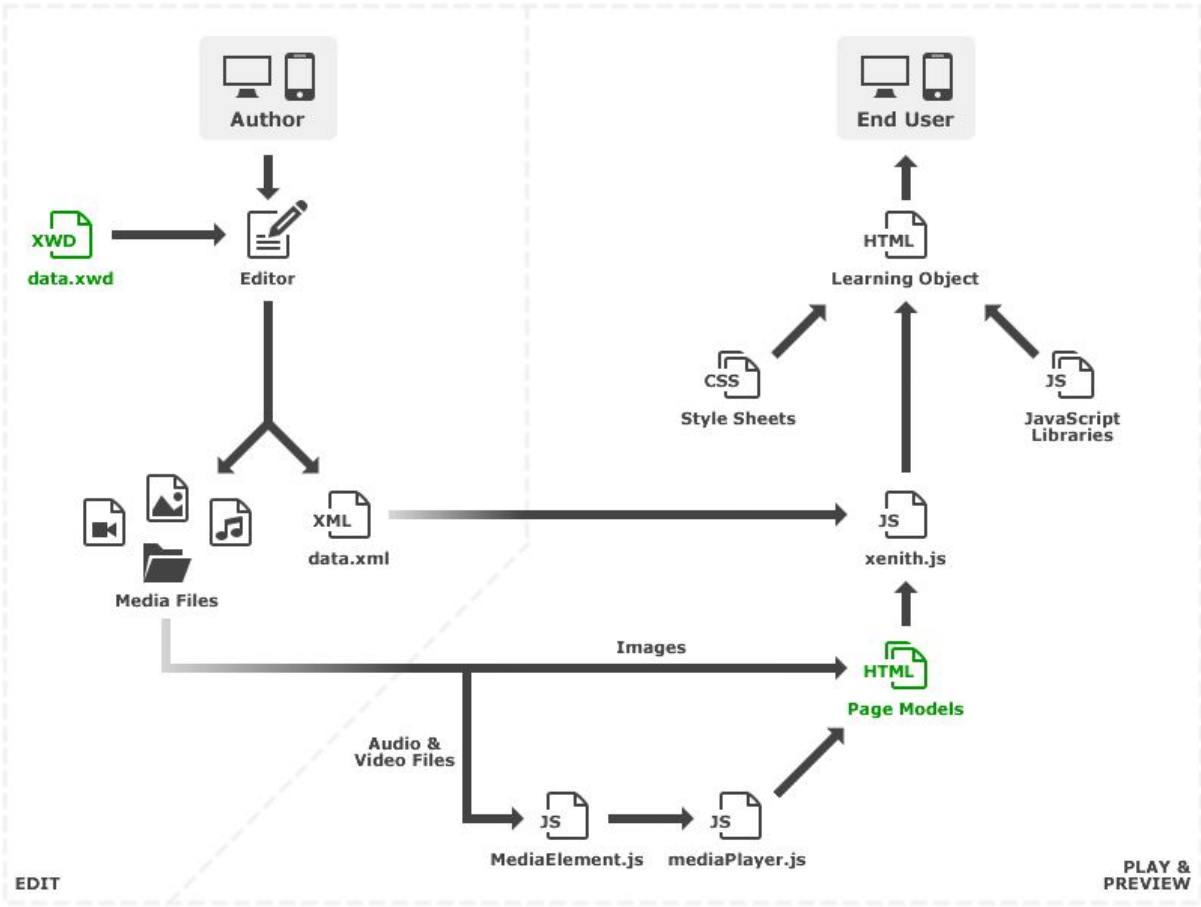
## How to Create a New Page Type

There are two files that you need to create to allow a new page type to be added to Toolkits projects. Let's look at the structure of the main files involved in editing and playing a Toolkits project:




The highlighted files show the two aspects of every page:

1. **XWD data** specifying how the page is created in the editor's wizard and saved to the XML file.
2. **HTML page model** specifying how the page appears in the interface to the end user.

So to create a new page you need to write the code for the page itself that the end user will see, and also create the logic behind it that allows the author to set the page up.

Let's begin with looking at the XWD data behind the editing process.

# XWD Data

The editor reads an XWD file to find out what editable fields should be presented to a project's author in a 'wizard'. Data entered in the wizard is saved to an XML file, to be used by the finished project.

The XWD file therefore contains an entry for each possible type of node you can have in the XML file. It contains details of all the possible project properties, every page, and every property and optional property that authors can edit or add to that page. It also defines style information about the type of control used to edit each piece of information in the wizard.

## The XWD Files

There are several XWD files within an installation.

Each module (e.g. Xerte Online Toolkit, Bootstrap, Decision Tree) has its own XWD file. There can also be multiple XWD files within each module if your installation gives the options of running in several languages.

This documentation focuses on XWD files for a Xerte Online Toolkit project, running in English, although the same principles apply to XWDs from other modules or languages.

The XWD file used by the editor is modules/xerte/parent_templates_Nottingham/wizards/en-GB/data.xwd.

When the editor runs it looks in this 'wizards' folder for the language it needs and uses the data.xwd file within that folder to set up the wizard. If the correct language folder is not found it will fall back to use the file modules/xerte/parent_templates/Nottingham/data.xwd which is in English.

## Adding a New Page to the XWD

Do not edit the main XWD file directly. The data.xwd file is made from merging many smaller XWD files together.

Follow these steps to add a new page to data.xwd:

1. **Create a new XWD file**
   There is an individual XWD file for each page type, as well as one containing the global properties for a project (project title etc.). These source files can be found at src/Nottingham/wizards/en-GB/ and they are named to match the associated page model.

   Create an XWD file in this source folder and give it a unique name. Its name will match the name you will later use for the associated page model (e.g. mcq.xwd & mcq.html).

   Often the quickest way of creating an XWD file is by duplicating an existing page's XWD that is structured similarly to the page you want to create. For example, for simple pages you could duplicate text.xwd and for pages that need nested pages duplicate accNav.xwd

1. **Write the XWD logic**
   Details of what the file should contain can be found in the 'Structure of an XWD File' section below.

2. **Merge the individual XWD files**
   You will need to rebuild the main data.xwd to include the XWD data for your new page. To do this you should run either build/rebuildNottingham.bat (Windows) or build/rebuildNottingham.sh (Linux).

   Full details on how to rebuild data.xwd and debug any errors can be found at src/Nottingham/buildXWD.txt.

3. **Test the wizard**
   Create a new project in Toolkits. You will see your new page on the insert menu of the editor. Add a page and check that all the fields appear as expected and that all the optional properties are available.

   You can also look at the XML file in USER_FILES to see if it contains the expected information that the project will use to create the page.

## Structure of an XWD File

XWD files are formatted as XML. This section describes the structure of an individual page's XWD file and uses the Graphics & Sound page (textGraphics.xwd) as an example.

All of the XWD data is contained within a **wizard** element with a **menus** attribute. This attribute says which category from the insert menu the page belongs to.

```
<wizard menus="Media">
</wizard>
```

The existing categories are Text, Media, Navigators, Connectors, Charts, Interactivity, Games and Misc but you can create a new one by using a different name here.

The data within wizard is then split into two sections:

1. **Wizard Data**
   This section contains details of how the page will appear in the editor. It describes how it will appear in the insert menu, as well as what fields will appear in a wizard when the page is added to a project.

2. **Initial XML Data**
   This section contains the data that will be written to a project's XML file as soon as the page is inserted.

### Wizard Data

Let's look at the wizard data for the Graphics & Sound page:

```
<wizard menus="Media">

  <textGraphics menu="Media" menuItem="Graphics and Sound"
icon="icPageWhitePicture" hint=" A page for presenting text and graphics. You can
also add an optional sound to this page." thumb="thumbs/textGraphics.jpg"
remove="true" duplicate="true">

  </textGraphics>

</wizard>
```

You can see that this section is contained within an element named after the XWD file name. In this case the XWD file is textGraphics.xwd so the element is called 'textGraphics'.

This page element has a number of required attributes relating to how the page will appear on the insert menu and page tree:

- **menu:** the category the page belongs to;

- **menuItem:** the full name of the page to appear in the insert menu's page list;
- **icon:** the small image that will appear next to the page's name (PNGs in modules/Xerte/icons/);
- **hint:** a description that appears when the page name is selected;
- **thumb:** a preview image that will appear with the hint (modules/Xerte/parent_templates/Nottingham/);
- **removed:** specifies whether pages of this type can be deleted;
- **duplicated:** specifies whether pages of this type can be duplicated.

None of these attributes are editable by the project author. They simply describe how the page appears in the editor.

The editable fields for a page are described by child nodes. Each child node represents an individual and editable field in the page wizard.

Here's the XWD for the Graphics & Sound page, now containing some of these child nodes:

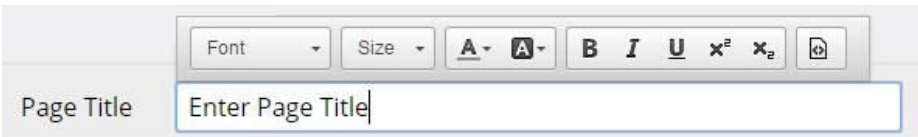
```
<wizard menus="Media">

  <textGraphics ... >

    <name label="Page Title" type="TextInput" wysiwyg="true" />
    <text label="Text" type="TextArea" height="250" />
    <url label="Image" type="media" />
    <tip label="Image Description" type="TextInput" />
    <align label="Align Text" type="ComboBox" options="Left,Right,Top,Bottom"
data="left,right,top,bottom" defaultValue="Left" />

  </textGraphics>

</wizard>
```

In order to create a Graphics & Sound page the project will need a number of basic pieces of information from the XML (remember that the XML is written by the editor in the format dictated here by the XWD file). It needs a page title, some text, an image, a description of the image for accessibility and information about how the page will be laid out. Each of these pieces of information is represented in the XWD above by a uniquely named element containing attributes that specify details of how the field should appear in the editor.

Let's look at some of these elements individually and how they translate into fields in the editor:

### Text Input

```
<name label="Page Title" type="TextInput" wysiwyg="true" />
```




You can see that the name node creates a one-line text input field. The WYSIWYG attribute means that the text format toolbar appears when you click in the text field.

### Text Area

```
<text label="Text" type="TextArea" height="250" />
```


A multi-line text input (TextArea) with a height of 250 pixels.

**Media**

```
<url label="Image" type="media" />
```


A field where authors can upload a file of their choice. The file selected will be written to the XML in the format "FileLocation + '[file_name]'"

**Combo Box**

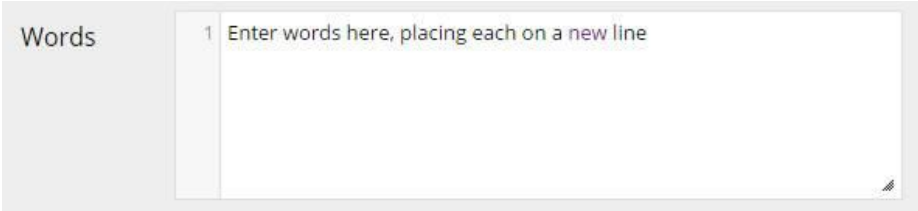
```
<align label="Align Text" type="ComboBox" options="Left,Right,Top,Bottom"
data="left,right,top,bottom" defaultValue="Left" />
```


A combo box where you can select from the list of words given in the options attribute. Although the combo box in the editor uses the options attribute, the data that is written to the XML file when an option is selected is taken from the data attribute list. This means that you can change the wording of text that appears in the editor (e.g. for translating to other languages) while not having to then change the project XML and page model files.

Other editable field types that can be used include:

**Script**

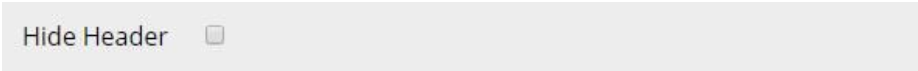
```
<word label="Words" type="script" height="100" />
```




A multi-line script field with a height of 100 pixels has been used. Otherwise similar to the TextArea type, the script fields do not allow the text entered in them to be formatted.

**Check Box**

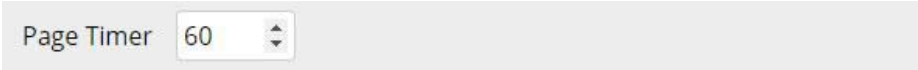
```
<hideFooter label="Hide Footer" type="CheckBox" defaultValue="false" />
```




A simple check box that will write "true" (checked) or "false" to the XML.

**Numeric Stepper**

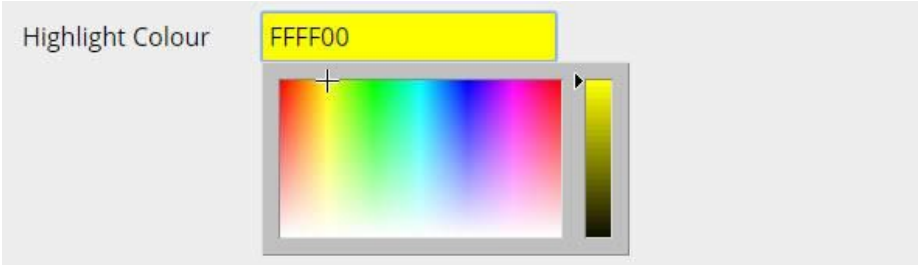
```
<timer label="Page Timer" type="NumericStepper" min="0" max="1200" step="1"
defaultValue="60" />
```




The numeric stepper field allows you to either manually enter a number (between the min and max values) or use the arrows to increase the current value by the step attribute value.

**Colour Picker**

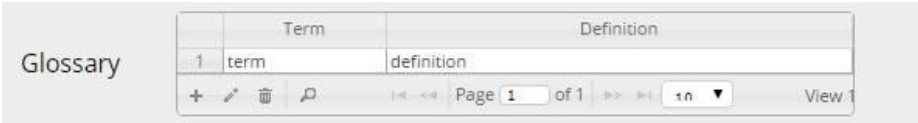
```
<colour label="Highlight Colour" type="ColourPicker" defaultValue="0xFFFF00" />
```




The colour picker allows the author to select a colour from a popup or enter a hex colour value.

**Data Grid**

```
<glossary label="Glossary" type="DataGrid" height="200" width="390" columns="2"
colWidths="100,295" editable="1,1" controls="1" headers="Term,Definition"
newRow="term,definition" rendererOptions="none,none"
defaultValue="term|definition"/>
```




The data grid allows the author to enter tabular data. You can use attributes in the XWD to specify the number and size of columns and set up default values for the initial cells. The data written to the XML file is a string where cell values are separated by "|" and rows are separated by "||" (e.g. "dog|woof||cat|meow||horse|neigh")

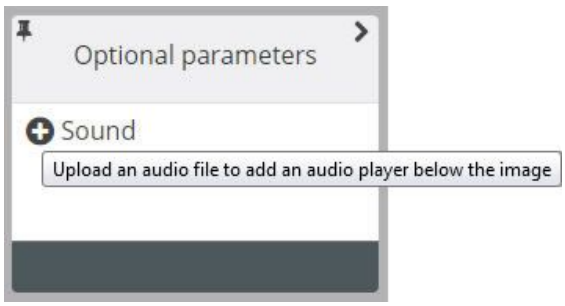
**Optional Properties**

As well as required properties that apply to all pages of that type, sometimes you might want to give the author the option of whether to add a property to a page. To do this you include child nodes in the same format as required fields, but add the attribute 'optional' to them. You also need to give them a default value as this won't be automatically set in the initial XML data that we will look at later.

For example, the Graphics & Sound page has an optional property that allows you to upload an audio file:

```
<wizard menus="Media">
  <textGraphics ... >


    <sound label="Sound" type="media" optional="true" defaultValue="Select a
File" tooltip="Upload an audio file to add an audio player below the image" />

  </textGraphics>
</wizard>
```

You can see that the sound node is set to be an optional property and its default value prompts the author to select a file to use. It does not appear on the wizard when the page is first added to a project but instead appears on the optional property panels to the right. The tooltip is a description of the property's purpose which appears when hovered over:



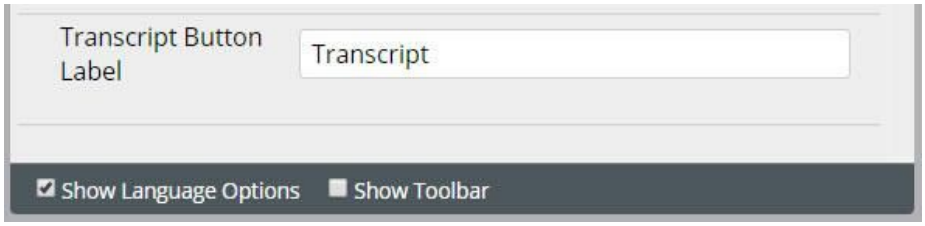

**Language Properties**

To give authors as much control as possible over their project you should not include any fixed strings in pages. You can make strings editable by including them as language options in a page's XWD. These are child nodes in the same format as the required fields, but with the extra attribute 'language'.

For example, if a transcript is added to a Graphics & Sound page there will be a button below the image that you press to view the transcript. Adding the following line to the XWD allows the button's label to be editable:

```
<wizard menus="Media">
  <textGraphics ... >


    <transcriptbuttonlabel label="Transcript Button Label" type="TextInput"
wysiwyg="true" language="true" />

  </textGraphics>
</wizard>
```

As language options don't need to be edited in order for a page to work (as long as a default value for them is set in the initial XML data) they don't appear by default in the editor. The author has to check the 'Language Options' check box at the bottom of the editor in order to view and edit them:




**Initial XML Data**

As well as information about how a page appears in the editor, the XWD must also contain details of what should be written to a project's XML file as soon as the page is inserted.

Let's look at this section of the Graphics & Sound page:

```
<wizard menus="Media">

  <pageWizard>
    <newNodes>
      <textGraphics><![CDATA[<textGraphics name="Enter Page Title" text="Enter
text for the page here" url="Select an Image" tip="Enter a Description for
Accessibility" align="Left"
transcriptbuttonlabel="Transcript"/>]]></textGraphics>
    </newNodes>
  </pageWizard>

  <textGraphics ... />

</wizard>
```

Appearing above the section describing the individual editor fields, this section of the XWD is contained within a **pageWizard** element with a child called **newNodes**. The newNodes element has one child, with a name to match the name you have given your XWD file. In this case the XWD file is textGraphics.xwd so the element in newNodes is called textGraphics. This element contains some **CDATA**.

When a project has a Graphics & Sound page added to it, the data in CDATA is written to the XML file. There is an element called textGraphics with an attribute for every required property and language property of the page. The attributes are named to match the nodes in the XWD that describe how each of these attributes is edited.

Each attribute has a default value that describes the state of the field in the editor before the author makes any changes. These attributes will be amended and added to as and when the author edits the wizard fields and adds optional properties.

For example, a new project is created and a Graphics & Sound page is added to it. The XML file for the project will contain the following:

```
<learningObject name="Learning Object Title" language="en-GB"
navigation="Linear">

  <textGraphics name="Enter Page Title" text="Enter text for the page here"
url="Select an Image" tip="Enter a Description for Accessibility" align="Left"
transcriptbuttonlabel="Transcript"></textGraphics>

</learningObject>
```

There is a **learningObject** element with attributes relating to the project's global properties. It contains a child node for every page in the project. This is where the textGraphics.xwd initial XML data appears.

After editing the page and adding some optional properties to it, the project XML will change accordingly:

```
<learningObject name="Learning Object Title" language="en-GB"
navigation="Linear">

  <textGraphics name="My First Page" text="Here is the text for my page..." url="
FileLocation + 'media/Koala.jpg'" tip=" A photograph of a koala " align="Right"
transcriptbuttonlabel="Transcript" sound="FileLocation + 'media/227 Dock Boggs -
Country Blues.mp3'"></textGraphics>

</learningObject>
```

You can see that the values of the textGraphics' attributes have changed and there is an extra attribute (sound) which represents an optional property that the author has added.

## Nested Pages

Nested pages are used where you want to give the author the option of adding a number of sub-pages to a page. These sub-pages could, for example, be used to add questions and answer options to a quiz (see Quiz: quiz.xwd) or to add nested pages to a navigator (see Accordion Navigator: accNav.xwd). The author can be given the option of different types of nested content to add (see Multiple Perspectives: perspectives.xwd) and can add several levels of nested content (see Tabbed Navigator+: tabNavExtra.xwd).

This example show how nested pages (used as answer options) are added to the MCQ page:

```
<wizard menus="Interactivity">

  <pageWizard ... />

  <mcq ... >

    <!-- this goes after the required, optional and language properties -->
    <newNodes>
      <option><![CDATA[<option text="Here is an option" feedback="Feedback for
this option" correct="false"/>]]></option>
    </newNodes>

  </mcq>
```

```
  <option menuItem="Option" icon="icBullet">

    <text label="Option" type="TextArea" height="100"/>
    <feedback label="Feedback" type="TextArea" height="100"/>
    <correct label="Correct" options="True,False" type="ComboBox"
data="true,false" />

  </option>

</wizard>
```

You can see that nested page XWD data is structured in the same way as the whole page's XWD data. There is a newNodes element in the section that describes the page in the editor. This contains CDATA with the default data to be written to the XML file for each nested page that is added. There is then a section describing the editor fields for the nested page.

In the same way that all page types must have a unique name, you need to make sure that any new nested pages you create are given a name that is not already used in other page's XWDs. The quickest way to do this is to search the main data.xwd file for the node name you want to use. If there are clashes with other names then the data will be overwritten when the page XWDs are merged.

# HTML Page Model

The HTML page model file is loaded into the project interface when the page is navigated to. It uses the XML data from USER-FILES to populate the page with content.

Initial steps to create a new HTML page model:

1. **Create the HTML file**
   Create a copy of modules/xerte/parent_templates/Nottingham/models_html5/examples/template.html in the models_html5 folder and rename it. The HTML file must have the same name as you have used for the associated XWD data (e.g. textGraphics.xwd & textGraphics.html).

2. **Edit the function names**
   Change the name of the first function in the file's script tag so that it matches the file name. Change the call to the init function so that it also references this.

   For example, in textGraphics.html the functions would be:

```
var textGraphics = new function() {...}

textGraphics.init();
```

You can now add content and functionality to the page using a combination of HTML elements, jQuery / JavaScript code and CSS.

## HTML Elements

Place fixed HTML elements for the page below the script and style tags. Additional HTML elements can be added in the code with jQuery.

In most page models the elements are contained within a div called pageContents. This div is not essential but makes the correct selection and creation of elements easier by ensuring you don't accidentally also select elements that are part of the main interface.

For example, the initial HTML in a page model could just be the pageContents div:

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

Additional elements could then be added dynamically with jQuery:

```
$("#pageContents").append('<button id="submitBtn">Submit</button>');

$("#pageContents").append('<button id="resetBtn">Reset</button>');
```

Creating a page containing these elements:

```
<div id="pageContents">
  <button id="submitBtn">Submit</button>
  <button id="resetBtn">Reset</button>
</div>
```

By selecting only elements within pageContents, you can avoid the accidental selection of the main interface buttons too (e.g. navigation buttons).

```
// individual buttons can be selected by unique id, e.g.
$("#submitBtn");


// all buttons on the page can be selected as children of #pageContents, e.g.
$("#pageContents button");
```

## jQuery / JavaScript Code

To protect the scope of the page code from that used to create the main interface you should write all of your jQuery / JavaScript within the function that matches the page name:

```
var textGraphics = new function() {
    // all of the code for the page should be in here
}
```

This means that functions created inside this function can only be called from outside by first referencing the page name. For example:

```
textGraphics.init();
```

And new variables created within the page cannot be inadvertently changed from outside it.

### Required Functions

Most of your code will probably go within the three functions required by every page model:

this.init(); this.pageChanged(); this.sizeChanged();

1. **init function:** Called when the page is first viewed and the page model has loaded into the interface. Write the code to initially set up the page here. Your code is likely to reference data from the project's XML file.

   For example, the Text page's init function loads the text for the current page into the pageContents element and calls the global function x_pageLoaded:

   ```
   this.init = function() {
       $("#pageContents").html(x_currentPageXML.childNodes[0].nodeValue);
       x_pageLoaded();
   }
   ```

2. **pageChanged function:** Called from xenith.js when the page is viewed again after it has first loaded, i.e. when a page has been seen, navigated away from, and then returned to. This is **not** called on the first view of the page.

By default, unlike the Flash version, all pages will appear in the exact state they were when previously viewed unless you add code to this function to reset or change things.

For example, on a Button Question page the button is hidden when clicked and the answer is revealed. When returning to a page of this type the controls should be reset so the button is visible and the answer is hidden again:

```
this.pageChanged = function() {
    $("#button").show();
    $("#answer").hide();
}
```

3. **sizeChanged function:** Called from xenith.js when the size of the project changes in the browser.

   This is immediately triggered in the code of the page currently being viewed if:

   - the screen size is changed using the interface button to toggle between 'full screen' and 'default';
   - the screen size is set to 'full screen' and the window size is changed;
   - the project is being viewed on a mobile device and the orientation is changed.

   It is also triggered (after this.pageChanged) if the page is returned to and the size has changed, for one of the reasons above, since it was last viewed.

   Ideally pages do not need any code in this function as the CSS should be set up so that it automatically adjusts itself for size changes. However, examples where code is needed include:

   - resizing panels where their size cannot be set just with CSS;
   - repositioning or resetting labels in drag and drop activities.

In addition to these three functions, all page models most include a call to x_pageLoaded.

### x_pageLoaded

The global function x_pageLoaded must be called from every page model when everything has finished loading. Pages are always hidden when initially loaded into the interface to prevent the end user seeing the page flicker (with images resizing etc.) as it is put together. This function makes the page fade in and be visible.

On many pages this can simply be called at the end of the init function. On other pages, for example those that contain images, it is best placed in an onload event so that the page only appears once everything is ready.

### Custom Functions

Create custom functions in the main page function following the same structure as init, pageChanged and sizeChanged. For example:

```
<script type="text/javascript">

    var textGraphics = new function() {
```

```
        this.pageChanged = function() {...};

        this.sizeChanged = function() {...};

        this.init = function() {
           // call the function within the page model as either:
           this.customFunction();
           // or:
           textGraphics.customFunction();
       };

        this.customFunction = function() {
           console.log("I'm your customFunction - you can call me whatever you
like");
       };

    }

    textGraphics.init();

</script>
```

### Global Functions

These functions from xenith.js are often used in page models:

1. **x_scaleImg(img, maxW, maxH)**
   Function scales an image, keeping proportions constrained, to a maximum of maxW x maxH.

   When an image is first scaled using this function, its original size is stored in the element's data (img.data("origSize")) so that if it is scaled again it can be done without exceeding its original dimensions.

   ```
   x_scaleImg($("#pageImg"), 500, 500);
   ```

2. **x_addLineBreaks(text)**
   Function swaps line breaks in XML data for <br> tags so they aren't lost when added to the page as text.

   ```
   $("#pageContents").html(x_addLineBreaks(x_currentPageXML.getAttribute("tex
   t")));
   ```

All global functions and variables in xenith.js are prefixed with 'x_' so that clashes with functions in code elsewhere (e.g. in page models) can easily be avoided.

### Getting XML Page Data

The variable **x_currentPageXML** contains the XML for the current page.

For example, the XML for the page could contain various attributes, as well as child nodes (nested pages) containing extra information:

```
<mcq linkID="PG1366742981022" name="MCQ" instruction="Answer this question"
prompt="What year was the Battle of Hastings?" type="Single Answer" align="Left"
panelWidth="Medium">
    <option text="1066" feedback="Yes, that is correct" correct="true" />
    <option text="1600" feedback="No, try again" correct="false" />
    <option text="1982" feedback="No, try again" correct="false" />
</mcq>
```

The jQuery used in a page model to access the data from the XML above would be:

```
x_currentPageXML.getAttribute("instruction"); // returns "Answer the question"

$(x_currentPageXML).children().each(function(i) {
    // loops through all child nodes of mcq (the options)
    this.getAttribute("text"); // returns "1066", "1600" etc.
});
```

### Getting Language Data

Strings used in page models should never be fixed. This allows authors to easily customise projects and for whole modules to be easily translated into other languages.

Most strings will therefore be accessible in the editor, saved in the project XML and retrieved like other attributes. However, where the same strings are used consistently across multiple page types they can found in the main language files in the root of XOT (e.g. languages/engine_en-GB.xml).

For example, this information can be found in engine_en-GB.xml:

```
<errorBrowser label="Your browser does not support this page type."/>
```

And it can be retrieved in a page model like this:

```
x_getLangInfo(x_languageData.find("errorBrowser")[0], "label", "Your browser does
not fully support this page type");

// x_getLangInfo(a, b, c); where a is the node, b is the attribute name and c is
the fallback text to use if the data isn't found in the language file.
```

### Saving Page Data

You cannot alter the state of a page or retrieve data about it when it is not visible. Therefore, if you want to refer to something from a previously viewed page you need to have saved that data outside of the page. You can use the global array **x_pageInfo** to store this data.

For example, to save data about the current page:

```
// x_pageInfo is an array containing information about all of the project's pages
// x_pageInfo[x_currentPage] is an object containing information about this page
x_pageInfo[x_currentPage].savedData = "the data I want to retrieve later";
```

And then retrieve it in the next page in the project:

```
x_pageInfo[x_currentPage - 1].savedData; // returns "the data I want to retrieve later"
```

See the modelAnswer and modelAnswerResults page models for an example of this being used.

If you want data from a page to be remembered when the same page is viewed again you can either save to x_pageInfo as above, or save data to elements within the page itself.

```
$("#pageContents").data("score", 10); // to store the data
$("#pageContents").data("score"); // to retrieve the data
```

### jQuery UI

XOT includes the jQuery UI library. This is useful for easily adding interactions, widgets and effects to pages.

You can find out more about them on the jQuery UI website or see examples of the library being used on a number of XOT pages (e.g. accordion widget in accNav.html and the sortable interaction in categories.html).

## CSS Styles

Add any page specific CSS styles to the page model's style tag. Be as specific as possible with CSS selectors to avoid selecting elements that make up the main interface.

For example:

```
<style type="text/css">

    <!-- style will only apply to buttons within pageContents -->

    #pageContents button {
        margin-left:    10px;
        float:          right;
    }

</style>
```

### Global Styles

There are a number of style sheets (modules/xerte/parent_templates/Nottingham/common_html5/css/...) that control the general appearance of the interface:

- **mainStyles.css** contains styles controlling the structure of the interface.
  It defines the layout of the header and footer bars and should not be changed as it is likely to stop parts of the interface working correctly.

- **themeStyles.css** contains the purely aesthetic parts of the interface CSS.
  It can be edited to rebrand all projects created from your XOT installation. For example, in this file you can change the colour and background images used for the footer and header bars.

- **desktopStyles.css** contains the CSS for when the interface is viewed on large screens and tablets.

- **mobileStyles.css** contains the CSS for when the interface is viewed on smaller screens.

Several classes in these style sheets can be useful in page models. These include the panel class (used to draw a panel and drop shadow around an element) and the splitScreen class (used on the navigators to create two columns of information).

## Hints and Tips

### Common Files

If your page includes images that aren't uploaded by the author, they should be saved to: modules/Xerte/parent_templates/Nottingham/common_html5/...

In this example, the tick image from the common_html5 folder is added to the page:

```
$("#pageContents").append('<img src="' + x_templateLocation +
'common_html5/tick.png' + '"/>');
```

x_templateLocation references the location of the template files within the XOT installation and should be used rather than a hardcoded file path.

### Embedding Media

Any files uploaded to a project via the editor are saved to the project's media folder (USER-FILES/[template_id]/media/…) with a reference to them in data.xml (FileLocation + '[file_name]').

Toolkits uses mediaElement.js to play audio and video files. In order for this to be done consistently across pages you should add media via Toolkits' mediaPlayer plugin. Amongst other things, this plugin means that with very little code in the page itself you can set up consistent controls, listen to media events and use YouTube and Vimeo URLs instead of uploaded files.

Use this code to embed media on a page:

```
// where pageMedia is the id of the element the media will load into
$("#pageMedia").mediaPlayer({

    // required properties:
    type    :"video", // or "audio"
    source  :x_currentPageXML.getAttribute("url"), // reference to media file

    // optional properties:
    width   :300, // number (pixels) or percentage e.g. "100%"
    height  :200, // video only
    startEndFrame :[10, 30], // to play a short section of the file [start,end]
    autoPlay: "true" // to automatically play when loaded

});
```

There are then two functions you can then add to page models to receive information from mediaPlayer.js:

```
this.mediaFunct = function(mediaElement, mediaSrc, eventType, data) {
    // where mediaElement is a reference to the media instance
```

```
    // you can then listen to its events in this way:
    mediaElement.addEventListener("timeupdate", function(e) { ... });

    // mediaSrc, eventType & data are useful if you are using Vimeo for videos
    // see the videoSynch page model for an example of these in use
}

// video only
this.mediaMetadata = function($video, dimensions) {
    // where:
    // $video is a reference to the mediaElement video instance
    // dimensions is an array containing dimensions of video [width, height]
}
```

### Testing

When a new page has been developed it's important to thoroughly test it in a number of browsers.

Does the page work as expected when:

- the page is first loaded?
- the page is returned to (particularly after you have viewed another page of the same type)?
- the project size is changed while viewing the page?
- the project size is changed while viewing a later page and then the page is returned to?
- viewed in different browsers and devices (specifically the CSS)?
- the mouse is not used? (Can you tab to all the controls and complete all interactions using just the keyboard?)
- all fields in the wizard for the page are completed? (What happens if no data has been entered in a field?)
- all optional properties for the page are added in the wizard?

### Upgrading Your Installation

When you upgrade your installation the data.xwd file may be overwritten with one that does not include any new pages you have created. Make sure you keep a copy your new pages' files and rebuild data.xwd to include them after upgrading.

### Additional Resources

The examples folder (modules/xerte/parent_templates/Nottingham/models_html5/examples/) contains a number of example page model files:

- **template.html** can be used as a starting point for new page models;
- **example_js.html** files give examples of code commonly used in page models, e.g. for accessing the xml data, scaling images and adding the media player;
- **example_css.html** files give examples of styles commonly used in page models.