

Tutorial: Building a simple website

Author: David Haasler

Table of contents

1	Building an information model.....	2
1.1	http://www.example.com.....	2
1.2	Creating schema for abstract type node.....	3
1.3	Creating a schema for the site.....	5
1.4	Creating a schema for the page.....	7
1.5	Child schemas.....	7
2	Creating a website structure.....	8
3	Editor customizing.....	10
3.1	Icons and display names in the editorial view.....	10
3.2	Adapting the labels of the editor fields.....	10
4	Transformations.....	11
4.1	Literal method »default«.....	13
4.2	HTML basic structure for rendering.....	16
4.3	Adding CSS.....	19
4.3.1	Creating the »head« method.....	20
4.3.2	Creating the »css« method.....	21
4.3.3	Integrating into the »default« method.....	22
4.4	Integrating and displaying the logo.....	23
4.4.1	Binary »default« method.....	23
4.4.2	Calling up the binary method for the logo.....	24
4.4.3	Integrating into the »default« method.....	26
4.5	Navigation.....	27
4.5.1	Path navigation.....	27
4.5.1.1	Creating the »path« method.....	27
4.5.2	Main navigation.....	30
4.5.2.1	Creating a »link« method.....	30
4.5.2.2	Creating the »navigation« method.....	31
4.5.2.3	Rendering the main navigation.....	33
4.6	Outputting content.....	35
4.6.1	Creating the »content« method.....	36
5	Closing remarks.....	38

Requirements

You should have already familiarised yourself with the onion.net Editor and perhaps have already gone through some tutorials for beginners.

Knowledge of XML, XML schema and XSLT as well as of any programming language is helpful.

We recommend that you complete the following tutorials beforehand:

- > [Define Information Model](#)
- > [Developing transformations](#)
- > [Editor customizing](#)

Description

In this tutorial, knowledge from previous tutorials is deepened and expanded. A simple website is structured step by step. The starting point is an empty onion.net system. The information model, structure, editor localisation and rendering are gradually built.

The result is a simple website with a navigation, an Ariadne thread and of course maintainable content. Thus the website only offers rudimentary functions, but is intended to make clear the general structure and procedure in onion.net.

Signs and symbols



Boxes marked with an arrow symbol and a green border contains instruction of what to do next.



This kind of boxes contains tips and tricks.

Source code is shown in blue boxes.

1 Building an information model

As a first step, we will create the schemas needed for the website. We need two different types of page for our website:

- > a *site* and
- > a *page*.

The *site* is used as the welcome page of the website and there can only be one of these on a website. In addition, contents can be maintained on the *site*, which are to apply for the entire website and be inherited. For example, the logo is to be visible on each page of the website. But the graphic itself should be stored in only one place as far as possible.

Under a *site*, it should be possible to create *pages*. The navigation is then later formed from this structure automatically. It is obvious that no further *sites* are to be created under the *site*, but just *pages*, i.e. content pages.

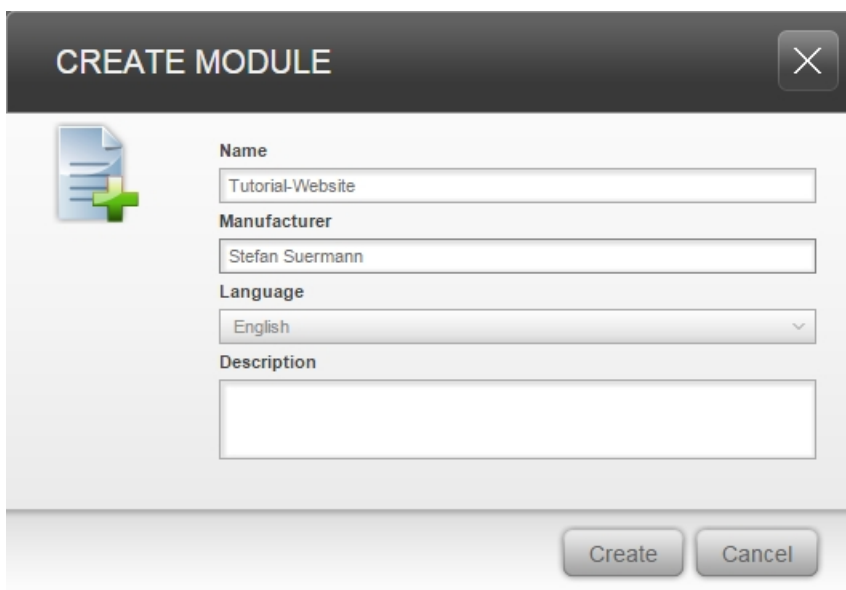
Since *site* and *page* are nevertheless very similar in principle, we will work with inheritance. Many contents are to be maintained both for a *site* and for *pages*, e.g. free text in the contents pane. Moreover, both types are visitable structure points. In order to implement the inheritance, we create in the schemas an object *node* from which *site* and *page* are derived.

1.1 <http://www.example.com>


We first create a module for our tutorial as well as an abstract schema <http://www.example.com>, under which the new schemas are to be grouped.



To do this, switch to the module administration in the onion.net Editor and create a new module. Call it **Tutorial-Website**.



CREATE MODULE



Name
Tutorial-Website

Manufacturer
Stefan Suermann

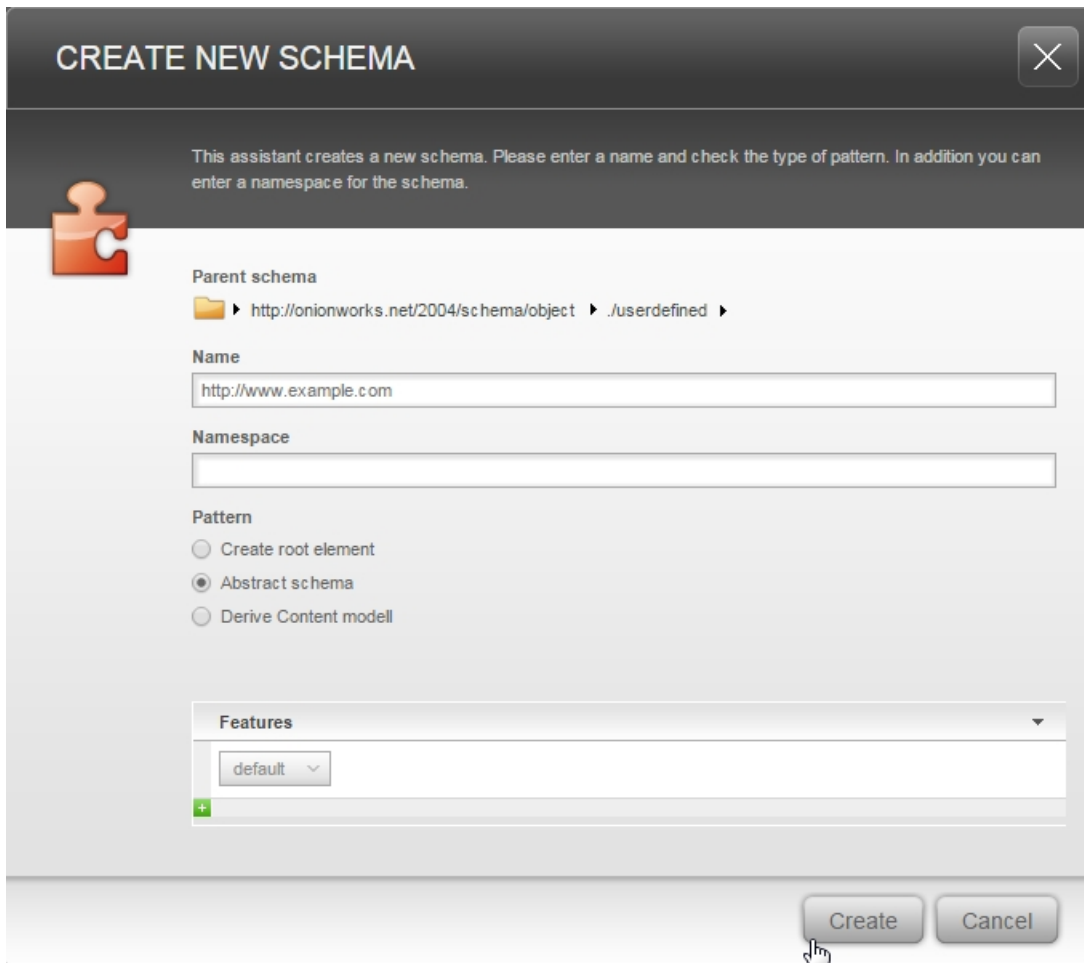
Language
English

Description

Create Cancel



Add the *schemata* group within the module and create the abstract schema **`http://www.example.com`** in it.



1.2 Creating schema for abstract type node



Underneath the schema **`http:// www. example.com`** you now create the schema **`http://www.example.com/node`**. This is an abstract schema, since it is not to be possible for concrete objects to be created by **`node`**.



When creating a new schema, the *SchemaLocation* of the parent schema is always assumed as a suggestion in the assistant. In this case, the field *Location* is therefore already initialised with *`http://www.example.com/`*. You now just have to attach the name of the new schema to be created at the end.

After creating the schema, you now define the elements to be contained by this schema and which are then available for both the **site** and **page**, derived from **node**. In our example there is a rich text field intended to assume the content of a page.

An abstract schema receives the following content by default:

```
<xs:schema
  xmlns:references="http://onionworks.net/2004/references"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:onion="http://onionworks.net/2004/schema"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  onion:schemaLocation="http://www.example.com"
></xs:schema>
```



Extend this content as follows:

```
<xs:schema
  xmlns:xhtml="http://www.w3.org/1999/xhtml"
  xmlns:references="http://onionworks.net/2004/references"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:onion="http://onionworks.net/2004/schema"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  onion:schemaLocation="http://example.com/node"
>
  <xs:element name="node" type="node" />
  <xs:complexType name="node">
    <xs:sequence>
      <xs:element name="text" type="xhtml:Flow" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

An element of the type **node** was added, the content of which was defined in the »*complexType name="node"*«. It consists of an element *text*, which is of the type *xhtml:Flow* and thus represents a rich text field.

The namespace *xhtml* has not yet been defined however, meaning saving the changes is not possible. You therefore enter the *xs:import* instruction for the name area *xhtml* in the second line:

```
<xs:import schemaLocation="http://www.w3.org/2002/08/xhtml/xhtml11-strict.xsd"
  namespace="http://www.w3.org/1999/xhtml" />
```

The **node** schema then looks as follows:


```
<xs:schema
  xmlns:xhtml="http://www.w3.org/1999/xhtml"
  xmlns:references="http://onionworks.net/2004/references"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:onion="http://onionworks.net/2004/schema"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  onion:schemaLocation="http://example.com/node"
>
  <xs:import schemaLocation="http://www.w3.org/2002/08/xhtml/xhtml1-strict.xsd"
  namespace="http://www.w3.org/1999/xhtml" />
  <xs:element name="node" type="node" />
  <xs:complexType name="node">
    <xs:sequence>
      <xs:element name="text" type="xhtml:Flow" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

1.3 Creating a schema for the site



Now create the schema **site** below the schema **node**. **Site** is to be derived from **node**. For this purpose, select the option *Derive content model* in the dialogue *Create new schema*. In a drop-down box, you will now be offered schemas which you can derive. In our case this is merely *node (ComplexType)*: Select this.

CREATE NEW SCHEMA



This assistant creates a new schema. Please enter a name and check the type of pattern. In addition you can enter a namespace for the schema.

Parent schema

▶ http://onionworks.net/2004/schema/object ▶ .userdefined ▶ http://www.example.com ▶ .node ▶

Name

http://www.example.com/node/site

Namespace

Pattern

☐ Create root element
☐ Abstract schema
☒ Derive Content modell

node (ComplexType) ▼

Features

default ▼

+

Create

Cancel

The schema of the derived object now looks as follows:

```

<xs:schema
  xmlns:references="http://onionworks.net/2004/references"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:onion="http://onionworks.net/2004/schema"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  onion:schemaLocation="http://www.example.com/node/site"
>
  <xs:redefine schemaLocation="http://www.example.com/node">
    <xs:complexType name="node">
      <xs:complexContent>
        <xs:extension base="node"></xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:redefine>
</xs:schema>

```

A *site* is to also be able to contain the logo of the website.



For this purpose, extend the element that is still empty `<xs:extension base="node">` to include the following attribute:

```
<xs:attribute name="logo" type="xlink:binaryReference" />
```

In order to be able to use these types, the *xlink* schema must be imported. In this case, you can drag the schema both into the attribute *schemaLocation* and *namespace*:

```
<xs:import schemaLocation="http://www.w3.org/1999/xlink"
namespace="http://www.w3.org/1999/xlink" />
```

Overall, the content should now look as follows:

```
<xs:schema
  xmlns:references="http://onionworks.net/2004/references"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:onion="http://onionworks.net/2004/schema"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  onion:schemaLocation="http://www.example.com/node/site"
>
  <xs:import schemaLocation="http://www.w3.org/1999/xlink"
  namespace="http://www.w3.org/1999/xlink" />
  <xs:redefine schemaLocation="http://www.example.com/node">
    <xs:complexType name="node">
      <xs:complexContent>
        <xs:extension base="node">
          <xs:attribute name="logo" type="xlink:binaryReference" />
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:redefine>
</xs:schema>
```

1.4 Creating a schema for the page

Now create a further derived schema **page** below **node**. Proceed as previously described under **site**.

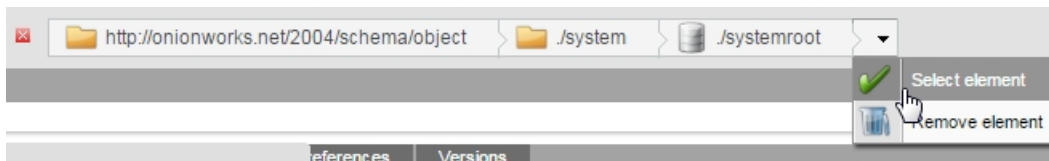
The **page** does not need any elements or attributes other than **node** - the default content of the derived schema can therefore be used.

1.5 Child schemas

Next, define which schemas may be created under which other schemas. First of all, it is to be possible to create a **site** below the *onion.net* root element, in order for a website of this type to be created in the onion.net Editor at all.



Navigate to the schema *site*. Then click on the tab *Structural reference* and add the *systemroot* schema (<http://onionworks.net/2004/schema/systemroot>).



Next, it must be defined that **pages** may be created under a **site** as well as further **pages** underneath a **page**.



Click the tab *Children schemata* in the **site** schema and drag the schema **page** into the white area.

It should also be possible to create a **page** schema below a **page** schema. **Page**-schemata must therefore also be defined as child schemas of a **page** schema. Proceed here as with the **site** schema.



2 Creating a website structure

You can now begin creating the first documents of your website. We choose the following simple structure:

- > Homepage
- > Services
- > Service group 1
- > Service group 2
- > Products
- > Product group 1

- Product group 2
- Contact
- Site notice

In order to create this structure, you must proceed as follows:



Switch to the editorial view. Now right-click on the root node (/) and select *<http://www.example.com/node/site>*. Allocate the name **Homepage**. As you can see, you can now already maintain content. Keep the content as simple as possible for the moment - a short sentence is enough. (e.g. "Lorem ipsum dolor sit amet."). Now save the document.

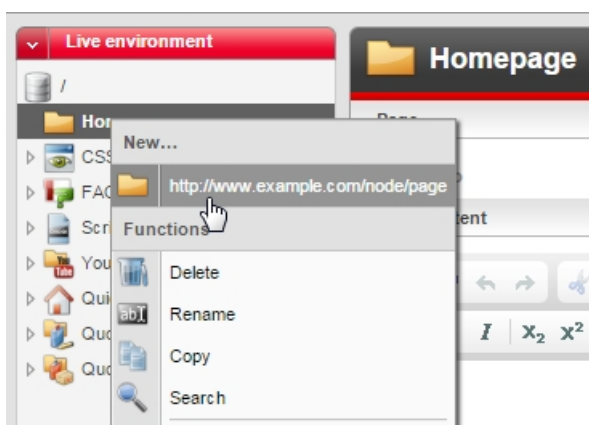
It will appear directly at the very bottom of the tree view with the name **Homepage**. You can drag and drop it right to the top of the tree view.



A grey bar will show you where the object will be put when you release the mouse button.



Now create further pages below the **Homepage** by clicking on the document **Homepage** with the right-hand mouse button and creating a *<http://www.example.com/node/page>* underneath it. Allocate a clear name for the pages also and maintain a short text on the page. Create the documents as suggested at the top of the tree structure.



3 Editor customizing

When creating the structure points you must have noticed that, in the context menu and in the object detail window, the terms and schema locations were displayed which we had previously allocated in the schema editor. This very technical display is not very pleasant for editors in particular. Moreover, the objects in the tree cannot be distinguished so well since they all have the same icon. We will therefore adapt the editor next.

3.1 Icons and display names in the editorial view

First of all, the objects that can be created by means of the schemas **site** and **page** are to be given icons and self-explanatory names. In this way they can be distinguished between better in the editor.



To do this, we switch to the module view and select the schema **site**. Select the *Settings* tab and then the sub-tab *Localization*. Select **Site** as the display name and allocate meaningful icons.



Many web pages offering free downloadable icons or iconsets for certain topics can be found on the web using relevant search engines.



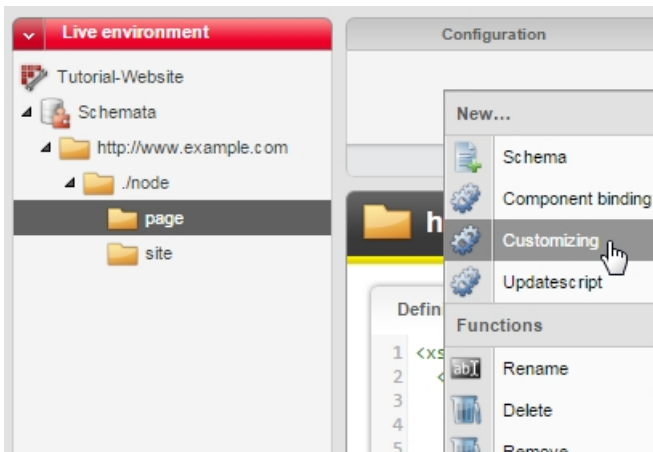
Follow the same procedure for the schemas *page* and *node* and allocate the names **Page** and **Node**.

3.2 Adapting the labels of the editor fields

So that the schemas also receive self-explanatory names when created, we now add language configurations.



To do this, navigate within the module view to the schema *site*. Right-click in the *Configuration* section (upper right side) and select *Customizing*. Select *English* as the language.



Insert the following code example within the `< o: settings>` element:

```
<o:component aspect="Label" match="@logo">
  <label>Logo</label>
</o:component>
```



Follow the exact same procedure for the schema **node** and enter the following code:

```
<o:component aspect="Label" match=".node">
  <label>Page</label>
</o:component>
<o:component aspect="Label" match="#node .text">
  <label>Content</label>
</o:component>
```

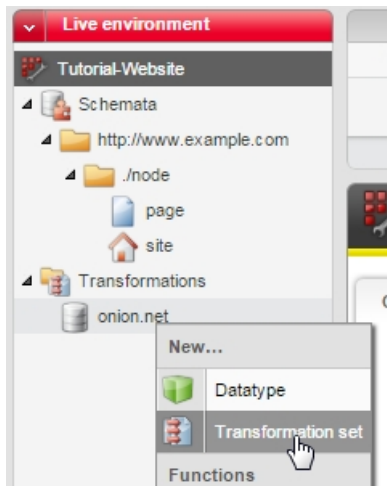
4 Transformations

Your website has now been created. Only it cannot yet be shown in the browser. We will now change this by starting to create transformations that will ensure the output of contents into the browser.

First of all, we will create a transformation group. All transformations for the tutorial website are to be located there.



For this purpose, right-click on the root node **Tutorial-Website** in the module view and select *transformation container*. Leave the name as **transformations**. Now right-click on the node that has just been created and select *data source*. Indicate **Onion** as the name. The type *Onion* is retained. Save the data source. Now create a transformation group by right-clicking beneath the data source *Onion* and call it **http://www.example.com** before then saving it.



Transformation groups are used for bundling logically identical data types

Now create the first transformation underneath.

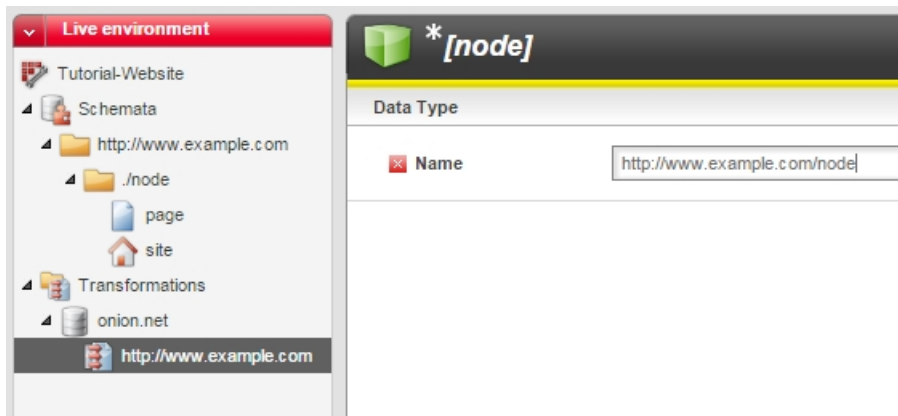


To do this, right-click on the transformation group that has just been created and select the item *Data type* from the context menu *New...* Select *node* here, since the transformation is to apply for our **node** schema.


No connection to the schema is established by the title alone.



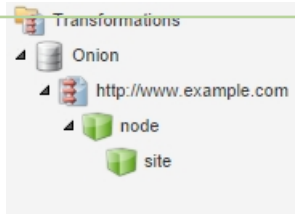
Click within the data type **node** on *Name* in order to create the optional field. The schema must now be inserted into this field. As before, you do this by dragging and dropping from the schema administration. So drag the schema **node** into the *Name* field and then return both the data type and the transformation group.




An XML schema is now clearly allocated to the transformation.



Now, in a similar way, create below **node** a further data type, which you name **site**. Drag the appropriate schema into the *Name* field. Then return this data type also.




So that you can now preview the transformations, the render engine on which you wish to allow a preview must be defined.



To do this, right-click on *Transformations* within the structure tree and select the point *Configure* in the context menu *Web server settings*. Now select *Preview* and confirm the dialogue by clicking on the button *Apply*.

4.1 Literal method »default«

We will now create a *literal method* for our data type. Literal methods are able to create text-based outputs (e.g. HTML) in the web browser.



To do this, right-click on the data type **node** and create a new literal method from the context menu. Give this method the name **default**.


A number of settings for this method can be made under the title. We must change the **access modifier** in this case. The access modifier controls from where the method may be called.


Access modifier	Description
Internal	Can only be called from other transformations
Protected	Can be called from other transformations and from .net code (e.g. extensions)
Public	Can be called via other transformations, from .net code and public interfaces (direct call using URL)




For the **default** method, we select *public* .

NEW DATAOBJECT






This assistant opens a new editing form. Please enter a title and check the data type.

Path
 node

Name

Datatype
 Literal method

Access modifier

Output cache

MIME-Type

Data view



A method with the name **default** is considered as a *default method* for the transformation system. As soon as a public literal method with this name is available for a data type, the point *Preview* appears automatically in the context menu. This opens, in a new browser window, the rendering of the method *default* for the appropriate object.

How our data is displayed can now be defined in the *Transformation* tab.



Click in the *Transformation* tab and press **[CTRL] + [Space]**. Confirm the dialogue without making a selection.

The following default transformation will be created:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:output
    method=" "
    omit-xml-declaration="yes"
    indent="no"
  />
  <xsl:template match="/"></xsl:template>
</xsl:stylesheet>
```



In the `<xsl:output>` element, enter the value **xml** into the *method* attribute. You enter the desired output within the `<xsl:template>` element. We will make do with **"Hello world!"** for the time being. Now save the transformation.

The complete transformation now looks as follows:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:output
    method="xml"
    omit-xml-declaration="yes"
    indent="no"
  />
  <xsl:template match="/"> Hello World </xsl:template>
</xsl:stylesheet>
```



In order to test the template directly, right click, in the content management, on the **site** or a **page** you created further above. Since both the **site** and the **page** are derived from **node**, this method **default** is taken for outputting if there is no direct **default** method for the schema **site** or **page**.

Since the editor has now found a *default* method for our website documents, it offers us the function *Preview* in the context menu.



You may have to return the **default** method once first and refresh the editor by pressing F5 in order to be displayed the *Preview* entry in the context menu.

If you click on this menu item, a browser window with the text "Hello world!" opens, exactly as indicated in the transformation.

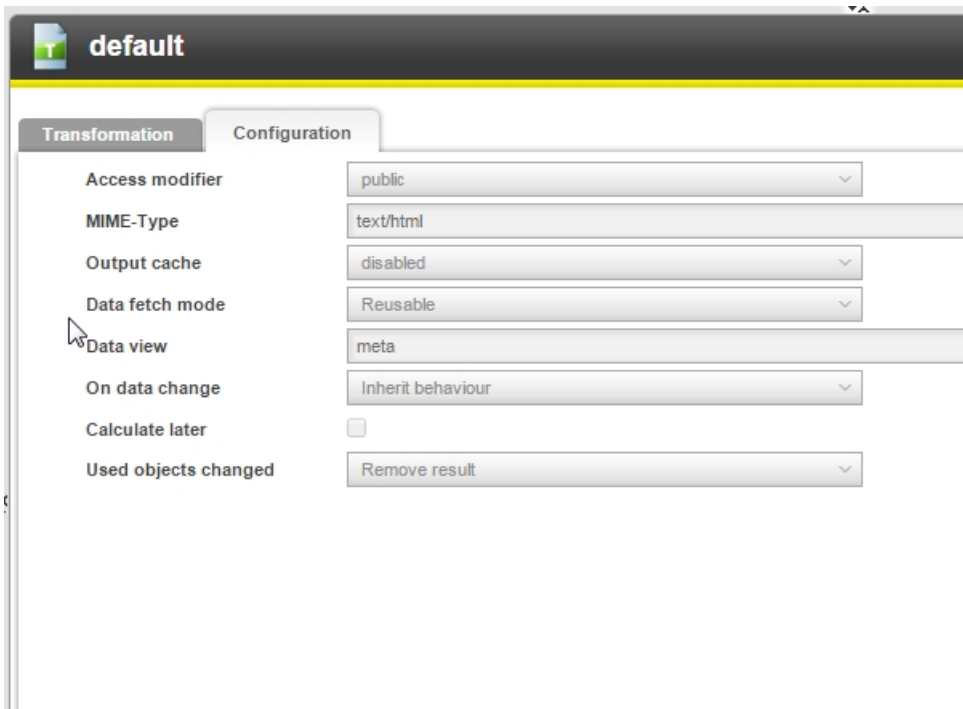
4.2 HTML basic structure for rendering

But we will of course not settle for the output "Hello world!". Instead, an HTML basic structure is to be output as early as in the first step, and we will gradually integrate the functionalities into this, such as navigation.

Since we would now like to output data from the displayed documents, we must change the **data view** of the method *default*.



To do this, click on the literal method **default** of the **node** data type. Select the value *meta* as the *Data view* in the tab *Configuration*. Save the changes.



Transformation	Configuration
Access modifier	public
MIME-Type	text/html
Output cache	disabled
Data fetch mode	Reusable
Data view	meta
On data change	Inherit behaviour
Calculate later	<input type="checkbox"/>
Used objects changed	Remove result

Now modify the content of the transformation as follows:

```
<xsl:stylesheet
  xmlns:web="http://onionworks.net/2004/renderengine/web"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:output
    method="xml"
    omit-xml-declaration="yes"
    indent="no"
    doctype-system="about:legacy-compact"
    encoding="utf-8"
  />
  <xsl:template match="/onion:object">
    <web:responseHeaders expires="0" />
    <html lang="de" xml:lang="de">
      <head>
        <title>
          <xsl:value-of select="@onion:name" />
        </title>
      </head>
      <body>
        <div id="page">
          <div id="contentNav">
            <div id="header">
              <p>Hier erscheint der Header</p>
            </div>
            <div id="path">
              <ul>
                <li>Hier fügen wir eine Pfadnavigation ein</li>
              </ul>
            </div>
            <div id="navigation">
              <p>Hier kommt die Navigation hin</p>
            </div>
            <div id="content">
              <h1>
                <xsl:value-of select="@onion:name" />
              </h1>
              <p>In diesem Bereich wird der Content der Seite ausgegeben</p>
            </div>
            <br class="clearBoth" />
          </div>
        </div>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Apart from changes to the `<xsl:output>` element for the purpose of rendering the correct *doctype* and inserting the element `<web:reponseHeaders />` for setting caching information, this template mainly contains static HTML code. The only exception is the output of the page title both in the *title* element in the *header* and in the *div* with the ID *content*.

Through having selected *meta* as the data view, we can access the attribute *onion:name*. The value we indicated in the field *title* as the document name will be in this attribute.



Now open the preview of a document of your website.

You will now already see the HTML basic structure. As well as the dummy texts, which identifies the sections of the website, some first fields of the documents are already output. In the title of the website on the one hand as well as in the heading *h1*.

4.3 Adding CSS

Next, the appearance of the HTML is to be improved by means of CSS. We could now insert a `<style type="text/css">` area into the `<head>` area in the **default** method, but that would make the method unnecessarily long and unclear. Instead, we will create a new literal method **head**, which calls a new literal method **css**.

We will create the method **css** under the data type **site** and not under **node**. This is for the following reason: The CSS contains no dynamic contents and is therefore exactly the same for each page. It should therefore be accessible under one URL only and not under a URL for each page of the website.

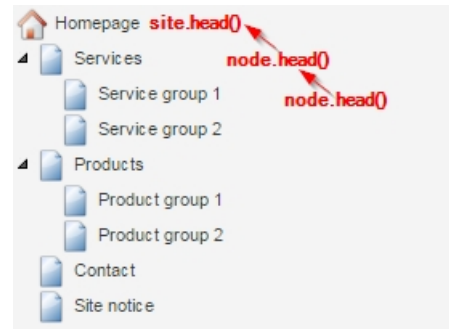
Due to inheritance however, when rendering a document, the method is looked for first in the data type corresponding to the document (e.g. a **page**). If there is no suitable method in this data type, the superordinate data type (in this case **node**) will be searched owing to the inheritance.

We therefore go via the intermediate method **head**, which then calls the **css** method on the **site** or creates the URL for integrating the stylesheet.

4.3.1 Creating the »head« method

We will call the method `css` in a further method `head` and in so doing proceed as follows:

Since we would like to find the **site** from a **page**, we create, under the data type **node**, a method **head**, which does nothing more than call a method with the same name on the parent element. Thus we go upwards along the structure, since the method **node.head()** applies both for the **site** and for the **page**.



So that this “going upwards” stops automatically at the **site**, we create the actual method **head** there we would like to call, since the method `css` is then ultimately called for the **site** in this method. We therefore also store this method **css** on **site**.

We begin with the actual method **head**, which is to integrate the CSS.



Now create the literal method **head** in the data type **site**. This does not have to be *public*, since it is only called internally. Also, *no data* view needs to be indicated since *no data* of the documents needs to be accessed.

In this method, an HTML `<link>` element is generated, which we will later load in the **default** method within the *HTML head* section. We do this via the following XSLT in the `head` method:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:output
    method="xml"
    omit-xml-declaration="yes"
    indent="no"
  />
  <xsl:template match="/">
    <link
      type="text/css"
      rel="stylesheet"
      href="{c.literalUri('css')}"
      media="screen, projection"
    />
  </xsl:template>
</xsl:stylesheet>
```

There is still no method **head** for the **page**. As described above, we will create a method under **node** for this. This has the advantage that it is also directly applicable for future page types that are derived from **node**.



Now create a literal method **head** below **node**. This requires the data view *meta*. Then assume the following transformation and save the method.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:output
    method="xml"
    omit-xml-declaration="yes"
    indent="no"
  />
  <xsl:template match="/onion:object">
    <c.literalCall id="{@onion:parent}" method="head" />
  </xsl:template>
</xsl:stylesheet>
```

An interesting thing about this method is the *c.literalCall*. The method **head** is called here, but with the ID of the parent document. So if the method is called on a **page**, it calls itself (or a method with the same name) on the parent. If the parent is the **site**, the method **head** is taken, which is to be found under **site**. This then takes care of the integration.

4.3.2 Creating the »css« method

As an *href* attribute of the element, a link to the literal method **css** is built here. We will now create this in the following.



Create the literal method **css** under the data type **site**. It must be possible for this method to be called publicly and it must have *text/css* as a MIME type. Enter the following content:

```

<xsl:stylesheet
  xmlns:web="http://onionworks.net/2004/renderengine/web"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:output method="text" />
  <xsl:template match="/">
    <web:responseHeaders expires="60" />
    body { color: #374041; background-color: #f5f5f5; font-family: Verdana, Arial,
    sans; font-size: 12px; } /* ##### reset styles ##### */ * { margin: 0; padding:
    0; } /* ##### Layout ##### */ #page { width: 980px; margin: 0 auto;
    background-color: #fff; margin-top: 5em; } /* Header */ #header #logo { border:
    0 none; margin: 1em; } /* Content */ #content { margin: 2em 2em 0 18em; padding:
    1em 2em 2em 2em; -moz-border-radius: 10px; -webkit-border-radius: 10px; border:
    1px solid #707070; background: -moz-linear-gradient(0 0, #ebf3fc, #fff);
    background: -webkit-gradient(linear, 0 0, 0 100%, from(#ebf3fc), to(#fff));
    /* IE 5.5 - IE 7 */ filter:
    progid:DXImageTransform.Microsoft.gradient(startColorstr=#ebf3fc,
    endColorstr=#FFFFFF); /* IE 8 */ -ms-filter:
    "progid:DXImageTransform.Microsoft.gradient(startColorstr=#ebf3fc,
    endColorstr=#FFFFFF)"; } /* ##### Styling ##### */ h1, h2, h3, h4, h5, h6 {
    color: #b51d48; text-shadow: 2px 2px 2px #99a7a8; } h1 { font-size: 2em;
    padding-bottom: 0.2em; border-bottom: 1px solid #374041; } p, ol, ul, h1, h2,
    h3 { margin-bottom: 0.5em; } ol, ul { margin-left: 3em; } a { text-decoration:
    none; color: #b51d48; } a.active { font-weight: bold; } a:hover {
    text-decoration: underline; } #contentNav { border: 1px solid #707070; overflow:
    hidden; } /* Navigation */ #path { font-size: 0.9em; overflow: hidden; padding:
    0.5em; border-top: 1px solid #707070; border-bottom: 1px solid #707070; }
    #path ul { list-style: none; margin-left: 0; } #path li { float: left;
    margin-right: 1em; } #navigation { float: left; margin-top: 1em; width: 150px;
    padding-left: 0.5em; } #navigation ul { list-style: none; margin-left: 0; }
    #navigation ul ul { margin-left: 1em; } #navigation li a { display: block;
    line-height: 1.2em; padding: 0.5em 0; } /* ##### helper ##### */ .clearBoth {
    clear: both; }    </xsl:template>
  </xsl:stylesheet>

```

4.3.3 Integrating into the »default« method

We now load the method **head** in our **default** method of the data type **node**.

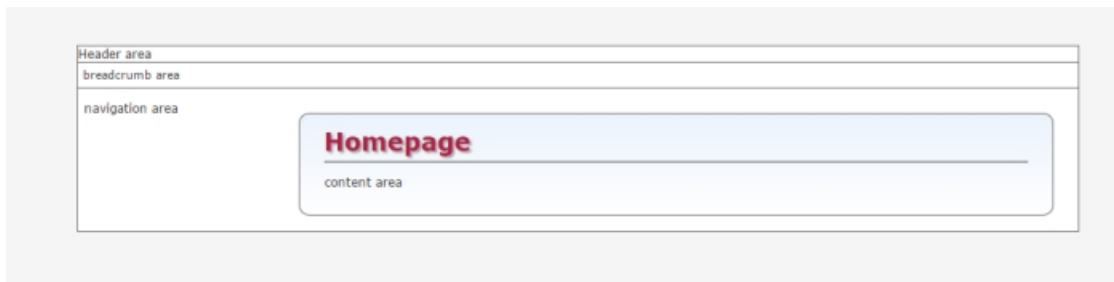


To do this, extend the element `<head>` in the **default** method to include the following call: `<c.literalCall method="head" />`

Overall, the head section in the **default** method will then look as follows:


```
<head>
  <title>
    <xsl:value-of select="@onion:name" />
  </title>
  <c.literalCall method="head" />
</head>
```

If you now preview one of the website documents, the stylesheets will now also be used, causing the website to be displayed differently.



4.4 Integrating and displaying the logo

In the schema **site** you have the possibility of integrating a logo. This is to also be integrated into the output in this step.



First load a logo into your **site Homepage** and save the document. Check out the **Homepage** for this purpose (if you have not already done so) and click on the field *logo* in order to show it. Then click on *select file* in order to select a graphic from the local computer. Use a graphic format for this which can be shown on the web, i.e. JPG (RGB colour space), PNG or GIF. So that the layout does not go to pieces, do not select too high a picture size. A size of 350x200 pixels for example would be suitable.

In order to show a binary document, we also need a binary method (as opposed to the literal method, which can output text).

4.4.1 Binary »default« method



Create a new *binary method* with the name of **binary.default** for the data type **node**. Indicate *public* as the *access modifier* and **content** as the data view. Use the default method below. This method simply delivers any kind of binary file and transfers it to the browser.



The prefix *binary.* is necessary in the file name since there is already a method with the name default on the data type node. The prefix *binary.* can therefore be placed in front of binary methods and is only evaluated systemically. In the object structure window you will see that the prefix does not have any influence on the actual method name. The same applies for XML methods with the prefix *xml.*

```
<xsl:stylesheet
  xmlns:reg="http://exslt.org/regular-expressions"
  xmlns:b="http://onionworks.net/2004/renderengine/binary"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:param name="select" />
  <xsl:template match="/">
    <xsl:variable name="ref" select="c.resolveNode($select)" />
    <xsl:value-of select="b.write($ref)" />
    <xsl:variable name="mimeType">
      <xsl:choose>
        <xsl:when test="count(reg:match($select, 'A\d', 'gi')) > 0">
          <xsl:variable name="element"
select="c.resolveNode(substring-before($select, 'A'))" />
          <xsl:value-of select="$element/@*[local-name() =
concat(local-name($ref), '.mimeType')]" />
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="$ref/@onion:mimeType" />
        </xsl:otherwise>
      </xsl:choose>
    </xsl:variable>
    <b:output mimeType="{ $mimeType }">
      <b:webResponse expires="60" />
    </b:output>
  </xsl:template>
</xsl:stylesheet>
```

4.4.2 Calling up the binary method for the logo

We now have a method that can output binary files. In the next step we will create a method which integrates this logo so that it can be shown on our website.

We follow exactly the same procedure here as with the **head** method, ensuring that under the data type **node** there is a method **logo** which forwards the query to the parent. On the data type **site** on the other hand, there is then the **logo** method, which ensures the output of the logo.



In this way, the logo is inherited from the welcome page to all subpages. The same also applies for the CSS.



In the data type **site**, create a literal method **logo** with the *Data view* **content** and the following content:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:output
    method="xml"
    omit-xml-declaration="yes"
    indent="no"
  />
  <xsl:template match="/node">
    <xsl:if test="count(@logo)">
      <a href="{c.literalUri()}">
        
      </a>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

In this method it is first checked whether the attribute *@logo* is filled at all. Since the attribute is not a mandatory field, it can also be empty. Without a check we would then have a *broken image*.

If a logo is present, it is integrated as a picture. In addition, the logo is linked with the welcome page. Calling *c.literalUri()* without specifying further parameters causes a URL to be built on the current document using the method **default()**.



Since the current document is the **site**, the logo is always linked with the welcome page.

What is interesting when integrating the logo as a picture is the filling of the `src` attribute . A binary method is called here which has the name **default**. This is our very **binary.default** created a short while ago for the displaying of binary data. As you can see, we do not need to indicate the prefix *binary*. here either. The parameter *select* is filled with the element of the document containing the binary data.



For a more detailed description of the method `c.generateId()`, we recommend the extension reference.

So that the method can work its way up to the site when a page is called, we need a **logo** method below **node** (similar to the **head** method).



Create the literal method **logo** in the data type **node**. In doing so, select meta as the data view. The content is nearly identical to that of the **head** method:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:output
    method="xml"
    omit-xml-declaration="yes"
    indent="no"
  />
  <xsl:template match="/onion:object">
    <c.literalCall id="{@onion:parent}" method="logo" />
  </xsl:template>
</xsl:stylesheet>
```

4.4.3 Integrating into the »default« method

The call of the **logo** method must now be integrated into our HTML structure in the **default** method.



To do this, replace the content of the *div* element with the *id* attribute *header* by calling the literal method **logo**: `<c.literalCall method="logo" />`

If you now preview a **page** or the **site**, the logo is rendered into the **head** area instead of the text. Moreover, it is linked with the **site**, meaning the welcome page can be switched to from every page.

4.5 Navigation

Two navigations are to now be created automatically using the structure of the created documents.

First, we of course need a *default* navigation, which shows all menu items of the first level below the site, all children of the menu item currently selected as well as all siblings of the levels inbetween. It serves for moving within the page structure, i.e. for “surfing” the website.

Secondly, we want to create a path navigation which shows all documents from the welcome page to the current page. This navigation is also called “Ariadne thread” or “breadcrumb path”. It serves for the user’s orientation, since it shows him the path to the current page at all times.

Since creating the path navigation is a bit simpler, we shall begin with this.

4.5.1 Path navigation

The path navigation represents the click path to the current page. This means that you basically only need to go one level upward each time from the current page up to the site in order to structure the path navigation. The illustration shows as an example the path navigation for the page *Product group 2*.

You are here: [Homepage](#) [Products](#) [Product group 2](#)

Like with the method **head**, we will structure a recursion. To do this, we create a method **path** which is called time and again on the respective parent. The first call (on the lowest **page**) only renders the title of the **page** that is not linked. The *title* of the **page** is also linked on the parent **pages**. Controlling whether a link is to be created or not takes place via a parameter which is transferred to the method at the time of calling. This is set to *false* by default (so also at the time of the first call), and is then explicitly transferred with *true* at the time of the parent calls.

So that the sequence is correct and the bottom page is not rendered first, the link is only actually output in the return of the recursion.

4.5.1.1 Creating the »path« method



Create a literal method **path** under the data type **node**. This requires the *data view meta* in order to access the parent.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:param
    name="link"
    c.type="Boolean"
    c IMPLIED="false"
  />
  <xsl:output
    method="xml"
    omit-xml-declaration="yes"
    indent="yes"
  />
  <xsl:template match="/onion:object">
    <xsl:call-template name="renderParent" />
    <li>
      <xsl:choose>
        <xsl:when test="$link">
          <a href="{c.literalUri()}">
            <xsl:value-of select="@onion:name" />
          </a>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="@onion:name" />
        </xsl:otherwise>
      </xsl:choose>
    </li>
  </xsl:template>
  <xsl:template name="renderParent">
    <c.literalCall
      id="{@onion:parent}"
      method="path"
      link="true"
    />
  </xsl:template>
</xsl:stylesheet>
```

This XSLT is divided into two templates. The first is the default template, which is called via the `<xsl:template match="...">` element. First of all, the template with the name **renderParent** is called. You will find this at the end of the method.

In this template, similarly to the **head** method, the method of the same name **path** is called again with the ID of the parent. The recursion is thus initiated here.

This is done until an own method **path** is found again under **site** and the recursion consequently stopped, since no further calls are taking place.

After the template call the title of the current object is then output. The link is created, or not created, depending on the control parameter *link*. Because this instruction only stands for the parent after the rendering, the correct sequence is created from top to bottom.

Since we want to create a semantically correct HTML, our path navigation must be a list. Therefore each *path* element is surrounded by a *li* element. The necessary *ul* element is added at the time of the initial call, which takes place in the **default** method.



Create a literal method **path** in the data type **site**. This requires the *data viewmeta*. Assume the following content:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:import href="../path" />
  <xsl:template name="renderParent" />
</xsl:stylesheet>
```

This method imports the content of the method **node.path()** (searching is carried out in the data types situated hierarchically above due to the "../" placed in front), but overwrites the template **renderParent** contained in it with an empty template. This means the output of the current document title takes place, but the call of the method **path** of the parent no longer does.



The import of the “general” method from **node** has the advantage that the markup stands for an element of the path navigation at only a single point. If something changes, e.g. a class is added to the *li* element, or the text is to be surrounded with a *span* element, this only has to be changed in one place.



Change the **default** method as follows: Replace the content of the *div* element with the *id* attribute **path** by calling the literal method **path** with a surrounding *ul* element and a describing text:

```
<ul>
  <li>
    <xsl:value-of select="'Sie befinden sich hier: ' " />
  </li>
  <c.literalCall method="path" />
</ul>
```

If you now call up a preview, you can see the clickable path navigation.

4.5.2 Main navigation

In the next step we will create the navigation. The navigation structure essentially consists of two parts:

One part is the output of all pages and subpages starting from the welcome page. This is relatively simple. We just need to call one method on the welcome page and then go through all children recursively. However, this is not exactly the effect we want to have. The sub-navigation of a page is to only be shown if we are on the page. The navigation should be identical to the illustration on the right. You can see at a glance: we are on the page *Products*. Therefore the two subpages can also be seen. Although the page *Services* also has subpages, these are not displayed however. For this functionality we need to know which path we are in.

Services
Products
 Product group 1
 Product group 2
 Contact
 Site notice

The second part, which is needed for the navigation rendering, is therefore based on the path navigation from the last chapter. For this part, we use a method **navigation**, which collects all documents from the current document up to the **site**. For the actual rendering of the navigation, the method **navigation.render** is then used. Called on the **site** at first, it goes through all children and highlights the current page if necessary or initiates the rendering of the sub-navigation.

4.5.2.1 Creating a »link« method

Since we must create a link for the respective navigation point in several places from the recursion, we first create a method **link**, which assumes this task. The reason for this is that the information from the *data view meta* is needed for creating the link (*title*). For building the structure however, the children must equally be gone through, for which the *data view children* is used.

For the data type **node**, create a literal method **link** with the *data view meta*.


```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:output
    method="xml"
    omit-xml-declaration="yes"
    indent="no"
  />
  <xsl:param
    name="highlight"
    c.optional="true"
    c.type="Boolean"
  />
  <xsl:template match="/onion:object">
    <a href="{c.literalUri()}">
      <xsl:if test="$highlight">
        <xsl:attribute name="class">active</xsl:attribute>
      </xsl:if>
      <xsl:value-of select="@onion:name" />
    </a>
  </xsl:template>
</xsl:stylesheet>
```

4.5.2.2 Creating the »navigation« method



Under **node**, create the literal method **navigation**. This requires the *data view meta*. Assume the following transformation:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:param name="pathItems" c.implicitValue="" />
  <xsl:output
    method="xml"
    omit-xml-declaration="yes"
    indent="no"
  />
  <xsl:template match="/onion:object">
    <c.literalCall
      id="{@onion:parent}"
      method="navigation"
      pathItems="{concat(c.id(), ' ', $pathItems)}"
    />
  </xsl:template>
</xsl:stylesheet>
```

This method does nothing more than the **path** method for our path navigation: It collects the **pages** of the click path from the current document to the **site**. However, unlike with the path navigation, we do not want to output this directly but process it further afterwards. Therefore the **path** elements are not output directly as list elements, but collected in the parameter *pathItems*. Each **page** of the click path extends this list to include its own ID at the beginning. If the method is then called on the **site**, the parameter contains, read from left to right, the *Ids* of the pages on which the user has clicked. These are simply separated with a blank. On the **site**, *pathItems* then contains roughly the following example value:

```
onion://data/objects/123 onion://data/objects/456 onion://data/objects/789
```



Now create a literal method for the data type **site**. Call this **navigation** and do not indicate a *data view*.

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:param name="pathItems" c.implicitValue="" />
  <xsl:output
    method="xml"
    omit-xml-declaration="yes"
    indent="no"
  />
  <xsl:template match="/">
    <c.literalCall method="navigation.render" pathItems="{ $pathItems}" />
  </xsl:template>
</xsl:stylesheet>
```

4.5.2.3 Rendering the main navigation

Now we will go about rendering the navigation. After we have gone through the click path from the bottom to the top via the method *navigation*, we now begin building the navigation on the welcome page via the method **navigation.render** .



Now create the literal method **navigation.render** for the data type **node**. Use **children** as the **data view**, since the child elements of the current menu item are to be displayed for navigation.

```

<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:param name="pathItems" />
  <xsl:output
    method="xml"
    omit-xml-declaration="yes"
    indent="no"
  />
  <xsl:template match="/onion:children">
    <xsl:variable name="currentPathItem" select="substring-before($pathItems,
' ')" />
    <xsl:variable name="followingPathItems" select="substring-after($pathItems,
' ')" />
    <xsl:if test="onion:object">
      <ul>
        <xsl:for-each select="onion:object">
          <li>
            <xsl:choose>
              <xsl:when test="@onion:href = $currentPathItem">
                <c.literalCall
                  id="{@onion:href}"
                  method="link"
                  highlight="true"
                />
                <c.literalCall
                  id="{@onion:href}"
                  method="navigation.render"
                  pathItems="{ $followingPathItems}"
                />
              </xsl:when>
              <xsl:otherwise>
                <c.literalCall id="{@onion:href}" method="link" />
              </xsl:otherwise>
            </xsl:choose>
          </li>
        </xsl:for-each>
      </ul>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>

```

Upon entering the method, the parameter *pathItems* contains the documents of the click path. The first ID in it represents exactly the active page of the first navigation level. Based on the above illustration, that would be the ID of the Products page.

This ID is saved in the variable *currentPathItem*. The rest is saved in the variable *followingPathItems* for later processing. We then go through all children of the current node. In the case of the *welcome page*, *Services*, *Products*, *Contact* and *Site Notice*. For each of these documents we check whether the element in the variable *currentPathItem* is concerned here, i.e. the active page in the first level.

If this is not the case, we simply call the auxiliary method **link**, in order to create the navigation point. Otherwise we create the navigation point of the respective page also, but then call the method **navigation.render** again recursively with the remaining path components (i.e. the value of the variable *followingPathItems*).

In this way, the next navigation level for the active path is created (if available).

The *if*-query, which surrounds the *ul*-block, serves for intercepting an empty *ul* element. This should only be created if there are actually navigation points.

Now all we have to do is commence the rendering of the navigation in the **default** method initially.



To do this, insert the call of the method **navigation** into the **default** method, so that the navigation is also rendered. Replace the content of the *div* element for this purpose with the ID **navigation** by way of the following call: **<c.literalCall navigation" />**

4.6 Outputting content

Next, we will deal with the output of the content, which consists of a rich text field in our example. A simple way of outputting would be to assume the content of the rich text field via *xslt* element *copy-of*. The problem here however is that no integrated pictures or links would then work.



Therefore, fill your documents with a few contents that go beyond normal text: Add pictures via the editor button "New picture" (in a web-compatible format such as JPG, PNG or GIF) and and interlink documents.



Linking documents is easiest if you highlight a text in the editor and then drag the target **page** from the tree view and drop it into the editor window. The highlighted text then gets the link automatically. (It is important not to drop the **page** directly on the highlighted text, but somewhere else in the editor window).

4.6.1 Creating the »content« method



Create the literal method **content** for the data type **node**. Select **content** as the *data view*. Add the namespace for the prefix *xhtml* (see code example) at the time of creation.

```
<xsl:stylesheet
  xmlns:xhtml="http://www.w3.org/1999/xhtml"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:onion="http://onionworks.net/2004/data"
  version="1.0"
>
  <xsl:output
    method="xml"
    omit-xml-declaration="yes"
    indent="yes"
  />
  <xsl:template match="/node">
    <xsl:apply-templates select="text/node()" />
  </xsl:template>
</xsl:stylesheet>
```

We will assume the above transformation as content to start with. This method would not output anything yet. The template thus ensures that, within our rich text field with the name *text*, all nodes (in this case HTML elements) are dealt with.

The HTML elements now need to be output.



To do this, extend the transformation to include further `<xsl:template>` elements before the closing `<xsl:stylesheet>` element.

First of all, we will deal with internal links to pictures and other documents:

The first template applies for all *img* elements found in the *text* element. All attributes (characterized by the @) that have been maintained in the rich text editor are assumed for the *img* tag of the output. In this way, editors can maintain meta information such as the alternative text on the picture. If an attribute is not to be assumed (for example a class), then it is simply omitted. In order to create a functioning link to a binary element, we generate, in the *src* attribute, a link to the binary method **default** which we already used for the logo output earlier on.



All XHTML elements generated by the rich text editor are allocated to the namespace **xhtml** (" <http://www.w3.org/1999/xhtml> "). Therefore, the namespace must also be matched for the templates which are to handle these elements. This is done by placing the namespace prefix *xhtml:img* in front.

The template underneath takes effect in the case of any links(a elements) with a link target (*href* attribute) beginning with *onion:.* This means we can be sure to only react to internal links in this template.

For the link element we generate, by means of *c.literalUri()*, a URL that can be called in the browser. Then all attributes of the *link* element of the editor are assumed, with the exception of the *href* and the *objectReference* attribute. In addition, all further nodes within the *link* element are added. These can be for example further texts or pictures.

You will find more on the core functions provided by onion.net in the extension reference.

All other elements can now be copied without special treatment for the time being.

```
<xsl:template match="node()">
  <xsl:element name="{local-name()}">
    <xsl:apply-templates select="@* | node()" />
  </xsl:element>
</xsl:template>
<xsl:template match="@*">
  <xsl:copy/>
</xsl:template>
<xsl:template match="text()">
  <xsl:copy>
    <xsl:apply-templates select="@* | node()" />
  </xsl:copy>
</xsl:template>
```

The first of these three templates creates, for each rich text element *node()*, an xhtml element of the same name (*local-name()*) and then assumes all attributes as well as all nodes underneath.

The second template ensures that attributes are assumed.

If a text node is concerned, the third of the above templates takes effect. It copies the content of the *text* element and reinserts all attributes and nodes under these.



If you have inserted all these templates into the **content** method, save and return it. Now insert the call of the **content** method into the **default** method. For this purpose, delete the following line from the **default** method:

```
<p>In diesem Bereich wird der Content der Seite ausgegeben</p>
```



Insert the following in the place where you have just deleted the above line: **<c.literalCall method="content" />**. Now return the method and open a preview of your website documents again. Now the content maintained in the rich text is output in addition to the other elements.

5 Closing remarks

Using this guide, you have now created a small and simple website which outputs maintainable contents and a functioning navigation and contains a path navigation for orientation within the website. Moreover, you have become acquainted with the fundamental approach for working in onion.net and how some things are connected.

With this knowledge you can now gradually extend the small and simple website and improve the rendering. This can be done in many places. Rendering the navigation for example is relatively time-consuming.

Moreover, the literal methods you have created over the course of this tutorial offer the possibility of setting a caching, which contributes to increasing the performance when rendering.

In addition, you can of course do a lot of playing around with the design. A number of extensions may still have to be made in the rendering for this. With tables for example it is advisable to highlight every other line in order to improve legibility.

With this tutorial, you have laid the foundation for dealing with the different tasks and possibilities of a website and for expanding on the result achieved with this tutorial so as to meet your needs.

The onion.net team hopes you have fun learning!

P.S.: The further tutorials may now be of interest to you.