Enonic XP Documentation

Release 5.3

Enonic AS

August 06, 2015

Contents

This is the home for Enonic XP 5.3 documentation. The documentation is available on github in a separate project.

CHAPTER 1

Installation

This section describes how to install Enonic XP. If you have any trouble, please look at our *Troubleshooting* section.

1.1 Requirements

Enonic XP is a self-contained Java Application with minimal requirements for infrastructure or other platform services.

General requirements are:

- Any OS supporting Java
- Java 1.8 (update 40 or above)
- Local filesystem
- At least 1Gb of available memory

For clustered deployments:

- Shared filesystem (for storing configuration and blobs)
- Open for network communication between the nodes

1.2 Download

First off, download the correct Enonic XP version and unzip it to a preferred directory.

Terminal users can do this:

```
$ curl -O http://repo.enonic.com/public/com/enonic/xp/distro/5.3.0/distro-5.3.0.zip
$ unzip distro-5.3.0.zip
$ cd enonic-xp-5.3.0
```

The top level of the unzipped directory will be referred to as `$XP_INSTALL` from now on.

1.3 Start the server

Now that the software has been downloaded, you're ready to start the server - start the respective file from command line.

Linux and OS X:

```
$ $XP_INSTALL/bin/server.sh
```

Windows:

```
$ $XP_INSTALL\bin\server.bat
```

This will start Enonic XP, which when successfully started will show something like the following at the end of the log:

```
12:53:14.302 INFO  c.e.x.l.framework.FrameworkService - Started Enonic XP in 7378 ms
```

1.4 Log in

Point your favorite (modern) browser to `http://localhost:8080`. Log in with username `su` and password `password`.

Congratulations on installing Enonic XP.

CHAPTER 2

Install Demo Module

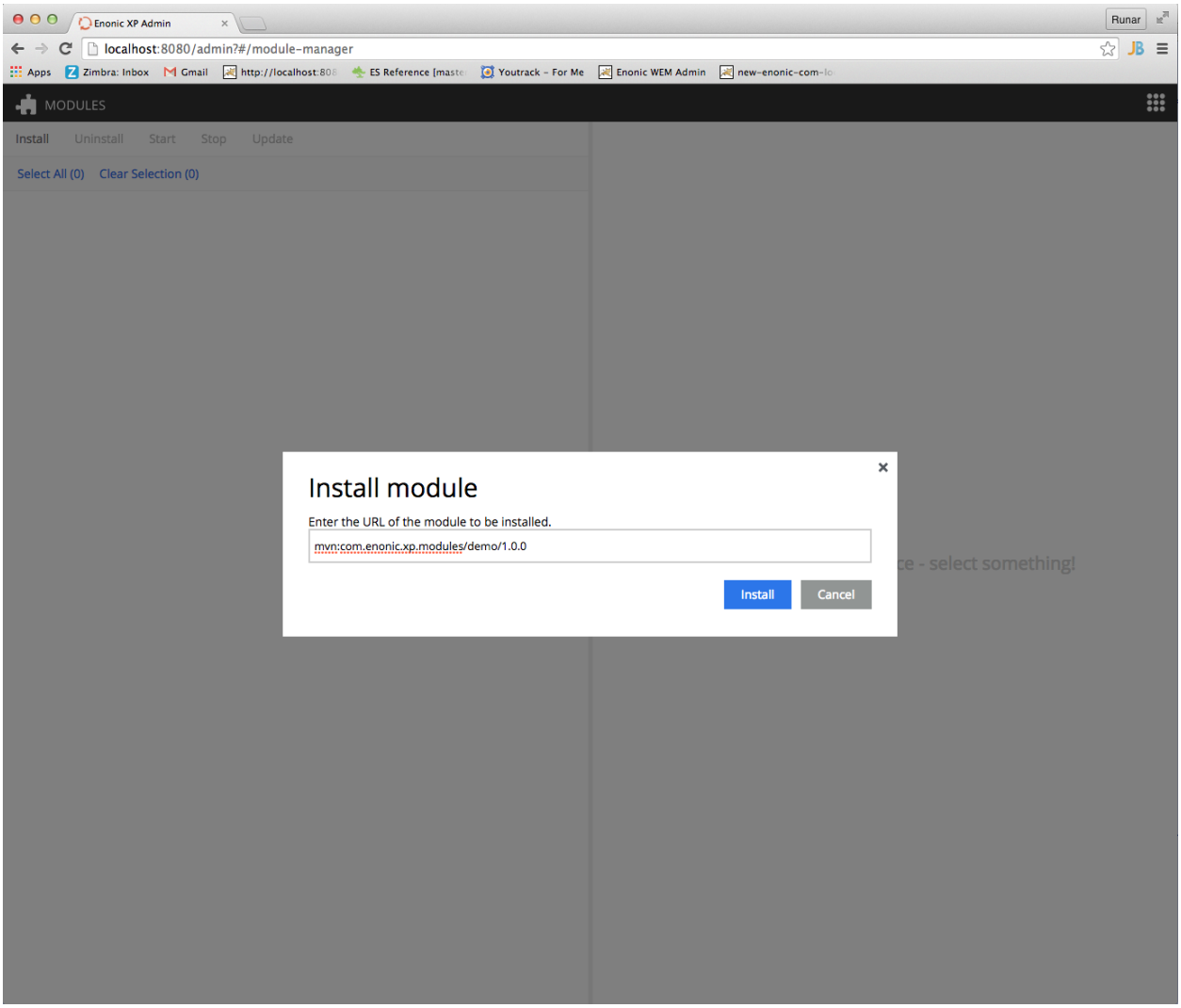
This section will guide you through the installation of the demo module.

2.1 Install via Admin Console

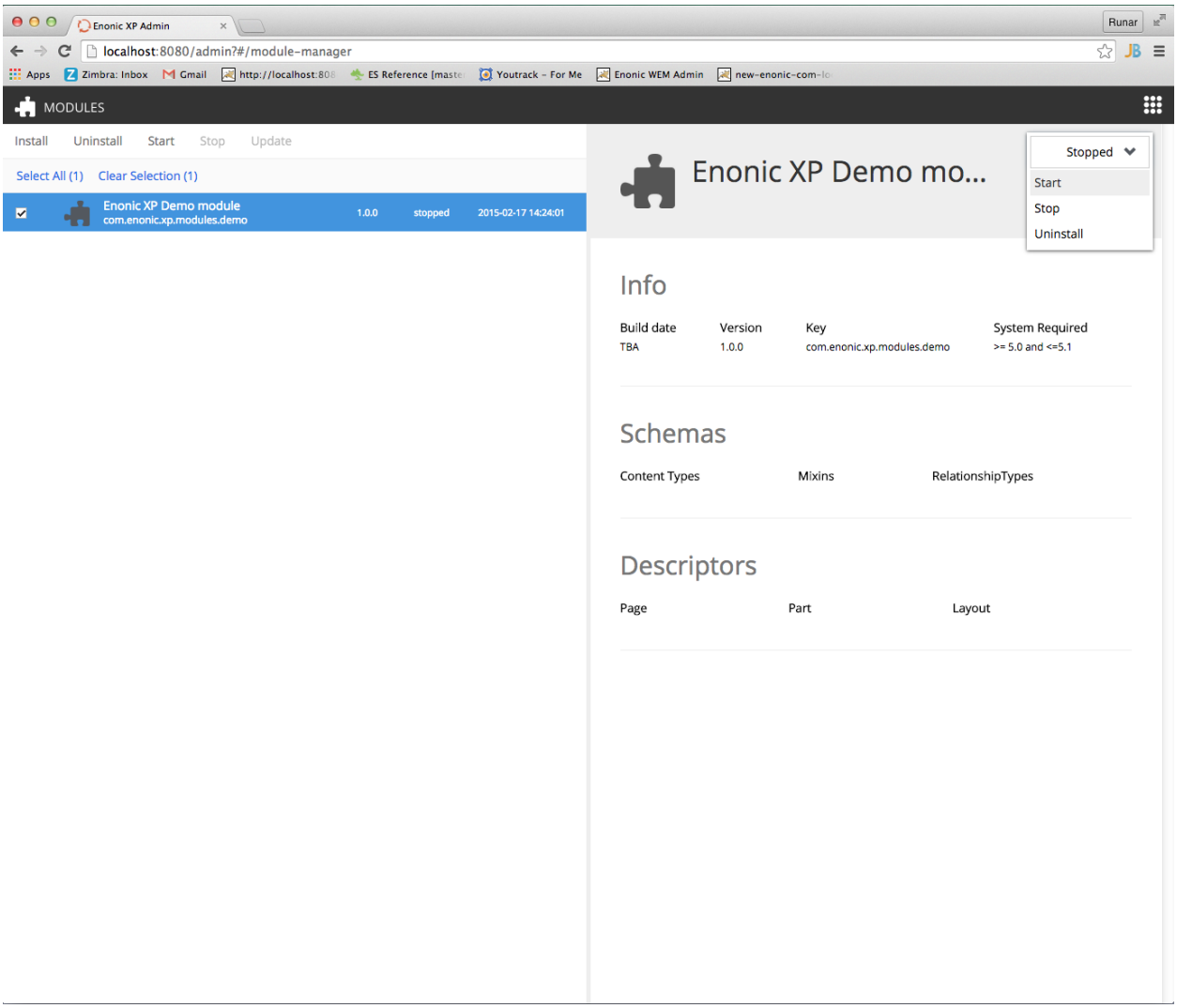
Sign into the admin console, and launch the 'Modules' App. You will need a user with the "Administrator" role (system.admin) to do this.

Click `Install` in the toolbar, and enter the URI below:

```
mvn:com.enonic.xp.modules/demo/1.1.0
```



Once the module is installed, it will appear in the list of modules. Select the module and start it from the menu.



2.2 Install from command line

If you prefer the devops way, modules can also be installed from the command line.

`$XP_INSTALL` is an alias for the directory where Enonic XP was installed. `$XP_HOME` is an alias for the Enonic XP home directory. The default location is `$XP_INSTALL/home`.

Download the module from Enonic Repository:

```
$ curl -O http://repo.enonic.com/public/com/enonic/xp/modules/demo/1.1.0/demo-1.1.0.jar
```

Then move the file into `$XP_HOME/deploy`.

If you have the modules application open, you will see that the module has been installed and started.

2.3 Accessing the demo content

Upon installation, the module will automatically create a complete site with content - a partial copy of the `Enonic.com` website.

Go to the `Content Manager` application to access and work with the site.

The source code for this module can be found on github.

CHAPTER 3

Your First Module

This guide will lead you through the basics of developing your first module. Creating a new module will require a text editor and gradle build-system.

3.1 Project structure

The project structure is similar to Maven. Create the folder structure you see below. All are folders except for `module.xml` and `build.gradle`:

```
my-first-module/
  build.gradle
  src/
    main/
      resources/
        module.xml
        cms/
          lib/
          pages/
          parts/
          layouts/
          view/
          assets/
          content-types/
          mixins/
          relationship-types/
          services/
```

Every file and folder has a specific function and meaning.

build.gradle Gradle script for building the module. This file describes the actual build process.

module.xml The `module.xml` file contains basic information to register the module in Enonic XP. Settings for the module can be defined in the `config` element and the values for these settings can be updated in the administration console. An example will be presented later in this document. Leave the `config` element blank for now.

```
<module>
  <config/>
</module>
```

cms/lib/ This is the last place the global `require` javascript-function looks, so it is a good place to put default libraries here.

cms/pages/ Page definitions should be placed here. They will be used to create page templates in the repository.

cms/parts/ Part definitions should be placed here. Parts are objects that can be placed on a page.

cms/layouts/ Layout definitions should be placed here. Layouts are definitions that restricts the placement of parts.

cms/views/ Views can generally be placed anywhere you want, just keep in mind what path to use when resolving them.

cms/assets/ Public folder for external css, javascript and static images.

cms/content-types/ Content schemas-types are placed here. Used to create structured content.

cms/mixins/ Mixin schema-types are placed here. A mixin can be used to add fields to a content-type.

cms/relationship-types/ Relationship-types are placed here. They are used to form relations between contents.

3.2 Building the module

The project is built using Gradle. Here is a simple build script to build `my-first-module`.

```
buildscript {
  repositories {
    jcenter()
    maven {
      url 'http://repo.enonic.com/public'
    }
  }

  dependencies {
    classpath 'com.enonic.xp.gradle:gradle-plugin:1.2.0'
  }
}

apply plugin: 'com.enonic.xp.gradle.module'
version = '1.0.0'

module {
  name = 'com.enonic.xp.modules.mymodule'
  displayName = 'My first module'
}
```

To build your module, write `gradle build` on the command line:

```
$ gradle build
```

This will produce a jar file that is located inside `build/libs` folder. That jar file is your build artifact you can install into Enonic XP.

3.3 Installing the module

After a module is built with Gradle, it must be deployed to your Enonic XP installation. There is not much of a module to build at this point, but you will want to build and check your project after each step of this tutorial. You can refer back to this section whenever you are ready.

To deploy your module, copy the produced artifact to your `$XP_HOME/deploy` folder:

```
$ cp build/libs/my-first-module-1.0.0.jar $XP_HOME/deploy/.
```

We have simplified this process by adding a `deploy` task to your build. Instead of manually copying in the deploy folder, you can execute `gradle deploy` instead:

```
$ gradle deploy
```

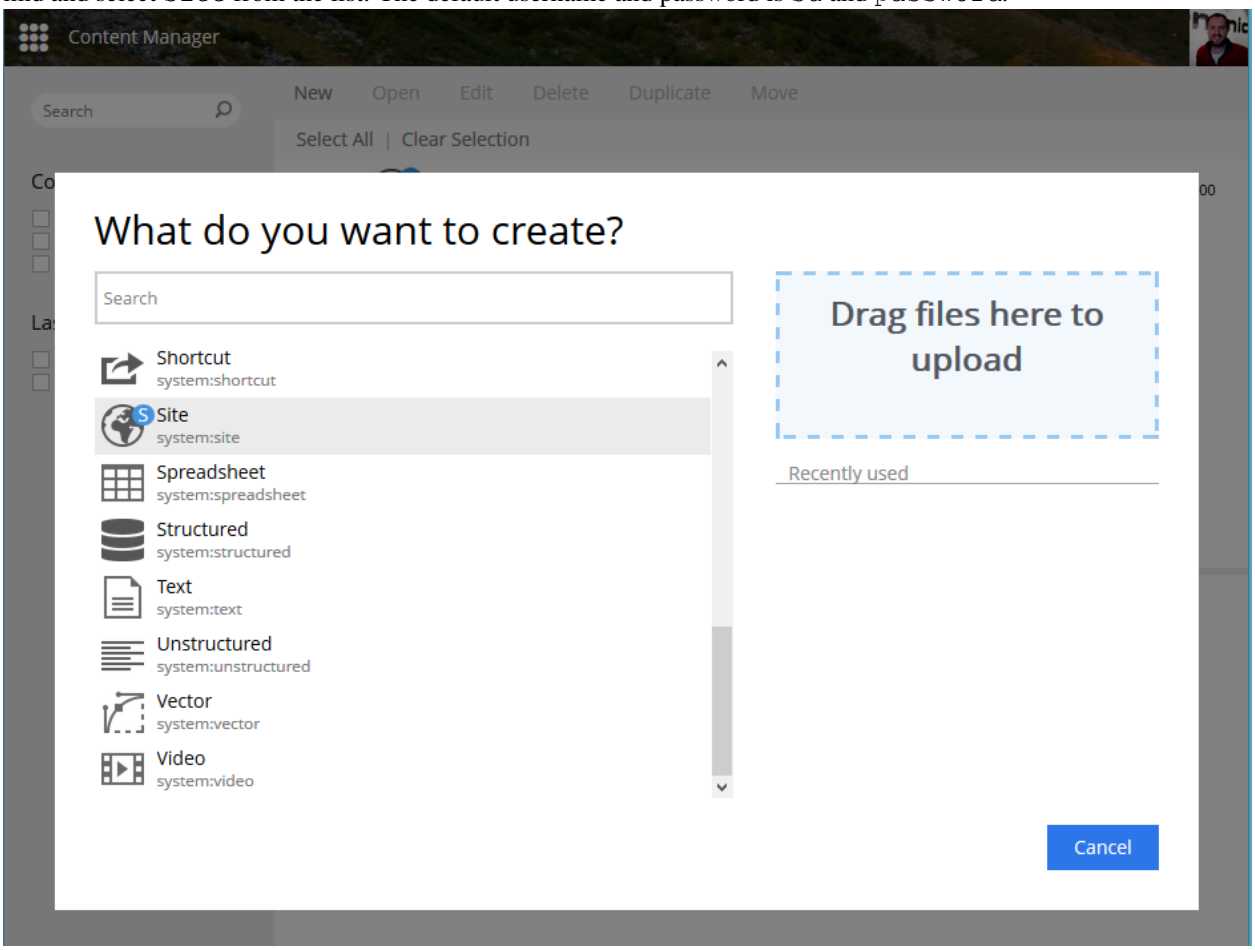
For this to work you have to set `XP_HOME` environment variable (in your shell) to your actual Enonic XP home directory.

To continuously build and deploy your module on changes, you can use the `gradle watch` task. This will watch for changes and deploy the changes to Enonic XP. The server will then pick up the changes and reload the module. This is probably the fastest way to develop your module:

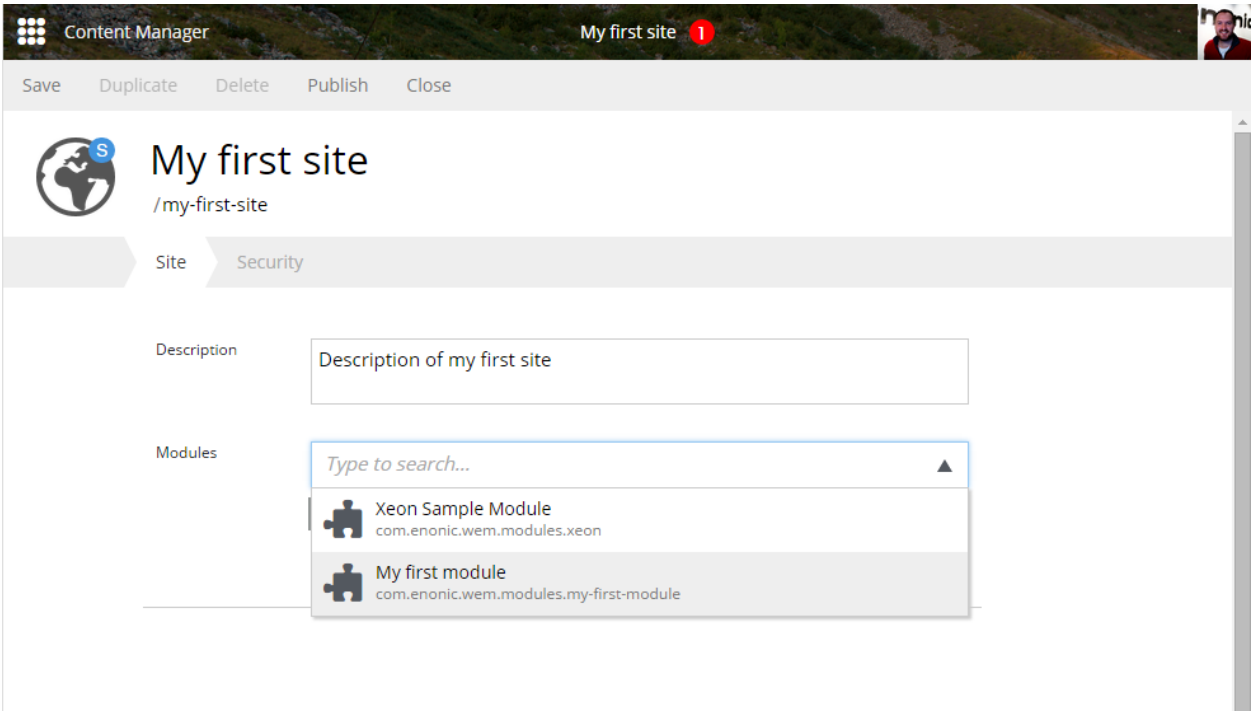
```
$ gradle watch
```

3.4 Setting up a site

First, log into the admin console and open the Content Manager app. Then click the `New` button and scroll down to find and select `site` from the list. The default username and password is `su` and `password`.



Fill in a site name and description and select the modules you want to add to the site. Finally, save the site.



3.5 Creating a simple page

To create a complete page in Enonic XP, we must grasp three different concepts: the page descriptor, the controller and the view. Start off by making a new folder in the page directory of the module and name it whatever you want. Inside this folder, we will place our controller and page descriptor.

The page descriptor is required to register the page and allows us to set up user input fields to configure the page. It also allows us to describe what regions are available in this page.

Create a page descriptor by making a file named `page.xml` in folder `page/my-first-page`.

```
<page-component>
  <display-name>My first page</display-name>
  <config/>
  <regions/>
</page-component>
```

3.6 Adding a Controller

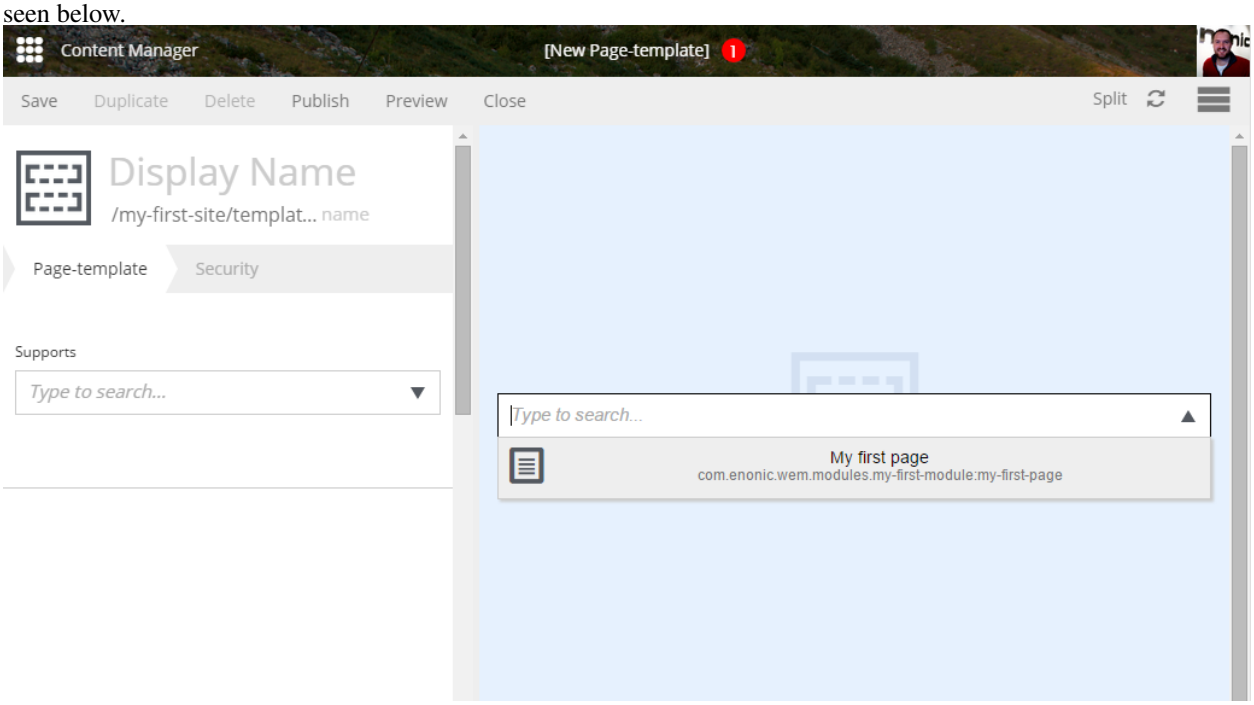
To be able to give a response to a request to this page, we will need to create a controller in the `page/my-first-page` folder. The controller is written in JavaScript and must be named `controller.js`. A controller exports a set of methods, one for each HTTP method that should be handled. The handle method has a request object as parameter and returns the result.

```
// Handles a GET request
exports.get = function(req) {}

// Handles a POST request
exports.post = function(req) {}
```

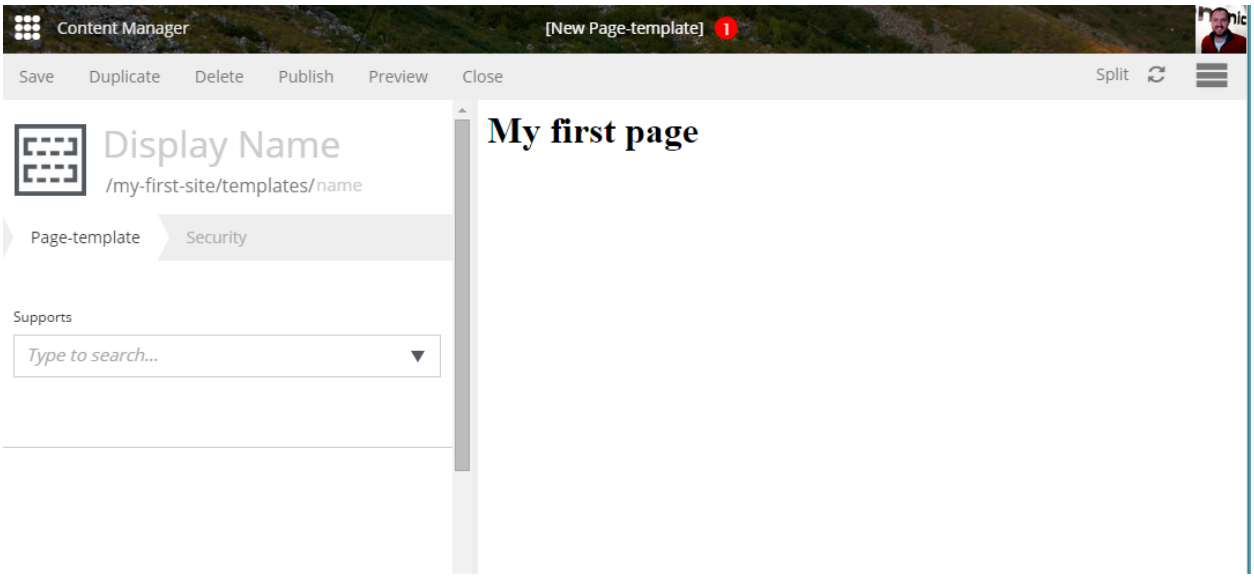
Create a simple `controller.js` file to render an HTML-page when it receives a get request.

```
exports.get = function(req) {

  return {
    body: '<html><head></head><body><h1>My first page</h1></body></html>',
    contentType: 'text/html'
  };

};
```

To view this page in Enonic XP, the project must first be built and then deployed. Then open the Content Manager app and find the site (that you created earlier). Click to expand the site and find the `Templates` folder. Click this folder and select `New` to create an new `Page template`. Select the `My-first-page` from the dropdown on the right as seen below.



The result should look something like this. Give it a name and save.



3.7 Rendering a view

If you feel like concatenating strings to create an entire webpage is a little too much hassle, Enonic XP also supports views. A view is rendered using a rendering engine; we currently support XSLT, Mustache and Thymeleaf rendering engines. This example will use Thymeleaf.

To make a view, create a file `my-first-page.html` in the `view` folder.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>My first page, with a view!</h1>
  </body>
</html>
```

In our `controller.js` file, we will need to parse the view to a string for output. Here is where the Thymeleaf engine comes in. Using the Thymeleaf rendering engine is easy; here is how we do it.

```
exports.get = function(req) {

  // Resolve the view
  var view = resolve('/cms/view/my-first-page.html');

  // Define the model
  var model = {
    name: "John Doe"
  };

  // Render a thymeleaf template
  var body = execute('thymeleaf.render', {
    view: view,
    model: model
  });

  // Return the result
```

```
  return {
    body: body,
    contentType: 'text/html'
  };

};
```

3.8 Adding dynamic content

We can send dynamic content to the view from the controller via the `model` parameter of the `render` function. We then need to use the rendering engine specific syntax to render it. The controller file above passed a variable called `name` and here is how to extract its value in the view using Thymeleaf syntax.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>My first page, with a view!</h1>
    <h2>Hello <span data-th-text="${name}">World</span></h2>
  </body>
</html>
```

More on how to use Thymeleaf can be found in the official Thymeleaf documentation.

3.9 Adding regions

To be able to add components like images, component parts, or text to our page via the live edit drag and drop interface, we need to create at least one region. To do this we will first modify the page descriptor (`page.xml`) to support one region called `main`.

```
<page-component>
  <display-name>My first page</display-name>
  <config />
  <regions>
    <region name="main"/>
  </regions>
</page-component>
```

Next, our controller must send the region data to the view.

```
// Get the current content. It holds the context of the current execution
// session, including information about regions in the page.
var content = execute('portal.getContent');

// Include info about the region of the current content in the parameters
// list for the rendering.
var mainRegion = content.page.regions["main"];

// Extend the model from previous example
var model = {
    name: "Michael",
    mainRegion: mainRegion
};
```

Finally, we need to modify the view to render all components inside the region. To make live-edit understand that an element is a region, we need an attribute called `data-portal-component-type` with the value `region`.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>My first page, with a view!</h1>
    <h2>Hello <span data-th-text="${name}">World</span></h2>
    <div data-portal-component-type="region">
      <div data-th-each="component : ${mainRegion.components}" data-th-remove="tag">
        <div data-portal-component="${component.path}" data-th-remove="tag" />
      </div>
    </div>
  </body>
</html>
```

We can now use the live edit drag and drop interface to drag components onto our page. Lets look into how we can create components.

3.10 Creating a part component

Creating a component is very similar to creating a page. We need a descriptor, a controller and a view. When they are done, we can add the part component to the page by using drag and drop in live edit.

Start by creating folders `part/mypart`, and add a `part.xml` descriptor. This is very similar to our `page.xml`, the only difference is that a part cannot contain any regions. We want this part to list our favorite things to do, so we will need to configure some input for the part. We will add a text input which can be repeated up to 5 times.

```
<part-component>
  <display-name>My favorite things</display-name>
  <config>
    <field-set name="things">
      <label>Things</label>
      <items>
        <input type="TextLine" name="thing">
          <label>Thing</label>
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="0" maximum="5"/>
        </input>
      </items>
    </field-set>
  </config>
</part-component>
```

Next, create a controller inside the same folder, `controller.js`.

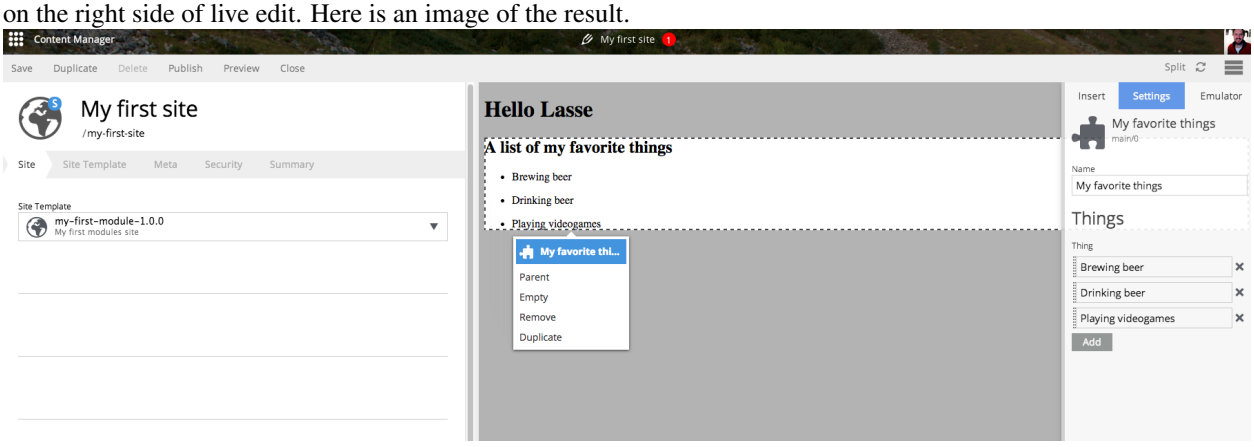
```
exports.get = function(portal) {

  // Find the current component from request
  var component = execute('portal.getComponent');

  // Find a config variable for the component
  var things = component.config["thing"] || [];
```

```
  // Define the model
  var model = {
    component: component,
    things: things
  };

  // Resolve the view
  var view = resolve('/cms/view/my-favorite-things.html');

  // Render a thymeleaf template
  var body = execute('thymeleaf.render', {
    view: view,
    model: model
  });

  // Return the result
  return {
    body: body,
    contentType: 'text/html'
  };

};
```

The part needs a root element with the attribute `data-portal-component-type`. In this case, it will be a `part`, but we can also resolve it dynamically as explained in the example. The `things` parameter is basically just JSON data, and we can loop it easily in Thymeleaf and print its value.

```
<section data-th-attr="data-portal-component-type=${component.type}">
  <h2>A list of my favorite things</h2>
  <ul class="item" data-th-each="thing : ${things}">
    <li data-th-text="${thing}">A thing will appear here.</li>
  </ul>
</section>
```

The part can now be added to the page via drag and drop. You will be able to configure the part in the *context window* on the right side of live edit. Here is an image of the result.

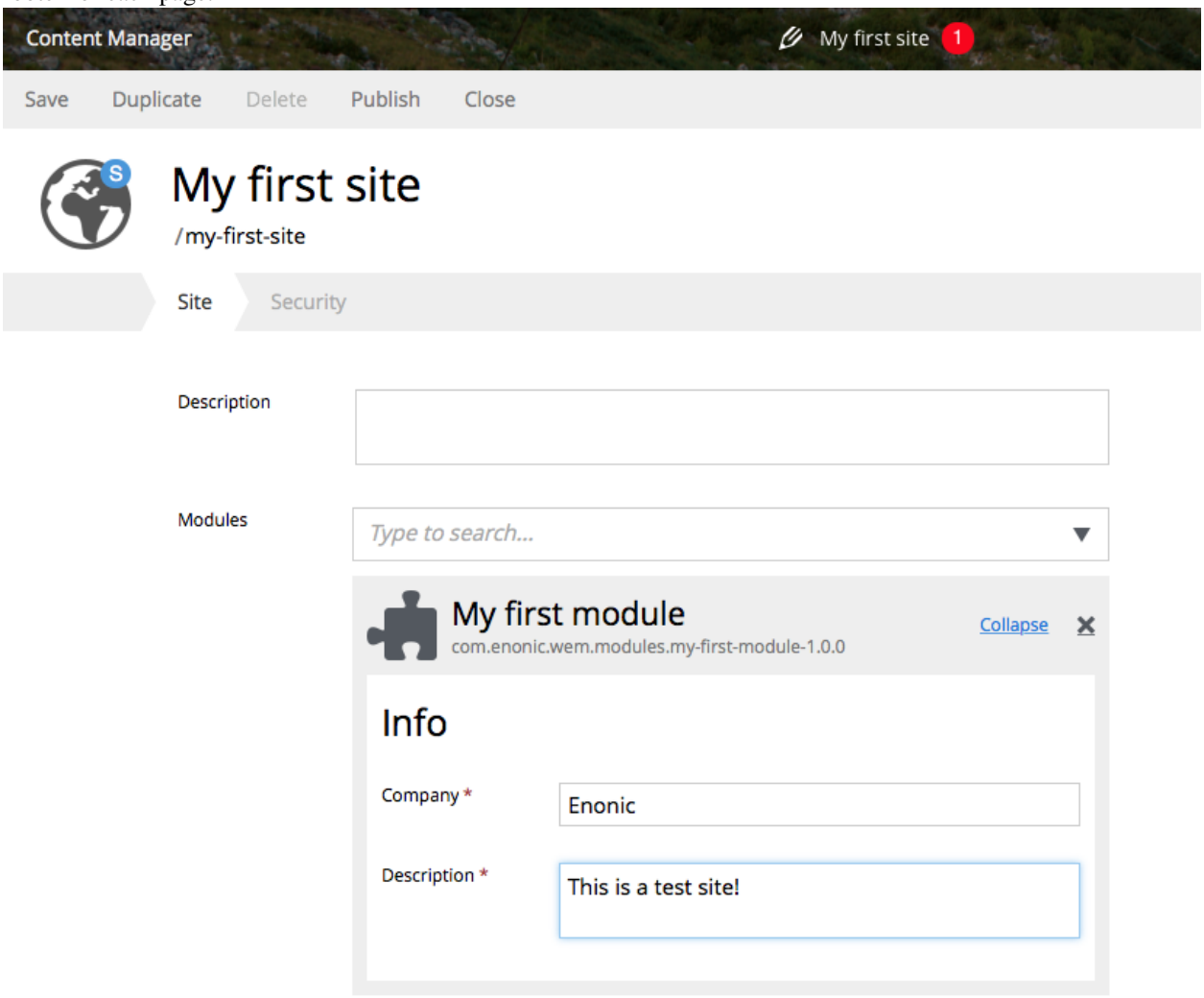


3.11 Configuring the module

Global configuration variables for the site may be defined in the config element of `module.xml`. Values for these settings can be filled in when you edit the site in the admin console.

```
<module>
  <config>
    <field-set name="info">
      <label>Info</label>
      <items>
        <input type="TextLine" name="company">
          <label>Company</label>
          <occurrences minimum="1" maximum="1"/>
        </input>
        <input type="TextArea" name="description">
          <label>Description</label>
          <occurrences minimum="1" maximum="1"/>
        </input>
      </items>
    </field-set>
  </config>
</module>
```

The company and description fields may now be filled out by editing the site. The values will be used in a standard footer for each page.



To use the module configuration values, the controller must retrieve the config element for the module and pass it to the Thymeleaf rendering method.

```
// Get the current site.
var site = execute('portal.getSite');

// Find the module configuration for this module in current site.
var moduleConfig = site.moduleConfigs['com.enonic.first.module'];

// Get the current content. It holds the context of the current execution
// session, including information about regions in the page.
var content = execute('portal.getContent');

// Include info about the region of the current content in the parameters
// list for the rendering.
var mainRegion = content.page.regions["main"];

// Extend the model from previous example
var model = {
    name: "Michael",
    mainRegion: mainRegion,
    moduleConfig: moduleConfig
};
```

Now we can update `my-first-page.html` file to render the configuration values that were passed from the controller.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>My first page, with a view!</h1>
    <h2>Hello <span data-th-text="${name}">World</span></h2>
    <footer id="footer">
      <div>
        &copy; 2015
        <span data-th-text="${moduleConfig['company']}">My Company</span> -
        <span data-th-text="${moduleConfig['description']}">All Rights Reserved</span>
      </div>
    </footer>
  </body>
</html>
```

3.12 Creating a layout

Layouts are used in conjunction with `regions` to organize the various component parts that will be placed on the page via live edit drag and drop. Layouts can be dropped into the page `regions` and then `parts` can be dragged into the layout. This allows multiple layouts (two-column, three-column, etc.) on the same page and web editors can change things around without touching any code. Layouts can even be nested. Making a layout is similar to making pages and part components. They require a descriptor, a controller, and a view. We will make a two column layout with the widths at 70% and 30%.

The layout descriptor defines regions within the layout where parts can be placed with live edit. First make a new folder `layouts/layout-70-30`. Within this folder, make a file called `layout.xml` as seen below.

```
<layout-component>
  <display-name>70/30</display-name>
  <config/>
  <regions>
    <region name="left"/>
    <region name="right"/>
  </regions>
</layout-component>
```

Also, create a file called `controller.js` in the `layout-70-30` folder.

```
exports.get = function(req) {

  // Find the current component.
  var component = execute('portal.getComponent');

  // Resolve the view
  var view = resolve('./layout-70-30.html');

  // Define the model
  var model = {
    component: component,
    leftRegion: component.regions["left"],
    rightRegion: component.regions["right"]
  };

  // Render a thymeleaf template
  var body = execute('thymeleaf.render', {
    view: view,
    model: model
  });

  // Return the result
  return {
    body: body,
    contentType: 'text/html'
  };

};
```

To make the layout view, create a file in the `layout-70-30` folder called `layout-70-30.html`.

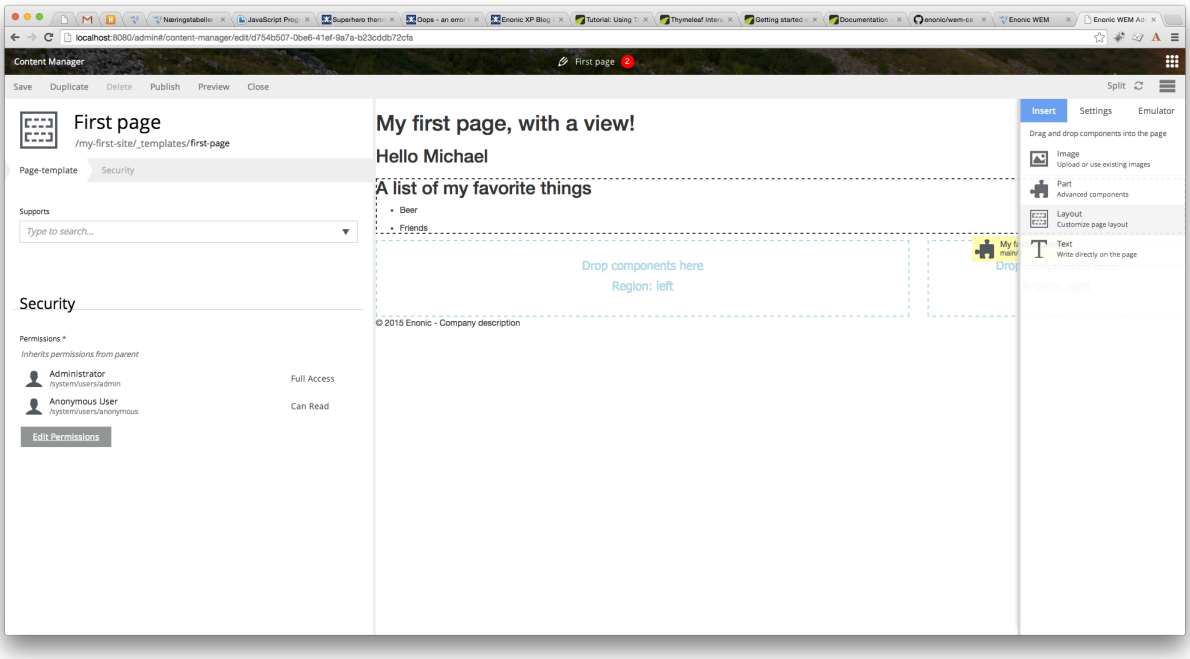
```
<div class="row" data-th-attr="data-portal-component-type=${component.type}">
  <div data-portal-component-type="region" class="col-sm-8">
    <div data-th-each="component : ${leftRegion.components}" data-th-remove="tag">
      <div data-portal-component="${component.path}" data-th-remove="tag" />
    </div>
  </div>

  <div data-portal-component-type="region" class="col-sm-4" >
    <div data-th-each="component : ${rightRegion.components}" data-th-remove="tag">
      <div data-portal-component="${component.path}" data-th-remove="tag" />
    </div>
  </div>
</div>
```

Some CSS must be added to style the page for the layout columns to work. Create a css folder inside the `assets` folder. Boostrap is used in this example so go ahead and download the minified CSS file and put it in the `assets/css` folder. Now the head element of the `view/my-first-page.html` file must be updated so that it includes the

bootstrap CSS file.

```
<head>
    <meta charset="utf-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
    <link data-th-href="${portal.assetUrl({'_path=css/bootstrap.min.css'})}" href="../assets/css/boot
</head>
```

After rebuilding the module, edit the page and click the `Insert` tab on the right. Then drag and drop a `Layout` to the `main` region. Then select the `70/30` layout from the dropdown. Now you can drag an image, text, or part into the layout regions.



3.13 Defining a Content-Type

Content-Types allow the creation of structured content. They are defined in XML and processed to generate forms for web editors to add content to the site without writing any code.

Create a folder called `person` in the `content-types` folder. Then create a file called `content-type.xml` inside the `person` folder. Fill it out as follows:

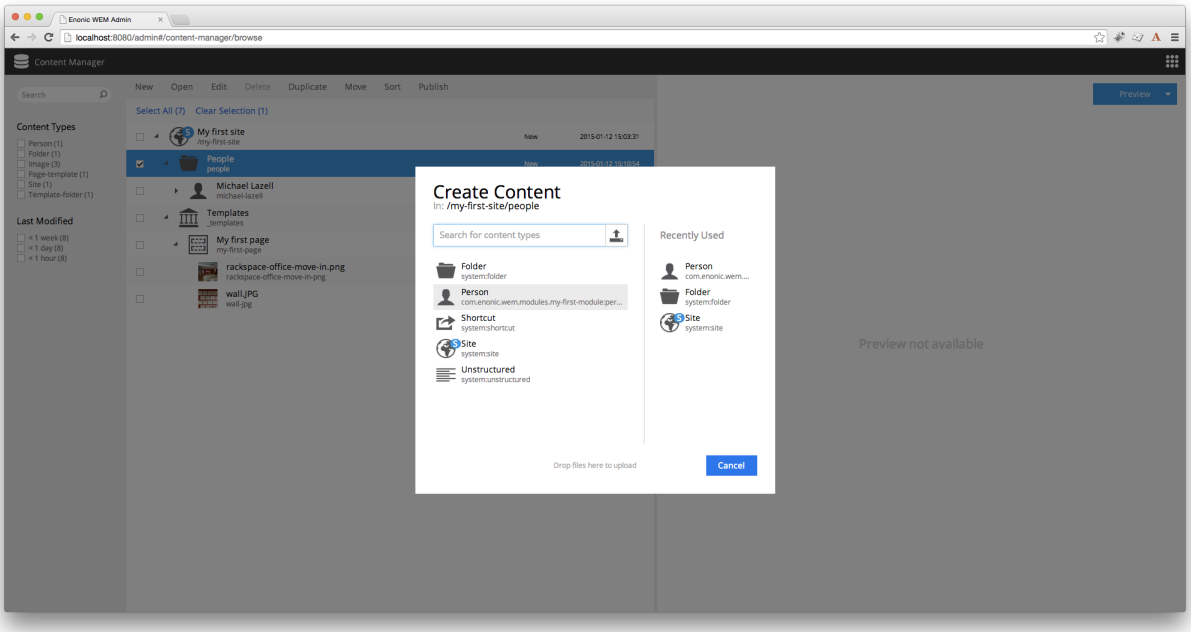
```
<content-type>
  <display-name>Person</display-name>
  <super-type>system:structured</super-type>
  <is-abstract>false</is-abstract>
  <is-final>true</is-final>
  <is-built-in>false</is-built-in>
  <allow-child-content>true</allow-child-content>
  <form>
    <field-set name="basic">
      <label>Person info</label>
      <items>
        <input type="TextLine" name="first-name">
          <label>First name</label>
```

```
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="1" maximum="1"/>
        </input>
        <input type="TextLine" name="last-name">
          <label>Last name</label>
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="1" maximum="1"/>
        </input>
        <input type="ImageSelector" name="image">
          <label>Photo</label>
          <immutable>false</immutable>
          <indexed>false</indexed>
          <occurrences minimum="1" maximum="1"/>
        </input>
        <input type="TextArea" name="bio">
          <label>Bio</label>
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="0" maximum="1"/>
        </input>
      </items>
    </field-set>
  </form>
</content-type>
```

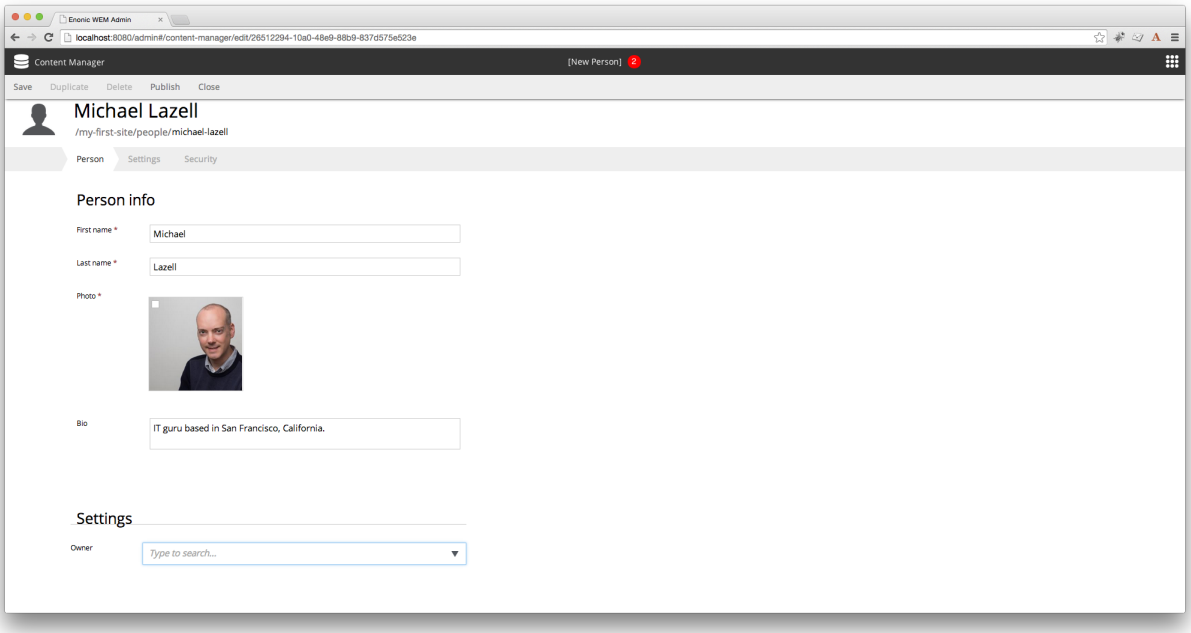
Save the following image with the name `thumb.png` and place it in the `content-type/person` folder. It will become the icon for the `person` content-type in the admin console.

3.14 Adding content

Once you rebuild the module, go to the content manager and select `New`. You will see `Person` listed as an available content-type in the administration console.



When you create a new `Person` content then you will see the form that was generated by the XML.



3.15 Displaying content

Now that a content-type has been defined and a content created, a new component part is needed to display the content. But first, a person content should be prepared in a specific location so that the code below can be copy/pasted and work properly. In the admin console content manager app, right-click on the `my-first-site` and select `new`. Then choose `folder` and name it `People`. Next, create a new person content in the *People* folder with the first name `Edvard` and the last name `Munch` and type in anything you like for the bio. Upload any image for the picture. Notice

the path of this content under the display name is `/my-first-site/people/edvard-munch`.

Now create a new folder in your project under the `parts` folder and name it `person-show`. Now create a part descriptor file named `part.xml`. This descriptor only needs a display-name for now.

```
<part-component>
  <display-name>Person</display-name>
  <config/>
</part-component>
```

Next, create the person-show controller named `controller.js`.

```
exports.get = function(req) {

  // Find the current component
  var component = execute('portal.getComponent');

  // Find the right content
  var person = execute('content.get', {
    key: '/my-first-site/people/edvard-munch'
  });

  // Combine the first and last name
  var personName = [
    person.data['first-name'],
    person.data['last-name']
  ].join(' ').trim();

  // Retrieve the image ID from the content used to create a image URL
  var imageId = person.data['image'];

  // Create a URL to the image
  var imageUrl = execute('portal.imageUrl', {
    id: imageId,
    filter: 'scaleblock(400,400)'
  });

  // Create view model
  var model = {
    component: component,
    person: {
      name: personName,
      image: imageUrl,
      bio: person.data['bio']
    }
  };

  // Return the result
  return {
    body: execute('thymeleaf.render', {
      view: resolve('person-show.html'),
      model: model
    }),
    contentType: 'text/html'
  }

};
```

Now a simple view file called `person-show.html` can be created to render the content data that is passed from the controller.

```
<div data-th-attr="data-portal-component-type=${component.type}">
  <p>
    <img class="img-responsive img-circle" data-th-src="${person.image}"
      src="../assets/images/person.jpg" alt="" />
  </p>
  <h3 data-th-text="${person.name}" data-th-remove="tag">Jane Doe</h3>
  <p data-th-text="${person.bio}">Person bio</p>
</div>
```

Rebuild the project and add the `Person` part to the page in *live edit*. This is a very simple example and not a very practical way to retrieve content. Errors would occur if the content name changed (due to the hardcoded content path), or if any of the content fields were missing values. The descriptor and controller will be modified in the next step to use a relationship type that will make the code more dynamic and robust.

3.16 Defining a Relationship-Type

Relationship-types allow you to create inputs that accept content as the value. These inputs can be used in content-types, module configurations, or the descriptors for component parts and pages. This section will explain how to set up a relationship type and modify the `Person` component part so that you can pick any `Person` content in the admin console and it will display correctly.

First create a folder in the `relationship-types` folder called `related-person`. Now make a file here called `relationship-type.xml`.

```
<relationship-type>
  <display-name>Person</display-name>
  <from-semantic>relates to person</from-semantic>
  <to-semantic>related of person</to-semantic>
  <allowed-from-types/>
  <allowed-to-types>
    <content-type>person</content-type>
  </allowed-to-types>
</relationship-type>
```

3.17 Use the Relationship-Type

Now the `Person` part descriptor can be updated to use the `Person` relationship type.

```
<part-component>
  <display-name>Person</display-name>
  <config>
    <input type="Relationship" name="person">
      <label>Person</label>
      <immutable>false</immutable>
      <indexed>false</indexed>
      <occurrences minimum="1" maximum="1"/>
      <config>
        <relationship-type>related-person</relationship-type>
      </config>
    </input>
  </config>
</part-component>
```

Once this is done, you will be able to choose a `Person` content for the part in the admin console. But the controller needs some updating to make it work dynamically. The following example shows how to get the related-person of the content.

```
/*
There are two ways to get content. This way only works when a page
with this `part` supports the content type. Then a URL that leads to
any `Person` content will reach this page and the content is
retrievable with `portal.getContent`. You can see this in action when
you click on a `Person` content in the admin console and it is
previewed in the page.
*/
var content = execute('portal.getContent');

// Find the related person id
var personId = component.config['person'];

// Fetch the actual person content
var person = execute('content.get', {
  key: personId
});
```

3.18 Defining a Mixin

Structures of data that are likely to be repeated in different content types can be defined in a single mixin. For example, a standard address form might be used in several content types. Each mixin definition file must be named `mixin.xml` and it must be placed in its own folder inside the `mixins` folder.

Go ahead and create the file below in a folder named `us-address`. The name of this folder will be used later when it is time to implement the mixin in another content type configuration.

```
<mixin>
  <display-name>U.S. Address format</display-name>
  <items>
    <form-item-set name="address">
      <label>Address</label>
      <immutable>false</immutable>
      <occurrences minimum="0" maximum="0"/>
      <items>
        <input type="TextLine" name="addressLabel">
          <label>Address Label</label>
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="0" maximum="1"/>
        </input>
        <input type="TextLine" name="addressLine">
          <label>Address Line</label>
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="1" maximum="2"/>
        </input>
        <input type="TextLine" name="city">
          <label>City</label>
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="1" maximum="1"/>
        </input>
```

```
        <input type="TextLine" name="state">
          <label>State</label>
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="0" maximum="1"/>
        </input>
        <input type="TextLine" name="zipCode">
          <label>Zip code</label>
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="1" maximum="1"/>
        </input>
      </items>
    </form-item-set>
  </items>
</mixin>
```

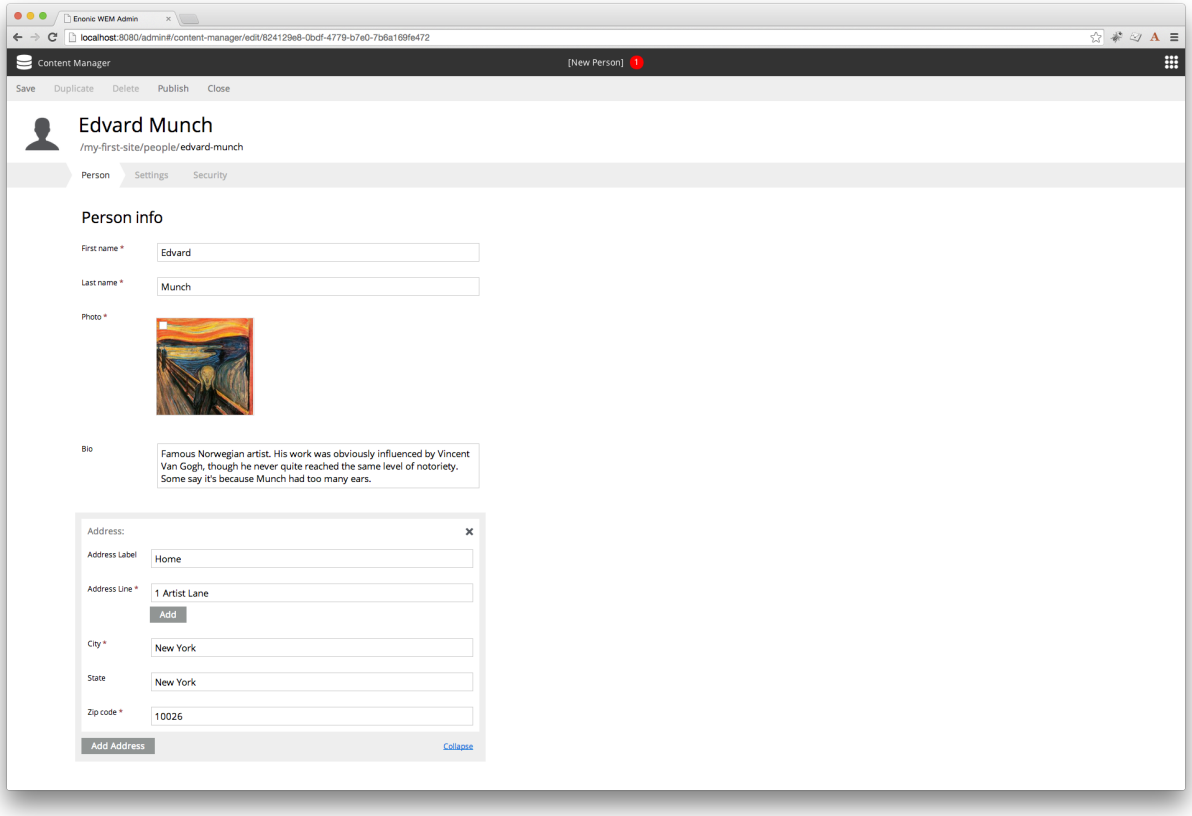
3.19 Using a Mixin

Now it is time to use the mixin in a content type. Below is the previous `Person` content type, modified to include an address. The mixin line as at the bottom.

```
<content-type>
  <display-name>Person</display-name>
  <super-type>system:structured</super-type>
  <is-abstract>false</is-abstract>
  <is-final>true</is-final>
  <is-built-in>false</is-built-in>
  <allow-child-content>true</allow-child-content>
  <form>
    <field-set name="basic">
      <label>Person info</label>
      <items>
        <input type="TextLine" name="first-name">
          <label>First name</label>
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="1" maximum="1"/>
        </input>
        <input type="TextLine" name="last-name">
          <label>Last name</label>
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="1" maximum="1"/>
        </input>
        <input type="ImageSelector" name="image">
          <label>Photo</label>
          <immutable>false</immutable>
          <indexed>false</indexed>
          <occurrences minimum="1" maximum="1"/>
        </input>
        <input type="TextArea" name="bio">
          <label>Bio</label>
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="0" maximum="1"/>
```

```
        </input>
      </items>
    </field-set>
    <x-data mixin="us-address"/>
  </form>
</content-type>
```

Now you can edit a `Person` content and see the new address fields.



CHAPTER 4

Node Domain

At the core of Enonic XP lies a distributed data storage - all persistent items in Enonic XP are stored as nodes.

4.1 Overview

Years of experience has taught us that traditional approaches to data storage (read SQL) are unsuited for the common requirements of modern cloud-based applications and platforms. A key goal of Enonic XP was to deliver a complete stack - virtually eliminating complex dependencies to 3rd party applications, and minimize requirements to infrastructure.

With the growing popularity of various so-called NoSQL (Not Only SQL) solutions, we evaluated many different technologies and found great inspiration in the following:

Git

- (+) Cherry picking
- (+) Branching
- (+) Pull requests
- (-) Performance search
- (-) Granularity of access (all or nothing)

Java Content Repository

- (+) Hierarchy
- (+) Granularity
- (+) Feature set
- (+) Unstructured
- (-) Performance
- (-) Complexity (not document oriented)
- (-) Attached data model
- (-) Requires additional storage backend

Elasticsearch

- (+) Document oriented
- (+) Scalability

- (+) Performance
- (+) Search
- (+) Aggregations
- (-) Search engine, not a database
- (-) No blob support
- (-) No security
- (-) Creates schemas

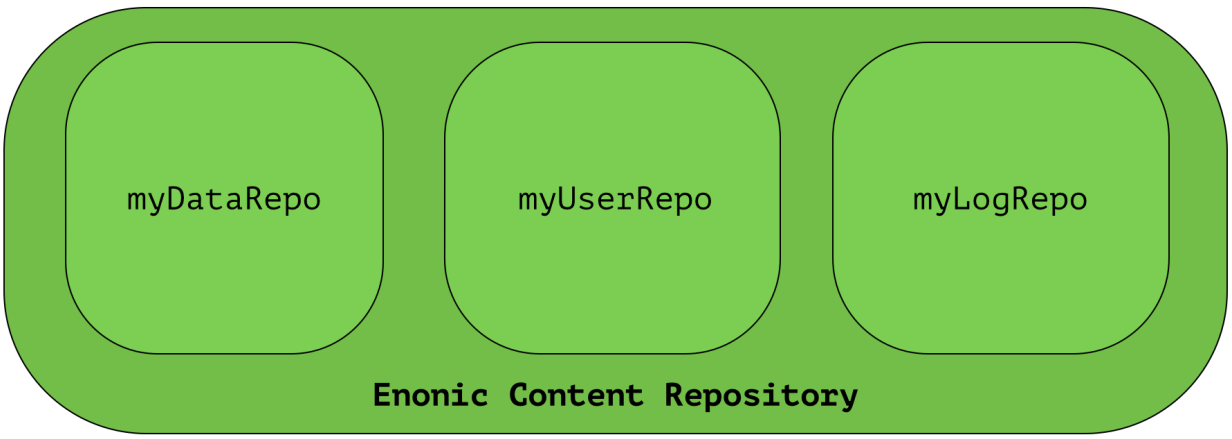
We were unable to find any single solution that was sufficiently simple and included our desired feature set - so we decided to build our own; the Enonic Content Repository.

The Enonic Content Repository is a place where you can store data, or more specific: *Nodes*.

It is built on top of Elasticsearch and exposes many of it's capabilities in search and aggregations and scalability - but in addition, provides the following capabilities:

- Hierarchical storage model
- Versioning support
- Complete Access Control and security model
- Blob support - using shared filesystem and append-only approach
- Repository and Branch concepts for content staging
- Schemaless - Add any property you like, at any time
- Rich set of value types (*HTMLPart*, *XML*, *Binary*, *Reference* etc..)
- SQL-like query syntax

The Enonic Content Repository itself contains one or more separate repositories based on the application need. For instance, an application could demand a setup having three repositories - one for application data, one for users and one for logging:



The reasons for having several separated repositories are many, and explained in detail in the *Repository* below.

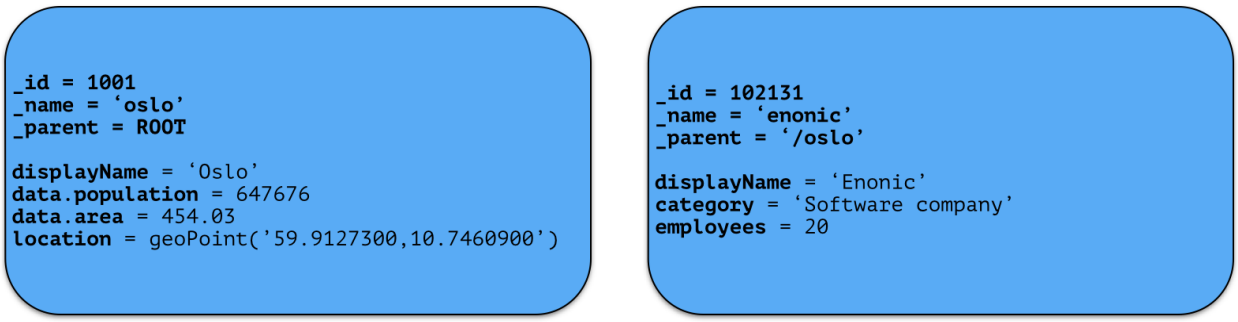
4.2 Nodes

A Node represents a single storable entity of data. It can be compared to a "row" in sql, or a "document" in document oriented storage models. Nodes are, as mentioned in the previous section, stored in a repository.

Every node has:

- a name
- a parent-reference
- an id
- a timestamp
- a (possibly empty) set of *Property* key/value.

Consider two nodes - one node representing the city "Oslo" and another representing the company, "Enonic":



The nodes have different properties. There is no schema to a node, so a node property value with the same property-name can have different value-types across nodes.

4.3 Property

Properties represent a placement of data in a node - following the simple `key = value` pattern. A property has a path. Elements in the path are separated by `.` (dot). Every property has also a type. See the complete list of *Value Types*.

```
myProperty
data.myProperty
cars.brands.skoda
```

For a property to be able to hold other properties, it has to be of type `Set`. In the above samples, `data`, `cars` and `brands` are properties of type `Set`.

Some characters are illegal in a property key. Here's a list of illegal characters:

- `_` is illegal as the first character, because it is a reserved prefix for *System Properties*.
- `.` is illegal as any character, since it is the path separator.
- `[` and `]` are also illegal as any character. These are used as array index indicators.

Here's an example of some properties:

```
first-name = "Thomas"
cities = ["Oslo", "San Francisco"]
city.location = geoPoint('37.785146,-122.39758')
```

```
person.age = 39
person.birth-date = localDate("1975-17-10")
```

4.4 Value Types

At the core of the node domain are value types. Every property to be stored in a node must have a value type. The value type enables the system to interpret and handle each piece of data specially - applying to both validation and indexing.

All value-types support arrays of values. All elements in an array must be of the same value-type.

Below is a complete list of all supported value-types.

String A character string.

Index value-type `String`

Example `'myString'`

GeoPoint Represents a geographical point, given in latitude and longitude.

Index value-type `GeoPoint`

Example `'59.9090442,10.7423389'`

Boolean A value representing `true` or `false`.

Index value-type `String`

Example `true`

Double Double-precision 64-bit IEEE 754 floating point.

Index value-type `Double`

Example `11.5`

Long 64-bit two's complement integer.

Index value-type `Double`

Example `1234`

Instant A single point on the time-line.

Index value-type `Instant`

Example `2015-03-16T10:00:02Z`

LocalDateTime A date-time representation without timezone.

Index value-type `String`

Example `2015-03-16T10:00:02`

LocalTime A time representation without timezone.

Index value-type `String`

Example `10:00:03`

HTMLPart Accepts a String containing valid HTML.

Index value-type `String`

Example `'<h1>my header</h1>'`

XML Accepts a String containing valid XML.

Index value-type `String`

Example `'<property>myPropertyValue</property>'`

Reference Holds a reference to other nodes in the same repository.

Index value-type `String`

Example `'0b7f7720-6ab1-4a37-8edc-731b7e4f439e'`

BinaryReference Reference to a binary object.

Index value-type `String`

Example `'my-binary-ref'`

Set A special value type that holds properties, allowing nested levels of properties.

4.5 System Properties

To reduce complexity, we explicitly dropped the use of namespaces. Thus, in order to separate system properties from user defined properties, we reserved _ as a starting character for system properties.

Below are the system properties explained.

_id Holds the id of the node, typically generated automatically in the form of a UUID.

_name Holds the name of the node. The name must be unique within its scope (all nodes with same parent).

_parentPath Reference to parent node path.

_path The path is resolved from the node name and parent path.

_timestamp The last change to the node version.

_nodeType Used to create collections for nodes in a repository.

_versionKey The id of the node version.

_state Used for keeping state of a node in a branch.

_permissions_read The principals that have read access.

_permissions_create The principals that have create access.

_permissions_delete The principals that have delete access.

_permissions_modify The principals that have modify access.

_permissions_publish The principals that have publish access.

_permissions_readpermissions The principals that have access to read the node permissions.

_permissions_writepermissions The principals that have access to change the node permissions.

4.6 Repository

A repository is a place where nodes can be stored. Data stored in a repository will typically belong to a common domain. Fetches and searches are by default executed against a single repository, so it makes sense to keep data from

different domains separated in different repositories. For instance, in the Enonic XP CMS, content and data concerning user management are separated into two repositories. The Content Manager application uses the `cms-repo` repository, and the User Manager application uses the `system-repo` repository.

When nodes are stored in the repository, two things happens:

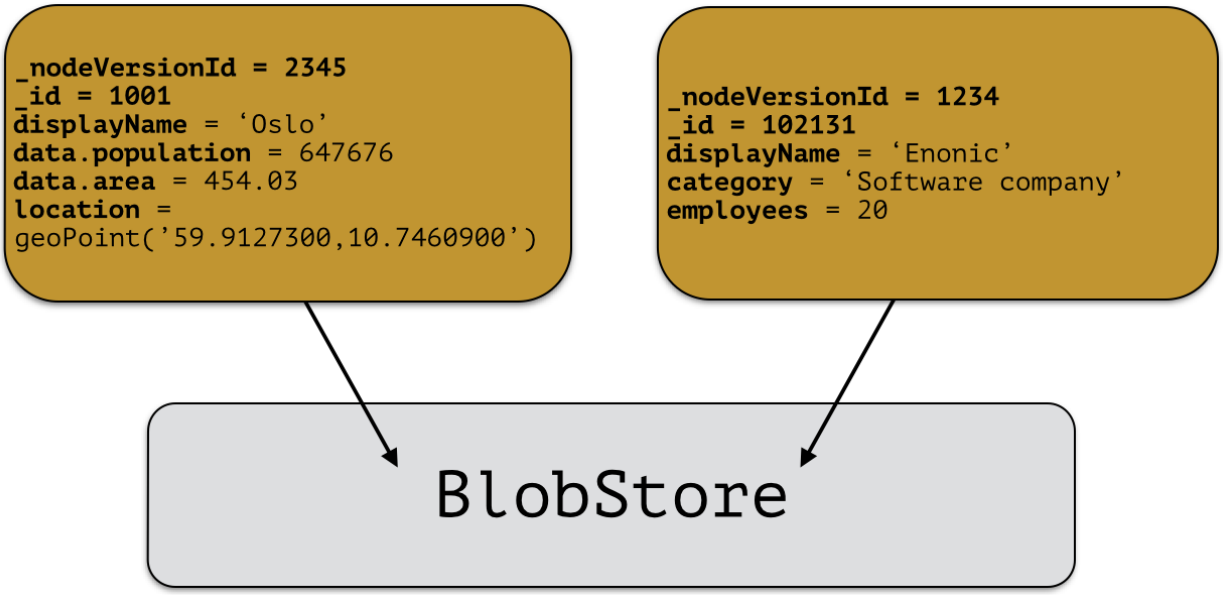
- The node properties are stored in a *Blobstore* as a *node-version*. A node-version is an entity representing the properties of the node, without name, parent and other meta-data.
- The node is inserted into a *branch*. The branch keeps track of a tree-structure referring to node-versions.

A repository will always contain a default branch, called 'master'. If there is more than one branch, API methods are used for resolving diff between and pushing changes from one branch to another.

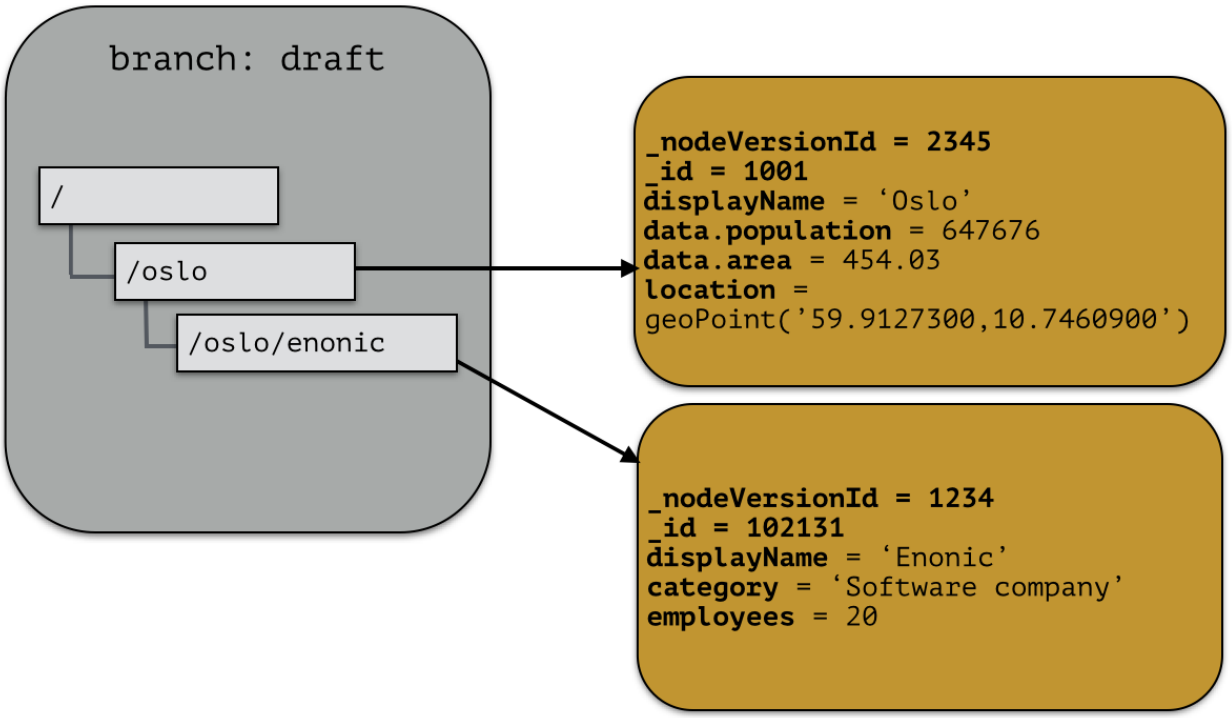
4.6.1 Node versions and branches

Consider the 'Oslo' and 'Enonic' nodes from earlier sections:

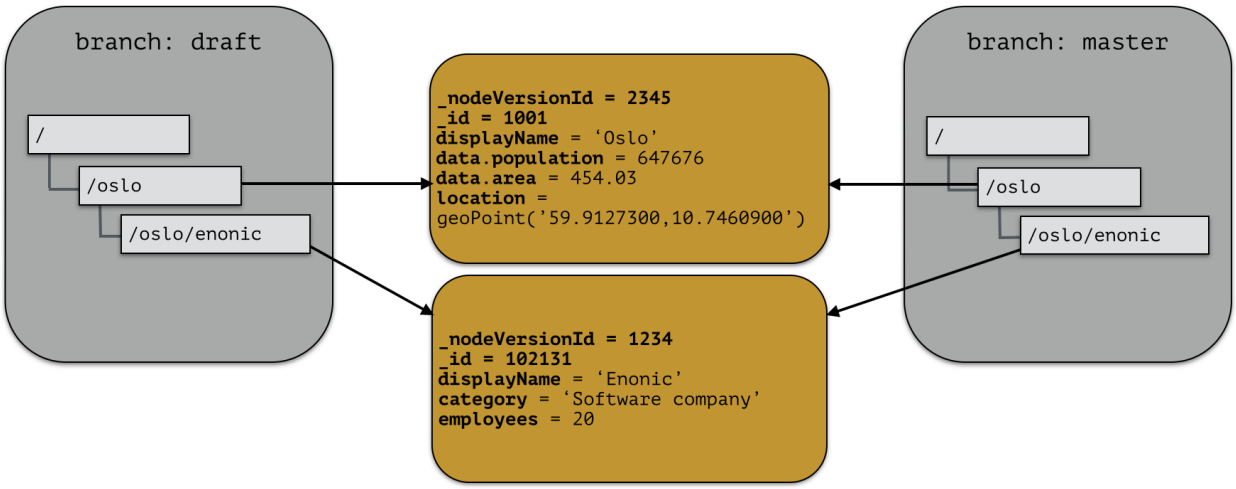
There will be two *node-versions* in the repository stored in the blobstore:



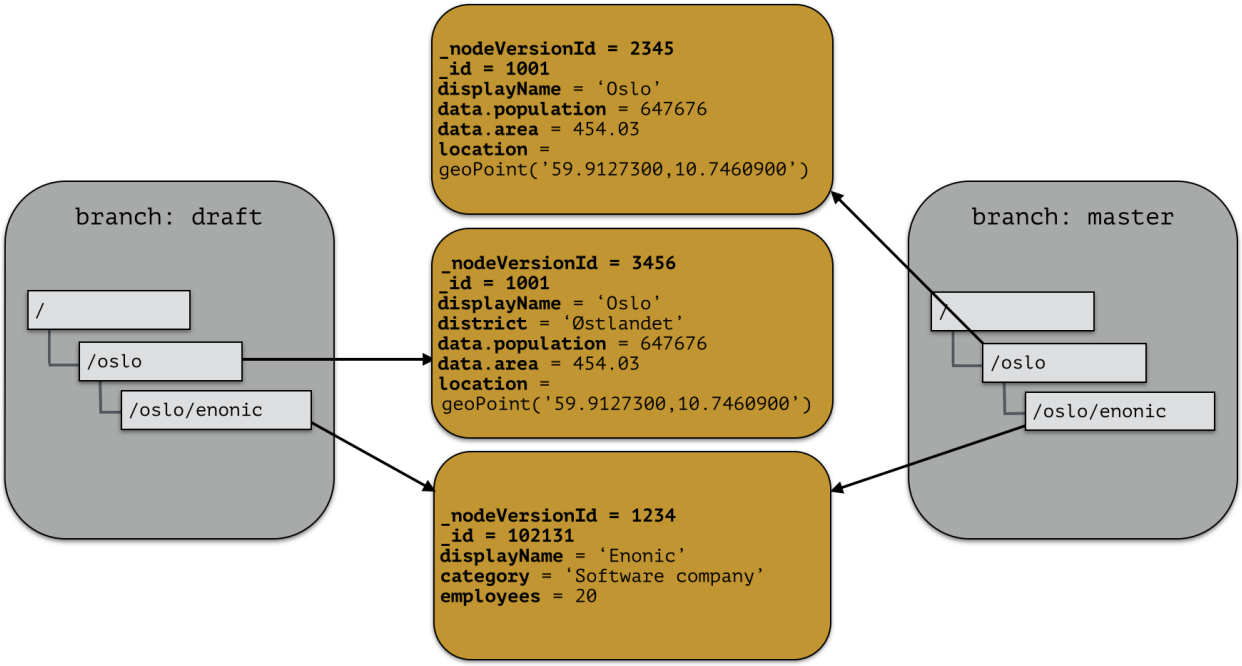
A node-version is a representation of a node's properties. A node-version has no knowledge of name, parent or other meta-data: just the properties of a node. At the same time, the targeted branch (named 'draft' in this example) gets two entries:



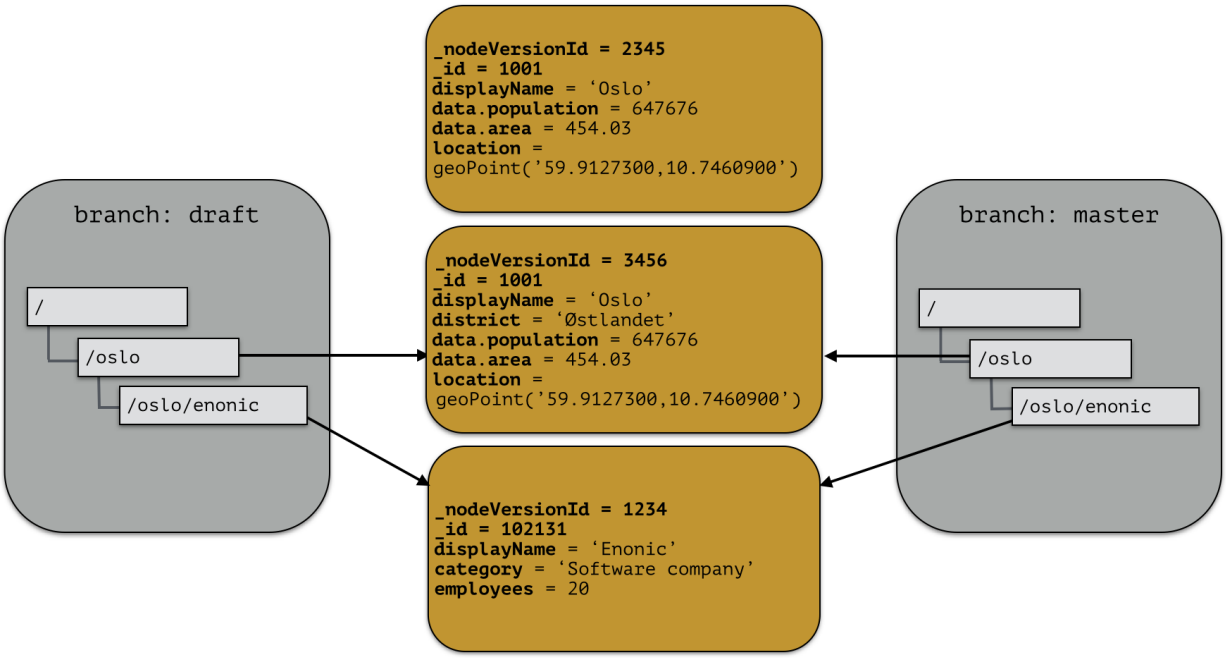
The node-versions are now a part of a tree-structure, based on the node's name and parent. If we *push* the content of branch 'draft' to the default branch 'master', we end up with something like this:



At the moment, there are two branches pointing to the same node-versions. This means that a single node version can exist in several branches with different structures. Now, consider that the 'oslo' - node is updated and stored to the 'draft'-branch, resulting in a new node-version with the same id and an updated pointer from the branch:



The two branches now point to different node-versions of the 'oslo' node. Again, doing a push-operation from 'draft' to 'master' will result in both nodes pointing to the same node-versions:



4.6.2 Repository characteristics

Note: Currently, there is no API for creating and managing repositories, so this information is for reference only at the moment.

A repository should be tuned to match the characteristic of the data you want to store, e.g:

- Expected number of documents
- Read or Write - optimized
- Real-time/near real-time/batch - data availability requirements
- Analyzing
- Archiving strategy

For instance; a log repository will have to be able to handle a large amount of data, but there will probably be no real-time requirements for data to be available. Also, archiving data will be needed to prevent the repository from growing infinitely.

4.7 Blobstore

The blobstore is a file system location defaulted to `$XP_HOME/blob`. The blobstore itself is split into one directory for nodes and one for binaries.

CHAPTER 5

Content Domain

Content is king, and Enonic XP ships with a complete CMS for your disposal.

5.1 Content vs Node

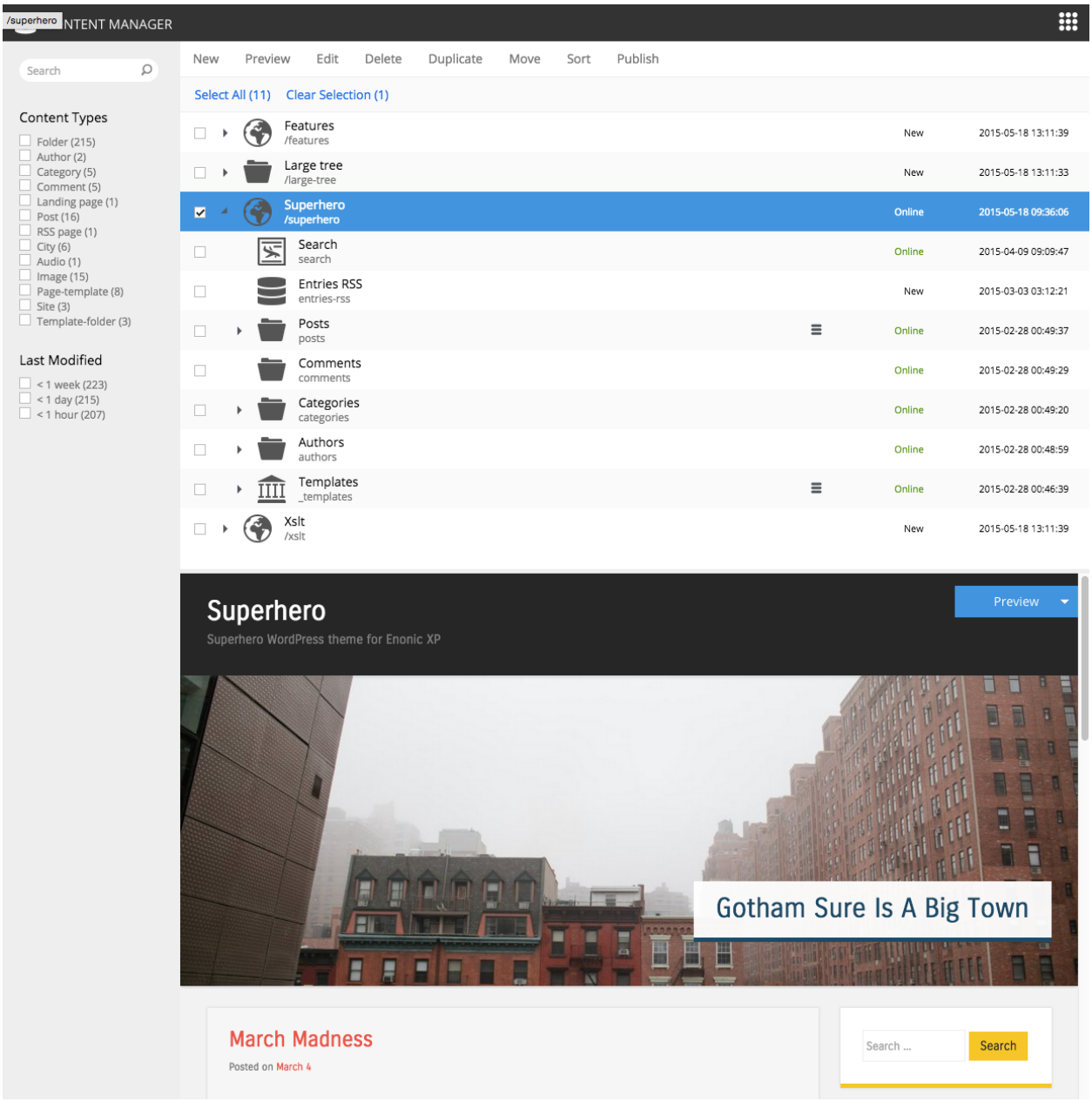
The foundation of Enonic XP is the *Node Domain*. In this domain, very general data can be stored and retrieved through the node-API. A node is schema-less, and contains a minimal set of set properties.

A content is a basically a node with a schema and a rich API on top of the node-domain. The schema is defined through *Content Types*.

Content nodes are stored in a provided *Repository* called `cms-repo`, and can be managed in the Enonic XP Content Manager application.

5.2 Content manager

Enonic XP ships with an application for managing content, called the "Content Manager".



In the above screenshot, we see a listing of content in the middle, a preview of a site-content in the portal and a faceted search on the right side.

5.3 Content repository

A built-in content repository called `cms-repo` is initialized when installing Enonic XP. This is where content is stored when working with content in the Content Manager application or the content-API.

Inside a repository exists something called branches. A branch is a tree-structure containing content. The `cms-repo`-repository has two branches:

- `draft`
- `master`

When working in Content Manager, the content seen is in the branch `draft`. Content in the portal is served from the `master`-branch.

Moving content from the `draft` branch to the `master`-branch are called publishing.

5.4 Content Types

Content Types provide developers with a rich, flexible and yet simple way to define interfaces and the resulting data models. Some highlights are:

- A rich set of widgets called Input Types
- Ability to group Input Types for tree-structured data
- Array support for everything
- Automatic generation of content display name from other fields
- Horizontal inheritance through Mixins

5.4.1 Base Content Types

Everything has to start somewhere, and for content types this starts with the base types. The base types are all installed and delivered with the system

Content types have a set of special tricks you need to know about:

- Content types are named with their module name, i.e. base:folder, where "base" is the module - but also have a nice display name like "Folder"
- `abstract` (default: false) means you cannot create content with this content type
- `final` (default: false) means it is not possible to create content types that "extend" this
- `allow-child-content` (default: true) if false, it will prevent users to create child items of content of this type.. (i.e. prevents creating child items of images)

Folder (base:folder)

- abstract: false
- final: false
- allow-child-content: true

Folders are simply containers for child content, with no other properties than their name and Display Name. They are helpful in organizing your content.

Media (base:media)

- abstract: true
- final: false
- allow-child-content: false

This content type serves as the abstract supertype for all content types that in their natural habitate are considered "files", these are listed below

Unstructured (base:unstructured)

- abstract: false
- final: true
- allow-child-content: true

The unstructured content type is a magical content type that permits the creation of any property and structure - without actually defining one first.

> **Caution:** There is currently no UI for unstructured content so they will appear as empty from the admin console.

Structured (base:structured)

- abstract: true
- final: false
- allow-child-content: true

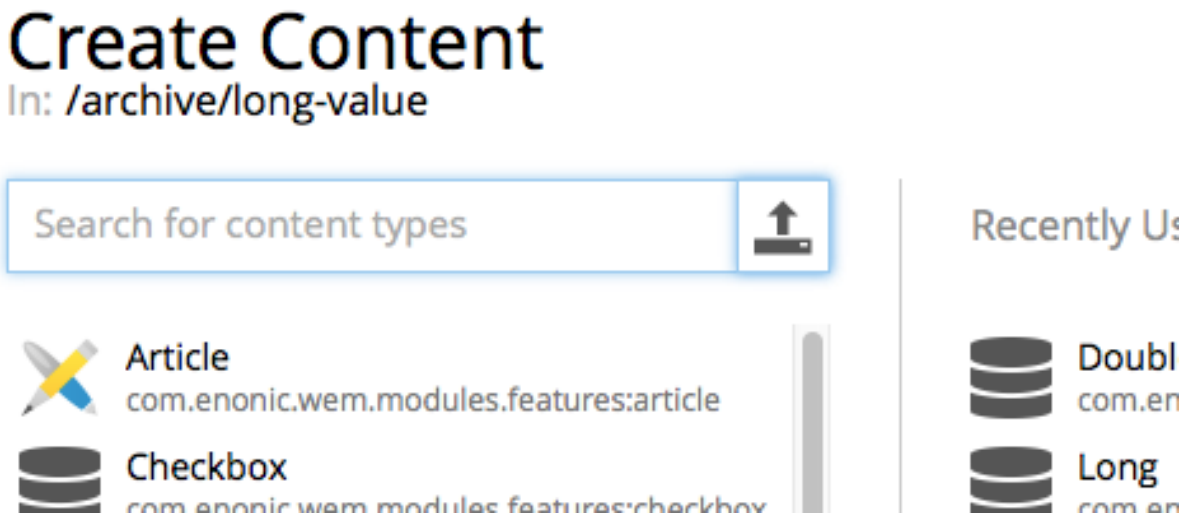
This is possibly the most commonly used base type for creating other content type. The structured content type is the foundation for basically any other structured content you can come up with, such as the `Person` content type above.

Media Content Types

- super-type: base:media
- abstract: false
- final: true
- allow-child-content: false

These settings apply to all the listed content types. When uploading a file to Enonic XP it will be transformed to one of the following content-types.

Tip: Enonic XP treats media content pretty much like any other content items - for instance the person, but they all have at least one attachment (namely the file).



Here are the various media content types that also come installed with Enonic XP:

Text (media:text) Plain text files.

Data (media:data) Misc binary file formats.

Audio (media:audio) Audio files.

Video (media:video) Video files.

Image (media:image) Bitmap image files.

Vector (media:vector) Vector graphic files like .svg.

Archive (media:archive) File archives like .zip, tar and jar.

Document (media:document) Text documents with advanced formatting, like .doc, .odt and pdf.

Spreadsheet (media:spreadsheet) Spreadsheet files.

Presentation (media:presentation) Presentation files like Keynote and Powerpoint.

Code (media:code) Files with computer code like .c, .pl or .java.

Executable (media:executable) Executable application files.

Unknown (media:unknown) Everything else.

5.4.2 Portal content types

In order to build sites in a secure and fashionable manner, Enonic XP also ships with a few special purpose content types.

Site (portal:site)

- super-type: base:structured
- abstract: false
- final: true
- allow-child-content: true

The Site content type is the root of a web-site, it's special trick is that you can add modules to it, and the portal will then be capable of presenting your content. Also, another important fact is that when a module is added to a site - you will also be able to create content from that specific module within it.

Page Template (portal:page-template)

- super-type: base:structured
- abstract: false
- final: true
- allow-child-content: true

Page templates are the equivalent of "master slides" in keynote and powerpoint. They enable you to set up pages that will be used when presenting other content types. From the sample content type above, the page template "Person Show" was taking care of the presentation.

Template folder (portal:template-folder)

- super-type: base:folder
- abstract: false
- final: true
- allow-child-content: `portal:page-template` only

This is the special content-type. Every site automatically creates a child content `_templates` of this type. that every Site has to hold the page templates of that site. It may not hold any other content type, and it may not be created manually in any other location.

Shortcut

Intended to be a URL with special functionality, but not ready for real use yet. The content type name is `base:shortcut`.

5.4.3 Custom Content Types

Custom Content Types can be created using Java or simple xml files - and deployed through modules.

When using xml, each content type must have a separate folder in the module resource structure. i.e. `cms/content-types/<my-content-type-name>`.

Each folder must then hold a file named `content-type.xml` and optionally an icon file, i.e. `thumb.png` The icon must be a Portable Network Graphics file (png). Other file formats will be supported in the future.

This is the basic structure of a `content-type.xml` file:

```
<content-type>
  <display-name>Choices</display-name>
  <content-display-name-script>$('firstName', ' ', 'lastName')</content-display-name-script>
  <super-type>base:structured</super-type>
  <is-abstract>false</is-abstract>
  <is-final>true</is-final>
  <allow-child-content>true</allow-child-content>
  <form>
```

```
    <input name="choice1" type="ComboBox">
      <label>Choice1</label>
      <occurrences minimum="0" maximum="1"/>
      <config>
          ...
      </config>
    </input>
  </form>
</content-type>
```

display-name The display name of the content type is used throughout the admin console to recognize it. But the technical name is the name of the folder the file is placed in.

super-type Many properties are inherited from the super-type. All custom content types must either inherit `base:structured` directly or indirectly. The icon and the general form to edit the fields of the content are important properties that are inherited from `base:structured`.

is-abstract If a content type is abstract, no content of this type may be instantiated. It may still be used as a super type for other content types.

is-final Final content types may not be used as super types of other content types.

allow-child-content If child content is allowed, it is legal to add nodes in the tree below a content of this type.

form Fields in the content type are defined as input elements which are placed inside the `form` element. All legal input types are described below.

input `name` and `type` are mandatory attributes of the input element. `label` and `occurrences` are mandatory child elements.

config Some input types have a complex configuration that is defined inside a `config` element. It is mandatory for the content types that need it.

content-display-name-script The name of a content may be generated by JavaScript from the values in the form.

5.5 Input Types

Each input type holds a specific type of data. A general input type is defined like this:

```
<input name="name" type="type-name">
  <label>Some label</label>
  <immutable>false</immutable>
  <indexed>true</indexed>
  <occurrences minimum="0" maximum="1"/>
</input>
```

@name The `name` attribute is the technical name used in templates and result sets to refer to this value.

@type The type refers to one of the many input types which are explained below.

label The label text will become the label for the input field in the editable form of the admin console.

immutable Immutable is not implemented yet. Setting this value to `true` will cause the value in such a field to become a constant when it is implemented.

indexed Indexed in not currently in use either, but required. Thought to indicate if the value should be indexed so it may be searchable, but will most likely be removed.

occurrences Detailed definition of how many times this field may be repeated inside one content. Set `minimum` to zero for fields that are not required, and `maximum` to zero for fields that have no restriction on the number of values.

5.5.1 Checkbox

A checkbox field has a value if it is checked, or no value if it is not checked. Therefore, the only values for occurrences that makes sense is a minimum of zero and a maximum of one. Any other value than zero for minimum is illegal.

5.5.2 ComboBox

A ComboBox needs a list of options.

```
<input name="name" type="ComboBox">
  <label>Required</label>
  <occurrences minimum="1" maximum="1"/>
  <config>
    <options>
      <option>
        <label>option A</label>
        <value>o1</value>
      </option>
    </options>
  </config>
</input>
```

config The config element lists the available options.

option Each option should have a `label` and a technical `value`. The `value` is passed to the server as a string. This input-type can have multiple options.

5.5.3 Date

A simple field for dates with a calendar pop-up box in the admin console. The default format is `yyyy-MM-dd`.

5.5.4 DateTime

A simple field for dates with time. A pop-up box with a calendar and time selector allows easy editing. The format is `yyyy-MM-dd hh:mm` for example, `2015-02-09T09:00`. The date-time could be of type `local` (no datetime) or with timezone. This is done using configuration:

```
<input name="datetime" type="DateTime">
  <config>
    <with-timezone>false</with-timezone>
  </config>
  <label>DateTime (no tz)</label>
  <indexed>true</indexed>
  <custom-text>Custom text</custom-text>
  <occurrences minimum="0" maximum="1"/>
</input>
<input name="requiredDatetime" type="DateTime">
  <config>
    <with-timezone>true</with-timezone>
  </config>
```

```
  <label>Required DateTime (with tz)</label>
  <indexed>true</indexed>
  <custom-text>Custom text</custom-text>
  <occurrences minimum="1" maximum="0"/>
</input>
```

with-timezone `true` if timezone information should be used.

5.5.5 Double

A double value input-type.

5.5.6 GeoPoint

Stores a GPS coordinate as two comma-separated decimal numbers.

- The first number must be a number between -90 and 90, where a negative number indicates a location south of equator and a positive is north of the equator.
- The second number must be a number between -180 and 180, where a negative number indicates a location in the western hemisphere and a positive number is a location in the eastern hemisphere.

5.5.7 HtmlArea

A field for entering html manually.

5.5.8 TinyMCE

A field for entering html in a WYSIWYG HTML editor.

5.5.9 ImageSelector

An ImageSelector is used to add images to a form. Existing image content may be selected, or a new image may be uploaded from the file system.

```
<input name="image" type="ImageSelector">
  <label>Non-required image</label>
  <occurrences minimum="0" maximum="1"/>
</input>
```

5.5.10 Long

A simple field for large integers.

5.5.11 ContentSelector

References to other content are specified by this input type.

```
<input name="cited-in" type="ContentSelector">
  <label>Cited In</label>
  <occurrences minimum="0" maximum="0"/>
  <config>
    <relationship-type>system:reference</relationship-type>
    <allow-content-type>citation</allow-content-type>
  </config>
</input>
```

relationship-type The `config` ' element is mandatory. It must contain the node `relationship-type` that contains the name of the relationship type. If no specific relationship type is required, use `system:reference`.

allow-content-type This optional element is used to limit the content types that may be selected for this input. Use one element for each content type. The full module name is required if the content type is defined in another module, like: `com.enonic.xp.modules.features:citation`

5.5.12 SingleSelector

An input type for selecting one of several options, defined in a mandatory `config` element. It may be either radiobutton, a dropdown list or a combobox. Example:

```
<input name="radio_selector" type="SingleSelector">
  <label>Radio Selector</label>
  <occurrences minimum="0" maximum="0"/>
  <config>
    <options>
      <option>
        <label>myOption 1</label>
        <value>o1</value>
      </option>
      <option>
        <label>myOption 2</label>
        <value>o2</value>
      </option>
    </options>
    <selector-type>RADIO</selector-type>
  </config>
</input>
```

occurrences When the occurrences element allows for multiple selectors, it is still only possible to select one item from each selector.

config The list of options for the selector is placed inside a `config` element.

option The `option` element must have a label and a value element. The label is what is presented to the user, while the value is the test that is stored on the back-end.

selector-type The `selector-type` element may have one of 3 values: `RADIO`, `DROPDOWN` and `COMBOBOX`.

5.5.13 Tag

An intuitive input format for specifying a set of simple strings.

5.5.14 TextArea

A field for inputting multi-line text.

5.5.15 TextLine

A field for inputting a single line of text.

5.5.16 Time

A simple field for time. A pop-up box allows simple selection of a certain time. The default format is `hh:mm`.

5.5.17 Field set

In order to group items visually, a field set may be used. This is an XML element with a label that will cause the form in the admin console to group the inputs inside the set under a heading from the label of the field set.

```
<field-set name="metadata">
  <label>Metadata</label>
  <items>
    <input name="tags" type="Tag">
      <label>Tags for tag cloud</label>
      <immutable>false</immutable>
      <indexed>false</indexed>
      <occurrences minimum="0" maximum="5"/>
    </input>
  </items>
</field-set>
```

@name The field set needs a name for reference.

label The label will appear as a heading above the inputs that are grouped inside.

items The fields inside the set must be listed inside an `items` element.

5.5.18 Form item set

It is possible to group inputs into logical units, either to allow them to repeat as a group, or just to visually separate items that belong together. A form item set is included in the content type config XML, at the level of the input node. Here is an example of a form item set with a regular input type before and after:

```
<form-item-set name="contact_info">
  <label>Contact Info</label>
  <items>
    <input name="label" type="TextLine">
      <label>Label</label>
      <occurrences minimum="0" maximum="1"/>
    </input>
    <input name="phone_number" type="TextLine">
      <label>Phone Number</label>
      <occurrences minimum="0" maximum="1"/>
    </input>
  </items>
  <immutable>false</immutable>
  <occurrences minimum="0" maximum="0"/>
</form-item-set>
```

name The set needs a name for reference in result sets.

label The set label is printed as a header on the box that will surround the group in the input form.

occurrences Occurrence configuration can be done at any level.

Tip: It is also possible to nest form item sets inside each other. Just include the nested set inside the `items` element, at the level of the `input` elements, just like at the top level.

5.6 Relationship Types

Custom content types may have relationships to each other or other content types. For instance, a person may have an image, or an employee may have a boss, or belong to a department. These relationships must be defined with a specific *relationship type*, then used in the custom content with an input type `ContentSelector`. The relationship type definition is an XML file. It must be placed in the folder, `cms/relationship-types/[name]` and be named `relationship-type.xml`. Here is an example of a relationship-type:

```
<relationship-type>
  <display-name>Citation</display-name>
  <from-semantic>citation in</from-semantic>
  <to-semantic>cited by</to-semantic>
  <allowed-from-types/>
  <allowed-to-types>
    <content-type>com.enonic.xp.modules.features:article</content-type>
  </allowed-to-types>
</relationship-type>
```

from-semantic Text to describe the "from" relationship.

to-semantic Text to describe the "to" relationship.

allowed-from-types Any content type may use this relationship-type.

allowed-to-types Wherever this relationship-type is used, only an article may be selected.

The content types have the format `module-name:content-type-name`. The module may be `system` for built-in types.

5.6.1 System relationship types

There are two default relationship types that may be used out of the box. These represent general relationship types that may be reused often.

`system:reference` No content type restriction, from-semantic = "relates to", to-semantic = "related of".

`system:parent` No content type restriction, from-semantic = "parent of", to-semantic = "child of".

5.7 Mixins

Structures of data that are repeated in many content types, like a set of address fields, or a combobox with a standard set of values, may be defined as mixins and reused in multiple content types. The mixin definition file must be named `mixin.xml` and placed in the folder `cms/mixins/[name]`. For example, `cms/mixins/us-address/mixin.xml`.

```
<mixin>
  <display-name>U.S. Address format</display-name>
  <items>
    <form-item-set name="address">
```

```
      <label>Address</label>
      <immutable>false</immutable>
      <custom-text/>
      <help-text/>
      <occurrences minimum="0" maximum="0"/>
      <items>
        <input type="TextLine" name="addressLine">
          <label>addressLine</label>
          <occurrences minimum="0" maximum="2"/>
        </input>
        <input type="TextLine" name="city">
          <label>City</label>
          <occurrences minimum="1" maximum="1"/>
        </input>
        <input type="TextLine" name="state">
          <label>State</label>
          <occurrences minimum="0" maximum="1"/>
        </input>
        <input type="TextLine" name="zipCode">
          <label>Zip code</label>
          <occurrences minimum="0" maximum="1"/>
        </input>
      </items>
    </form-item-set>
  </items>
</mixin>
```

5.7.1 Using a mixin

Below is an example of a simple contenttype that uses the `us-address` mixin. Notice that the name of the mixin folder is used and not the mixin's Display Name.

```
<content-type>
  <display-name>Using mixin: us-address</display-name>
  <super-type>base:structured</super-type>
  <is-abstract>false</is-abstract>
  <is-final>true</is-final>
  <is-built-in>false</is-built-in>
  <allow-child-content>true</allow-child-content>
  <form>
    <field-set name="basic">
      <label>Status</label>
      <items>
        <inline mixin="us-address"/>
      </items>
    </field-set>
  </form>
</content-type>
```

inline The mixin inputs are included with the `inline` element and the attribute `mixin` with the name of the mixin.

5.8 Content Structure

A content has a finite set of possible properties.

_id The content id.

_name The content name.

_parent The parent content path.

attachment If content contains attachments, a list of attachment-names and properties.

displayName Name used for display purposes.

contentType The content schema type.

creator The user principal that created the content.

createdTime The timestamp when the content was created.

data A property-set containing all user defined properties defined in the content-type.

x A property-set containing properties from mixins.

form The form defining the input type elements in the content.

language The locale-property of the content.

modifiedTime Last time the content was modified.

owner The user principal that owns the content.

page The page property contains page-specific properties, like template and regions. This will typically be reference to a page-template that supports the content-type.

site If content of type `portal:site`, this will contain site-specific information.

thumbnail A thumbnail representing the content.

type A property used to identify content in a node repository.

When creating a content, all defined properties in the content-type are stored under the content's data property. These properties will get the prefix `data`.

For example, a content-type article is defined like this:

```
<content-type>
  <display-name>Article</display-name>
  <super-type>base:structured</super-type>
  <form>
    <input name="title" type="TextLine">
      <label>My property text</label>
      <occurrences minimum="1" maximum="1"/>
    </input>
  </form>
</content-type>
```

The property "title" is now available in queries under the path `data.title`:

```
data.title = 'Fish and cheese'
```

5.9 Content Indexing

All content are indexed when stored. The properties of a content is indexed based on a set of rules:

- _id = string
- _name = fulltext

- _parent = string
- attachment = string
- displayName = fulltext
- contentType = string
- creator = string
- createdTime = datetime
- data = type
- x = type
- form = none
- language = string
- modifiedTime = datetime
- owner = string
- page = minimal
- page.regions = none
- site = none
- thumbnail = none
- type = string

5.9.1 Rules

When storing a content, the properties are indexed based on a set of rules:

string Indexed as a string-value only, no matter what type of data.

datetime Indexed as a datetime-value and string, no matter what type of data. If not able to parse value as date-time, no value will be indexed.

numeric Indexed as a numeric (double) and string, no matter what type of data. If not able to parse value as number, no value will be indexed.

minimal Indexed as a string-value only, no matter what type of data.

type Indexing are done based on type; e.g numeric values are stored as both string and numeric.

none Value not indexed.

ngram nGram-indexed fields are available for search by using the nGram-function. A nGram-analyzed field will index all substring values from 2 to 15 characters.

Consider this value of a property of type `text-line`:

```
"article"
```

This is split into the following tokens when analyzed:

```
'ar', 'art', 'arti', 'artic', 'articl', 'article'
```

For more information about how the nGram-function works, check out the nGram-function.

fulltext Fulltext-indexed fields are available for search by using the fulltext-function. A fulltext-analyzed field will be split into tokens.

Consider this value of a property of type `text-line`:

```
"This article contains information test-driven development"
```

This is split into the following tokens when analyzed:

```
'this', 'article', 'contains', 'information', 'about', 'test', 'driven', 'development'
```

For more information about how the fulltext-function works, check out the fulltext-function.

CHAPTER 6

Modules

Modules are the core components in Enonic XP. A module is a standalone package of code, content and structure that can be deployed into any instance of Enonic XP.

6.1 Project structure

The project structure is similar to Maven. Create the folder structure you see below. All are folders except for `module.xml` and `build.gradle`:

```
my-first-module/
  build.gradle
  src/
    main/
      resources/
        module.xml
        cms/
          lib/
          pages/
          parts/
          layouts/
          view/
          assets/
          content-types/
          mixins/
          relationship-types/
          services/
```

Every file and folder has a specific function and meaning.

build.gradle Gradle script for building the module. This file describes the actual build process.

module.xml The `module.xml` file contains basic information to register the module in Enonic XP. Settings for the module can be defined in the `config` element and the values for these settings can be updated in the administration console. An example will be presented later in this document. Leave the `config` element blank for now.

```
<module>
  <config/>
</module>
```

cms/lib/ This is the last place the global `require` javascript-function looks, so it is a good place to put default libraries here.

cms/pages/ Page definitions should be placed here. They will be used to create page templates in the repository.

cms/parts/ Part definitions should be placed here. Parts are objects that can be placed on a page.

cms/layouts/ Layout definitions should be placed here. Layouts are definitions that restricts the placement of parts.

cms/views/ Views can generally be placed anywhere you want, just keep in mind what path to use when resolving them.

cms/assets/ Public folder for external css, javascript and static images.

cms/content-types/ Content schemas-types are placed here. Used to create structured content.

cms/mixins/ Mixin schema-types are placed here. A mixin can be used to add fields to a content-type.

cms/relationship-types/ Relationship-types are placed here. They are used to form relations between contents.

6.2 Building the module

The project is built using Gradle. Here is a simple build script to build a module.

```
buildscript {
  repositories {
    jcenter()
    maven {
      url 'http://repo.enonic.com/public'
    }
  }

  dependencies {
    classpath 'com.enonic.xp.gradle:gradle-plugin:1.2.0'
  }
}

apply plugin: 'com.enonic.xp.gradle.module'
version = '1.0.0'

module {
  name = 'com.enonic.first.module'
  displayName = 'My first module'
}
```

To build your module, write `gradle build` on the command line:

```
$ gradle build
```

This will produce a jar file that is located inside `build/libs` folder. That jar file is your build artifact you can install into Enonic XP.

6.3 Installing the module

After a module is built with Gradle, it must be deployed to your Enonic XP installation. There is not much of a module to build at this point, but you will want to build and check your project after each step of this tutorial. You can refer back to this section whenever you are ready.

To deploy your module, copy the produced artifact to your `$XP_HOME/deploy` folder:

```
$ cp build/libs/[artifact].jar $XP_HOME/deploy/.
```

We have simplified this process by adding a `deploy` task to your build. Instead of manually copying in the deploy folder, you can execute `gradle deploy` instead:

```
$ gradle deploy
```

For this to work you have to set `XP_HOME` environment variable (in your shell) to your actual Enonic XP home directory.

To continuously build and deploy your module on changes, you can use the `gradle watch` task. This will watch for changes and deploy the changes to Enonic XP. The server will then pick up the changes and reload the module. This is probably the fastest way to develop your module:

```
$ gradle watch
```

6.4 Controllers

A controller is a fragment of JavaScript code executed on the server that processes and responds to HTTP requests.

Every page, part, layout and service must have a controller which consists of a JavaScript file named `controller.js`.

6.4.1 Methods

A controller must have one or more functions to handle requests, one for each different http method: `GET`, `POST`, `DELETE`, etc. For every request sent to the controller the appropriate function will be called.

The functions that handle the requests can be exposed with the `exports` keyword.

```
// Handles a GET request
exports.get = function(req) {}

// Handles a POST request
exports.post = function(req) {}
```

A handler function receives a parameter with a `request` object, and returns a `response` object.

```
exports.get = function(request) {

  if (request.mode === 'edit') {
    // do something...
  }

  var name = request.params.name;
  log.info('Name = %s', name);

  return {
    body: 'Hello ' + name,
    contentType: 'text/plain'
  };

};
```

6.4.2 Request

The `request` object represents the HTTP request and current context for the controller.

```
{
  "mode": "edit",
  "uri": "http://enonic.com/my/page",
  "method": "GET",
  "branch": "master",
  "params": {
    "debug": "true"
  },
  "formParams": {
    "user": "dymmy",
    "password": "secret"
  },
  "headers": {
    "Language": "en",
    "Cookies": "mycookie=123; other=abc;"
  },
  "cookies": {
    "mycookie": "123",
    "other": "abc"
  }
}
```

mode Rendering mode, one of: `edit`, `preview`, `live`.

uri URI of the request.

method HTTP method of the request.

branch Name of the repository branch, one of: `draft`, `master`.

params Name/value pairs with the URI query parameters from the request.

formParams Name/value pairs with the form parameters submitted in the request.

headers Name/value pairs with the HTTP request headers.

cookies Name/value pairs with the HTTP request cookies.

6.4.3 Response

The `response` object is the value returned by a controller handler function. It represents the HTTP response.

```
{
  "status": 200,
  "body": "Hello World",
  "contentType": "text/plain",
  "headers": {
      "key": "value"
  },
  "redirect": "/another/page",
  "pageContributions": {}
}
```

status HTTP response status code. Default to `200`.

body HTTP message body of the response that can either be a string or a JavaScript object.

contentType MIME type of the body. Defaults to `text/plain; charset=utf-8`.

headers Name/value pairs with the HTTP headers to be added to the response.

redirect URI to redirect to. If specified the value will be set in the "Location" header and the status will be set to 303 ("See other").

pageContributions HTML contributions that can be provided from a component to a page. Will be described in a later section.

6.4.4 Page Contributions

Page contributions are fragments of HTML that a component (part or layout) can contribute to the page in which it is contained. The idea is to allow components to add Javascript or CSS stylesheets globally in the page, although it is not restricted to scripts or styles.

Page contributions help solving 2 problems:

- Allow components to insert scripts or styles in specific positions in the page where it is often required.

  For example, a component providing web analytics might require that a script is inserted at the end of the page `<body>`. Or a stylesheet needed for a component must be inserted in the `<head>` tag.

- Avoid duplicating script libraries or stylesheets required for a component. Even if the same component is included multiple times in a page, the library script contributed will only be added once.

Any part or layout controller can contribute content to the page. The values from all component contributions will be included in the final rendered page. Duplicated values will be ignored. There are 4 positions where content can be contributed to and inserted in the page:

- `headBegin`: After the `<head>` opening tag.
- `headEnd`: Bbefore the `</head>` closing tag.
- `bodyBegin`: After the `<body>` opening tag.
- `bodyEnd`: Before the `</body>` closing tag.

```
{
  "body": "<html>...</html>",
  "pageContributions": {
    "headEnd": "value",
    "bodyEnd": [
      "value1", "value2"
    ]
  }
}
```

Some remarks:

- All the `pageContributions` fields are all optional. The `pageContributions` object is optional, and each property inside is optional.
- The value for a contribution can be a string or an array of strings.
- The values are unique within an injection point (or tag position). If the same string is contributed from different parts, or from the same part that exists multiple times in the page, the value will only be inserted once. E.g. if 2 parts include a script for jQuery, it will be included once. But if one part is contributing to `headBegin` and another one contributes the same value to `bodyEnd`, then it will be inserted 2 times.
- If the tag does not exist in the rendered page, the value is ignored. I.e. if there is no `<head>` tag, the contributions to `headBegin` and `headEnd` will just be ignored.

- The contributions are inserted in a post-processing step during rendering. That means that there will not be any processing of Thymeleaf tags or similar, contributions are treated as plain text.

6.5 Configuring the module

Global configuration variables for the site may be defined in the config element of `module.xml`. Values for these settings can be filled in when you edit the site in the admin console.

```
<module>
  <config>
    <field-set name="info">
      <label>Info</label>
      <items>
        <input type="TextLine" name="company">
          <label>Company</label>
          <occurrences minimum="1" maximum="1"/>
        </input>
        <input type="TextArea" name="description">
          <label>Description</label>
          <occurrences minimum="1" maximum="1"/>
        </input>
      </items>
    </field-set>
  </config>
</module>
```

To use the module configuration values, the controller can read the values and use it.

```
// Get the current site.
var site = execute('portal.getSite');

// Find the module configuration for this module in current site.
var moduleConfig = site.moduleConfigs['com.enonic.first.module'];
```

6.6 Page

A page can be composed of parts and layouts or just be a simple page without the compositions. To create a page, you will have to add a descriptor, a controller and optionally - a view.

6.6.1 Descriptor

The page descriptor is required to register the page and allows us to set up user input fields to configure the page. It also allows us to describe what regions are available in this page.

```
<page-component>
  <display-name>My first page</display-name>
  <config/>
  <regions/>
</page-component>
```

display-name A simple human readable display name.

config The `config` element is where input fields are defined for configurable data that may be used on the page.

regions This is where regions are defined. Various component parts can be dragged and dropped into regions on the page.

6.6.2 Controller

A page controller handles requests to the page. The controller is written in JavaScript and must be named `controller.js`. A controller exports a set of methods, one for each HTTP method that should be handled. The handle method has a request object as parameter and returns the result.

```
// Handles a GET request
exports.get = function(req) {}

// Handles a POST request
exports.post = function(req) {}
```

Here's a simple controller that acts on the `GET` request method.

```
exports.get = function(req) {

  return {
    body: '<html><head></head><body><h1>My first page</h1></body></html>',
    contentType: 'text/html'
  };

};
```

6.6.3 Rendering a view

If you feel like concatenating strings to create an entire webpage is a little too much hassle, Enonic XP also supports views. A view is rendered using a rendering engine; we currently support XSLT, Mustache and Thymeleaf rendering engines. This example will use Thymeleaf.

To make a view, create a file `my-first-page.html` in the `view` folder.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>My first page, with a view!</h1>
  </body>
</html>
```

In our `controller.js` file, we will need to parse the view to a string for output. Here is where the Thymeleaf engine comes in. Using the Thymeleaf rendering engine is easy; here is how we do it.

```
exports.get = function(req) {

  // Resolve the view
  var view = resolve('/cms/view/my-first-page.html');

  // Define the model
  var model = {
    name: "John Doe"
  };

  // Render a thymeleaf template
```

```
  var body = execute('thymeleaf.render', {
    view: view,
    model: model
  });

  // Return the result
  return {
    body: body,
    contentType: 'text/html'
  };

};
```

6.6.4 Adding dynamic content

We can send dynamic content to the view from the controller via the `model` parameter of the `render` function. We then need to use the rendering engine specific syntax to render it. The controller file above passed a variable called `name` and here is how to extract its value in the view using Thymeleaf syntax.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>My first page, with a view!</h1>
    <h2>Hello <span data-th-text="${name}">World</span></h2>
  </body>
</html>
```

More on how to use Thymeleaf can be found in the official Thymeleaf documentation.

6.6.5 Regions

To be able to add components like images, component parts, or text to our page via the live edit drag and drop interface, we need to create at least one region. Regions can be declared in the page descriptor.

```
<page-component>
  <display-name>My first page</display-name>
  <config />
  <regions>
    <region name="main"/>
  </regions>
</page-component>
```

You will also need to handle regions in your controller.

```
// Get the current content. It holds the context of the current execution
// session, including information about regions in the page.
var content = execute('portal.getContent');

// Include info about the region of the current content in the parameters
// list for the rendering.
var mainRegion = content.page.regions["main"];

// Extend the model from previous example
var model = {
```

```
    name: "Michael",
    mainRegion: mainRegion
};
```

To make live-edit understand that an element is a region, we need an attribute called `data-portal-component-type` with the value `region` in our HTML.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>My first page, with a view!</h1>
    <h2>Hello <span data-th-text="${name}">World</span></h2>
    <div data-portal-component-type="region">
      <div data-th-each="component : ${mainRegion.components}" data-th-remove="tag">
        <div data-portal-component="${component.path}" data-th-remove="tag" />
      </div>
    </div>
  </body>
</html>
```

We can now use the live edit drag and drop interface to drag components onto our page.

6.7 Part

A part is a building block that can be placed on a page or into a region. As with pages, a part is composed of a descriptor, a controller and and optionally - a view.

The part descriptor is required to register the part and allows us to set up user input fields to configure the part. A part cannot contain any regions.

```
<part-component>
  <display-name>My favorite things</display-name>
  <config>
    <field-set name="things">
      <label>Things</label>
      <items>
        <input type="TextLine" name="thing">
          <label>Thing</label>
          <immutable>false</immutable>
          <indexed>true</indexed>
          <occurrences minimum="0" maximum="5"/>
        </input>
      </items>
    </field-set>
  </config>
</part-component>
```

To drive this part, we will also need a controller `controller.js`.

```
exports.get = function(portal) {

  // Find the current component from request
  var component = execute('portal.getComponent');

  // Find a config variable for the component
```

```
  var things = component.config["thing"] || [];

  // Define the model
  var model = {
    component: component,
    things: things
  };

  // Resolve the view
  var view = resolve('/cms/view/my-favorite-things.html');

  // Render a thymeleaf template
  var body = execute('thymeleaf.render', {
    view: view,
    model: model
  });

  // Return the result
  return {
    body: body,
    contentType: 'text/html'
  };

};
```

The part needs a root element with the attribute `data-portal-component-type`. In this case, it will be a `part`, but we can also resolve it dynamically as explained in the example. The `things` parameter is basically just JSON data, and we can loop it easily in Thymeleaf and print its value.

```
<section data-th-attr="data-portal-component-type=${component.type}">
  <h2>A list of my favorite things</h2>
  <ul class="item" data-th-each="thing : ${things}">
    <li data-th-text="${thing}">A thing will appear here.</li>
  </ul>
</section>
```

The part can now be added to the page via drag and drop. You will be able to configure the part in the *context window* in live-edit.

6.8 Layout

Layouts are used in conjunction with `regions` to organize the various component parts that will be placed on the page via live edit drag and drop. Layouts can be dropped into the page `regions` and then `parts` can be dragged into the layout. This allows multiple layouts (two-column, three-column, etc.) on the same page and web editors can change things around without touching any code. Layouts can even be nested. Making a layout is similar to making pages and part components.

Layout contains - like pages and parts - a descriptor, a view and a controller, and should be placed in under the layouts-folder in the module: `layouts/<layoutname>/`

6.8.1 Descriptor

The layout descriptor defines regions within the layout where parts can be placed with live edit.

```
<layout-component>
  <display-name>70/30</display-name>
  <config/>
  <regions>
    <region name="left"/>
    <region name="right"/>
  </regions>
</layout-component>
```

6.8.2 Controller

The layout controller composes the view of the layout based on http-requests.

```
exports.get = function(req) {

  // Find the current component.
  var component = execute('portal.getComponent');

  // Resolve the view
  var view = resolve('./layout-70-30.html');

  // Define the model
  var model = {
    component: component,
    leftRegion: component.regions["left"],
    rightRegion: component.regions["right"]
  };

  // Render a thymeleaf template
  var body = execute('thymeleaf.render', {
    view: view,
    model: model
  });

  // Return the result
  return {
    body: body,
    contentType: 'text/html'
  };

};
```

6.8.3 View

A layout view defines the markup for the layout-component. The sample view below is created in Thymeleaf, but you could create it in any view that's supported.

```
<div class="row" data-th-attr="data-portal-component-type=${component.type}">
  <div data-portal-component-type="region" class="col-sm-8">
    <div data-th-each="component : ${leftRegion.components}" data-th-remove="tag">
      <div data-portal-component="${component.path}" data-th-remove="tag" />
    </div>
  </div>

  <div data-portal-component-type="region" class="col-sm-4" >
```

```
    <div data-th-each="component : ${rightRegion.components}" data-th-remove="tag">
      <div data-portal-component="${component.path}" data-th-remove="tag" />
    </div>
  </div>
</div>
```

6.8.4 Styling

For a layout to have any meaning, some styling must be applied to the view. Depending on preferences, the needed css should be placed in the `/assets`-folder of the module, and included in the page where the layout should be supported, e.g the view my-first-page.html should support bootstrap layouts:

```
<head>
  <meta charset="utf-8"/>
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <link data-th-href="${portal.assetUrl({'_path=css/bootstrap.min.css'})}" href="../assets/css/bootst
</head>
```

6.9 Service

With services you can create REST-like services without binding it to some content. Place a `controller.js` file within a named folder under the `services` folder and this will be accessable with a fixed url:

```
*/_/service/[module]/[name]
```

Where `module` is the module name (without version) and `name` is the name of the service.

Here's a simple service that increments a counter and returns it as JSON.

```
var counter = 0;

exports.get = function(req) {

  counter++;

  return {
    body: {
      time: new Date(),
      counter: counter
    },
    contentType: 'application/json'
  };

};
```

6.10 Rendering a View

Instead of composing the HTML output in your controller, it's much easier to pass the model down to a view. We support pluggable view technologies and support the following out of the box:

- Thymeleaf (see *thymeleaf.render*)
- Mustache (see *mustache.render*)

- Xslt (see *xslt.render*)

6.11 Localization

Multi-language support for text on a site can be enabled by adding a localization resource bundle to the module. The localization resource bundle contains files with custom localized phrases which can be accessed through `localize` functions in the "controllers" and "view" files.

To see how this is used in a controller, see *i18n.localize* function. See *localize* function on how to use this in the various views.

6.11.1 Resource Bundle

The resource-bundle consists of a collection of files containing the phrases to be used for localization. The resource-bundle should be placed in a folder named `i18n` under the module resource root.

Each locale to be localized should be represented by a single resource, e.g this could be a structure for a module supporting

- 'English' (default)
- 'English US'
- 'Norwegian'
- 'Norwegian Nynorsk'

```
i18n/phrases.properties
i18n/phrases_en_us.properties
i18n/phrases_no.properties
i18n/phrases_no_nn.properties
```

The filename of a resource determines what locale it represents:

```
phrases[_languagecode][_countrycode][_variant].properties
```

Caution: The filename should be in lowercase.

The `languagecode` is a valid ISO Language Code. These are the two-letter codes as defined by ISO-639. You can find a full list of these codes at a number of sites, such as: http://www.loc.gov/standards/iso639-2/php/English_list.php.

The `countrycode` is a valid ISO Country Code. These are the two-letter codes as defined by ISO-3166. You can find a full list of these codes at a number of sites, such as: http://www.iso.ch/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html

A sample phrases.properties file would look like this:

```
user.greeting = Hello, {0}!
complex_message = Good to see you. How are you doing?
message_url = http://localhost:8080/{0}
message_multi_placeholder = My name is {0} and I live in {1}
message_placeholder = Hello, my name is {0}.
med_\u00e6_\u00f8_\u00e5 = This contains the norwegian characters æ, ø and å
```

Placeholders

Placeholders are marked with `{<number>}`. The given number corresponds with the function argument named `values` and the placement of the parameter. See below for an example.

Encoding and special characters

The encoding of localization resource bundle files must be ISO-8859-1, also known as Latin-1. All non-Latin-1 characters *in property-keys* must be entered using Unicode escape characters, e.g u00E6 for the Norwegian letter 'æ'. The values may also be encoded, but this is not required.

6.11.2 Resolving locale

A locale is composed of language, country and variant. Language is required, country and variant are optional.

The string-representation of a locale is:

```
LA[_CO][_VA]
```

where

- `LA` = two letter language-code
- `CO` = two letter country-code
- `VA` = two letter variant-code.

The variant argument is a vendor or browser-specific code. For example; WIN for Windows, MAC for Macintosh, and POSIX for POSIX. Where there are two variants, separate them with an underscore, and put the most important one first. For example, a Traditional Spanish collation might construct a locale with parameters for language, country and variant as: "es", "ES", "Traditional_WIN".

When a localize function is called upon, a locale is resolved to decide which localization to use.

The following is considered, in this order:

- Given as argument to function
- Site-language

6.11.3 Finding best match

When localizing a keyword, a best match pattern will be applied to the resource bundle to select the localized phrase. If the locale for a request is resolved to "en-US", these files will be considered in given order:

- `phrases_en_us.properties`
- `phrases_en.properties`
- `phrases.properties`

If the locale for a request would have been resolved to `en`, the `phrases_en_us.properties` file would not have been considered when localizing a keyword.

If the locale does not match a specific file, the default `phrases.properties` will be used.

If no matching localization key is found in any of the files in a bundle, a default `NOT_TRANSLATED` will be displayed.

CHAPTER 7

Search

How to find data in the Enonic Content Repository. For a system that deals with storing and retrieving data, a rich search-API is paramount.

7.1 Overview

When searching in Enonic XP, you are searching for nodes, or content if working in context of the CMS-module. This documentation is general and intended for the nodes-domain, but except for some built-in property-values and the addition of some convenience parameters in the content domain, everything is valid for both domains.

In general, the search-API's deals with a number of basic parameters:

- start
- count
- query
- filter
- aggregations

7.1.1 Start & count

When searching, the result will contain a number of matching nodes. This number if given by the provided `count` parameter in the query. The result will also contain a value indicating the total number of hits for the search: `total`. The `start` parameter indicates from what position in the result set we should start retrieving results.

Lets consider a search matching 1000 documents. Usually, one does not to retrieve all these results at once, but rather a subset of the result - and fetch the next subset of the result if necessary. This type of data-retrieving is called paging.

Typically, one will decide the number of wanted results for each iteration, e.g 100:

- start = 0
- count = 100

Then, for the next iteration, we will start from the first result not retrieved in the first iteration:

- start = 100
- count = 100

The `total` return field can be used to create page-navigation for the search result, by dividing the `total` hits by the page-size (`count`) to get the needed number of pages.

7.1.2 Query

The query-part of a search is where the constraints are defined. If query parameter is empty, all nodes in the repository will match. The query is defined in *Query Language* section.

The results matching the query constraint will be assigned a score. This is imperative for fulltext-type queries. The score of a matching documents depends on how the constraint is defined, e.g which fulltext-like function is used. See the *Query Functions* section for details.

7.1.3 Filter & query-filter

A filter also applies constraints. The difference between a filter-constraint and a query-constraint, is that the hits matching the filter are not scored. Scoring hits is a costly operation, and makes no sense for typical filter constraints like "price > 10", so its a good way of optimizing searches by appending non-fulltext operations to the filter-constraint instead of the query-constraint.

There are also two different kinds of filters. A *query-filter* is a part of the query-constraint, meaning that aggregations results are also affected by these constraints. A *filter* on the other hand, is not considered in the aggregations calculations, meaning that applying a filter will not impact the aggregation result.

7.1.4 Aggregations

An aggregation is a function, or something that is executed, on a collection of search results. The search-results are defined by the query and query-filter of the search request. See the *Aggregations* section for details.

7.2 Query Functions

Here's a description of all functions that can be used in a query.

7.2.1 fulltext

The fulltext function is searching for words in a field, and calculates relevance scores for matches based on a set of rules (e.g number of occurences, field-length).

Tip: Only fields analyzed as text are considered when applying the fulltext-function. This includes, as default, all text-based fields in the content-domain.

Syntax

```
fulltext(<fields>, <search-string>, <operator>)
```

Fields

Fields is a string containing a comma-separated list of fields to include in the search. Wildcards are supported in field-names.

You can boost - thus increasing or decreasing hit-score pr field basis - if providing more than one field to the query by appending a weight-factor: `^N`:

```
fulltext('title^5,description', 'my search string', 'AND')
```

Operator

The allowed operators are:

- `OR` Matches if any of the words in the search-string matches.
- `AND` Matches only if all words in search-string matches.

Search-string syntax

The search-string supports a set of operator:

- `+` signifies AND operation.
- `|` signifies OR operation.
- `-` negates a single token.
- `*` at the end of a term signifies a prefix query.
- `(` and `)` signify precedence.
- `~N` after a word signifies edit distance (fuzziness) with a number representing Levenshtein distance.
- `~N` after a phrase signifies slop amount.

Examples

Match if "myField" contains any of the given words.

```
    fulltext("myField", "cheese fish cake onion", "OR")
```

Match if any field with path starting with "myData.myProperties" contains any of the given words.

```
    fulltext("myData.myProperties.*", "cheese fish cake onion", "OR")
```

Match if "myField" contains any of the given words and "myCategory" = "soup".

```
    myCategory = "'soup" AND fulltext("myField", "cheese fish cake onion", "OR")
```

Match if "myField" contains all the given words.

```
    fulltext("myField", "cheese fish cake onion", "AND")
```

Match if "myField" contains "Levenshtein" with a fuzziness distance of 2.

```
    fulltext("myField", "Levenshtein~2", "AND")
```

Match if "myField" contains "fish" and not "boat".

```
    fulltext("myField", "fish -boat", "AND")
```

Match if any field under data-set data contains "fish" and not "boat".

```
    fulltext("data.*", "fish -boat", "AND")
```

7.2.2 nGram

An n-gram is a sequence of n letters from a string. The nGram-function are used to search for words or phrases beginning with a given search string. Typically, find-as-you-type searches will use this function.

Tip: Only fields analyzed as text are considered when applying the ngram-function. This includes, as default, all text-based fields in the content-domain.

Syntax

```
ngram(<field>, <search-string>, <operator>)
```

Operator

The allowed operators are:

- `OR` Matches if any of the words in the search-string matches.
- `AND` Matches only if all words in search-string matches.

Examples

Matches if "myField" contains any word beginning with "lev", e.g "Levenshteins Algorithm".

```
    ngram("myField", "lev", "AND")
```

Matches if "myField" contains words beginning with "lev" and "alg", e.g "Levenshteins Algorithm".

```
    ngram("myField", "lev alg", "AND")
```

Matches if "myField" contains words beginning with "fish" or "boat", e.g "fishpond" or "boatman".

```
    ngram("myField", "fish boat", "OR")
```

7.3 Order Functions

Here's a description of all functions that can be used in order-by clause.

7.3.1 geoDistance

The geoDistance-function enables you to order the results according to distance to a given geo-point.

Tip: Documents with no geo-point property with the given path will be ordered last if matching the query.

Syntax

```
geoDistance(<field>, <location>)
```

Field Field-argument accepts a path to a property containing geoPoint data.

Location The location is a geoPoint from which the distance factor should be calculated, formatted as "latitude,longitude".

Examples

Order by distance from "shopLocation" to the fixed location.

```
    ORDER BY geoDistance("shopLocation", "59.9127300,10.7460900")
```

7.4 Aggregations

An aggregation is a function, or something that is executed, on a collection of search results. The search-results are defined by the query and query-filter of the search request.

For instance, consider a query returning all nodes that has a property "price" less than, say, 100$. Now, we want to divide the result nodes into ranges, say 0-25$, 25-50$ and so on. We also would like to know the average price for each category. This could be done by doing multiple separate queries, and calculate the average manually, but this would be very ineffective and cumbersome. Luckily, aggregations solves these types of problems easily.

In some API functions it's possible to send in a aggregations expression object. This object is either in Java or a JSON like the following:

```
"aggregations" : {
  "[name]" : {
    "[type]" : {
      ... body ...
    },
    "aggregations": {
      ... sub-aggregations ...
    }
  }
}
```

There are two different types of aggregations:

- Bucket aggregations: A bucket aggregation places documents matching the query in a collection - a bucket. Each bucket has a key.
- Metrics aggeregations: A metric aggeregation computes metrics over a set of documents.

Typically, you will divide data into buckets, and then use metric aggregations to calculate e.g average values, sum, etc for each bucket if necessary.

7.4.1 terms

The 'terms' aggergation places documents into bucket based on property values. Each unique value of a property will get its own bucket. Here's a list of properties:

field (string) The property path.

size (int) The number of bucket to return, ordered by the given orderType and orderDirection. Default to `10`.

order (string) How to order the results, type and direction. Default to `_term ASC`.

Types:

- `_term`: Alphabetic ordering of bucket keys.

- `_count`: Numeric ordering of number of document in buckets.

Here's an example of the terms aggregation:

```
"aggregations": {
  "categories": {
    "terms": {
      "field": "myCategory",
      "order": "_count desc",
      "size": 10
    }
  }
}
```

The above example gives the following result:

```
"aggregations": {
  "categories": {
    "buckets": [{
      "doc_count": 132,
      "key": "articles"
    },
    {
      "doc_count": 101,
      "key": "documents"
    },
    {
      "doc_count": 43,
      "key": "case-studies"
    }]
  }
}
```

7.4.2 range

The range aggregation query defines a set of ranges that represents a bucket. Here's a list of properties:

field (string) The property path.

ranges (range[]) The range-buckets to create.

range (from: number, to: number) Defines a range to create a bucket for. From-value is included in bucket, to is excluded.

Here's an example of the range aggregation:

```
"price_ranges": {
  "range": {
    "field": "price",
    "ranges": [
      { "to": 50 },
      { "from": 50, "to": 100 },
      { "from": 100 }
    ]
  }
}
```

The above example gives the following result:

```
    "price_ranges": {
      "buckets": [{
        "doc_count": 2,
        "key": "a",
        "to": 50
      },
      {
        "doc_count": 4,
        "from": 50,
        "key": "b",
        "to": 100
      },
      {
        "doc_count": 4,
        "from": 100,
        "key": "c"
      }]
    }
```

7.4.3 date_range

The date_range aggregation query defines a set of date-ranges that represents a bucket. Only documents with properties of type 'DateTime' will considered in the `date_range` aggregation buckets. Here's a list of properties:

field (string) The property path.

format (string) The date-format of which the buckets will be formatted to on return. Default to `YYYY-MM-DDThh:mm:ssTZD`.

ranges (range[]) The range-buckets to create.

range (from: number, to: number) Defines a range to create a bucket for. From-value is included in bucket, to is excluded. The from and to follows a special date-math explained below.

Here's an example of the date_range aggregation:

```
    "my_date_range": {
      "date_range": {
        "field": "date",
        "format": "MM-yyy",
        "ranges": [{
          "to": "now-10M"
        },
        {
          "from": "now-10M"
        }]
      }
    }
```

The above example gives the following result:

```
    "price_ranges": {
      "buckets": [{
        "doc_count": 2,
        "key": "a",
        "to": 50
      },
      {
        "doc_count": 4,
```

```
        "from": 50,
        "key": "b",
        "to": 100
      },
      {
        "doc_count": 4,
        "from": 100,
        "key": "c"
      }]
    }
```

Date-math expression

The range fields accepts a date-math expression to calculate the time-spans.

Now minus a day:

```
now-1d
```

The given date minus 3 days plus one minute:

```
2014-12-10T10:00:00Z||-3h+1m
```

Range describing now plus one day and thirty minutes, rounded to minutes:

```
now+1d+30m/m
```

7.4.4 date_histogram

The date-histogram aggregation query defines a set of bucket based on a given time-unit. For instance, if querying a set of log-events, a `date_historgram` aggregations query with interval `h` (hour) will divide each log event into a bucket for each hour in the time-span of the matching events. Here's a list of properties:

field (string) The property path.

interval (string) The time-unit interval for creating bucket. Supported time-unit notations:

- `y` = Year
- `M` = Month
- `w` = Week
- `d` = Day
- `h` = Hour
- `m` = Minute
- `s` = Second

format (string) Output format of date string.

minDocCount (int) Only include bucket in result if number of hits <= `minDocCount`.

Here's an example of the date_histogram aggregation:

```
    "by_month": {
      "date_histogram": {
        "field": "init_date",
        "interval": "1M",
```

```
        "minDocCount": 0,
        "format": "MM-yyy"
      }
    }
```

The above example gives the following result:

```
    "by_month" : {
      "buckets" : [{
        "doc_count" : 8,
        "key" : "2014-01"
      }, {
        "doc_count" : 10,
        "key" : "2014-02"
      }, {
        "doc_count" : 12,
        "key" : "2014-03"
      }]
    }
```

7.4.5 stats

The stats-aggregations calculates the following statistics for the parent-aggregation buckets:

- avg
- min
- max
- count
- sum

Here's a list of properties:

field (string) The property path.

Here's an example of the stats aggregation:

```
    {
      "start": 0,
      "count": 0,
      "aggregations": {
        "products": {
          "terms": {
            "field": "data.product.category",
            "order": "_count desc",
            "size": 10
          },
          "aggregations": {
            "priceStats": {
              "stats": {
                "field": "data.product.price"
              }
            }
          }
        }
      }
    }
```

The above example gives the following result:

```
    "products": {
      "buckets": [{
        "key": "tv",
        "doc_count": 123,
        "priceStats": {
          "count": 123,
          "min": 2599,
          "max": 87944,
          "avg": 7400,
          "sum": 578100
        }
      },
      {
        "key": "blu-ray player",
        "doc_count": 42,
        "priceStats": {
          "count": 42,
          "min": 699,
          "max": 5999,
          "avg": 1548,
          "sum": 65016
        }
      },
      {
        "key": "reciever",
        "doc_count": 12,
        "priceStats": {
          "count": 12,
          "min": 2999,
          "max": 26950,
          "avg": 5548,
          "sum": 66756
        }
      }]
    }
```

7.5 Querying date and time

Querying against date and time-fields may require some knowledge on how data is stored and indexed.

7.5.1 LocalDate

LocalDate represents a date without time-zone in the ISO-8601 calendar, e.g `2015-03-19`. LocalDate-properties are stored as a ISO LocalDate-formatted string in the index, thus all searches are done against string-values.

LocalDate string-format:

```
yyyy-MM-dd
```

Given a node with a property named 'myLocalDate' of type `localDate` and value `2015-03-19`, all of the following queries will match:

```
myLocalDate = '2015-03-19'
myLocalDate > '2015-03-18'
myLocalDate <= '2015-03-19'
```

7.5.2 LocalTime

LocalTime represents a time without time-zone in the ISO-8601 calendar, e.g `11:39:49`. LocalTime-properties are stored as a ISO LocalTime-formatted string in the index, thus all searches are done against string-values.

LocalTime string-format:

```
HH:mm[:ss[.SSS]]
```

LocalTime string value examples:

```
09:30
10:00
10:00:30
10:00:30.142
```

Since the queries are matching string-values, the input time in query must either adhere the same string-format restrictions, or be wrapped in a function `time` which accepts a time-formatted string as input.

Given a node with a property named 'myLocalTime' of type `localTime` and value = `09:36:00`, all the following queries will match:

```
myLocalTime > '09:00'
myLocalTime = '09:36'
myLocalTime = '09:36:00'
myLocalTime LIKE '09:*'
myLocalTime < '09:36:01'
myLocalTime < '09:36:00.1'
```

This must be wrapped in time-function since its not padded with a leading 0:

```
myLocalTime > time('9:00')
```

If optional fractions of seconds are given, the string format will also contain this even if 0, and expression will not match unless wrapped in time-function:

```
myLocalTime = time('09:36:00.0')
```

Even if the string-matching will do the job 99% of the time, the safest bet is to always go with the time-function when applicable.

7.5.3 LocalDateTime

LocalDateTime represents a date-time without time-zone in the ISO-8601 calendar, e.g `2015-03-19T11:39:49`. LocalDateTime-properties are stored as a ISO LocalDateTime-formatted string in the index, thus all searches are done against string-values.

LocalDateTime string-format:

```
yyyy-MM-ddTHH:mm[:ss[.SSS]]
```

Since the queries are matching string-values, the input dateTime in query must either adhere the same string-format restrictions, or be wrapped in a function `dateTime` which accepts a dateTime-formatted string as input.

Given a node with a property named 'myLocalDateTime' of type `localDateTime` and value `2015-03-19T10:30:00`, all of the following queries will match:

```
myLocalDateTime = '2015-03-19T10:30:00'
myLocalDateTime = dateTime('2015-03-19T10:30')
myLocalDateTime < dateTime('2015-03-19T10:30:00.001')
```

7.5.4 DateTime / Instant

DateTime represents a date-time with time-zone in the ISO-8601 calendar, e.g `2015-03-19T11:39:49+02:00`. Its possible to query properties of with value-type *DateTime* both as an ISO instant and as ISO dateTime, using the provided built-in functions `instant` and `dateTime`.

Instant string-format (instant always given in UTC-time):

```
yyyy-MM-ddTHH:mm[:ss[.SSS]Z
```

Instant string value examples:

```
2015-03-19T16:30:20Z
2015-03-19T16:30:20.123Z
```

DateTime string-format (*Z* for UTC, else offset in hours and minutes):

```
yyyy-MM-ddTHH:mm[:ss[.SSS](Z|+hh:mm|-hh:mm)
```

DateTime string value examples:

```
2015-03-19T16:30:20Z
2015-03-19T16:30:20+01:00
2015-03-19T16:30:20-01:30
2015-03-19T16:30:20.123-01:30
```

Given a node with a property named 'myDateTime' of type `dateTime` and value `2015-03-19T10:25:00+02:00`, all of the following queries will match:

```
myDateTime = instant('2015-03-19T08:25:00Z')
myDateTime = dateTime('2015-03-19T08:25:00Z')
myDateTime = dateTime('2015-03-19T10:25:00+02:00')
myDateTime = dateTime('2015-03-19T11:25:00+03:00')
```

7.6 Querying paths

All nodes have 3 system-properties concerning the node placement in a branch, all of type `String`:

- `_name`: The node name without path.
- `_parentPath`: The parent node path.
- `_path`: The full path of the node.

Finds node with path `/content/mySite/myCategory/myContent`.

```
    _path = '/content/mySite/myCategory/myContent'
```

Finds all nodes with name `myContent` in a folder named `myCategory`, e.g `/content/test/thisIsMyCategory/myContent` and `/content/myCategory/myContent`.

```
_name = 'myContent' AND _parentPath LIKE '*myCategory'
```

Finds all nodes under the path `/content/mySite/myCategory` including children of children.

```
_path LIKE '/content/mySite/myCategory/*'
```

Finds only first level children under the path `/content/mySite/myCategory`.

```
_parentPath = '/content/mySite/myCategory'
```

CHAPTER 8

Folder Structure

The unpacked Enonic XP distribution will have the following structure:

```
enonic-xp-[version]
 |- bin/
 |- home/
   |- config/
   |- deploy/
   |- logs/
 |- lib/
 |- system/
 |- toolbox/
 |- repo/
 |- data/
```

Root installation folder is referred to as `XP_INSTALL`. Here's an explanation of all the other folders:

bin/ Contains the scripts for starting and stopping Enonic XP.

home/ Home directory, also called `XP_HOME`.

- **config/** Configuration files are placed here, including system.properties.
- **deploy/** Hot deploy directory. Place modules to install in this directory.
- **logs/** Default location for logs.

lib/ Contains the bootstrap code used to launch Enonic XP.

system/ System OSGi bundles are placed here.

toolbox/ Command-line interface tool to manage the server. See *Toolbox CLI*.

repo/ Repository data (blobs and indexes).

data/ Additional data like exports, snapshots and dumps.

CHAPTER 9

Configuration

Enonic XP, system modules and 3rd party modules can easily be configured using the files in `$XP_HOME/config/` directory.

When changing files ending with `.cfg`, their respective modules will automatically be restart with their new configuration. Files ending with `.properties` require a full restart of Enonic XP to be applied. In a clustered environment each node must be re-started.

9.1 System Configuration

The default `system.properties` are listed below.

```
#
# Installation settings
#
xp.name = demo

#
# Configuration FileMonitor properties
#
felix.fileinstall.poll = 1000
felix.fileinstall.noInitialDelay = true

#
# Remote shell configuration
#
osgi.shell.telnet.ip = 127.0.0.1
osgi.shell.telnet.port = 5555
osgi.shell.telnet.maxconn = 2
osgi.shell.telnet.socketTimeout = 0

#
# Initial http service properties
#
org.osgi.service.http.port = 8080
org.ops4j.pax.web.session.timeout = 1440
org.ops4j.pax.web.session.cookie = JSESSIONID
```

9.2 Virtual Host Configuration

Virtual hosts are configured in the file `com.enonic.xp.web.vhost.cfg` and is automatically updated on changes. A sample virtual host configuration is listed below.

```
enabled = true

mapping.a.host = localhost
mapping.a.source = /status/a
mapping.a.target = /full/path/status/a

mapping.b.host = enonic.com
mapping.b.source = /
mapping.b.target = /portal/master/enonic.com

mapping.admin.host = enonic.com
mapping.admin.source = /admin
mapping.admin.target = /admin
```

In this example file, three mappings are configured.

host Host-name to match.

source Requested path to match.

target Path to which the request is sent.

In the second example, mapping `b`, a site is mapped to the root of the URL, which would be normal in production environments.

In the third example, the admin site is mapped to `enonic.com/admin`.

CHAPTER 10

Docker

Docker allows you to package an application with all of its dependencies into a standardized unit for software development. We provide standard images for each version so that it's really easy to start Enonic XP for development or production.

To start using Docker you will have to set up a Docker host and install Docker client libraries. Please follow get started with Docker guide to set up everything you need.

After everything is installed, you can easily start up Enonic XP on your container host like this:

```
docker run -d -p 8080:8080 --name xp-app enonic/xp-app:5.3.0
```

This will download the Enonic XP image and start it up and map it to port `8080` on your docker-host. To look at the log, use `docker logs` command:

```
docker logs xp-app
```

Shutdown your server by executing the `docker stop` command:

```
docker stop xp-app
```

And, finally, kill the container if you do not need it anymore. This will destroy all data on the image as well:

```
docker kill xp-app
```

If you want to see more options for how to use the provided Enonic XP image, go to our project page at Docker Hub.

CHAPTER 11

Backup and Restore

Backing up your Enonic XP data is vital for any installation. There are several ways to secure your data.

- *Export/import:* see *Export and Import*.
- *Snapshot/restore:*: see below.
- *Backup ‘‘$XP_HOME/repo‘‘-folder*: Only for non-clustered environments.

All the data in an Enonic XP installation is stored in `$XP_HOME/repo`. This directory has two folders: `blob` and `index`. The `blob` folder contains all files needed by the system to manage your data, while the `index` folder contains the Elasticsearch index folders. These are dependent on each other in the sense that one is not much use without the other.

Caution: Technical details: When fetching data in Enonic XP, the index is scanned and a set of “blob-keys” are returned. These blob-keys refer to files in the *blob* folder where the actual contents are fetched. Each file in the *blob* folder is a binary blob or a serialized “node”, which is the low level structure of your data.

That leaves us with ensuring that two elements are safely stored for retrieval in an emergency:

- `$XP_HOME/repo/blobs`
- `$XP_HOME/repo/index`

11.1 Backing up indexes

Backing up the indices is a bit more complex than just copying the index-folder since it involves floating data with state, especially in a clustered environment. We have a number of rest-resources available at your disposal for snapshot operations:

http://<your-installation>/admin/rest/repo/snapshot Stores a snapshot of the current indices state.

http://<your-installation>/admin/rest/repo/list Returns a list of available snapshots for the installation.

http://<your-installation>/admin/rest/repo/restore Restore a snapshot of the indices state.

http://<your-installation>/admin/rest/repo/delete Deletes a single or a group of snapshots.

11.2 Snapshot

The snapshot rest-service accepts a JSON in this format:

```
{
  "repositoryId": "<repository-id>"
}
```

A snapshot of the given repository will be created for later retrieval. Each subsequent snapshot will store the changes between this snapshot and the last snapshot of the given repository. This means that only changed data are stored when doing subsequent snapshots. The snapshots will be stored in `$XP_HOME/data/snapshot`. A name of the snapshot will be given at snapshot-time, and returned in the snapshot-result.

To ease the process, we have provided a *snapshot* tool.

11.3 Restore

The restore rest-service accepts a JSON in this format:

```
{
  "snapshotName": "<snapshot-name>",
  "repository" : "<repository-id>"
}
```

The indices will be closed for the duration of a restore operation, meaning that no request will be accepted while the restore in running. To ease the process, we have provided a *restore* tool.

> **Warning:** Restoring a snapshot will restore data to the exact state of the indices at the snapshot-time, meaning all other changes will be lost.

11.4 Delete

The delete rest-service accepts a JSON in this format:

```
{
  "snapshotNames": ["name1", "name2"],
  "before" : "<timestamp>"
}
```

Deletes either all snapshots before timestamp, or given snapshots by name. To ease the process, we have provided a *deleteSnapshots* tool.

CHAPTER 12

Export and Import

Exporting and importing data in your Enonic XP installation is useful both for securing data and migrating between installations. Enonic XP ships with a set of *Toolbox CLI* to ease the operation of exporting and importing data from the system.

Caution: At the moment, exporting and importing data can only be done to and from files on the server running Enonic XP server.

12.1 Export

The export operation will extract data for a given content URL and store it as XML in a sub-folder under `$XP_HOME/data/export`. The REST service for export is found at the following URL:

```
http://<host>:<port>/admin/rest/export/export
```

The export REST service accepts a JSON in this format:

```
{
  "sourceRepoPath": "<source-repo-path>",
  "exportName": "<name>",
  "importWithIds": <true|false>,
  "dryRun": <true|false>
}
```

To ease the process, we have provided an *export* tool.

12.2 Import

The import will take data from a given export directory and load it into Enonic XP at the desired content path. The REST service for import is found at the following URL:

```
http://<host>:<port>/admin/rest/export/import
```

The import REST service accepts a JSON in this format:

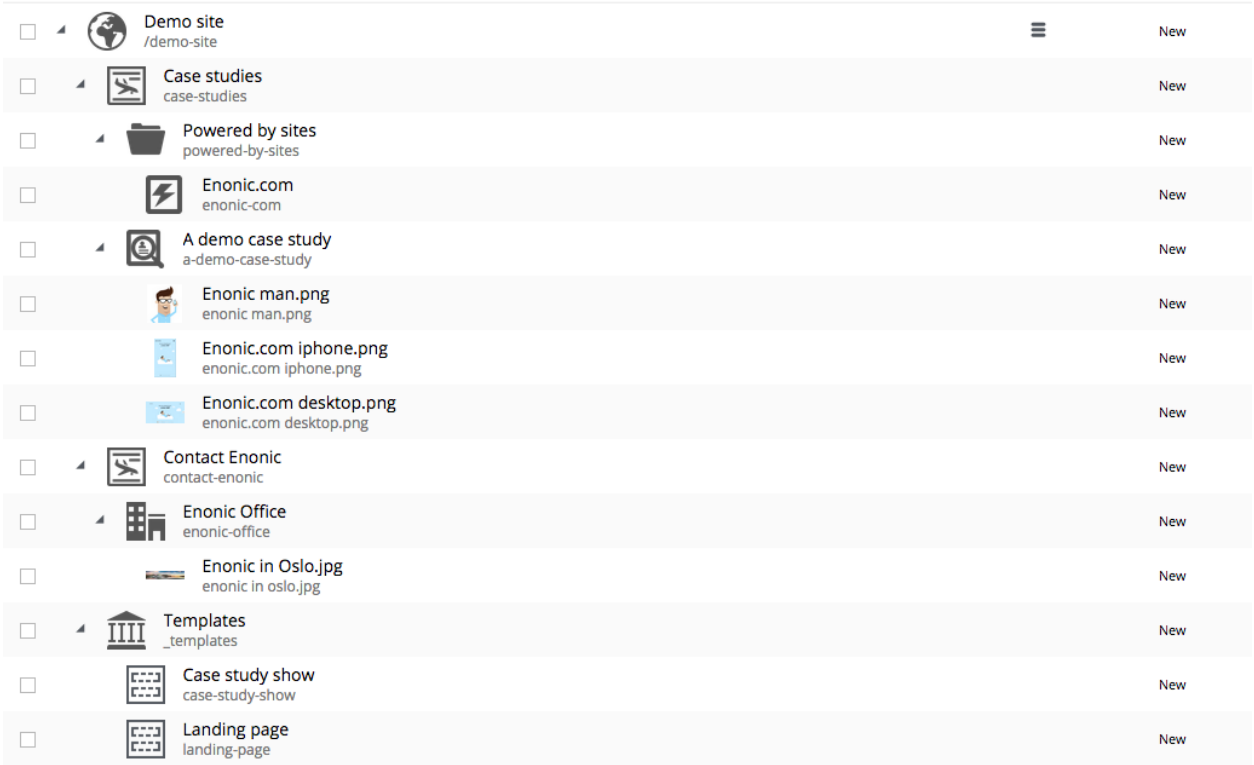
```
{
  "exportName": "<name>",
  "targetRepoPath": "<target-repo-path>",
  "importWithIds": <true|false>,
```

```
  "dryRun": <true|false>
}
```

To ease the process, we have provided an *import* tool.

12.3 Export data structure

Let's look at how this works. The following structure will be exported:



Run the export command:

```
$ ./export.sh -u su:password -s cms-repo:draft:/ -t myexport \
  -n -i false
```

Below is the resulting structure in the export folder `$XP_HOME/data/export/myexport`:

```
./content
./content/_
./content/_/node.xml
./content/demo-site
./content/demo-site/_
./content/demo-site/_/manualChildOrder.txt
./content/demo-site/_/node.xml
./content/demo-site/_templates
...
./content/demo-site/case-studies
./content/demo-site/case-studies/_
./content/demo-site/case-studies/_/node.xml
./content/demo-site/case-studies/a-demo-case-study
...
```

```
./content/demo-site/case-studies/a-demo-case-study/enonic man.png
./content/demo-site/case-studies/a-demo-case-study/enonic man.png/_
./content/demo-site/case-studies/a-demo-case-study/enonic man.png/_/bin
./content/demo-site/case-studies/a-demo-case-study/enonic man.png/_/bin/Enonic man.png
...
./content/demo-site/case-studies/powered-by-sites
...
./content/demo-site/contact-enonic
...
```

content The base folder of the export. All content in `cms-repo` has this as root path.

content/_ All folders named _ are system folders for the data at the current level.

content/_/node.xml The definition of the node, e.g. all data for the current node

content/demo-site This is the site from the screenshot above.

content/demo-site/_/manualChildOrder.txt Our demo-site has manually ordered children, this file contains an ordered list of children.

content/demo-site/case-studies This 'case-studies' content is the first element in the site.

content/demo-site/case-studies/a-demo-case-study/enonic man.png/_/bin The `A demo case study` content has a binary attachment called `Enonic man.png`. The folder `_/bin` contains the actual binary files.

12.4 Changing export data

It is possible to make manual changes to the exported data before importing.

Using the above export as an example, the `demo-site` displayName can be changed to something more suitable:

```
myExport $ vi content/demo-site/_/node.xml

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<node xmlns="urn:enonic:xp:export:1.0">
  <id>2dfbdc41-af98-4b3c-a2a9-9dc4814d003a</id>
  <childOrder>_manualordervalue DESC</childOrder>
  <nodeType>content</nodeType>
  <data>
    <boolean name="valid">true</boolean>
    <string name="displayName">My much nicer demo-site!</string>
    <string name="type">portal:site</string>
    <string name="owner">user:system:su</string>
```

After some data has been changed, it can be imported again:

```
$ ./import.sh -u su:password -s myExport -t cms-repo:draft:/
```

Caution: Editing exported data is experimental at the moment and will potentially cause trouble if not done carefully. For exports without ids, references will be broken and must be fixed manually. When importing *with* ids onto existing data, renaming and changing manual order will not yet work as expected.

CHAPTER 13

Server JavaScript

Our server-side javascript engine has some well-known global variables and functions. All the global variables and functions are documented in this section.

13.1 log

This object holds the logging methods. It's one method for each log level and takes the same number of parameters.

`log.`**`debug`**(*message*, *args*)

Arguments

- **`message`** (*string*) – Message to log as a debug-level message.
- **`args`** (*array*) – Optional arguments used in message format.

`log.`**`info`**(*message*, *args*)

Arguments

- **`message`** (*string*) – Message to log as a info-level message.
- **`args`** (*array*) – Optional arguments used in message format.

`log.`**`warning`**(*message*, *args*)

Arguments

- **`message`** (*string*) – Message to log as a warning-level message.
- **`args`** (*array*) – Optional arguments used in message format.

`log.`**`error`**(*message*, *args*)

Arguments

- **`message`** (*string*) – Message to log as a error-level message.
- **`args`** (*array*) – Optional arguments used in message format.

Examples:

```
// Log a simple message
log.debug('Hello World');

// Log a formatting message
log.debug('Hello %s', 'World');
```

```
// Log a formatting message
log.debug('%s %s', 'Hello', 'World');
```

13.2 resolve

This function resolves a fully qualified path to a local path based on your current location. It will never check if the path exists, just resolve it. This function supports both relative (with dot-references) and absolute paths.

resolve (*path*)

Arguments

- **path** (*string*) – Path to resolve using current location.

Returns The fully qualified resource path of the location.

Examples:

```
// Absolute path
var path1 = resolve('/cms/views/myview.html');

// Relative path
var path2 = resolve('myview.html');

// Relative path (same as above)
var path3 = resolve('./myview.html');

// Relative path
var path4 = resolve('../myview.html');
```

13.3 require

This function will load a javascript and return the exports as object. The function implements parts of the CommonJS Modules Specification.

require (*path*)

Arguments

- **path** (*string*) – Path to the javascript to load.

Returns The loaded javascript object exports.

Examples:

```
// Absolute path
var lib1 = require('/cms/lib/mylib.js');

// Relative path
var lib2 = require('mylib.js');

// Relative path (same as above)
var lib3 = resolve('./mylib.js');

// Relative path
var lib4 = resolve('../mylib.js');
```

13.4 execute

This function will execute a Java script command and return any result. It is used to allow the users to execute exposed Java script commands. See *Script Commands*.

execute (*name*, *params*)

Arguments

- **name** (*string*) – Name of command to execute.
- **object** (*params*) – Parameters passed down to the execute method.

Returns The result of the execute method.

Examples:

```
// Execute command
var result = execute('thymeleaf.render', {
  view: resolve('myview.html'),
  model: {
    friut: 'apple',
    stock: 10
  }
});
```

CHAPTER 14

Script Commands

Script commands is used for interfacing Java functionality from our server-side JavaScript. This document includes a complete reference of all standard script commands.

14.1 content.get

This command fetches a content.

Arguments:

key ***(string)*** Path or id to the content.

branch ***(string)*** Set by portal, depending on context, to either `draft` or `master`. May be overridden, but this is not recommended. Default is the current branch set in portal.

Example:

```
var result = execute('content.get', {
    key: '/my/content'
});

if (result) {
    log.info('Display Name = ' + result.displayName);
} else {
    log.info('Content was not found');
}
```

Result:

```
{
  "_createdTime": "1970-01-01T00:00:00Z",
  "_creator": "user:system:admin",
  "_id": "123456",
  "_modifiedTime": "1970-01-01T00:00:00Z",
  "_modifier": "user:system:admin",
  "_name": "mycontent",
  "_path": "/a/b/mycontent",
  "data": {
    "a": [1],
    "b": ["2"],
    "c": [
      {
        "d": [true]
```

```
      },
      {
        "d": [true],
        "e": [
          "3",
          "4",
          "5"
        ],
        "f": [2]
      }
    ]
  },
  "displayName": "My Content",
  "draft": false,
  "hasChildren": false,
  "x": {
    "mymodule": {
      "myschema": {
        "a": ["1"]
      }
    }
  },
  "page": {
    "config": {
      "a": ["1"]
    },
    "controller": "mymodule:mycontroller",
    "regions": [
      {
        "components": [
          {
            "config": {
              "a": ["1"]
            },
            "descriptor": "mymodule:mypart",
            "name": "mypart",
            "path": "top/0",
            "type": "part"
          },
          {
            "config": {
              "a": ["1"]
            },
            "descriptor": "mymodule:mylayout",
            "name": "mylayout",
            "path": "top/1",
            "regions": [
              {
                "components": [
                  {
                    "config": {
                      "a": ["1"]
                    },
                    "descriptor": "mymodule:mypart",
                    "name": "mypart",
                    "path": "top/1/bottom/0",
                    "type": "part"
                  }
```

```
                ],
                "name": "bottom"
              }
            ],
            "type": "layout"
          }
        ],
        "name": "top"
      }
    ]
  },
  "type": "system:unstructured"
}
```

14.2 content.getChildren

This command returns children of a content.

Arguments:

key *(string)* Path or id to the parent content.

start *(integer)* Start index (used for paging). Default is 0.

count *(integer)* Number of contents to fetch. Default is 10.

sort *(string)* Sorting expression.

branch *(string)* Set by portal, depending on context, to either "draft" or "master". May be overridden, but this is not recommended. Default is the current branch set in portal.

Example:

```
var result = execute('content.getChildren', {
    key: '/a/b/mycontent',
    start: 5,
    count: 3,
    sort: '_modifiedTime ASC'
});

log.info('Found ' + result.total + ' number of contents');

for (var i = 0; i < result.contents.length; i++) {
    var content = result.contents[i];
    log.info('Content ' + content._name + ' loaded');
}
```

Result:

```
{
  "contents": [
    {
      "_createdTime": "1970-01-01T00:00:00Z",
      "_creator": "user:system:admin",
      "_id": "111111",
      "_modifiedTime": "1970-01-01T00:00:00Z",
      "_modifier": "user:system:admin",
      "_name": "mycontent",
      "_path": "/a/b/mycontent",
```

```
      "data": {},
      "displayName": "My Content",
      "draft": false,
      "hasChildren": false,
      "x": {},
      "page": {},
      "type": "system:unstructured"
    },
    {
      "_createdTime": "1970-01-01T00:00:00Z",
      "_creator": "user:system:admin",
      "_id": "222222",
      "_modifiedTime": "1970-01-01T00:00:00Z",
      "_modifier": "user:system:admin",
      "_name": "othercontent",
      "_path": "/a/b/othercontent",
      "data": {},
      "displayName": "Other Content",
      "draft": false,
      "hasChildren": false,
      "x": {},
      "page": {},
      "type": "system:unstructured"
    }
  ],
  "total": 2
}
```

14.3 content.query

This command queries content. See *Query Language* for details.

Arguments:

start ***(integer)*** Start index (used for paging). Default is 0.

count ***(integer)*** Number of contents to fetch. Default is 10.

query ***(string)*** Query expression.

sort ***(string)*** Sorting expression.

aggregations ***(object)*** Aggregations expression.

contentTypes ***(string[])*** Content types to filter on.

branch ***(string)*** Set by portal, depending on context, to either "draft" or "master". May be overridden, but this is not recommended. Default is the current branch set in portal.

Example:

```
var result = execute('content.query', {
    start: 0,
    count: 100,
    query: "type = 'article' AND fulltext('myField', 'searching for cheese', 'AND')",
    sort: "modifiedTime DESC, geodistance('p1', 'p2')",
    contentTypes: [
        "mymodule:article",
        "mymodule:comment"
```

```
    ],
    aggregations: {
        genders: {
            terms: {
                field: "gender",
                order: "_count asc",
                size: 2
            }
        },
        by_month: {
            date_histogram: {
                field: "init_date",
                interval: "1m",
                minDocCount: 0
            }
        }
    }
});

log.info('Found ' + result.total + ' number of contents');

for (var i = 0; i < result.contents.length; i++) {
    var content = result.contents[i];
    log.info('Content ' + content._name + ' loaded');
}
```

Result:


```
{
  "aggregations": {
    "by_month": {
      "buckets": [
        {
          "doc_count": 8,
          "key": "2014-01"
        },
        {
          "doc_count": 10,
          "key": "2014-02"
        },
        {
          "doc_count": 12,
          "key": "2014-03"
        }
      ]
    },
    "genders": {
      "buckets": [
        {
          "doc_count": 10,
          "key": "male"
        },
        {
          "doc_count": 12,
          "key": "female"
        }
      ]
    }
  },
```

```
  "contents": [
    {
      "_createdTime": "1970-01-01T00:00:00Z",
      "_creator": "user:system:admin",
      "_id": "111111",
      "_modifiedTime": "1970-01-01T00:00:00Z",
      "_modifier": "user:system:admin",
      "_name": "mycontent",
      "_path": "/a/b/mycontent",
      "data": {},
      "displayName": "My Content",
      "draft": false,
      "hasChildren": false,
      "x": {},
      "page": {},
      "type": "system:unstructured"
    },
    {
      "_createdTime": "1970-01-01T00:00:00Z",
      "_creator": "user:system:admin",
      "_id": "222222",
      "_modifiedTime": "1970-01-01T00:00:00Z",
      "_modifier": "user:system:admin",
      "_name": "othercontent",
      "_path": "/a/b/othercontent",
      "data": {},
      "displayName": "Other Content",
      "draft": false,
      "hasChildren": false,
      "x": {},
      "page": {},
      "type": "system:unstructured"
    }
  ],
  "total": 2
}
```


14.4 content.delete

This command deletes a content.

Arguments:

key ***(string)*** Path or id to the content.

branch ***(string)*** Set by portal, depending on context, to either `draft` or `master`. May be overridden, but this is not recommended. Default is the current branch set in portal.

Example:

```
var result = execute('content.delete', {
    key: '/my/content'
});

if (result) {
    log.info('Content deleted');
} else {
```

```
    log.info('Content was not found');
}
```

14.5 content.create

This command creates a content.

name *(string)* Name of content.

parentPath *(string)* Path to place content under. Default is '/'.

displayName *(string)* Display name. Default is same as <name>.

requireValid *(boolean)* The content has to be valid to be created. Default is *(true)*.

contentType *(string)* Content type to use.

data *(object)* Actual content data.

x *(object)* eXtra data to use.

branch *(string)* Set by portal, depending on context, to either "draft" or "master". May be overridden, but this is not recommended. Default is the current branch set in portal.

Example:

```
var result = execute('content.create', {
    name: 'mycontent',
    parentPath: '/a/b',
    displayName: 'My Content',
    draft: true,
    contentType: 'system:unstructured',
    data: {
        a: 1,
        b: 2,
        c: ['1', '2'],
        d: {
            e: {
                f: 3.6,
                g: true
            }
        }
    },
    x: {
        test: {
            a: 1
        }
    }
});

log.info('Content created with id ' + result._id);
```

Result:

```
{
  "_id": "123456",
  "_name": "mycontent",
  "_path": "/a/b/mycontent",
  "data": {
```

```
    "a": [1],
    "b": [2],
    "c": [
      "1",
      "2"
    ],
    "d": [
      {
        "e": [
          {
            "f": [3.6],
            "g": [true]
          }
        ]
      }
    ]
  },
  "displayName": "My Content",
  "draft": true,
  "hasChildren": false,
  "x": {
    "mymodule:": {
      "test:": {
        "a": [1]
      }
    }
  },
  "page": {},
  "type": "system:unstructured"
}
```

14.6 content.modify

This command modifies a content.

key ***(string)*** Path or id to the content.

editor ***(function)*** Editor callback function.

branch ***(string)*** Set by portal, depending on context, to either "draft" or "master". May be overridden, but this is not recommended. Default is the current branch set in portal.

Example:

```
function editor(c) {
    c.displayName = 'Modified';
    c.data.a++;
    c.data.z = '99';

    c.x['other'] = {
        name: 'test'
    };

    return c;
}

var result = execute('content.modify', {
```

```
    key: '/my/content',
    editor: editor
});

if (result) {
    log.info('Content modified. New title is ' + result.displayName);
} else {
    log.info('Content not found');
}
```

Result:


```
{
  "_createdTime": "1970-01-01T00:00:00Z",
  "_creator": "user:system:admin",
  "_id": "123456",
  "_modifiedTime": "1970-01-01T00:00:00Z",
  "_modifier": "user:system:admin",
  "_name": "mycontent",
  "_path": "/a/b/mycontent",
  "data": {
    "a": [2.0],
    "b": ["2"],
    "c": [
      {
        "d": ["true"]
      },
      {
        "d": ["true"],
        "e": [
          "3",
          "4",
          "5"
        ],
        "f": ["2"]
      }
    ],
    "z": ["99"]
  },
  "displayName": "Modified",
  "draft": false,
  "hasChildren": false,
  "x": {
    "mymodule": {
      "myschema": {
        "a": ["1"]
      },
      "other": {
        "name": ["test"]
      }
    }
  },
  "page": {},
  "type": "system:unstructured"
}
```

14.7 portal.getContent

This command returns the content corresponding to the current execution context. It is meant to be called from a page, layout or part controller.

Example:

```
var result = execute('portal.getContent');

log.info('Current content path = ' + result._path);
```

Result:


```
{
  "_createdTime": "1970-01-01T00:00:00Z",
  "_creator": "user:system:admin",
  "_id": "123456",
  "_modifiedTime": "1970-01-01T00:00:00Z",
  "_modifier": "user:system:admin",
  "_name": "mycontent",
  "_path": "/a/b/mycontent",
  "data": {
    "a": [1],
    "b": ["2"],
    "c": [{
      "d": [true]
    }, {
      "d": [true],
      "e": ["3", "4", "5"],
      "f": [2]
    }]
  },
  "displayName": "My Content",
  "draft": false,
  "hasChildren": false,
  "x": {
    "mymodule": {
      "myschema": {
        "a": ["1"]
      }
    }
  },
  "page": {
    "config": {
      "a": ["1"]
    },
    "controller": "mymodule:mycontroller",
    "regions": {
      "top": {
        "components": [{
          "config": {
            "a": ["1"]
          },
          "descriptor": "mymodule:mypart",
          "name": "mypart",
          "path": "top/0",
          "type": "part"
        }, {
          "config": {
```

```
            "a": ["1"]
          },
          "descriptor": "mymodule:mylayout",
          "name": "mylayout",
          "path": "top/1",
          "regions": {
            "bottom": {
              "components": [{
                "config": {
                  "a": ["1"]
                },
                "descriptor": "mymodule:mypart",
                "name": "mypart",
                "path": "top/1/bottom/0",
                "type": "part"
              }]
            }
          },
          "type": "layout"
        }]
      }
    }
  },
  "type": "system:unstructured"
}
```

14.8 portal.getComponent

This command returns the component corresponding to the current execution context. It is meant to be called from a layout or part controller.

Example:

```
var result = execute('portal.getComponent');

log.info('Current component name = ' + result.name);
```

Result:

```
{
  "config": {
    "a": ["1"]
  },
  "descriptor": "mymodule:mylayout",
  "name": "mylayout",
  "path": "main/-1",
  "regions": {
    "bottom": {
      "components": [
        {
          "config": {
            "a": ["1"]
          },
          "descriptor": "mymodule:mypart",
          "name": "mypart",
          "path": "main/-1/bottom/0",
          "type": "part"
```

```
        }
      ]
    }
  },
  "type": "layout"
}
```

14.9 portal.getSite

This command returns the parent site of the content corresponding to the current execution context. It is meant to be called from a page, layout or part controller.

Example:

```
var result = execute('portal.getSite');

log.info('Current site module config = ' + result.moduleConfigs['myModule']);
```

Result:

```
{
  "_id": "100123",
  "_name": "my-content",
  "_path": "/my-content",
  "data": {
    "moduleConfig": [
      {
        "config": [
          {
            "Field": [42]
          }
        ],
        "moduleKey": ["mymodule"]
      }
    ]
  },
  "draft": false,
  "hasChildren": false,
  "x": {},
  "moduleConfigs": {
    "mymodule": {
      "Field": [42]
    }
  },
  "page": {},
  "type": "system:unstructured"
}
```

14.10 portal.assetUrl

This command generates a URL pointing to a static file.

Arguments:

path ***(string)*** Path to the asset.

module *(string)* Other module to reference to. Defaults to current module.

params *(object)* Custom parameters to append to the url.

Example use in controller:

```
var url = execute('portal.assetUrl', {
  path: 'styles/main.css'
});
```

14.11 portal.imageUrl

This command generates a URL pointing to an image.

Arguments:

path *(string)* Path to the asset.

module *(string)* Other module to reference to. Default is current module.

params *(object)* Custom parameters to append to the url.

Example use in controller:

```
var url = execute('portal.imageUrl', {
  id: '1234',
  filter: 'scale(1,1)'
});
```

14.12 portal.componentUrl

This command generates a URL pointing to a component.

Arguments:

id *(string)* Id to the page.

path *(string)* Path to the page.

component *(string)* Path to the component. If not set, the current path is set.

params *(object)* Custom parameters to append to the url.

Example:

```
var url = execute('portal.componentUrl', {
  component: 'main/0'
});
```

14.13 portal.attachmentUrl

This command generates a URL pointing to an attachment.

Arguments:

id *(string)* Id to the content holding the attachment.

path*(string)* Path to the content holding the attachment.

name *(string)* Name to the attachment.

label *(string)* Label of the attachment. Default is `source`.

download *(boolean)* Set to true if the disposition header should be set to attachment. Default is `false`.

params *(object)* Custom parameters to append to the url.

Example:

```
var url = execute('portal.attachmentUrl', {
  download: true
});
```

14.14 portal.pageUrl

This command generates a URL pointing to a page.

Arguments:

id *(string)* Id to the page. If id is set, then path is not used.

path *(string)* Path to the page. Relative paths is resolved using the context page.

params *(object)* Custom parameters to append to the url.

Example:

```
var url = execute('portal.pageUrl', {
  path: '/my/page',
  params: {
    a: 1,
    b: [1, 2]
  }
});
```

14.15 portal.serviceUrl

This command generates a URL pointing to a service.

Arguments:

service *(string)* Name of the service.

module *(string)* Other module to reference to. Default is current module.

params *(object)* Custom parameters to append to the url.

Example:

```
var url = execute('portal.serviceUrl', {
  service: 'myservice',
  params: {
    a: 1,
    b: 2
  }
});
```

14.16 portal.imagePlaceholder

This command generates a URL to an image placeholder.

Arguments:

width *(string)* Desired width of the placeholder.

height *(string)* Desired height of the placeholder.

params *(object)* Custom parameters to append to the url.

Example:

```
var url = execute('portal.imagePlaceholder', {
  width: '300',
  height: '200'
});
```

14.17 portal.processHtml

This command replaces abstract internal links contained in an HTML text by generated URLs.

Argument:

value *(string)* Html value string to process.

Example use in controller:

```
var processedHtml = execute('portal.processHtml', {
  value: '<a href="content://123" target="">Content</a>' +
         '<a href="media://inline/123" target="">Inline</a>' +
         '<a href="media://download/123" target="">Download</a>'
});
```

Result:

```
<a href="/admin/portal/preview/draft/features/content" target="">Content</a>
<a href="/admin/portal/preview/draft/features/content/_/attachment/inline/123/image.jpg" target="">In
<a href="/admin/portal/preview/draft/features/content/_/attachment/download/123/image.jpg" target="">
```

14.18 i18n.localize

This command localizes a phrase.

key *(string)* The property key.

locale *(string)* A string-representation of a locale. If the locale is not set, the site language is used.

values *(string[])* Optional placeholder values.

Example:

```
var message_multi_placeholder = execute('i18n.localize', {
  key: 'menu',
});

var message_multi_placeholder = execute('i18n.localize', {
```

```
  key: 'greetings',
  locale: "no",
  values: ["John", "London"]
});
```

14.19 thymeleaf.render

This command renders a view using thymeleaf.

Arguments:

view ***(string)*** Location of the view. Use `resolve(..)` to resolve a view.

model ***(object)*** Model that is passed to the view.

Example:

```
var view = resolve('view/fruit.html');

var result = execute('thymeleaf.render', {
    view: view,
    model: {
        fruits: [
            {
                name: 'Apple',
                color: 'Red'
            },
            {
                name: 'Pear',
                color: 'Green'
            }
        ]
    }
});
```

14.20 xslt.render

This command renders a view using XSLT.

Arguments:

view ***(string)*** Location of the view. Use `resolve(..)` to resolve a view.

model ***(object)*** Model that is passed to the view. This model is converted to XML.

Example:

```
var view = resolve('view/fruit.xsl');

var result = execute('xslt.render', {
    view: view,
    model: {
        fruits: [
            {
                name: 'Apple',
                color: 'Red'
            },
```

```
            {
                name: 'Pear',
                color: 'Green'
            }
        ]
    }
});
```

14.21 mustache.render

This command renders a view using mustache.

Arguments:

view ***(string)*** Location of the view. Use `resolve(..)` to resolve a view.

model ***(object)*** Model that is passed to the view.

Example:

```
var view = resolve('view/fruit.html');

var result = execute('mustache.render', {
    view: view,
    model: {
        fruits: [
            {
                name: 'Apple',
                color: 'Red'
            },
            {
                name: 'Pear',
                color: 'Green'
            }
        ]
    }
});
```

CHAPTER 15

View Functions

Both Thymeleaf and XSLT supports a set of view functions. All the functions are described here with examples. All view functions described here has the same parameters described in the associated *Script Commands* but transformed into an array instead with the following conventions:

1. All parameters is not prefixed by anything. `params.a` is just *a*.
2. Every other parameter is prefixed by _. `key` will be `_key`.

15.1 assetUrl

This generates a URL pointing to a static file. See parameters for *portal.assetUrl*.

Usage in Thymeleaf:

```
<a data-th-href="${portal.assetUrl({'_path=css/main.css'})}">Link</a>
```

Usage in XSLT:

```
<xsl:stylesheet version="2.0" exclude-result-prefixes="#all"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:portal="urn:enonic:xp:portal:1.0">

  <xsl:template match="/">
    <xsl:value-of select="portal:assetUrl('_path=a')"/>
  </xsl:template>

</xsl:stylesheet>
```

15.2 attachmentUrl

This generates a URL pointing to an attachment. See parameters for *portal.attachmentUrl*.

Usage in Thymeleaf:

```
<a data-th-href="${portal.attachmentUrl({'_download=true'})}">Link</a>
```

Usage in XSLT:

```
<xsl:stylesheet version="2.0" exclude-result-prefixes="#all"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:portal="urn:enonic:xp:portal:1.0">

  <xsl:template match="/">
    <xsl:value-of select="portal:attachmentUrl('_download=true')"/>
  </xsl:template>

</xsl:stylesheet>
```

15.3 componentUrl

This generates a URL pointing to a component. See parameters for *portal.componentUrl*.

Usage in Thymeleaf:

```
<a data-th-href="${portal.componentUrl({'_component=main/1'})}">Link</a>
```

Usage in XSLT:

```
<xsl:stylesheet version="2.0" exclude-result-prefixes="#all"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:portal="urn:enonic:xp:portal:1.0">

  <xsl:template match="/">
    <xsl:value-of select="portal:componentUrl('_component=main/1')"/>
  </xsl:template>

</xsl:stylesheet>
```

15.4 imageUrl

This generates a URL pointing to an image. See parameters for *portal.imageUrl*.

Usage in Thymeleaf:

```
<img data-th-src="${portal.imageUrl({'_id=11'})}"/>
<img data-th-src="${portal.imageUrl({'_name=test'})}"/>
```

Usage in XSLT:

```
<xsl:stylesheet version="2.0" exclude-result-prefixes="#all"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:portal="urn:enonic:xp:portal:1.0">

  <xsl:template match="/">
    <xsl:value-of select="portal:imageUrl('_id=11')"/>
    <xsl:value-of select="portal:imageUrl('_name=test')"/>
  </xsl:template>

</xsl:stylesheet>
```

15.5 pageUrl

This generates a URL pointing to a page. See parameters for *portal.pageUrl*.

Usage in Thymeleaf:

```
<a data-th-href="${portal.pageUrl({'_path=/my/page', 'a=3'})}">Link</a>
```

Usage in XSLT:

```
<xsl:stylesheet version="2.0" exclude-result-prefixes="#all"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:portal="urn:enonic:xp:portal:1.0">

  <xsl:template match="/">
    <xsl:value-of select="portal:pageUrl('_path=/my/page', 'a=3')"/>
  </xsl:template>

</xsl:stylesheet>
```

15.6 serviceUrl

This generates a URL pointing to a service. See parameters for *portal.serviceUrl*.

Usage in Thymeleaf:

```
<a data-th-href="${portal.serviceUrl({'_service=myservice', 'a=3'})}">Link</a>
```

Usage in XSLT:

```
<xsl:stylesheet version="2.0" exclude-result-prefixes="#all"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:portal="urn:enonic:xp:portal:1.0">

  <xsl:template match="/">
    <xsl:value-of select="portal:serviceUrl('_service=myservice', 'a=3')"/>
  </xsl:template>

</xsl:stylesheet>
```

15.7 imagePlaceholder

This command generates a URL to an image placeholder. See parameters for *portal.imagePlaceholder*.

Usage in Thymeleaf:

```
<img data-th-src="${portal.imagePlaceholder({'width=10','height=10'})}"/>
```

Usage in XSLT:

```
<xsl:stylesheet version="2.0" exclude-result-prefixes="#all"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:portal="urn:enonic:xp:portal:1.0">

  <xsl:template match="/">
```

```
    <xsl:value-of select="portal:imagePlaceholder('width=10','height=10')"/>
  </xsl:template>

</xsl:stylesheet>
```

15.8 localize

This localizes a phrase. See parameters for *i18n.localize*.

Usage in Thymeleaf:

```
<div data-th-text="${portal.assetUrl({'_key=mystring'})}">Not translated</div>
```

Usage in XSLT:

```
<xsl:stylesheet version="2.0" exclude-result-prefixes="#all"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:portal="urn:enonic:xp:portal:1.0">

  <xsl:template match="/">
    <xsl:value-of select="portal:localize('_key=mystring')"/>
  </xsl:template>

</xsl:stylesheet>
```

15.9 processHtml

This function replaces abstract internal links contained in an HTML text by generated URLs.. See parameters for *portal.processHtml*.

Usage in Thymeleaf:

```
<div data-th-text="${portal.processHtml({'_value=some text'})}">Text</a>
```

Usage in XSLT:

```
<xsl:stylesheet version="2.0" exclude-result-prefixes="#all"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:portal="urn:enonic:xp:portal:1.0">

  <xsl:template match="/">
    <xsl:value-of select="portal:processHtml('_value=some text')"/>
  </xsl:template>

</xsl:stylesheet>
```

CHAPTER 16

Query Language

When finding nodes and content you will be using our query language. It is based on SQL and looks very similar.

16.1 queryExr

Grammar:

```
queryExpr = [ constraintExpr ] [ orderExpr ] ;
```

- If no constraint-expression is given, all documents will match.
- If no order-expression is given, results will be ordered by *timestamp* descending.

Examples:

```
myCategory = 'article'
myCategory = 'article' ORDER BY title DESC
ORDER BY title
```

16.2 constraintExpr

Grammar:

```
constraintExpr = compareExpr
               | logicalExpr
               | dynamicConstraint
               | notExpr ;
```

16.3 compareExpr

Grammar:

```
compareExpr   = fieldExpr operator valueExpr ;
fieldExpr     = propertyPath ;
operator      = '=', '!=', '>', '>', '<', '<=', 'LIKE', 'NOT LIKE', 'IN', 'NOT IN' ;
valueExpr     = string | number | valueFunc ;
valueFunc     = geoPoint | instant | time | dateTime, localDateTime ;
geoPoint      =  '"' lat ',' lon '"' ;
```

```
instant       = 'instant(' string ')' ;
time          = 'time(' string ')' ;
dateTime      = 'dateTime(' string ')' ;
localDateTime = 'localDateTime(' string ')' ;
```

Examples:

```
user.myCategory = "articles"
user.myCategory IN ("articles", "documents")
user.myCategory != "articles"
user.myCategory LIKE "*tic*"
myPriority < 10
myPriority <= 10
myPriority > 10
myPriority < 100
myPriority != 10
myInstant = instant('2014-02-26T14:52:30.00Z')
myInstant <= instant('2014-02-26T14:52:30.00Z')
myInstant <= dateTime('2014-02-26T14:52:30.00+02:00')
myTime = time('09:00')
myLocalDateTime = time('2014-02-26T14:52:30.00')
myLocation = '59.9127300,10.7460900'
myLocation IN ('59.9127300,10.7460900','59.2181000,10.9298000')
```

16.4 logicalExpr

Grammar:

```
logicalExpr = constraintExpr operator constraintExpr ;
operator    = 'AND' | 'OR' ;
```

Examples:

```
myCategory = "articles" AND myPriority > 10
myCategory IN ("articles", "documents") OR myPriority <= 10
```

16.5 dynamicConstraint

Grammar:

```
dynamicConstraint = functionExpr ;
```

Examples:

```
fulltext('myCategory', 'Searching for fish', 'AND')
ngram('description', 'fish boat', 'AND')
```

16.6 notExpr

Grammar:

```
notExpr = 'NOT' constraintExpr ;
```

Examples:

```
NOT myCategory = 'article'
```

16.7 orderExpr

Grammar:

```
orderExpr = 'ORDER BY' ( fieldOrderExpr | dynamicOrderExpr )
            ( ',' ( fieldOrderExpr | dynamicOrderExpr ) )* ;
```

16.8 fieldOrderExpr

Grammar:

```
fieldOrderExpr = propertyPath [ direction ] ;
direction>     = 'ASC' | 'DESC' ;
```

Examples:

```
_name ASC
_timestamp DESC
title DESC
data.myProperty
```

16.9 dynamicOrderExpr

Grammar:

```
dynamicOrderExpr = functionExpr [ direction ] ;
direction        = 'ASC' | 'DESC' ;
```

Examples:

```
geoDistance('59.9127300,10.746090')
```

16.10 propertyPath

Grammar:

```
propertyPath = pathElement ( '.' pathElement )* ;
pathElement  = ( [ validJavaIdentifier - '.' ] )* ;
```

Examples:

```
myProperty
data.myProperty
data.myCategory.myProperty
```

Tip: Wildcards in propertyPaths are supported in functions `fulltext` and `ngram` only at the moment. When using these functions, expressions like this are valid:

```
myProp*
*Property
data.*
*.myProperty
data.*.myProperty
```

16.11 functionExpr

Grammar:

```
functionExpr = functionName '(' arguments ')' ;
```

16.12 Examples

Find all documents where property 'myCategory' is populated with a value, and the value does not equals 'article'.

```
myCategory LIKE '*' AND NOT myCategory = 'article'
```

Find all document where property 'myCategory' is either 'article' or 'document' and title starts with 'fish'.

```
myCategory IN ('article', 'document') AND ngram('title', 'fish', 'AND')
```

Find all documents where any fulltext-analyzed property contains 'fish' and 'spot', and order them ascending by distance from Oslo.

```
fulltext('_allText', 'fish spot', 'AND') ORDER BY
geoDistance('location', '59.9127300,10.7460900') ASC
```

Find all documents where any property under data-set 'data' contains 'fish' and 'spot', and order them ascending by distance from Oslo.

```
fulltext('data.*', 'fish spot', 'AND') ORDER BY
geoDistance('location', '59.9127300,10.7460900') ASC
```

CHAPTER 17

Toolbox CLI

The toolbox is a CLI (command line interface) tool that is used to do administration tasks. Toolbox executables are located in `$XP_INSTALL/toolbox` folder. Use `toolbox.sh` for mac/unix environments and `toolbox.bat` for windows environments.

To get help for the commands, just type the following:

```
$ toolbox.sh

usage: toolbox <command> [<args>]

The most commonly used toolbox commands are:
    delete-snapshots   Deletes snapshots, either before a given timestamp or by name.
    dump               Export every branch in specified repository.
    export             Export node from a branch in a repository.
    help               Display help information
    import             Import nodes from an export into a repository branch.
    list-snapshots     Returns a list of existing snapshots with name and status.
    reindex            Reindex content in search indices for the given repository and branches.
    restore            Restores a snapshot of a previous state of the repository.
    snapshot           Stores a snapshot of the current state of the repository.

See 'toolbox help <command>' for more information on a specific command.
```

To get help for a specific command, you can type `toolbox.sh help <command>`, like:

```
$ toolbox.sh help import
```

Here's a list of all the commands that you can do with the toolbox:

17.1 snapshot

Create a snapshot of a single repository while running. The snapshots will be stored in the `$XP_HOME/data/snapshot` directory.

Usage:

```
NAME
        toolbox snapshot - Stores a snapshot of the current state of the
        repository.

SYNOPSIS
        toolbox snapshot -a <auth> [-h <host>] [-p <port>] -r <repository>
```

```
OPTIONS
        -a <auth>
            Authentication token for basic authentication (user:password).

        -h <host>
            Host name for server (default is localhost).

        -p <port>
            Port number for server (default is 8080).

        -r <repository>
            the name of the repository to snapshot.
```

Example:

```
$ ./toolbox.sh snapshot -a su:password -r cms-repo
```

17.2 restore

Restore a named snapshot. The snapshots are located in the `$XP_HOME/data/snapshot` directory.

Usage:

```
NAME
        toolbox restore - Restores a snapshot of a previous state of the
        repository.

SYNOPSIS
        toolbox restore -a <auth> [-h <host>] [-p <port>] -r <repository>
                -s <snapshotName>

OPTIONS
        -a <auth>
            Authentication token for basic authentication (user:password).

        -h <host>
            Host name for server (default is localhost).

        -p <port>
            Port number for server (default is 8080).

        -r <repository>
            The name of the repository to restore.

        -s <snapshotName>
            The name of the snapshot to restore.
```

Example:

```
$ ./toolbox restore -a su:password -r cms-repo \
  -s cms-repo2015-07-02t11:53:13.224z
```

17.3 list-snapshots

List all the snapshots for the installation.

Usage:

```
NAME
        toolbox list-snapshots - Returns a list of existing snapshots with name
        and status.

SYNOPSIS
        toolbox list-snapshots -a <auth> [-h <host>] [-p <port>]

OPTIONS
        -a <auth>
            Authentication token for basic authentication (user:password).

        -h <host>
            Host name for server (default is localhost).

        -p <port>
            Port number for server (default is 8080).
```

Example:

```
$ ./toolbox.sh list-snapshots -a su:password
```

17.4 deleteSnapshots

Deletes all snapshots before the given timestamp.

Usage:

```
NAME
        toolbox delete-snapshots - Deletes snapshots, either before a given
        timestamp or by name.

SYNOPSIS
        toolbox delete-snapshots -a <auth> -b <before> [-h <host>] [-p <port>]

OPTIONS
        -a <auth>
            Authentication token for basic authentication (user:password).

        -b <before>
            Delete snapshots before this timestamp.

        -h <host>
            Host name for server (default is localhost).

        -p <port>
            Port number for server (default is 8080).
```

Example:

```
$ ./toolbox.sh delete-snapshots -a su:password -b 2015-02-14t14:24:20.618z
```

17.5 export

Extract data for a given repo and content path. The result will be stored in the `$XP_HOME/data/export` directory.

Usage:

```
NAME
        toolbox export - Export node from a branch in a repository.

SYNOPSIS
        toolbox export -a <auth> [-h <host>] [-p <port>] -s <sourceRepoPath>
                [--skipids] -t <exportName>

OPTIONS
        -a <auth>
            Authentication token for basic authentication (user:password).

        -h <host>
            Host name for server (default is localhost).

        -p <port>
            Port number for server (default is 8080).

        -s <sourceRepoPath>
            Path of data to export. Format:
            <repo-name>:<branch-name>:<node-path>.

        --skipids
            Flag that skips ids in data when exporting.

        -t <exportName>
            Target name to save export.
```

Example:

```
$ ./toolbox.sh export -a su:password -s cms-repo:draft:/ -t myExport
```

17.6 import

Import will take data from a named export and load it into Enonic XP at the desired content path.

Usage:

```
OPTIONS
        -a <auth>
            Authentication token for basic authentication (user:password).

        -h <host>
            Host name for server (default is localhost).

        -p <port>
            Port number for server (default is 8080).

        -s <exportName>
            A named export to import.

        --skipids
```

```
            Flag that skips ids.

        -t <targetRepoPath>
            Target path for import. Format:
            <repo-name>:<branch-name>:<node-path>. e.g 'cms-repo:draft:/'
```

Example:

```
$ ./toolbox.sh import -a su:password -s myExport -t cms-repo:draft:/
```

17.7 reindex

Reindex the content in the search indices for the given repository and branches.

Usage:

```
NAME
        toolbox reindex - Reindex content in search indices for the given
        repository and branches.

SYNOPSIS
        toolbox reindex -a <auth> -b <branches>... [-h <host>] [-i] [-p <port>]
                -r <repository>

OPTIONS
        -a <auth>
            Authentication token for basic authentication (user:password).

        -b <branches>
            A comma-separated list of branches to be reindexed.

        -h <host>
            Host name for server (default is localhost).

        -i
            If flag -i given true, the indices will be deleted before recreated.

        -p <port>
            Port number for server (default is 8080).

        -r <repository>
            The name of the repository to reindex.
```

Example:

```
$ ./toolbox.sh reindex -a su:password -b draft -i -r cms-repo
```

17.8 dump

Export every branch in specified repository. This is used to backup the entire repository when doing a upgrade. The result will be stored in the `$XP_HOME/data/dump` directory.

Usage:

```
NAME
        toolbox dump - Export every branch in specified repository.

SYNOPSIS
        toolbox dump -a <auth> [-h <host>] [-p <port>] -t <target>

OPTIONS
        -a <auth>
            Authentication token for basic authentication (user:password).

        -h <host>
            Host name for server (default is localhost).

        -p <port>
            Port number for server (default is 8080).

        -t <target>
            Dump name.
```

Example:

```
$ ./toolbox.sh dump -a su:password -t myDump
```

CHAPTER 18

JavaDoc API

You can either download the JavaDoc as a zip or view it directly in your browser.

CHAPTER 19

Notable Changes

This document describes the notable changes for each version.

19.1 Notable 5.3 Changes

Here's a list of some notable changes. For every change see the full changelog.

New Toolbox CLI We have removed the UNIX shell script tools and replaced it with a cross-platform Java tool instead. It has the same functionality but can now be used with all supported OS platforms.

Named exports/imports Earlier, the export/import used a path where to store/retrieve the data. It's now just a name and the export is stored under `$XP_HOME/data/export` directory.

CHAPTER 20

Frequently Asked Questions

20.1 What's the latest release?

Latest official release is |version|.

20.2 Where can I get the source code?

All source code for Enonic XP are published on our GitHub project page.

20.3 Do you publish changelogs?

Yes. You can go to the releases tab on GitHub to read the changelog for all versions. If you want to see what's comming, you can go to our GitHub wiki page.

CHAPTER 21

Troubleshooting

This document is an up-to-date list of known problems (and how to fix them) for our current release and development releases.

21.1 Wrong Java version

Verify that Java 1.8 (update 40 or higher) is installed and that this version is actually used.

Run *java -version* in the shell where you attempt to start Enonic XP:

```
$ java -version
java version "1.8.0_40"
Java(TM) SE Runtime Environment (build 1.8.0_40-b26)
Java HotSpot(TM) 64-Bit Server VM (build 25.40-b25, mixed mode)
```

The boot log will also output the version of Java that was actually used.

If the Java version does not match your expected version, make sure that `JAVA_HOME` is set correctly. For OS X and Linux users - execute the following in your command line:

```
export JAVA_HOME=`/usr/libexec/java_home -v 1.8`
```

Optionally add the line to your `~/.properties` file to make the change persistent.

21.2 Port 8080 already taken

A lot of different web software defaults to port 8080. If you find that the log is complaining about this, simply identify the other software you have running on this port and stop it.

Optionally, you may set a different port for Enonic XP, but this is the topic of *Configuration*.

E

L

R