typical DOM server to client sync approach we are used.

Because there is no document data stored in server by ItsNat, you can shutdown your servlet container and start again with a fresh instance (no session data restored) and stateless events still can be sent to the server by an already loaded stateless document by the previous instance of the servlet container.

By this way multiple symmetric servers can work together with no need of server memory or session data storage and node replication (anyway user code can make use of session for storing custom data as in any conventional classic web application).

Stateless mode can be SEO compatible because the initial page load is performed by ItsNat as usual using fast-mode load for SEO compatibility, and so you can generate plain HTML in load time of any possible client state executing the same Java DOM code.

This new mode does not make full use of ItsNat however many powerful features are still ready to be used in this mode like pure markup templating, powerful DOM manipulation for view logic and components (without AJAX/script events).

8.2 DEVELOPMENT LIFECYCLE AND EXAMPLE

8.2.1 Development lifecycle

Stateless mode is based on a per-document configuration, that is, there is no “global stateless mode”, if a ItsNat document is loaded stateless this document will be stateless ready to receive stateless events, in the same time and servlet other documents can be stateful with no problem of sharing.

This is the lifecycle on development a stateless document:

1) Register an ItsNat document template to be used as the initial page

   To ensure is not kept in server disable event support calling `ItsNatDocumentTemplate.setEventsEnabled(boolean)` with `false`. Because the document cannot receive client events is not kept in server memory nor in session.

2) Add Java code to generate the initial page accordingly

3) Register *other document templates to process stateless events*. In simple cases the template for the initial page can also be used to process stateless events.

4) Add Java code to recreate the client state and to process received stateless events sent by client

5) Add manually JavaScript event listeners to your required markup elements with code to send stateless events calling the ItsNat JavaScript method `dispatchEventStateless`. Stateless events must specify the ItsNat template going to receive them in server

8.2.2 Stateless example

Let's put everything together in a concrete example. Most of concepts of stateful use of ItsNat still apply but be ready to a new radically different way of using ItsNat.

First of all we create a new servlet:

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

    ItsNatHttpServlet itsNatServlet = getItsNatHttpServlet();

    //ItsNatServletConfig itsNatConfig = itsNatServlet.getItsNatServletConfig();
    itsNatServlet.addItsNatServletRequestListener(
            new StatelessGlobalDocumentLoadListener());
    itsNatServlet.addEventListener(new StlessGlobalEventListener());

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

    ItsNatDocumentTemplate docTemplate;
    docTemplate = itsNatServlet.registerItsNatDocumentTemplate(
       "manual.stless.example","text/html",
       pathPrefix + "stless_example.html");
    docTemplate.addItsNatServletRequestListener(
            new StlessExampleInitialDocLoadListener());
    docTemplate.setEventsEnabled(false); // Stateless

    docTemplate = itsNatServlet.registerItsNatDocumentTemplate(
       "manual.stless.example.eventReceiver","text/html",
       pathPrefix + "stless_example_event_receiver.html");
    docTemplate.addItsNatServletRequestListener(
            new StatelessExampleForProcessingEventDocLoadListener());
    docTemplate.setEventsEnabled(false); // Stateless

    ItsNatDocFragmentTemplate docFragDesc;
    docFragDesc = itsNatServlet.registerItsNatDocFragmentTemplate(
      "manual.stless.example.fragment","text/html",
      pathPrefix + "stless_example_fragment.html");
  }
}
```

The following code snippet registers the template of the initial page:

```
    ItsNatDocumentTemplate docTemplate;
    docTemplate = itsNatServlet.registerItsNatDocumentTemplate(
       "manual.stless.example","text/html",
       pathPrefix + "stless_example.html");
    docTemplate.addItsNatServletRequestListener(
            new StlessExampleInitialDocLoadListener());
    docTemplate.setEventsEnabled(false); // Stateless
```

Note the `setEventsEnabled(false)` call, this configuration method call disables client to server *stateful* remote events, because there is no need of a `ItsNatDocument` kept in server to receive events, ItsNat does not kept the document in memory or in session, this setting is necessary if you want a true stateless document.

The `stless_example.html` file:

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
xmlns:itsnat="http://itsnat.org/itsnat">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>ItsNat Stateless Example in Manual</title>
    <script>
    function sendEventStateless()
    {
        var counterElem = document.getElementById("counterId");
        var userEvt = document.getItsNatDoc().createEventStateless();
        userEvt.setExtraParam('counter',counterElem.firstChild.data);
        userEvt.setExtraParam('itsnat_doc_name','manual.stless.example.eventReceiver');
        document.getItsNatDoc().dispatchEventStateless(userEvt,3 /*XHR_ASYNC_HOLD*/, 1000000);
    }
    </script>

</head>
<body>

    <h3>ItsNat Stateless Example in Manual</h3>

    <h4 id="presentationId" itsnat:nocache="true"></h4>

    <br /><br />
    <a href="javascript:sendEventStateless()">Send stateless event</a>

    <br /><br />
    <div>
        <div>Num. Events: <b id="counterId" itsnat:nocache="true">0</b></div>
    </div>

    <div>
        <div>
            <div id="insertHereId" />
        </div>
    </div>

    <br />
</body>
</html>
```

This document template will be the initial page of our micro Single Page Interface Stateless example, remember this template is registered with *stateful* remote events disabled, but watching this JavaScript code line:

```
document.getItsNatDoc().dispatchEventStateless(userEvt,3 /*XHR_ASYNC_HOLD*/, 1000000);
```

We realize there is another kind of remote events: stateless events.

This method accepts three parameters:

1) An event object: following the same conventions as in custom events seen before in this manual

2) A transport mode: an integer with the same values of `org.itsnat.core.CommMode` constants. Most of the time you're going to pass 3 parameter (XHR_ASYNC_HOLD = AJAX asynchronous queued events).

3) Timeout in milliseconds

Transport mode and timeout parameters should be familiar to you because they are typically provided for configuration of a stateful document template.

This line is interesting:

```
userEvt.setExtraParam('itsnat_doc_name','manual.stless.example.eventReceiver');
```

ItsNat recognizes this standard "document load" parameter also in a stateless event, when received this stateless event in server, ItsNat server behavior is the following:

1) A new `ItsNatDocument` is created based on the specified template. This document instance is *stateless* no matter the template was configured (`setEventsEnabled(false)` is just recommended for clarity and to avoid stateful loads using conventional URLs)

2) Document load processing is as usual with a radical difference: *final DOM state when loading phase ends is not serialized as HTML and initial JavaScript code generated in this phase is ignored/lost*.

3) The global event listener registered in load phase calling `ItsNatDocument.addEventListener(EventListener)` is executed *receiving a stateless event*. DOM modifications performed in this phase are sent to the client as return of the event in a similar way of a stateful remote event.

Used to the conventional, stateful mode of ItsNat, this stateless behavior seems non-sense, however, it has a lot of sense…

The main purpose of stateless event processing is:

1) To reconstruct fully or partially the DOM client state in server: this is the purpose of the load phase, the final DOM state/tree of the document after loading must match fully or partially the client state.

2) Modify accordingly the DOM state of the document to generate the necessary JavaScript to bring the client a new state: this is the purpose of the event processing phase.

As you easily can figure out, fully reconstructing the client state per-event may be very costly, in general stateless ItsNat mode is more costly on server processing than stateful, but it has the benefit of the unlimited scalability of stateless, adding more nodes and distributing events between nodes with no problem of data sharing or server affinity (this is an ItsNat point of view, use of session for your own custom data is up to you).

Fortunately you do not need to fully reconstruct the client page in server, you just can reconstruct only the parts being to be changed by the concrete event, furthermore, ItsNat stateless mode is exceptional for injecting new markup to the client page.

Continuing our example, Java code to generate the initial page:

```
public class StlessExampleInitialDocLoadListener implements ItsNatServletRequestListener
{
  public StlessExampleInitialDocLoadListener()
  {
  }

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

public class StlessExampleInitialDocument
{
    protected ItsNatHTMLDocument itsNatDoc;

    public StlessExampleInitialDocument(ItsNatHTMLDocument itsNatDoc)
    {
        this.itsNatDoc = itsNatDoc;

        HTMLDocument doc = itsNatDoc.getHTMLDocument();
        Text node = (Text)doc.createTextNode(
             "This the initial stateless page (not kept in server)");
        Element presentationElem = doc.getElementById("presentationId");
        presentationElem.appendChild(node);
    }
}
```

This code is just an excuse to show how the initial page is a conventional ItsNat-generated page (but stateless), nothing especial here.

The interesting stuff starts with `stless_example_event_receiver.html` template registered with the name `manual.stless.example.eventReceiver` going to process stateless events:

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
xmlns:itsnat="http://itsnat.org/itsnat">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>ItsNat Stateless Document For Stateless Event Processing</title>
</head>
<body>

    <b id="counterId" itsnat:locById="true" itsnat:nocache="true">(num)</b>

    <div id="insertHereId" itsnat:locById="true" itsnat:nocache="true" />

</body>
</html>
```

Before continuing with the associated Java code, let’s stop to compare the initial page template:

```
    <div>
        <div>Num. Events: <b id="counterId" itsnat:nocache="true">0</b></div>
    </div>

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

and this template:

```
    <b id="counterId" itsnat:locById="true" itsnat:nocache="true">(num)</b>

    <div id="insertHereId" itsnat:locById="true" itsnat:nocache="true" />
```

Both pairs of elements have the same id, this is not casual, for instance the containing of the `insertHereId` element of the initial page template is going to be the same as the "event processor" template, as you can see only the tag name (div) and the id attribute are the same, note how both elements are located in a different place (markup order related to body element) in the page, further we will explain this is the reason of the ItsNat attribute `locById` set to true.

The Java code associated to this template:

```
public class StatelessExampleForProcessingEventDocLoadListener
        implements ItsNatServletRequestListener
{
    public StatelessExampleForProcessingEventDocLoadListener()
    {
    }

    public void processRequest(ItsNatServletRequest request,ItsNatServletResponse response)
    {
            new StatelessExampleForProcessingEventDocument(
                    (ItsNatHTMLDocument)request.getItsNatDocument(),request,response);
    }
}

public class StatelessExampleForProcessingEventDocument
        implements Serializable,EventListener
{
    protected ItsNatHTMLDocument itsNatDoc;
    protected Element counterElem;

    public StatelessExampleForProcessingEventDocument(ItsNatHTMLDocument itsNatDoc,
                ItsNatServletRequest request, ItsNatServletResponse response)
    {
        this.itsNatDoc = itsNatDoc;

        if (!itsNatDoc.isCreatedByStatelessEvent())
            throw new RuntimeException(
                    "Only to test stateless, must be loaded by a stateless event");

        // Counter node with same value (state) than in client:
        String currCountStr = request.getServletRequest().getParameter("counter");
        int counter = Integer.parseInt(currCountStr);

        HTMLDocument doc = itsNatDoc.getHTMLDocument();
        this.counterElem = doc.getElementById("counterId");
        ((Text)counterElem.getFirstChild()).setData(String.valueOf(counter));

        itsNatDoc.addEventListener(this);
    }

    public void handleEvent(Event evt)
    {
        ItsNatEventDOMStateless itsNatEvt = (ItsNatEventDOMStateless)evt;

        Text counterText = (Text)counterElem.getFirstChild();
        String currCountStr = counterText.getData();
        int counter = Integer.parseInt(currCountStr);
        counter++;
        counterText.setData(String.valueOf(counter));

        Document doc = itsNatDoc.getDocument();

        Element elemParent = doc.getElementById("insertHereId");
        ScriptUtil scriptGen = itsNatDoc.getScriptUtil();
        String elemRef = scriptGen.getNodeReference(elemParent);
        ClientDocument clientDoc = itsNatEvt.getClientDocument();
        clientDoc.addCodeToSend(elemRef + ".innerHTML = '';");
        clientDoc.addCodeToSend("alert('Currently inserted fragment removed before');");

        ItsNatServlet servlet = itsNatDoc.getItsNatDocumentTemplate().getItsNatServlet();
        ItsNatHTMLDocFragmentTemplate docFragTemplate =
             (ItsNatHTMLDocFragmentTemplate)servlet.getItsNatDocFragmentTemplate(
                   "manual.stless.example.fragment");

        DocumentFragment docFrag = docFragTemplate.loadDocumentFragmentBody(itsNatDoc);

        elemParent.appendChild(docFrag); // docFrag is empty now

        // Umm we have to celebrate/highlight this insertion
        Element child1 = ItsNatTreeWalker.getFirstChildElement(elemParent);
        Element child2 = ItsNatTreeWalker.getNextElement(child1);
        Text textChild2 = (Text)child2.getFirstChild();
        Element bold = doc.createElement("i");
        bold.appendChild(textChild2); // is removed from child2
        child2.appendChild(bold);
        child2.setAttribute("style","color:red");
         // <h3 style="color:red"><i>Inserted!</i></h3>
    }
}
```

In summary this code updates the `counterId` element with the current value in page and inserts a template with name `manual.stless.example.fragment` into the element with id `insertHereId`, into the document loaded for processing the stateless event. Later we will see that this insertion finally happens into the final client page.

The markup of the template registered with `manual.stless.example.fragment` is:

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"
            xmlns:itsnat="http://itsnat.org/itsnat">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Fragment</title>
</head>
<body>

    <h4 style="color:green"><b>Fragment...</b></h4>
    <h3>Inserted!</h3>

</body>
</html>
```

Because this template is inserted calling `docFragTemplate.loadDocumentFragmentBody(itsNatDoc)` ignore everything but the `<body>` content.

Let's study the code:

```
        if (!itsNatDoc.isCreatedByStatelessEvent())
            throw new RuntimeException(
                "Only to test stateless, must be loaded by a stateless event");
```

This check avoids trying to load this document, designed for processing stateless events, as a conventional page specifying a URL containing the `itsnat_document` parameter with the name of the template. The method `ItsNatDocument.isCreatedByStatelessEvent()` only returns true when the document has been loaded as part of processing of a stateless event.

Another more important use of `ItsNatDocument.isCreatedByStatelessEvent()` is to distinguish when the document was created to load a page as usual or to process a stateless event, because as you can easily figure out, in this very simple example the template for the initial page is so light than it also could be used for processing stateless events, thus this method could be used to separate the initial page load code (the presentation element construction) from the document load phase code when processing a stateless event (in this case registering the template-level event listener to receive the stateless event).

Note we are receiving a parameter `counter` with the current integer into the `counterId` element:

```
        // Counter node with same value (state) than in client:
        String currCountStr = request.getServletRequest().getParameter("counter");
        int counter = Integer.parseInt(currCountStr);

        HTMLDocument doc = itsNatDoc.getHTMLDocument();
        this.counterElem = doc.getElementById("counterId");
        ((Text)counterElem.getFirstChild()).setData(String.valueOf(counter));
```

With this code we "reconstruct" the DOM state of the `counterId` element in client.

Finally in the load phase:

```
        itsNatDoc.addEventListener(this);
```

It registers "this" document object as the event listener to process the stateless event causing the load of this document. After the phase load, this listener will be immediately called and the stateless event passed.

```
    public void handleEvent(Event evt)
    {
        ItsNatEventDOMStateless itsNatEvt = (ItsNatEventDOMStateless)evt;
```

This cast sounds familiar, ItsNat extends DOM Events with other kind of ItsNat-defined events, in this case stateless events are a new kind of "extended DOM event".

The interface `ItsNatEventDOMStateless` inherits from `ItsNatEventStateless`, an object implementing `ItsNatEventStateless` interface is "a stateless event". Later we will see a case of stateless event implementing `ItsNatEventStateless` but not `ItsNatEventDOMStateless`.

Now is time to really enter into the stateless approach of ItsNat:

```
        Text counterText = (Text)counterElem.getFirstChild();
        String currCountStr = counterText.getData();
        int counter = Integer.parseInt(currCountStr);
        counter++;
        counterText.setData(String.valueOf(counter));
```

We said before that the objective of the load phase when processing the stateless event is to bring the loaded document to the same DOM state than the client part we are going to change when processing the stateless event into the event listener. As you can see the text node has been updated with a new integer value, the necessary JavaScript code will be generated and sent to client to update the `counterId` element in client as the result of the stateless event processing.

More actions:

```
        ItsNatDocument itsNatDoc = itsNatEvt.getItsNatDocument();
        Document doc = itsNatDoc.getDocument();
        Element elemParent = doc.getElementById("insertHereId");
        ScriptUtil scriptGen = itsNatDoc.getScriptUtil();
        String elemRef = scriptGen.getNodeReference(elemParent);
        ClientDocument clientDoc = itsNatEvt.getClientDocument();
        clientDoc.addCodeToSend(elemRef + ".innerHTML = '';");
        clientDoc.addCodeToSend("alert('Currently inserted fragment removed before');");
```

In this case there is nothing done in the load phase related to `insertHereId` element because we want to reinsert the same DOM into this element again and again and custom JavaScript code is easier to clean the current state in client (ItsNat stateless mode ItsNat is much more client centric and JavaScript knowledge is recommended).

This code requires more explanation, the document we are modifying was created based on the template with name `manual.stless.example.eventReceiver`, this template is similar *but not the same* as `manual.stless.example`, notwithstanding the JavaScript generated is sent to modify the later.

Let's repeat again the important (dynamic) elements:

- `manual.stless.example.eventReceiver`

```
    <div>
        <div>Num. Events: <b id="counterId">0</b></div>
    </div>

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

- `manual.stless.example`

```
    <b id="counterId" itsnat:locById="true" itsnat:nocache="true">(num)</b>

    <div id="insertHereId" itsnat:locById="true" itsnat:nocache="true" />
```

As you know ItsNat node localization is based on "counting nodes in the tree" from <html> to the concrete node, of course node localizing is smart and automatic caching happens to avoid traversing the tree from the <html> root node. Anyway in some occasion full traversing happens, according to previous localization of elements in the first template is not the same as the second template, unless we explicitly avoid top-down traversing from root. This is the reason of ItsNat attribute `locById`.

The `locById` attribute tells ItsNat to use the id attribute/property in generated JavaScript to locate the node just calling `getElementById()`. By using this trick you can make ad-hoc lighter templates with just the enough markup aligned with the client DOM state.

Some code remains to explain:

```
ItsNatServlet servlet = itsNatDoc.getItsNatDocumentTemplate().getItsNatServlet();
ItsNatHTMLDocFragmentTemplate docFragTemplate =
     (ItsNatHTMLDocFragmentTemplate)servlet.getItsNatDocFragmentTemplate(
           "manual.stless.example.fragment");

DocumentFragment docFrag = docFragTemplate.loadDocumentFragmentBody(itsNatDoc);

elemParent.appendChild(docFrag); // docFrag is empty now

// Umm we have to celebrate/highlight this insertion
Element child1 = ItsNatTreeWalker.getFirstChildElement(elemParent);
Element child2 = ItsNatTreeWalker.getNextElement(child1);
Text textChild2 = (Text)child2.getFirstChild();
Element bold = doc.createElement("i");
bold.appendChild(textChild2); // is removed from child2
child2.appendChild(bold);
child2.setAttribute("style","color:red");
 // <h3 style="color:red"><i>Inserted!</i></h3>
```

This conventional ItsNat code inserts a modified version of the markup inside <body> of the loaded template `manual.stless.example.fragment` into the element `insertHereId`. The interesting detail of this code is most of the DOM modifications happen *after* insertion in document (remember that corresponding JavaScript DOM code is generated on the fly), any node involved, `child1`, `child2` and `textChild2`, need to be located after insertion, are they located traversing the full tree from <html>?

No!

As said before, location caching avoids fully traversing the tree from top, for instance ItsNat looks for parent nodes already in tree and cached, in this case the node `insertHereId` is the candidate, or any other child node cached when inserted. These techniques provide a powerful server centric Java based manipulation of the client DOM tree beyond simple one-time insertion.

8.2.3 The global ItsNatServletRequestListener and the second opportunity

Our example is not finished, more code is pending to analyze and a bit more code will be added.

In our servlet example mwe registered some global listeners:

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

    ItsNatHttpServlet itsNatServlet = getItsNatHttpServlet();

    //ItsNatServletConfig itsNatConfig = itsNatServlet.getItsNatServletConfig();
    itsNatServlet.addItsNatServletRequestListener(
            new StatelessGlobalDocumentLoadListener());
    itsNatServlet.addEventListener(new StlessGlobalEventListener());
```

In this case our focus is `StatelessGlobalDocumentLoadListener`. This is the code:

```
public class StatelessGlobalDocumentLoadListener implements ItsNatServletRequestListener
{
    public StatelessGlobalDocumentLoadListener()
    {
    }

    public void processRequest(ItsNatServletRequest request,
       ItsNatServletResponse response)
    {
        ItsNatDocument itsNatDoc = request.getItsNatDocument();

        if (itsNatDoc != null)
        {
            String docName = itsNatDoc.getItsNatDocumentTemplate().getName();
            System.out.println("Loading " + docName);
        }
        else
        {
            ServletRequest servReq = request.getServletRequest();
            String itsNatAction = servReq.getParameter("itsnat_action");
            if ("event_stateless".equals(itsNatAction))
            {
                String docName =
                    servReq.getParameter("itsnat_doc_name_second_opportunity");
                if (docName != null)
                    servReq.setAttribute("itsnat_doc_name", docName);
                           // Second opportunity
            }
            else
            {
                String docName = servReq.getParameter("itsnat_doc_name");
                if (docName == null)
                    docName = (String)servReq.getAttribute("itsnat_doc_name");
                if (docName != null)
                {
                    System.out.println("Page not found " + docName);
                    try
                    {
                        Writer out = response.getServletResponse().getWriter();
                        out.write("<html><body><h1>Page not found: \"" + docName + "\"");
                        out.write("</h1></body></html>");
                    }
                    catch(IOException ex) { throw new RuntimeException(ex); }
                }
                else
                {
                    throw new RuntimeException("Unexpected");
                }
            }
        }
    }
}
```

Consider we add the following code to `manual.stless.example.eventReceiver` template:

```
<script>
function sendEventStatelessCustomAddedTemplateInEvent()
{
    var userEvt = document.getItsNatDoc().createEventStateless();
    var counterElem = document.getElementById("counterId");
    var userEvt = document.getItsNatDoc().createEventStateless();
    userEvt.setExtraParam('counter',counterElem.firstChild.data);
    userEvt.setExtraParam('itsnat_doc_name_second_opportunity',
                          'manual.stless.example.eventReceiver');
    document.getItsNatDoc().dispatchEventStateless(userEvt,3 /*XHR_ASYNC_HOLD*/,1000000);
}
```

```
</script>

<a href="javascript:sendEventStatelessCustomAddedTemplateInEvent()">Send stateless custom
event, added template in event</a>
```

The only significative difference with previous code is this line:

```
userEvt.setExtraParam('itsnat_doc_name_second_opportunity',
                      'manual.stless.example.eventReceiver');
```

In this case we are not sending an `itsnat_doc_name` parameter, the parameter name `itsnat_doc_name_second_opportunity` is invented for this example.

A global servlet-level `ItsNatServletRequestListener` listener is ever executed by ItsNat for any ItsNat request, including stateless (event) requests. In this listener we can detect there was not provided a template name, and we have a second opportunity:

```
            ServletRequest servReq = request.getServletRequest();
            String itsNatAction = servReq.getParameter("itsnat_action");
            if ("event_stateless".equals(itsNatAction))
            {
                String docName =
                    servReq.getParameter("itsnat_doc_name_second_opportunity");
                if (docName != null)
                    servReq.setAttribute("itsnat_doc_name", docName);
                           // Second opportunity
            }
```

The `itsnat_action` parameter is a parameter familiar for us because it has some special uses, in this case the value `event_stateless` is standard and can be used to detect when we are receiving a stateless event. Setting the `itsnat_doc_name` as attribute we say to ItsNat to use this attribute instead of the missing parameter (we used this technique before in custom requests and pretty URLs). One interesting use of this technique is to avoid specifying the template going to be used in client side as a sort of stateless request dispatcher.

8.2.4 The global EventListener and the custom stateless events

In previous examples we generated and sent JavaScript code to the client based on a stateless document template loaded by a stateless event.

We also have the option of avoiding the document template artifact and generate and send back custom JavaScript to client.

Now our focus is:

```
   itsNatServlet.addEventListener(new StlessGlobalEventListener());
```

`StlessGlobalEventListener` code:

```
public class StlessGlobalEventListener implements EventListener
{
    public StlessGlobalEventListener()
    {
    }

    public void handleEvent(Event evt)
    {
        ItsNatEventStateless itsNatEvt = (ItsNatEventStateless)evt;

        if (itsNatEvt.getItsNatDocument() == null)
        {
            ClientDocument clientDoc = itsNatEvt.getClientDocument();
            //ServletRequest request =
             // itsNatEvt.getItsNatServletRequest().getServletRequest();
            String docName = (String)itsNatEvt.getExtraParam("itsnat_doc_name");
            if (docName != null)
                clientDoc.addCodeToSend("alert('Received stateless event with not found
itsnat_doc_name: " + docName + " from the page with title: " +
itsNatEvt.getExtraParam("title") + "');");
            else
                clientDoc.addCodeToSend("alert('Received a custom stateless event from
the page with title: " + itsNatEvt.getExtraParam("title") + "');");
        }
    }
}
```

If we send a stateless event without specifying the `itsnat_doc_name` (and avoiding any kind of second opportunity seen before), no document based on a template is instantiated and only this event listener and the code in bold is executed.

This is an example of code in client:

```
  <script>
  function sendEventStatelessCustom()
  {
    var userEvt = document.getItsNatDoc().createEventStateless();
    userEvt.setExtraParam('title',document.title);
    document.getItsNatDoc().dispatchEventStateless(userEvt,3 /*XHR_ASYNC_HOLD*/,1000000);
  }
  </script>

  <a href="javascript:sendEventStatelessCustom()">Send stateless event custom</a>
```

8.2.5 Conclusion

We have shown how we can build Single Page Interface Stateless Server Centric Web Sites with ItsNat. The previous example is just a proof of concept, another more complete example is in The Feature Showcase using components!! And a basic example of a stateless SPI SEO compatible web site in ItsNat web.

Enjoy SPI Stateless SEO Compatible, enjoy ItsNat!!