Vendio Stores
RST

Template Language Reference

Version 2.1, 09/07/2009

Contents

Introduction

This document describes the language used to create online stores with the Vendio Stores product, called **Really Simple Templates** (or **RST** on short). It covers a wide range of topics, from basic HTML integration to advanced widget programming, that will allow each store owner to get the exact look and feel that he or she needs for his online store.

The Vendio Stores

The Vendio Stores are Vendio's premium service that enables sellers to create and manage an online store. Its goal is to provide the ultimate platform for sellers to use and enhance, and therefore the foundation of the platform exposes full access to all of its three components:

- The **Stores API** is a standalone interface over the seller's store data, that can be used with any store front end (such as an online store, a desktop widget or a mobile application). The API uses industry standard XML and HTTP technologies to enable any type of application to retrieve and display the store data.
- The **Store Templates** are an absolutely unique offer on the market: sellers can **fully** customize the look and feel of their store: they can organize their store by creating **any number of pages**, with the **design and layout they like**. There are **no limitations** about what the stores can look and behave like. Vendio offers a selection of **highly customizable, in-house made templates** to choose from, as well as the availability to **create and upload a 100% custom made template**. Moreover, sellers can choose to have their store **reliably hosted by Vendio**, or to install their store on **any web hosting sytem** they prefer. Template designers can partner with Vendio to create and publish their templates within the Vendio system. Their templates will be available to all users, along with those created by the Vendio designers.
- The **Really Simple Templates** language (RST) is a powerful, yet extremely simple markup language that enables sellers to instantly integrate their store data in their store templates. The language was designed with simplicity in mind and it's light and intuitive, so that any template designer can start writing a Vendio Stores template without any prior knowledge or programming skills.

Assumptions and prerequisites

The topics in this document will focus on *how to write a 100% custom made template* for the Vendio Stores. They assume you already have set up your Vendio Stores, if you haven't you can go now through the following steps:

- Go to `www.vendio.com` and sign up for a **free account with Vendio Stores**
- Go to the Store Info page, choose a store name and choose to host your store at Vendio
- Design a template using **your favorite authoring tool** (or text editor), pack the template files in a **zip archive** and upload the archive in the Store Template page. Preview your template to see the results. Repeat this until you are satisfied with your

store.

> **Inline notes**
> Certain topics may require some previous knowledge on web technologies, such as *HTML*, *XML* or *Ajax*. While it's beyond the scope of this document to cover in detail every collateral technology employed or referred, brief inline notes will describe the basics of such technologies whenever they're mentioned, and will provide additional links to relevant documentation on the web.

There... are you ready to write your first Vendio Stores template?

1. RST Basics | Step by step template building

1.1 HTML just works!

A template is a collection of HTML files and other assets (images, CSS files, Flash movies etc.). The *only* requirement that a Vendio Stores template has is to contain a file called **home.html**. This is the beauty of the Vendio Stores templates: you can use any HTML code to create a store page and it just works!

> **What is HTML?**
> From Wikipedia: *HTML, an initialism of HyperText Markup Language, is the predominant markup language for Web pages*. Now really, you're writing web templates, you knew what HTML was, didn't you?

> **Web authoring tools**
> *Web authoring* is just a fancy word for designing a web site. There are a lot of software programs out there to help designers, programmers or even complete non-technical people to build a collection of HTML pages, which compose a site. These programs are called *web authoring tools*.

Hello, World!

Use your favorite text editor, or authoring tool, to create a simple (or complex!) file called **home.html**. Here is an example of how your page might look like:

home.html

```
<html>
<body>
<h1>Hello, World!</h1>
Welcome to my store.
</body>
</html>
```

Now pack the file in a .zip file and upload it in the Store Template page. Use the store preview feature to see how your page looks like.

Adding template assets

Your template can have any number of HTML files, as well as images, flash movies or CSS files. Vendio Stores do offer this unique feature. You can organize the files using any directory structure you like; your imagination is the only limit of what your template looks like and what it does! If you're using a web authoring tool to build your template, that has a feature to save, or export, or publish your composition as HTML, just pack the file structure that your tool created in a .zip file (make sure your main page is called home.html), upload it and it will work right out of the box!

1.2 Adding data to your store using XML feeds

Unless you're building a very simple web site, your site will need to display real data, and that's when the RST language comes in. XML feeds are a popular way to exchange information over the web, and that's why the Vendio Stores templates have been designed so that information from an XML feed be extremely easy to integrate.

> **What is XML?**
> XML stands for *Extensible Markup Language*. XML's purpose is *to aid information systems in sharing structured data, especially via the Internet* (Wikipedia). Basically, XML is a language that allows information (such as the information about your online store and items) to be packed and exchanged over the web, between its storage location (such as your online inventory) and an application displaying it (such as your online store being visited by a customer). The XML quickly became a very popular format in conjunction with the practice of *syndicating content* over the web, using *web feeds*.

> **Web feeds**
> A web feed represents an access channel to data that is updated frequently. Originally a way to distribute news over the Internet, web feeds and web APIs are now a common way to publish (or *syndicate*) content from a central source to a wide range of *subscribers*. It its simplest and most commonly encountered form, a web feed is an URL that, when accessed from a browser or from some other application, returns an XML-formatted piece of information.
> Example: Open the URL below in your browser to see the Yahoo News Top Stories in XML format.
> `http://rss.news.yahoo.com/rss/topstories`

The Vendio Stores templates can display any information available on the web in XML format. With XML web feeds being so popular, this means your templates can display virtually anything, as long as there's a web service that offers that information as a web feed. While displaying Yahoo News on your website may not be very useful, showing your inventory items *is* a critical feature for an online store, and that's where the Vendio Stores API comes into play.

The Vendio Stores API

The Vendio Stores API is a collection of URLs (or HTTP calls) that return information about your store and inventory items in XML format. Any application can retrieve (public) information regarding your store settings, store categories, or items. The API can also be used to manage your store's shopping carts and, in the future, the API will be extended to fully integrate the checkout process. You can refer to the Stores API Reference to find more information about the available calls.

> **The Vendio inventory**
> Vendio offers you an *online inventory* service, which is a database to store and manage all your items for sale. To set up your inventory, go to the My Items page on the Vendio site. The nice thing about the Vendio inventory is that all your

> inventory data can be retrieved as an XML feed via the Vendio Stores API.

Adding content from an XML feed

Let's suppose that you wrote a template page that looks like this:

home.html

```
<p>
<h1>Item Title 123</h1>
<span>
   Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
   incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
   exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure
   dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
   Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit
   anim id est laborum.
</span>
Price: <b>$18.49</b>
</p>
```

This is a fragment that displays the title, the description and the price of a dummy (fake) product. Although oversimplified for clarity reasons, the fragment above is actually quite similar to what the output of a web authoring tool might look like. When you load the home.html file in your browser, you browser will format the dummy data according to the HTML markup, and will display it. What you need now is to *replace* the dummy data with real data about an item.

First, you need an XML feed that returns the data about the item. The URL for such a feed looks like:

```
http://sell.vendio.com/GetHostedStoresInfoServlet?
verb=GII&GII.store=mystorename&GII.lid=123
```

> **The Stores API URLs**
> The URL above triggers a `GetItemsInformation` call (represented by the `GII` verb). The URL contains the name of the web service that returns the data, the call's name (or verb), the seller's store name and the item's listing ID. Every Vendio Stores user has a unique store name, that identifies his store among other stores. Each inventory item available in store has a unique listing id, that identifies it among other inventory items in the seller's store.

The XML response of this call looks like the one below:

```
<Storefront>
 <ItemsInfo>
   <Status>Success</Status>
   <Items count="1" totalItems="1" totalPages="1">
```

```
    <Item featured="Y" id="123">
      <Title>Puppy Dog</Title>
      <ImageUrl caption="Cute Puppy" fullSizeImageUrl="http://someserver/someimage.jpg"/>
      <Price currency="$">29.99</Price>
      <Description>Hey, come and get this lovely puppy.</Description>
    </Item>
  </Items>
 </ItemsInfo>
</Storefront>
```

By looking at the above response, you can easily identify the data that we need to display: the item's title is *Puppy Dog*, the item's description is *Hey, come and get this lovely puppy.* and the item's price is *$29.99*. What we want is to include these values in our HTML mockup code.

Remember, the code we used to display the dummy data was (copy & paste from above):

```
<p>
<h1>Item Title 123</h1>
<span>
   Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
   incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
   exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure
   dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
   Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit
   anim id est laborum.
</span>
Price: <b>$18.49</b>
</p>
```

Now, let's replace it with:

```
<p rst:xml="http://sell.vendio.com/GetHostedStoresInfoServlet?
verb=GII&GII.store=mystorename&GII.iid=123">
<h1 rst:content="/Storefront/ItemsInfo/Items/Item/Title">Item Title 123</h1>
<span rst:content="/Storefront/ItemsInfo/Items/Item/Description">
   Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
   incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
   exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure
   dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
   Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit
   anim id est laborum.
</span>
Price: <b rst:content="${/Storefront/ItemsInfo/Items/Item/Price}">$18.49</b>
</p>
```

That's it! As you can see, the added information does not affect the layout of your template

when you preview the page locally. All you did was to add some HTML tag attributes starting with a *rst:* prefix, which are not visible in the template itself. However, if you upload your template and use it for your store, you will see the actual store data replacing the dummy values. I.e. the dummy title, description and price will be *magically* replaced with the real data from the XML feed. RST is that simple: that what Really Simple Template stands for!

> **HTML tags and attributes**
> From a structural point of view, a HTML document consists of *content* and *markup*. The *content* represents any meaningful information displayed in the document, such as a text, an image or a movie. The *markup* is a special text, not visible to the document's reader, but specifying and affecting the way the information is *formatted* within the document. The markup specifies things like font, size, color, alignment, transparency or layout. The markup usually consists of words surrounded by `< >` signs, called *tags*, as in `<b>`, `<img>` or `<input>`, optionally accompanied by additional key-value pairs within the same `< >` signs, as in `<form action="doAction.html" method="post">`
>
> What RST does is leverage this feature of the HTML markup and "inject" data-related information (markup) as HTML attributes, so that the human-readable representation of the template is not affected in any way by this modification. RST was designed to be **an intuitive, non-intrusive, HTML-friendly way to inject data from XML feeds into a web page**.

> **The RST rendering engine**
> So you might wonder: what exactly is behind the "magic" of turning RST tags into actual data? The answer is simple: **the rendering engine**. Your template files are *stored* for free on the Vendio servers, but this is not all -- they are *parsed* and *rendered* by a special application called... *the engine*. So an RST-based web site relies on the following components:
> - one or more XML data sources accessible via HTTP ("the API")
> - a collection of HTML pages enriched with RST attributes ("the template")
> - the RST rendering engine
>
> The RST attributes are special HTML attributes that are added to the HTML tags in a template to bind data from the XML feeds to those tags. In the absence of the RST rendering engine, the template will be a valid HTML mockup (with dummy data). When the RST engine parses the template, it will generate actual dinamic pages based on the XML data.

So let's recap one more time the changes needed to display XML data in the template:

1. Add a **rst:xml** attribute to any HTML tag in your template. Set the value of this attribute to the URL of the XML feed:

```
<p rst:xml="http://sell.vendio.com/GetHostedStoresInfoServlet?verb=GII&GII.store=mystorename&GII.lid=123">
```

2. Add **rst:content** attributes to any HTML tag to replace the *content* of that tag with a value from the XML feed. Set the value of this attribute to the *XPath* expression referring the

information within the XML feed:

```
<h1 rst:content="/Storefront/ItemsInfo/Items/Item/Title">Item Title 123</h1>
```

> **What is XPath?**
> You may have noticed the "path" notation used to refer a piece of information (or *node*) within the XML document. This notation is called **XPath**, and it's another widely used technology used in conjunction with XML to display data from a feed. XPath enables us to "traverse" the nodes of an XML feed in order to "extract" the data we're interested in, using a syntax very similar to common file system paths. For instance, to extract the "title" from the XML example above, you can use the following XPath expression: `/Storefront/ItemsInfo/Items/Item/Title`. If you look at the XML feed as if it were a directory structure, then the XPath expressions become pretty straightforward.
> Explaining XPath in detail is beyond this topic's scope, but you can find many references on the web and the official XPath page at W3C is a good start.

> **RST Path: an RST extension to XPath**
> There is one limitation of the XPath way to extract information from an XML feed: using an XPath expression, you can select any *node* from the document, and therefore you can refer its *content*, but you cannot access the value of an *attribute* of that node. Therefore, the RST specification adds a simple extension to the XPath syntax, in order to retrieve the value of a tag attribute, which is appending a @ sign followed by the name of the attribute, after the node selector. For instance, to refer the "currency" in the GII response, you would use the following RSTPath expression: `/Storefront/ItemsInfo/Items/Item/Title@currency`.
> See **Chapter 3. Language specifications** for the complete specification of the RST Path expressions.

Loading data from a local XML file

In certain cases, you may want to store some template data (such as number of items per page, promotional texts or store policies, font or color schemes) in a place where they can be easily accessed and modified. One option would be to embed them in the template, and edit the template files whenever you want to change them. A more convenient way would be to store the data in an XML file, along with the template files, and access the data from the XML file just as if it were a web feed. That's because it's virtually no difference between an XML file that you store locally with your template and an XML file that you fetch via an URL, except for needing to specify that the local XML file is... local. And you do that by adding an **rst:xmlsource="template"** attribute next to the **rst:xml** attribute that loads the file, and set the **rst:xml** value to the name of the local XML file:

```
<p rst:xml="settings.xml" rst:xmlsource="template">
...
</p>
```

In the example above, a file called *settings.xml* would be loaded as an XML source.

Loading multiple XML data sources

Complex pages need to display several types of data -- categories, items, cart contents -- and to retrieve this information you need to access more than one URL. RST allows you to define as many XML sources as you like, and in order to distinguish them you need to add *identifiers* to each of them, as follows:

```
<p rst:xml:somename="http://path/to/some/xml"
  rst:xml:someothername="http://path/to/some/other/xml"
  ...>
<b rst:content="somename:/xpath/to/data/in/first/xml">Some name</b>
<i rst:content="someothername:/xpath/to/data/in/the/second/xml">Some other name</i>
</p>
```

The names added after the **rst:xml** attribute (*somename*, *someothername*) are called **identifiers**. They can be used later to *refer* the corresponding XML feed. There are a few limitations on what characters may appear in an identifier, but using only characters should be a simple rule to keep you out of trouble.

You can add any number of **rst:xml** attributes in any HTML tag. You can also place each **rst:xml** attribute in its own tag:

```
<span rst:xml:somename="http://path/to/some/xml"></span>
<span rst:xml:someothername="http://path/to/some/other/xml"></span>
...
<b rst:content="somename:/xpath/to/data/in/first/xml">Some name</b>
<i rst:content="someothername:/xpath/to/data/in/the/second/xml">Some other name</i>
</p>
```

Speed up data loading: caching XML data

For each **rst:xml** attribute in your template that points to a distinct URL, a call will be made to that URL in order to retrieve the data. This occurs on every page load, and if your store has many visitors this may impact your site's performance. Some data, for instance the category tree, change infrequently, and therefore reusing the fetched data between requests would speed up page loading, without creating any problem. To enable data caching, use the **rst:ttl** attribute and set it to a reasonable number of seconds to cache the data for. A low value will ensure high responsiveness to data changes, a high value will ensure increased loading speed. A value of 0 (zero) specifies that the data should never be cached -- this should be set for accuracy-critical information, such as cart contents or available selling quantity.

1.3 Replacing HTML attributes with real data

We've seen how tag contents can be replaced with data from an XML feed using the

rst:content attribute. What if we need to replace a tag's *attribute* with data from an XML feed? For instance, if we want to display an image for an item, we would need to set the `src` attribute of the `<IMG>` tag to the actual image URL.

Suppose our mockup contains the following HTML code to display the image of an item:

```
<img src="thumb.jpg" alt="Item 123" title="Item 123"/>
```

and we're using the GII call to retrieve the item's information (copy and paste from above):

http://sell.vendio.com/GetHostedStoresInfoServlet?verb=GII&GII.store=mystorename&GII.lid=123

```
<Storefront>
 <ItemsInfo>
  <Status>Success</Status>
  <Items count="1" totalItems="1" totalPages="1">
   <Item featured="Y" id="123">
    <Title>Puppy Dog</Title>
    <ImageUrl caption="Cute Puppy"
fullSizeImageUrl="http://someserver/someimage.jpg"/>
    <Price currency="$">29.99</Price>
    <Description>Hey, come and get this lovely puppy.</Description>
   </Item>
  </Items>
 </ItemsInfo>
</Storefront>
```

What we need to do is:

- load the XML data from `http://sell.vendio.com/GetHostedStoresInfoServlet?verb=GII&GII.store=mystorename&GII.lid=123`
- replace the **src** attribute with `http://someserver/someimage.jpg` (from the "`ImageUrl`" value in the XML feed)
- replace the **alt** and **title** attributes with *Cute Puppy* (from the "`caption`" value in the XML feed)

and here's how we do it:

```
<img rst:xml:iteminfo="http://sell.vendio.com/GetHostedStoresInfoServlet?
verb=GII&GII.store=mystorename&GII.lid=123"
src="thumb.jpg"
rst:attr:src="iteminfo:/Storefront/ItemsInfo/Items/Item/ImageUrl"
alt="Item 123"
rst:attr:alt="iteminfo:/Storefront/ItemsInfo/Items/Item/ImageUrl@currency"
title="Item 123"
rst:attr:title="iteminfo:/Storefront/ItemsInfo/Items/Item/ImageUrl@currency"/>
```


Again, what happens is that loading the HTML code above in a browser will not affect the original layout, but when you upload it to the Vendio server and preview your template or visit your store, the RST attributes will do their job and *replace* the dummy data. For each HTML attribute that you want to replace (or set) to a value from an XML feed, you need to set an **rst:attr** attribute with the tag's name, i.e. `rst:attr:tagname="xpath value"`. If you specify an **rst:attr** attribute, it's up to you whether you specify the original attributes (**src**, **alt** and **title** in the example above) or not. The only difference is what you see when loading the template directly in your browser, it doesn't have any effect when the template is rendered by the RST engine.

1.4 The pseudo-tag: <NOTAG>

RST enables template designers to include data from XML feeds in their HTML mockups. This is done by adding RST attributes (HTML attributes starting with the **rst:** prefix) to the HTML tags. RST attributes are parsed and interpreted during the template rendering and control the content of the HTML tags they are attached to. What if you don't want to output any HTML tag whatsoever, but only data from the XML feed? In this case you'll use a special tag called `<NOTAG>`. `NOTAG` will behave like any other HTML tag, i.e. any number of RST attributes can be added to one `NOTAG` tag and any number of `NOTAG` tags can be added to the template document -- however, the tag itself will *not* be rendered in the final output.

Example:

home.html

```
<h1>Item details</h1>
<notag rst:xml:iteminfo="http://sell.vendio.com/GetHostedStoresInfoServlet?
verb=GII&GII.store=mystorename&GII.lid=123"/>
Title: <notag rst:content="/Storefront/ItemsInfo/Items/Item/Title">Title 123</notag>
```

The generated output will look like:

```
<h1>Item details</h1>
Title: Puppy Dog
```

What happened is that an XML source was loaded and a value from the XML feed was rendered, yet its containing tag NOTAG was not rendered.

1.5 Using context data

When browsing a site, the displayed content changes based on the *context* of your navigation. The navigation context includes the page name (e.g. the home page, the item details page, the about us page, or the checkout page), the page number (e.g. when navigating through multiple pages of search results), search terms, a product ID or a category name etc.

Some context data is explicitly set by clicking on a link, or specifying a value in the URL of the

page. For instance, an item detail page URL may look like: `http://www.vendio.com/stores/mystorename/item/item-with-a-funny-title/lid=123`. In this case, the `lid=123` part of the URL defines an explicit context data, the which is item's ID. The item ID uniquely identifies one item, which in turn generates a page content specific to that item.

Other context data is implicit, calculated automatically, such as the store's URL, the prev / next page URLs, the number of results per page or the transfer protocol (HTTP / HTTPS). RST refers to the implicit context data as **magical** values.

Any context value can be used in RST attributes by referring it as **context:variable_name**. For instance, let's suppose that searching for "Ipod Nano" triggers the following URL: `http://www.vendio.com/stores/mystorename/search/keywords=Ipod%20Nano/` and you want to display a text in your page that looks like:

```
<div>Searching the store for <b>Canon PowerShot</b> ...</div>
```

where you want to replace the "Canon PowerShot" text with the actual search terms. Here's how you do it:

```
<div>Searching the store for <b rst:content="context:keywords">Canon PowerShot</b> ...</div>
```

For a complete list of "magical" context keys, refer to **Chapter 4.3 Access context information**.

1.6 Adding navigation to your site: linking pages

Almost any page of a store contains links to other pages. These may be links to external pages (such as PayPal), links to other store pages (such as About Us), links to parametrized store pages (such as links to showing all products in a category, or showing the item details for a product), or navigation links to a different page number in a list. RST offers a built in mechanism for generating links, through the use of the **rst:href** attribute. A few examples:

Link to About Us:

```
<a rst:href:pagename="about">About Us</a>
```

Link to the details page for the current item:

```
<a rst:href:pagename="details" rst:href:lid="context:lid">Details</a>
```

Navigation links to the first/prev/next/last pages of results in a list:

```
<a rst:href:page="1">First</a>
<a rst:href:page="context:__prevpage">Previous</a>
<a rst:href:page="context:__nextpage">Next</a>
```



```
<a rst:href:page="searchresultsxml://Results@pageCount">Last</a>
```

Use the **rst:href:pagename** attribute to specify the page to link to. The name of the page is the name of the .html file in your template directory, without the .html extension (so for example `rst:href:pagename="about"` would link to `about.html`).

Use **rst:href:param="value"** to add any `param=value` context data to your link. For instance, use "`lid`" to specify the ID of a product and set its value to a value taken from an XML feed or from the context data.

1.7 Display conditional content

Sometimes you need to display some content only if certain condition is met (for instance show a Google checkout button only if the Google checkout is enabled and available for an item), or hide some content if a certain condition is met (for instance hide links to store categories that don't have any items associated), or display alternate content based on a certain condition (for instance, show a Buy Now button if there's enough quantity for an item, or a Sold Out button if the quantity is zero). RST provides two conditional control attributes, **rst:if** and **rst:ifnot**. The value of this attribute can be the value from an XML feed (in this case, an empty value or a value of 0 evaluates to false, while any other value evaluates to true) or a more complex comparison between two values (several operators are available: equals to, does not equal to, is less than, is greater than or equal etc.). Refer to **Chapter 3.3 The RST syntax** for a complete list of comparison operators.

```
<a rst:if="{context:page} gt 1" rst:ifnot="{context:__prevpage} eq 1" rst:href:page="1">First</
a>
<a rst:if="{context:page} gt 1" rst:href:page="context:__prevpage">Previous</a>
<a rst:if="{context:page} lt {{searchresultsxml://Results@pageCount} - 1}"
rst:href:page="context:__nextpage">Next</a>
<a rst:if="{context:page} lt {{searchresultsxml://Results@pageCount} - 1}"
  rst:ifnot="{context:__nextpage} ne {searchresultsxml://Results@pageCount}"
  rst:href:page="searchresultsxml://Results@pageCount">Last</a>
```

In the example above, the "First" link is displayed only if the current page is not the first one and if the first page is different than the previous page; the "Previous" link is displayed only if the current page is not the first one; the "Next" link is displayed only if the current page is not the last one; and the "Last" link is displayed only if the current page is not the last one and the last page is different than the next page.

> **Nested RST attribute values**
> In its simplest form, a RST attribute can take the value of an XPath (such as `/Storefront/ItemsInfo/Items/Item/Title`), or a context variable (such as `context:keywords`). When grouping more than one such values within the same attribute, they are included in braces `{ }`. Examples:
>
> - To display concatenated values:
>
> ```
> <b rst:content="Page {context:page} of
> ```

```
{searchresultsxml://Results@pageCount}">Page 1 of 10</b>
```

- To evaluate conditions:

```
<a rst:if="{context:page} ne
{searchresultsxml://Results@pageCount}">More...</a>
```

- Deambiguation:

```
<a rst:href:lid="{context:lid}">Permalink</a>
```

rst:if will skip the containing tag completely if the condition is evaluated as false, while **rst:ifnot** has the opposite effect: will display the tag's content if the condition is not met.

1.8 Show repetitive content

Store data (and its XML feed representation) often contains data belonging to a list of similar entities. For instance, retrieving information on the store categories will return the same information (e.g. name, id, item count) for every category in the store. Or, retrieving the information on an item will return the URLs of all the images associated to that item. Or, search results will include the same type of information for every item matching the search criteria. The results are typically displayed on the page in a similar manner, as a list, either one after another, or grouped as multiple items on a row, or grouped as multiple items on rows with alternating background colors. In every case, the format is based on a *repeating pattern*. The pattern may repeat after each item, or after a group of items, or after a fixed number of groups of items. RST is again a great helper in providing a simple, yet powerful way to manage repetitive data.

Node lists in XML
A list of items is typically represented in an XML document as a sequence of nodes having the same name, grouped inside a *container* node:

```
<ContainerNode>
  <ItemNode>...</ItemNode>
  <ItemNode>...</ItemNode>
  ...
  <ItemNode>...</ItemNode>
</ContainerNode>
```

The `<ItemNode>` node may contain some data or some child nodes of its own. Optionally, nodes in the list may contain a diferentiating attribute, such as the node's index within the list, or a unique ID, but this is not mandatory:

```
<ContainerNode childCount="5">
  <ItemNode index="1" itemID="123">...</ItemNode>
  <ItemNode index="2" itemID="456">...</ItemNode>
  <ItemNode index="3" itemID="789">...</ItemNode>
  <ItemNode index="4" itemID="abc">...</ItemNode>
  <ItemNode index="5" itemID="xxx">...</ItemNode>
</ContainerNode>
```

Accessing lists using XPath
When nodes having the same name are grouped as a list in an XML document, accessing them by name is no longer possible. Take for instance the following XML feed:

```
<ContainerNode childCount="5">
  <ItemNode index="1" itemID="123">...</ItemNode>
  <ItemNode index="2" itemID="456">...</ItemNode>
  <ItemNode index="3" itemID="789">...</ItemNode>
  <ItemNode index="4" itemID="abc">...</ItemNode>
  <ItemNode index="5" itemID="xxx">...</ItemNode>
</ContainerNode>
```

How do you access the information belonging to the 3rd item in the list? Or the one belonging to item having `itemID="abc"`? The answer is: using XPath.

- To retrieve the 3rd node in the list: `/ContainerNode/ItemNode[3]`
- To retrieve the node with itemID="abc":

`/ContainerNode/ItemNode[@itemID='abc']`

- To retrieve the item ID of the 3rd node:

`/ContainerNode/ItemNode[3]@itemID`

Display all items in a list

In its simplest form, displaying repetitive content means iterate through all the nodes in a list and display the information belonging to those nodes. Let's say we have an XML feed containing the following data:

http://sell.vendio.com/GetHostedStoresInfoServlet?verb=GII&GII.searchString=test&GII.store=mystorename

```
<Storefront>
 <ItemsInfo>
  <Items count="10" totalItems="10" totalPages="1">
   <Item featured="Y" id="644">
    <Title>Test </Title>
    <Subtitle/>
    <ImageUrl/>
    <Price currency="$">23.00</Price>
   </Item>
   <Item featured="Y" id="862">
    <Title>Test Inventory</Title>
    <Subtitle/>
    <ImageUrl/>
    <Price currency="$">10.00</Price>
   </Item>
   <Item featured="Y" id="611">
    <Title>This is test item for sku1</Title>
    <Subtitle/>
```

```
      <ImageUrl/>
      <Price currency="$">2.00</Price>
    </Item>
    <Item featured="Y" id="616">
      <Title>This is test item for sku2</Title>
      <Subtitle/>
      <ImageUrl/>
      <Price currency="$">2.00</Price>
    </Item>
  </Items>
  <Status>Success</Status>
 </ItemsInfo>
</Storefront>
```

and we want to display a simple list with the item titles, linking to the item details pages. The HTML mockup for such a list may look like:

```
<ul>
 <li><a href="#">Item 1</a></li>
 <li><a href="#">Item 2</a></li>
 <li><a href="#">Item 3</a></li>
</ul>
```

To replace the list above with the actual content of the XML feed, we write:

```
<ul rst:xml="http://sell.vendio.com/GetHostedStoresInfoServlet?
verb=GII&GII.searchString=test&GII.store=mystorename"
   rst:repeat:item="/Storefront/ItemsInfo/Items/Item">
 <li><a href="#" rst:href:pagename="details"
              rst:href:lid="item1:@id"
              rst:content="item1:/Title">Item 1</a></li>
 <li><a href="#" rst:href:pagename="details"
              rst:href:lid="item2:@id"
              rst:content="item2:/Title">Item 2</a></li>
 <li><a href="#" rst:href:pagename="details"
              rst:href:lid="item3:@id"
              rst:content="item3:/Title">Item 3</a></li>
</ul>
```

Loading the page directly in the browser shows the same content as the mockup. What we did was to add a few RST attributes:

- an **rst:xml** attribute to load the XML data
- an **rst:repeat** attribute to *iterate* through the item nodes. The attribute has an identifier (*item*) that will be later referred in order to access the information in the node. The attribute's value (`/Storefront/ItemsInfo/Items/Item`) is an XPath pointing to the repetitive node name. If you remember, referring the Nth node in a list is made by appending a `[N]` suffix to the XPath pointing to the node. The node containing the **rst:repeat** attribute will form a **loop**: its contents (from the opening tag to the closing tag, exclusively) will be *repeated* for each occurrence of the nodes referred by the identifier, until no more nodes are found. In this case, the content will iterate through all the `<Item>` nodes.
- inside the loop section, information belonging to a node is simply accessed by appending a number to the identifier's name. *item1* means "first node referred by identifier item", *item2* means "second node referred by identifier item", "item3" means "third node referred by identifier item"

What happens if the items in the XML feed are less than the number of items in a loop section? For instance, what if there were only 2 items returned in the feed, what would *item3* have pointed to? The answer is: the *item3* line would have been skipped and the loop would have been terminated. Whenever the looping reaches the end of the list in the data feed, the loop section is terminated.

What happens if the items in the XML feed are more than the number of items in a loop section? Then the loop section will repeat, starting with the next available node. That is, after completing the first loop, *item1* will point to the 4th node (the first loop was completed after rendering 3 nodes), *item2* to the 5th and *item3* to the 6th. Then, if more than 6 items were available in the feed, *item1* would point to the 7th node, and so on.

You may wonder, is it really necessary to write 3 lines of code? It seems that removing the second and third line from the loop section would have endered the same output: *item1* would have pointed to the first node in the XML feed, then the loop section would have ended and repeated itself until the last node in the feed: *item1* would have pointed to the second node, than to the third, and so on, for all the nodes. That's actually right: you don't need to write identical lines for more than one node. Then why did we do it? For two reasons:

1. The mockup contained 3 lines. If we'd removed the 2nd and 3rd line from the template, the rendered template would've looked ok, but the dummy template, loaded directly in the browser, would've displayed only one line instead of 3. If preserving the mockup is important (for showcasing, or for future editing), then adding RST attributes to 3 lines instead of one may be worth the effort.
2. As you're about to see, there are times when subsequent items are *not* rendered identically. Examples may include: several items on a row in a table, or alternating colors of the row background.

Limit the number of displayed items in a list

There are cases when the XML feed may return a larger (or unknown) number of items than a page needs to display. In this cases, iterating blindly through all the items in the list may break

the layout of the page, or even the customer's browser. Adding a maximum limit to the number of items to iterate through, as well as an initial offset is done by using two RST attributes: **rst:max** and **rst:start**.

Example:

```
<ul rst:xml="http://sell.vendio.com/GetHostedStoresInfoServlet?
verb=GII&GII.searchString=test&GII.store=mystorename"
   rst:repeat:item="/Storefront/ItemsInfo/Items/Item"
   rst:start="0"
   rst:max="9">
...
</ul>
```

Using different formatting for items in a list

If you want to display a list of items, 3 items in a row, then you need to include each item in a cell but also include every three items in a row. How do you do that? The answer is simple: design your mockup and then just add RST attributes. Let's say your mockup looks like:

```
<table>
 <tr>
   <td>Item1</td>
   <td>Item2</td>
   <td>Item3</td>
 </tr>
</table>
```

To turn the above code into a repetitive section using XML feed data, you will write:

```
<table>
 <tr rst:xml="http://sell.vendio.com/GetHostedStoresInfoServlet?
verb=GII&GII.searchString=test&GII.store=mystorename"
     rst:repeat:item="/Storefront/ItemsInfo/Items/Item">
   <td rst:content="item1:/Title">Item1</td>
   <td rst:content="item2:/Title">Item2</td>
   <td rst:content="item3:/Title">Item3</td>
 </tr>
</table>
```

Brilliant? Simplicity always has that! Let's recap:

- You define a looping section, by indicating the nodes to loop through
- You write the HTML markup for one or more loops. You use numbers to define node offsets within a loop
- You let the RST engine to do the magic!

1.9 Dynamically refresh content with AJAX

Unless you've been living on a different web planet than Internet, you heard about AJAX. Some praise it, some blame it, but the reality is that everybody uses it. So it would be a shame if RST templates were any different, wouldn't it?

> **What does AJAX do?**
> The AJAX definition ranges from *"a coined acronym for an old, poorly used technology"* to *"the core technology of web2.0"*. If you're interested in the origins, history and core technology behind AJAX start from Wikipedia. If you want to learn AJAX, start by reading some tutorials. If you just want to know what AJAX can do for your store, this is it: it can load content faster, so that people spend less time waiting for your store pages to load and focus more on the information that the store has to offer. It offers a more pleasant UI experience, as well as a more effective way of doing business online.
>
> For example, it can replace a classic "shopping cart page" that shows the cart's content every time you add a product to the cart, with a "shopping cart widget", a small companion displayed in every page, which refreshes every time the cart's content is changed, without requiring the visitor to wait for two full page refreshes.

Essentially, what you do with AJAX is: load content dynamically, in a specified area of the current page, as a result of a user action (also called *event*). Example: refresh the shopping cart widget whenever the visitor adds an item to the cart. Achieving this with RST is as simple as having said that: *refresh the shopping cart widget whenever the visitor adds an item to the cart*!

The RST attributes available for refreshing content via AJAX are:

- **rst:ajax:event** Name of event that triggers the AJAX call
- **rst:ajax:load** ID of element to load the AJAX content into
- **rst:ajax:pagename** store page name to load via AJAX
- **rst:ajax:param** parameters sent to the store page
- **rst:ajax:loader** ID of element to display while loading the AJAX content
- **rst:ajax:onload** function name to call when the AJAX call completes

To load content dynamically using AJAX, you need:

- a flyout div to load content into
- an element and an event to trigger the content load

The element that triggers the content load should have the following RST attributes:

- **rst:ajax:event** - the name of the event
- **rst:ajax:load** - the ID of the div to load content into
- **rst:ajax:pagename** - the store page name to load the content from
- any number of **rst:ajax:param="name=value"** attributes defining the parameters for the store page to load on demand

Any additional code (show/hide the div, position the flyout relative to other elements in the page etc.) should be specified normally in the **onXXX** event handlers.

Example:

flyoutcontent.html

```
<notag rst:if="{context:item}">You added item <notag rst:content="{context:item}"/></notag>
```

home.html

```
<div id="flyout" rst:include="flyoutcontent">Flyout content goes here</div>

<notag rst:xml="http://sell.vendio.com/GetHostedStoresInfoServlet?
verb=GII&GII.searchString=test&GII.store=mystorename"
      rst:repeat:item="/Storefront/ItemsInfo/Items/Item">
 <div rst:ajax:event="onclick"
      rst:ajax:load="flyout"
      rst:ajax:pagename="flyoutcontent"
      rst:ajax:param="item={item1:/Title}">Click to add<br/>
   <notag rst:content="item1:/Title">item</notag>
 </div>
</notag>
```

1.10 Split your template in smaller files: using includes

Designing a template leads almost always to the need of duplicating content across pages. Whether it's about a common layout, or about a common color palette, or basic sections that are present on all pages, one can quickly identify portions of code that can be isolated and "reapplied", in the exact same form or slightly modified (*parametrized*), in more places within your template. Where this leads further is that you want to include the repetitive content in a separate file and have a way to tell the template renderer: include my repetitive section here.

With RST, this is as simple as:

- any .html file can be included in another .html file
- to include a .html file, use the **rst:include** attribute, with the value set to the name of the included file, without the .html extension

Example:

leftnav.html

```
<td width="20%">Some content</td>
```

rightnav.html

```
<td width="20%">Some content</td>
```

home.html

```
<table width="100%">
 <tr>
   <notag rst:include="leftnav"/>
   <td>Home Page</td>
   <notag rst:include="rightnav"/>
```

```
  </tr>
</table>
```

about.html

```
<table width="100%">
  <tr>
    <notag rst:include="leftnav"/>
    <td>About Us</td>
    <notag rst:include="rightnav"/>
  </tr>
</table>
```

In this (oversimplified) example, two different pages (Home and About Us) share the same left nav and right nav. This is achieved by including the left nav and right nav contents in separate files and including the files in the two pages.

2. Advanced RST | Tweaking the templates

This chapter describes a few advanced techniques that will help you build more complex templates.

2.1 Debugging your template

We designed the Store templates with speed and simplicity in our minds, so that you can apply a template to your store literally within seconds. This comes at a price: **you are responsible** for the content of your template. If your template is broken, your store will be broken. Therefore, we recommend that you turn your Store **not publicly visible** while you customizing it, or otherwise your changes will be publicly visible to your store's visitors. You can fix and **re-upload** your template as many times as you want, or you can **revert to one of Vendio's templates**.

To **test the layout** of your template, use the preview feature of the Store Template page. If you need layout changes, you do them in your local directory, repack the template, re-upload it and preview it again to see the changes.

If your template has a proper layout but the integration with the store data is broken, your store may become unusable. In this case, visiting your store will show a fatal error page and you'll have no clue what the error might be. Luckily, there's a way to inspect the errors that your template may contain. Just go to your store's URL and append a *?__debug=127* parameter and you will enter the **store debug mode**. Your store will no longer display an error page, but rather show you useful information about how the store data is retrieved and integrated in the template. If you see any messages in red, those are the errors that prevent your store from appearing properly.

> **The template debug mode**
> Appending a `?__debug=127` parameter to your page URL will show the maximum available information related to how your template fetches and integrates the store data. Debug messages related to broken XML feeds, invalid paths within the XML documents, invalid conditions or loops will show in red, while debug messages related to XML data sources, file inclusions, and data evaluations will show in green.
> You can tweak the amount of debug information that is shown in the page. Thus, the value you assign to the `__debug` parameter is calculated as a sum of the following partial detail levels:
> - 1 - show API errors (invalid XML responses)
> - 2 - show template errors (orphan tags, parse errors)
> - 4 - show loop info
> - 8 - show evaluated conditions
> - 16 - show trace info
> - 32 - show evaluated values
> - 64 - show XML feeds

> *Example:* to show API errors, template errors and XML feeds you will need to pass a debug value of 1 + 2 + 64 = 67, as in:
> `http://www.vendio.com/stores/your_store_name?__debug=67`

2.2 Design constraints in Beta phase

While we're offering Vendio Stores as a Beta service, we're continually working on improving it and enhancing its features. We're also working on eliminating a few legacy constraints that currently affect the way templates need to be designed. For maximum compliance, designers should be aware (and respect) the following constraints:

- While in Beta phase, all templates should contain at least the following pages:
  - **home.html** - the store home page
  - **category.html** - the items by category page (also used as search results page)
    - receives the *catId* or *searchString* parameters
  - **item.html** - the item detail page
    - receives the *lid* parameter
  - **cart.html** - the cart contents page
    - receives the *cartId* parameter
  - **trackorder.html** - track order page
    - receives the *order_number* and *email* parameters
  - **policy.html** - store policies page
  - **about.html** - about us page
  - **contact.html** - contact us page
  - **error.html** - fatal error page
  - **checkout_api.html** - the checkout page header & footer in XML format (see below)

- The checkout page is a special page. Although it has a .html extension, the **checkout_api.html** page needs to have the following format:

```
<notag rst:nohtml="1" rst:header="Content-Type: text/xml"/>
<CheckoutPageUIResponse><Top><![CDATA[
... checkout page top HTML ...
]]></Top><Bottom><![CDATA[
... checkout page bottom HTML ...
]]></Bottom></CheckoutPageUIResponse>
```

  - Only the page header and footer can be fully customized in the checkout page. The middle section, the checkout form, has a fixed format and its appearance can be modified through CSS
  - The checkout page can **NOT** contain a `<DOCTYPE>` definition (because both the header and the footer are included inside the `BODY` tag of the page)
  - All template assets (CSS, JS, or image files) referred from the header and footer must have absolute URLs, that is prepend the src attributes with **{context:__templatebase}**

- All template pages except the checkout page must contain a Vendio tracking image:

```
<!-- Tracking hit URL -->
<img rst:if="{context:__pagename} ne checkout_api" alt=""
width="1" height="1" rst:attr:src="{context:__trackurl}"/>
```

- The item detail page (*item.html*) may contain a Vendio counter image:

```
<img rst:xml:storeinfo="{context:__storesapiurl}?verb=GSI&GSI.store={context:__store}"
rst:attr:src="{context:__counterurl}.{storeinfo:/Storefront/StorefrontInfo/AccountId}.
{context:lid}"/>
```

- The following size limitations apply to the template file storage:
  - Maximum number of templates, per account: 50
  - Maximum size, per template: 8MB
  - Maximum size, per account: 200MB

3. Language specifications

This chapter contains the full, condensed reference of the RST language: concepts, definitions, rules, syntax. For a pocket guide of RST, please refer to the RST Quick Language Reference document.

3.1 The RST data sources

During the rendering process of a page, several data sources are used.

XML data sources

Remote XML data sources are defined by a HTTP URL to access the data from. The XML data is fetched from the given URL and stored for later access. Content is usually accessed via XPath expressions. One or more XML data feeds can be defined for each template page. The XML data is cached for a given period, controllable via the RST attributes. Local XML data sources can also be defined to refer static XML documents within the template's directory.

Context variables

Context variables represent transient data specific to a page view: the page name, pagination info (items per page, page number, search parameters), navigation info (current page, previous page, next page) etc. Technically, the context variables consist of all the GET data available in a page request plus several "magic" variables that can be computed based on the existing ones (such as the links to prev/next pages in a search results page).

Session variables

Session variables are data that the server can store between multiple page accesses. Typical usage includes "last item viewed", "user id" etc. Session variables are stored using browser cookies.

Data pointers

Shortcuts or pointers to a data source can be defined at any moment for later referencing. This is particularly useful to display recursive information (such as a category tree) or to shorten data paths.

3.2 The RST features

The RST language and the RST render engine offer the following functionality to a template:

Display dynamic data

Data from any data source (XML feed, context or session) can be accessed, processed and

displayed within the template, overriding any dummy data from the mockup.

Conditional display

Content can be conditionally displayed or hidden based on a simple test performed on the data.

Repetitive display

Similar content (such as item details for every item) can be displayed by looping on a given data source container. Moreover, iterations can be done in groups of items, so it is possible to give different formatting to different items in a group (in order to display multiple lines with 3 items on a line for instance).

Navigation support

The RST language offers an easy way to define links between pages and/or form submits, based on the context data.

AJAX support

The RST language provides an easy way to bind dynamic content fetching via AJAX to specific HTML events (for instance, to show a flyout box on mouseover).

3.3 The RST syntax

The RST language is based on adding special HTML attributes (that have a special rst: prefix) to certain HTML tags in the template.

RST paths

A RST path defines a way to access data in an XML data source. The RST path format is: **identifier:xpath@attribute**

- **identifier:**: pointer to a RST data source
- **xpath**: XPath expression to search through an XML data source
- **@attribute**: attribute name related to the XML node

Any of the parts can be missing:

- A RST path may consist of an identifier. Example: "`category`"
- A RST path may lack the identifier name. Example: "`/sf_subcateg[0]/category@id`"
- A RST path may lack the attribute. Example: "`/sf_subcateg[0]/category/name`"
- A RST path may lack the XPath expression. Example: "`category:@id`"

An XPath expression followed by an attribute is called an **extended XPath**.

RST expressions

A RST expression is any value that can be stored in a RST Attribute. A RST expression can consist of:

- an RST path: **identifier:xpath@attribute**. Example: "`category:/sf_subcateg[0]/category@id`"
- references to data containers: **context:**, **session:**. Example: "`context:page`"
- a combination of more RST expressions: **{}**. Example: "`Page {pagination:@page} of {pagination:@totalPages}`"
- nested RST expressions: "`Page {pagination:pages[{pagination:@page}]}`"
- arithmetical operations: **+**, **-**, *****, **/**. Example: "`Items {{pagination:@pageSize} * ({pagination:@page} - 1)}`"
- conditions: **eq**, **ne**, **gt**, **ge**, **lt**, **le, or, and**. Example: "`{pagination:@page} eq 1`"
- functions: **rst:q**, **rst:cbw**, **rst:pw**, **rst:uc**, **rst:ucfirst**, **rst:lc**, **rst:html**, **rst:url**, **rst:length**, **rst:substr**, **rst:round**, **rst:amount**, **rst:date, rst:seourl, rst:seokeywords**, **rst:anyof**. Example: "`{rst:substr({category:@name}, 0, 5)}...`"
- strings. Example: "`Loading page, please wait...`"

RST attributes

A RST attribute is a HTML attribute that is parsed by the RST engine and has the following format: **rst:type:identifier="expression"**. Example: "`rst:repeat:item="/storefront/categories/category`"

- **rst:**: prefix, the attribute prefix that makes it recognizable by the parser
- **type**: RST type, the type of the RST attribute
- **identifier**: RST identifier, the name of the RST entity represented by the RST attribute
- **expression**: a RST expression

4. Full language reference

This chapter describes all the RST features, in detail. Using RST attributes, you can perform the following data binding tasks:

4.1 Define XML data sources

- **rst:xml**
  - **rst:ttl** XML cache validity, in seconds (default 3600, set to 0 to prevent caching)
  - **rst:method** HTTP method to retrieve the data source (GET or POST, default GET)
  - **rst:xmlsource** XML source location (template, default: remote through http)

4.2 Display data

- **rst:content**

4.3 Access context information

- **context:**: access context information (GET, POST and "magic")
  - "Magic":
    - **context:__pagename** store page name (home, item etc.)
    - **context:page** page number in pagination data
    - **context:__prevpage** previous page number in pagination data
    - **context:__nextpage** next page number in pagination data
    - **context:__currenturl** full URL of current page
    - **context:__currentpageurl** URL of current page, without any GET parameters
    - **context:__lasturl** full URL of last store page visited
    - **context:__errmsg** fatal error message, in the **error** page
    - **context:__store** name of store owner (store URL)
    - **context:__template** name of the template
    - **context:__storebase** URL of current store (e.g. `http://www.vendio.com/stores/MyStore/`)
    - **context:__storesecurebase** HTTPS URL of current store (e.g. `https://www.vendio.com/stores/MyStore/`)
    - **context:__storehttpbase** HTTP URL of current store (e.g. `http://www.vendio.com/stores/MyStore/`)
    - **context:__storehomeurl** URL to store home page
    - **context:__jsbase** URL to the javascript libraries available on the Vendio Stores domain (e.g. `http://stores.beta.vendio.com/javascript/`)
    - **context:__templatebase** URL of template content (e.g. `http://www.vendio.com/stores/.template/MyStore/current/`)

- **context:__checkouturl** URL to Vendio Checkout
- **context:__storesapiurl** URL to Vendio Stores API
- **context:__trackurl** URL to Vendio tracking URL. Ends with `cid=`, need to append cid value from XML
- **context:__helpers** HTML code to include Javascript helper scripts and `BASE` tag

4.4 Access session information

- **rst:session**

4.5 Display content conditionally

- **rst:if** display content if condition matched (e.g. `<notag rst:if="{/items@count} gt 0" rst:content="/items@count"/>`)
- **rst:ifnot** display content if condition not matched
  - Supported operators: `eq, ne, gt, ge, lt, le, or, and`
- **rst:assert** abort page rendering with fatal error if assertion fails (e.g. `<notag rst:assert="/Response/Status">`)

4.6 Define data pointers

- **rst:define**
  - **rst:local** Make the define specific for the block it was defined in, unset / revert to previous value when the block ends

4.7 Set HTTP headers

- **rst:header** Set HTTP headers (e.g. `<notag rst:header="Content-Type: text/xml"/>`)
- **rst:redirect** Redirect (e.g. `<notag rst:redirect="http://www.yahoo.com"/>`)
- **rst:nohtml** Suppress helpers output (for AJAX or XML pages)
- **rst:error** Abort page rendering with fatal error (e.g. `<notag rst:error="Invalid quantity">`)
  - Error message is only logged, not displayed to users

4.8 Define loop sections

- **rst:repeat**
  - **rst:start**
  - **rst:max**

4.9 Override HTML attributes

- **rst:attr**

4.10 Navigation & SEO

- **rst:href**
- **rst:href:pagename**
- **rst:seo**
- **rst:action**
- **rst:action:pagename**

4.11 AJAX support

- **rst:ajax:event** Name of event that triggers the AJAX call
- **rst:ajax:load** ID of element to load the AJAX content into
- **rst:ajax:pagename** store page name to load via AJAX
- **rst:ajax:param** parameters sent to the store page
- **rst:ajax:loader** ID of element to display while loading the AJAX content
- **rst:ajax:onload** function name to call when the AJAX call completes

To load content dynamically using AJAX, you need:

- a flyout div to load content into
- an element and an event to trigger the content load

The element that triggers the content load should have the following RST attributes:

- **rst:ajax:event** - the name of the event
- **rst:ajax:load** - the ID of the div to load content into
- **rst:ajax:pagename** - the store page name to load the content from
- any number of **rst:ajax:param="name=value"** attributes defining the parameters for the store page to load on demand

Any additional code (show/hide the div, position the flyout relative to other elements in the page etc.) should be specified normally in the **onXXX** event handlers.

4.12 Include sections

- **rst:include**

4.13 The special RST tag

- **<notag>**

4.14 Set up multiple RST attributes in one tag

For each tag, the order of evaluation is:

- check tag conditions (**rst:if**, **rst:ifnot**, **rst:assert**). Skip tag if any of the conditions is not met
- set variables (**rst:define**, **rst:session**)
- process headers (**rst:error**, **rst:header**, **rst:redirect**)

- fetch XML data (**rst:xml**, **rst:xmlsource**, **rst:ttl**, **rst:method**)
- process output modifiers (**rst:nohtml**)
- process content specifiers (**rst:include**, **rst:content**, **rst:repeat**)
- process attribute modifiers (**rst:attr**, **rst:href**, **rst:seo**, **rst:action**, **rst:ajax**, **rst:src**)

4.15 Javascript frameworks

All major javascript frameworks are available under the `/javascript` directory.

- `/javascript/ext/ext-base.js`
- `/javascript/jquery/jquery-latest.js`
    - `/javascript/jquery/plugins`
    - `/javascript/jquery/plugins/interface`
    - `/javascript/jquery/plugins/ui`
- `/javascript/mootools.js`
- `/javascript/slimbox.js`
- `/javascript/thickbox.js`

September 7th, 2009
The Vendio Stores Team