

---

# **Enonic XP Documentation**

*Release 6.2*

**Enonic AS**

March 31, 2016



<b>1</b>	<b>Getting Started Guide</b>	<b>3</b>
<b>2</b>	<b>Developer Guide</b>	<b>17</b>
<b>3</b>	<b>Operations Guide</b>	<b>89</b>
<b>4</b>	<b>API and Reference Guide</b>	<b>103</b>
<b>5</b>	<b>Release Notes</b>	<b>169</b>
<b>6</b>	<b>Upgrading to 6.2</b>	<b>173</b>
<b>7</b>	<b>Frequently Asked Questions</b>	<b>175</b>
<b>8</b>	<b>Glossary</b>	<b>177</b>



Congratulations - you've found the official Enonic XP 6.2 documentation. We really hope you like our technology and use it to build amazing things :-).

We've put together some funky documentation - we recommend starting with the cozy stuff like *Getting Started Guide* or *My First App*. The more savvy will probably enjoy our *API and Reference Guide*.

Enjoy! - *The Enonic Development Team*

# Dive into our DOCUMENTATION



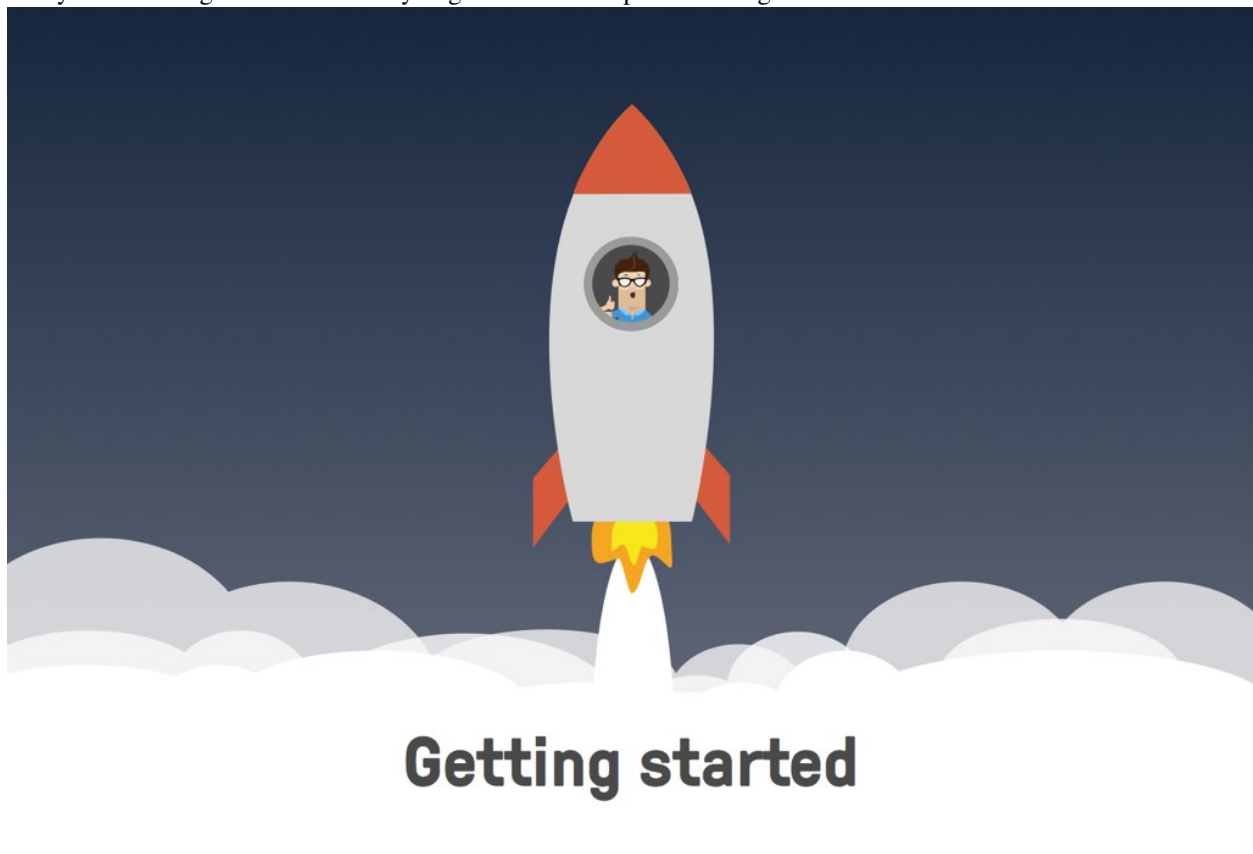
component enonic toolbox.sh Content Manager  
Enonic Content Repository domain  
./gradlew deploy GeoPoint live edit  
experience platform Docker  
Content types Mixins portal.getContent()  
Unstructured relationship-type  
init-app require('/lib/

---

## Getting Started Guide

---

So - you're looking for the fastest way to get Enonic XP up and running?



Select **ONE** of the options below to get going:

### 1.1 Use Enonic Cloud

Enonic Cloud is a one-stop-shop hosted version of Enonic XP, available on demand.

**Complete the following steps:**

- *Request Free Cloud Trial*
- *Log In*
- *Superhero Blog*
- *Next Steps*

### 1.1.1 Request Free Cloud Trial

We're offering a time limited free trial - running your very own cloud instance of Enonic XP. All you need to do is:

- Request a [Free Cloud Trial of Enonic XP](#)

---

**Note:** Enonic also offers paid cloud instances, ranging from “Developer Cloud” to “Platium Cloud”. [Check out our offerings](#)

---

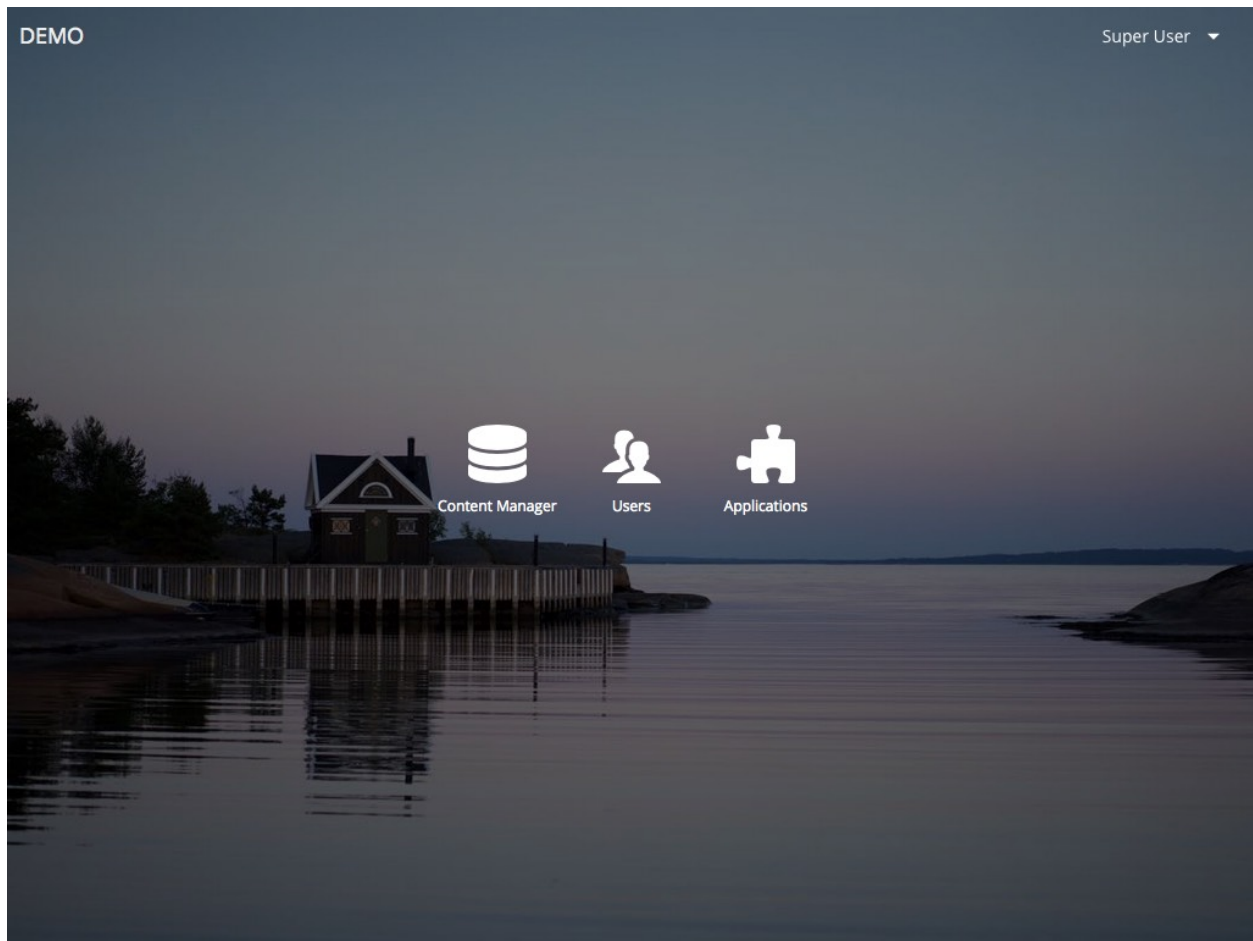
### 1.1.2 Log In

After requesting a Trial, you should receive an informative e-mail about your installation.

- Click the link in your e-mail to reach the administrative interface, it should be in the following form:  
`http://<my-email-com>.tryme.enonic.io`
- Log in with username `su` and password `password`.

After logging in you should see the following screen:





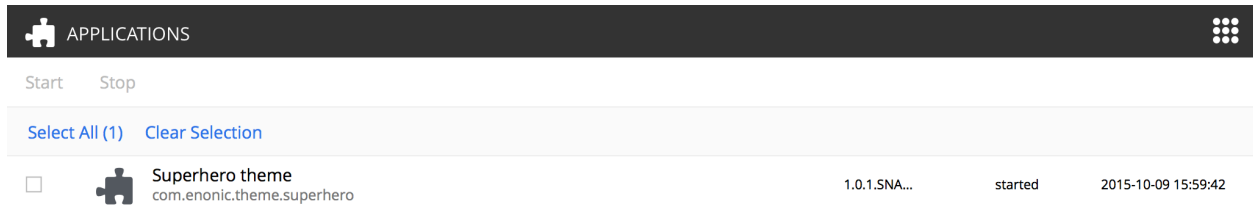
### 1.1.3 Superhero Blog

For your enjoyment, we've created the Superhero Blog application and pre-installed it on the trial instance.

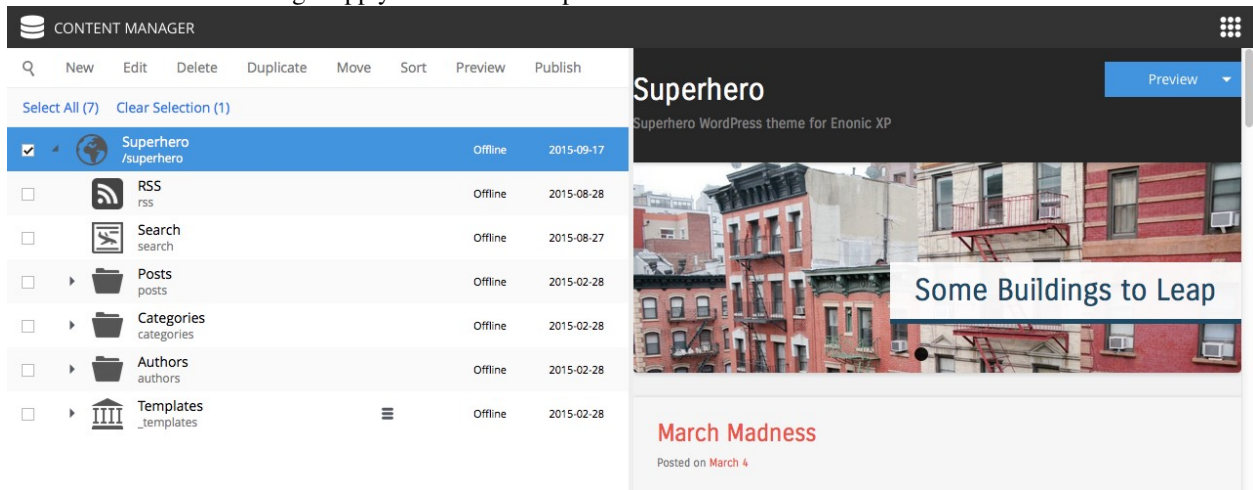
This is a simple blog, basically emulating Wordpress - even using one of their nice themes. Follow these steps to try it out:

#### Start blogging

- From the Admin interface, verify that Superhero Blog has been installed using the Applications App, it should look something like this:



- In the Content Manager app you will find a superhero site and all it's content



### Watch Video

To learn quickly how Enonic XP works, and especially the Superhero Blog Application, watch this video:



Full source code for the Superhero Blog can be found on GitHub: <https://github.com/enonic/app-superhero-blog>

### 1.1.4 Next Steps

Congratulations on getting started :-)

If you're ready for some more fun, we recommend looking into the following:

- Watch the [Platform Video](#)
- Hook up with the community on [our forum](#)
- Build your first Application with *My First App*
- Get under the hood of the *Content Domain*

## 1.2 Install on my Computer

This section describes how to install Enonic XP on your own computer. If you have any problems, please look at our [Troubleshooting](#) section.

---

### Note: General Requirements

- Any OS supporting Java (Mac, Linux, Windows etc)
  - Java JDK 1.8 (update 40 or newer)
  - At least 1 GB of available memory
  - HTTP port 8080 should be available (this can be changed if needed, see [Configuration](#))
-

## 1.2.1 Install Java

**Warning:** To run Enonic XP, you need Java Development Kit (JDK) 1.8.40 JDK or newer.

### Check JDK Version

If you're not sure what JDK version you have (or even if you have one), run the following in your terminal/shell:

```
javac -version
```

This should produce a response such as: "javac 1.8.0\_60"

Having problems with your existing Java installation? Check out our troubleshooting\_java documentation.

### Optionally Install Java

If it turns out you're on the wrong java version, follow these steps

- Download it from <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

The screenshot shows the Oracle Java SE Downloads page. The main content area is titled "Java SE Downloads" and features two download buttons: "Java Platform (JDK) 8u60" and "NetBeans with JDK 8". A red arrow points to the "Java Platform (JDK) 8u60" button with the text "Click here". Below the buttons, there is a section for "Java Platform, Standard Edition" with details about "Java SE 8u60". On the right side, there are two vertical lists of links: "Java SDKs and Tools" and "Java Resources".

- Follow the instructions for your respective operating system

## 1.2.2 Download Enonic XP

Enonic XP is available in a simple universal distribution file - running on all platforms (Windows, Linux, OSX etc)

- [Download Enonic XP distribution](#)
- Unzip the file to a suitable location

### Hint: Commandline Ninja version

```
curl -O http://repo.enonic.com/public/com/enonic/xp/distro/6.2.1/distro-6.2.1.zip
unzip distro-6.2.1.zip
cd enonic-xp-6.2.1
```

Next - let's get the server started

### 1.2.3 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 Installation Folder]/bin/server.sh
```

Windows:

```
[XP Installation Folder]\bin\server.bat
```

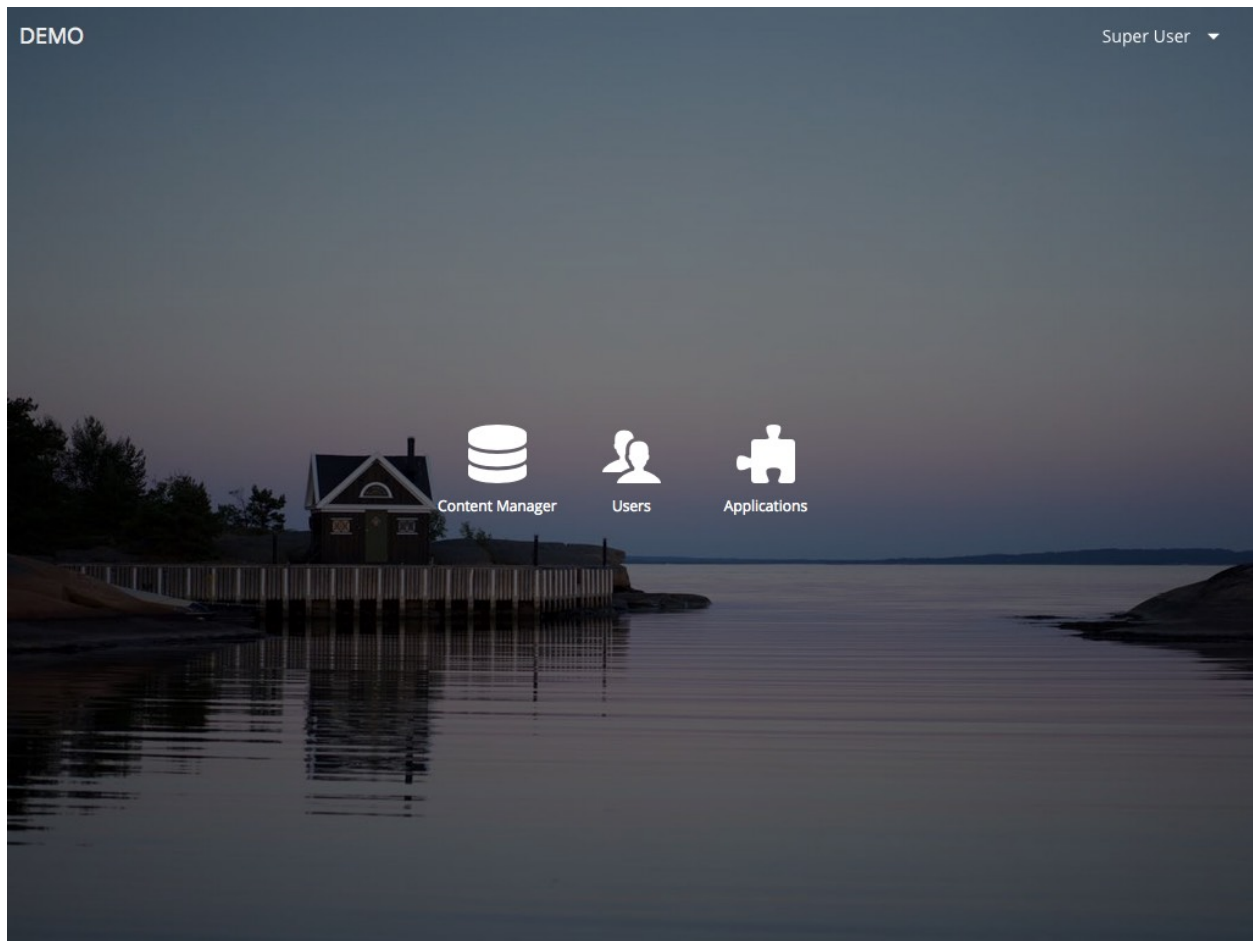
This will start Enonic XP. When successfully started, the following will appear at the end of the log:

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

### 1.2.4 Log In

- Point your browser to `http://localhost:8080`
- Log in with username `su` and password `password`.

After logging in you should see the following screen:



## 1.2.5 Install Superhero Blog

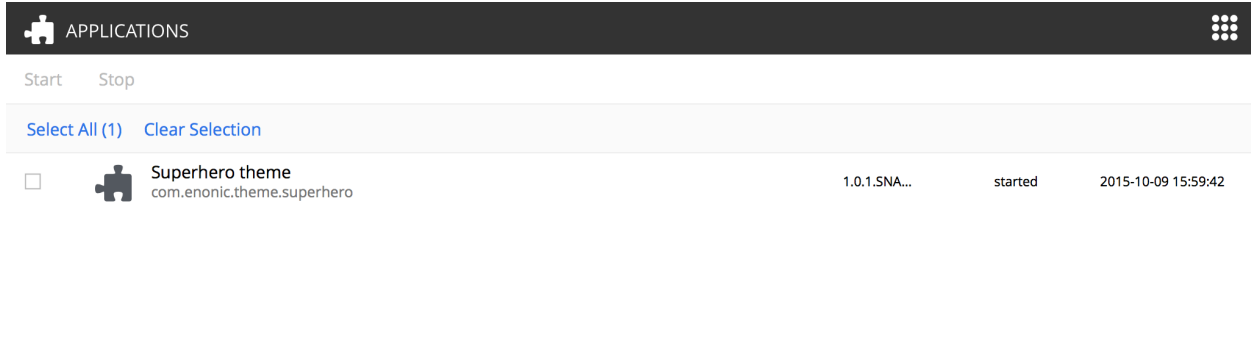
This is a simple blog, basically emulating Wordpress - even using one of their nice themes. Follow these steps to try it out:

### Deploy App

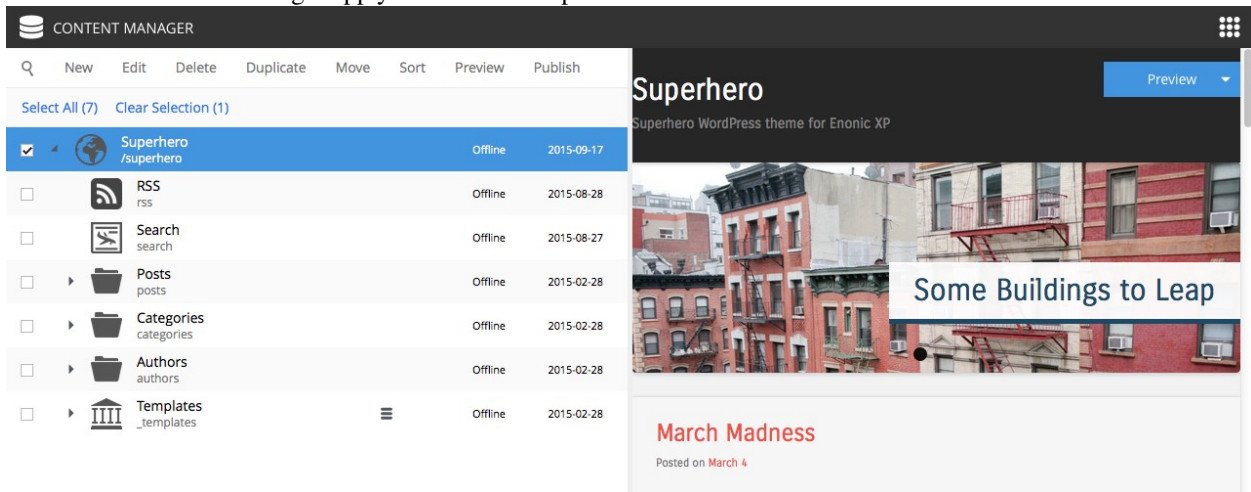
- [Download the Superhero Blog Application](#)
- Copy the application file into [XP Installation Folder]/home/deploy/

### Start blogging

- From the Admin interface, verify that Superhero Blog has been installed using the Applications App, it should look something like this:



- In the Content Manager app you will find a superhero site and all it's content



### Watch Video

To learn quickly how Enonic XP works, and especially the Superhero Blog Application, watch this video:



Full source code for the Superhero Blog can be found on GitHub: <https://github.com/enonic/app-superhero-blog>

### 1.2.6 Next Steps

Congratulations on getting started :-)

If you're ready for some more fun, we recommend looking into the following:

- Watch the [Platform Video](#)
- Hook up with the community on [our forum](#)
- Build your first Application with *My First App*
- Get under the hood of the *Content Domain*

## 1.3 Install with Docker

We're huge devop fans - and devops love [Docker](#). Docker is the most popular application container platform in the world. For your convenience, we build Docker images of every Enonic XP release.



**Complete the following steps:**

- [Install Docker](#)
- [Start Server](#)
- [Log In](#)
- [Install Superhero Blog](#)
- [Next Steps](#)

### 1.3.1 Install Docker

Running Enonic XP with Docker actually requires access to a Docker container - now, that's a surprise!

---

**Note:** Docker version 1.8.1 or newer is required to complete this guide

---

If you don't already have a Docker up and running, we recommend reading the brilliant documentation on how to get started with Docker:

- [Docker for Windows](#)
- [Docker for OSX](#)
- [Docker for Linux](#)

### 1.3.2 Start Server

#### Launch Enonic XP on Docker

With Docker up and running, installing Enonic XP is as smooth as baby skin. Execute the commands below in your terminal/shell to get going.

- Create a storage container for configuration files, applications and data (XP\_HOME)

```
docker run -it --name xp-home enonic/xp-home
```

- Install and start Enonic XP, mounting the xp-home volume

```
docker run -d -p 8080:8080 --volumes-from xp-home --name xp-app enonic/xp-app
```

This will download the latest stable Enonic XP image, start it, and map it to port 8080 on your docker-host. You can optionally add `:<versionnumber>` at the end of the command to launch a specific version of Enonic XP - i.e.

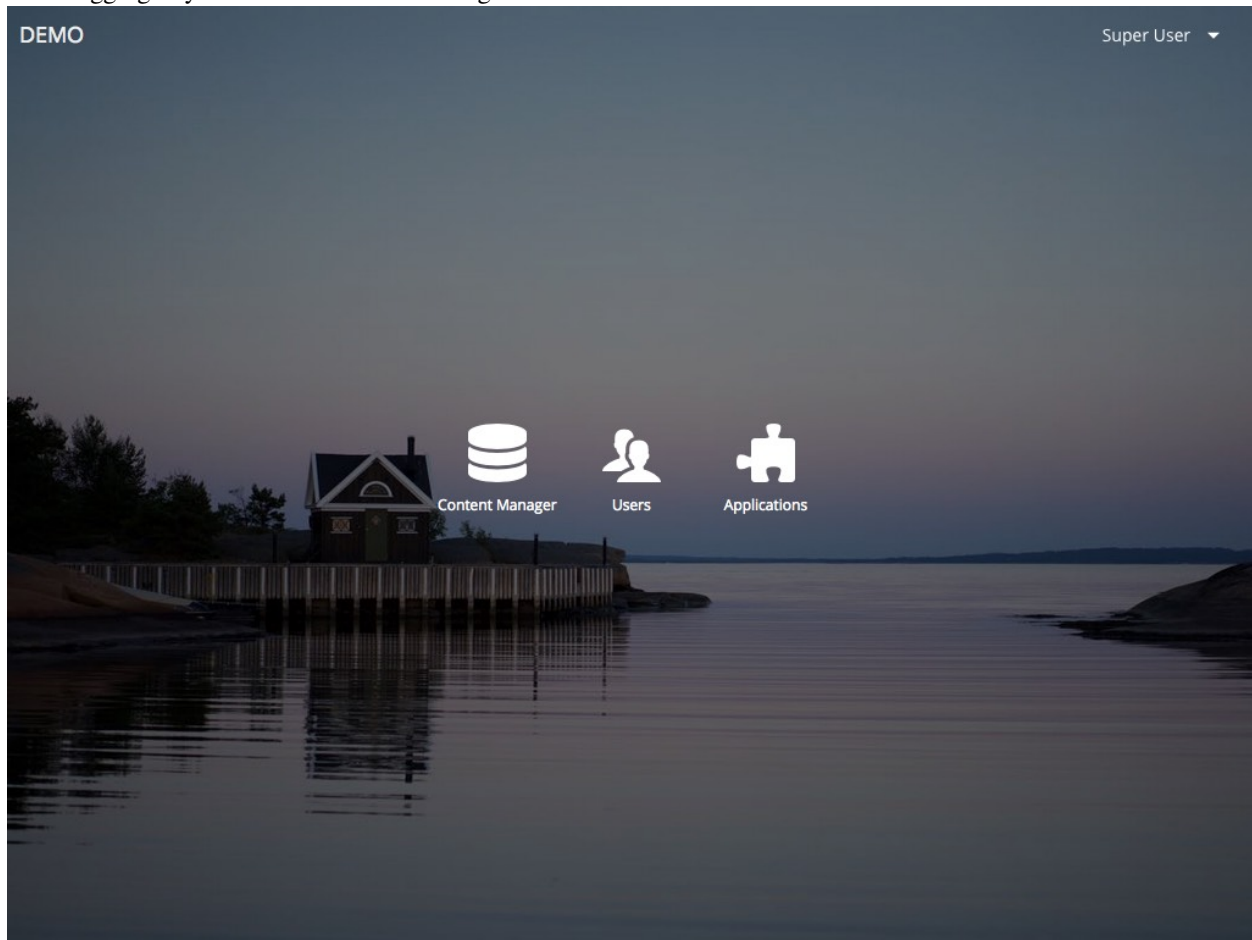
```
docker run -d -p 8080:8080 --volumes-from xp-home --name xp-app enonic/xp-app:6.1.0
```

Check out our Project page at Docker Hub for more info.

### 1.3.3 Log In

- Point your browser to `http://<mydockercontainer>:8080`
- Log in with username `su` and password `password`.

After logging in you should see the following screen:



### 1.3.4 Install Superhero Blog

This is a simple blog, basically emulating Wordpress - even using one of their nice themes. Follow these steps to try it out:

#### Deploy App

- [Download the Superhero Blog Application](#)
- Open your terminal/shell and change directory to where the file was downloaded

- Copy the application file into XP\_HOME storage volume with the following command

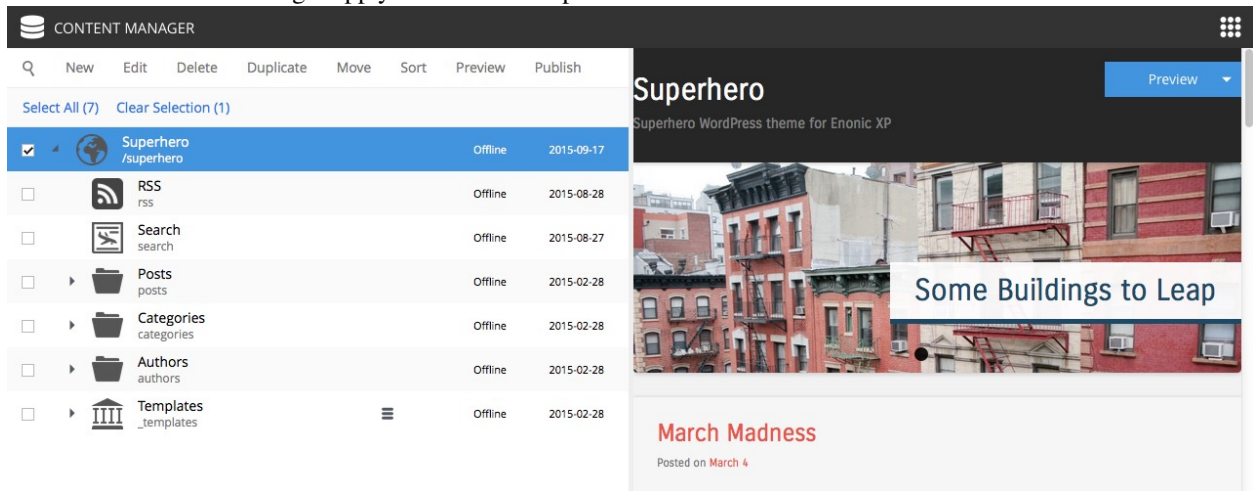
```
docker cp superhero-1.0.0.jar xp-home:/enonic-xp/home/deploy/.
```

### Start blogging

- From the Admin interface, verify that Superhero Blog has been installed using the Applications App, it should look something like this:



- In the Content Manager app you will find a superhero site and all it's content



### Watch Video

To learn quickly how Enonic XP works, and especially the Superhero Blog Application, watch this video:



Full source code for the Superhero Blog can be found on GitHub: <https://github.com/enonic/app-superhero-blog>

### 1.3.5 Next Steps

Congratulations on getting started :-)

If you're ready for some more fun, we recommend looking into the following:

- Watch the [Platform Video](#)
- Hook up with the community on [our forum](#)
- Build your first Application with *My First App*
- Get under the hood of the *Content Domain*

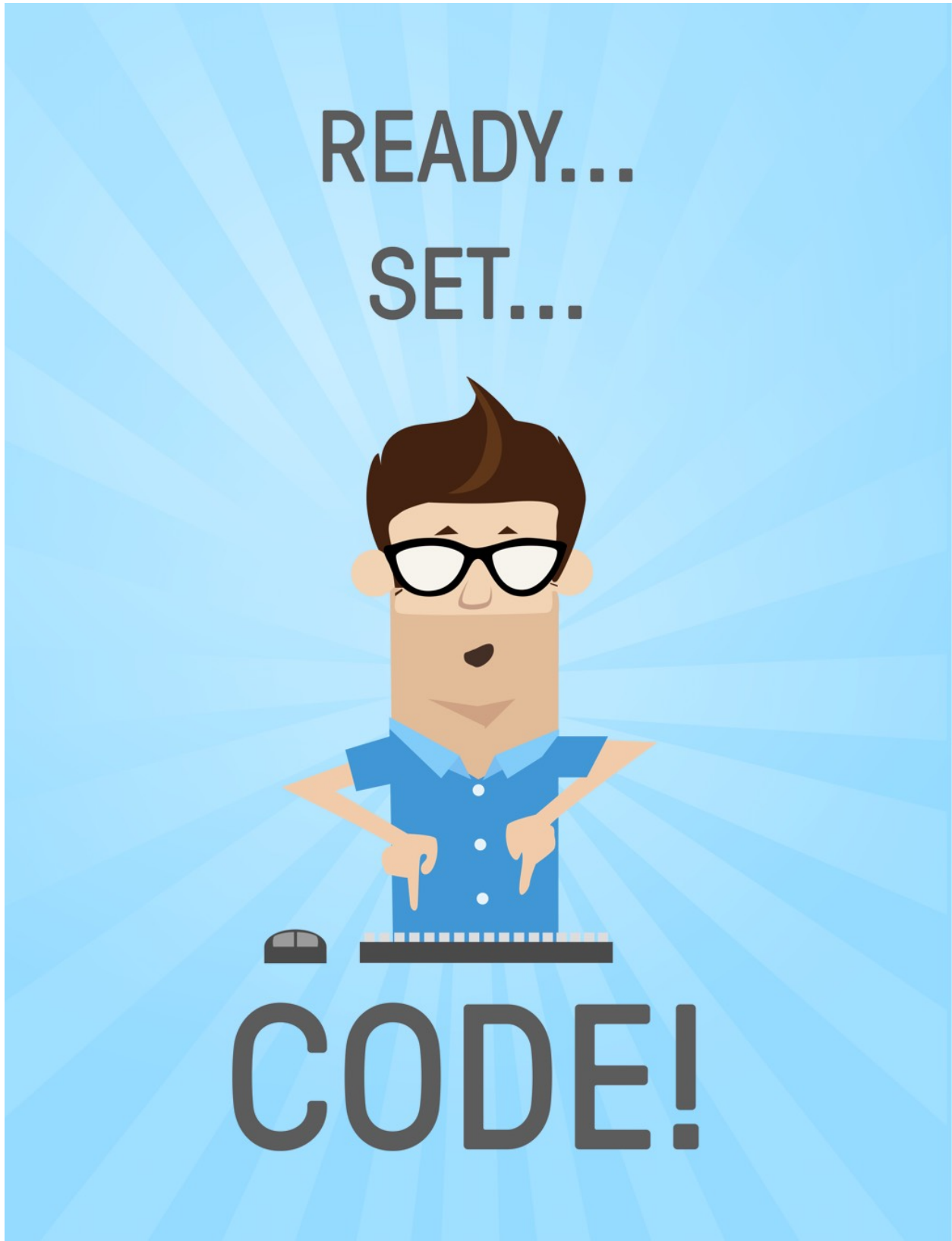
---

## Developer Guide

---

### We <3 Developers!

We're thrilled to see you here - If you're a first timer to Enonic XP, we recommend starting on *My First App* - or if you're very familiar with XP, how about drilling into the *Node Domain* chapter. If you're actually looking for APIs - you'll find them over here *API and Reference Guide*.



## 2.1 My First App

*This guide will lead you through the required steps to build the “Hello World” app for Enonic XP.*

In this tutorial, you will learn how to initialize new application projects and deploy them. We will create a simple website that displays a list of countries and cities with a Google map of each city. Upon completion, you will be familiar with content types, page and part components, page templates, regions, and the Content Manager app. You won't be writing any code - just copy/paste from the examples.

The screen-shots below show the final product of this tutorial.

---

**Note:** To complete this tutorial, you will need a local running installation (see *Install on my Computer*) of Enonic XP and a text editor of your choice. All terminal actions assume you're using OSX or Linux.

---

### 2.1.1 Initialize project

Enonic XP includes the *Toolbox CLI* which can perform several useful operations. The *init-project* operation will initialize a new application project with the standard structures required (see *Project structure*).

1. Create a new folder at a suitable location on your filesystem for the application project files. e.g. `/Users/<username>/project/myapp` This will be the project root.
2. Go to this folder in the terminal and run the following command:

```
[$XP_INSTALL]/toolbox/toolbox.sh init-project -n com.company.myapp -r starter-base
```

---

**Tip:** Only basic characters (a-z, 0-9 and .) must be used for application names, and the name must be globally unique. We recommend following standard Java package naming conventions such as `com.mycompany.myapp`.

---

Your project folder will now be filled with the standard folder structure for developing an app.

### 2.1.2 Build and Deploy

Now that we have set up a project, we should test that it builds and deploys successfully. But before deploying the app, the `$XP_HOME` environment variable must be set to the path of the home folder of the XP installation.

1. Run the following command in the terminal, replacing `[$XP_INSTALL]` with your installation location (no brackets): `export XP_HOME=[$XP_INSTALL]/home`
2. Execute the following command from the project root directory: `./gradlew deploy`

If you don't already have *Gradle* installed, the Gradle wrapper will be download first. Next it will build the app and then attempt to deploy it to your installation.

The deployment step simply moves the result of the build (the application JAR file) into the `$XP_HOME/deploy` directory. From there, Enonic XP will detect, install and start the application automatically.

You will need to access the Administrative console to check that the app has installed and started.

3. Log in to the Administrative console (<http://localhost:8080>) with the Administrative user credentials (userid **su** and password **password**).
4. Navigate to the Applications App. The application you just deployed should be listed.
5. Click the app listed as “Myapp” to see information about it and confirm that it has started.

**Note:** You can change the display name of the application by editing the `gradle.properties` file.

---

### 2.1.3 Create the Hello World Site

Our next goal is to set up a “Hello World” site in the Content Manager app, but first we must add some initial configuration to our project.

#### Site descriptor

An application can serve many purposes and building sites is just one of them. The `site.xml` file is the descriptor that will let Enonic XP know that this app can be added to a site. Site-wide configurations can be defined in this file but we will leave the `config` element empty for now (see *Configuring a site*).

A basic `site.xml` file was automatically created by the `init-project` script:

```
[project-root]/src/main/resources/site/site.xml
```

---

**Note:** All of the files we will be working with are below the “site” directory in the project folder - `src/main/resources/site`. All file paths from now on will begin with “site”.

---

#### Page Component

Page components are the most basic building blocks of websites in Enonic XP (see *Page*). They require a JavaScript controller and optionally an XML descriptor and an HTML view. This first example does not need a descriptor file.

A page controller (see *Page*) is a JavaScript file that handles requests such as GET and POST. Controllers usually pass JavaScript objects with data to be dynamically rendered in an HTML view. No data is passed in the example below, but the view file is specified and rendered as static HTML.

1. Create a folder called `hello` inside the `site/pages` directory.
2. Create the page controller and page view files specified below inside the `hello` folder:

Listing 2.1: Hello page controller - `site/pages/hello/hello.js`

```
var thymeleaf = require('/lib/xp/thymeleaf'); // Import the thymeleaf render function

// Handle the GET request
exports.get = function(req) {

  // Specify the view file to use
  var view = resolve('hello.html');

  // Render HTML from the view file
  var body = thymeleaf.render(view, {});

  // Return the response object
  return {
    body: body
  }
};
```



The *view* below is a simple HTML file. This file will be updated later to handle dynamic content.

Listing 2.2: Hello page view - site/pages/hello/hello.html



```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello world</title>
  </head>
  <body data-portal-component-type="page">
    <h1>Hello world</h1>
  </body>
</html>
```

3. Once these files are in place, redeploy the app from the terminal with `./gradlew deploy`.

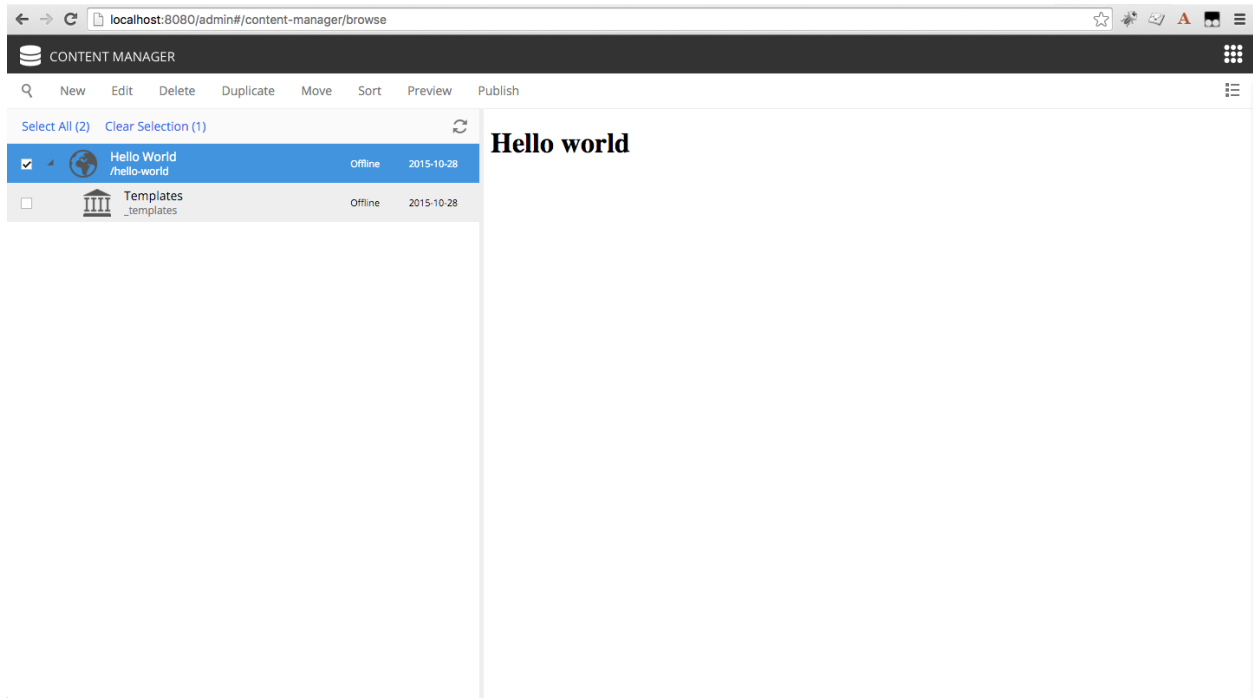
**Tip:** Each page controller must reside in its own folder under the `site/pages` directory. The name of the controller JavaScript file must be the same as the directory that contains it. The HTML view file can reside anywhere in the project and have any valid file name. This allows view files to be shared between components.

## Create Site

Now that the files are in place, we can create the site in the Administration console *Content manager* admin app. You can switch between admin apps with the button on the far right of the toolbar that looks like a square made of dots.

1. In your browser, navigate to the Content Manager admin app. (Use the square dots icon  in the toolbar to switch between admin apps.)
2. Click “New” and select “Site” from the list of content types. This opens a tab within the page for editing the *site* content.
3. Fill in the form with Display Name: “Hello World”.
4. Select your “MyApp” application in the “Applications” dropdown.
5. Open the Page Editor with this button  in the toolbar.
6. Use the dropdown in the blue area that just appeared to select the “hello” page.
7. Click the “Save draft” button in the toolbar (top-left).
8. Now close the “Hello World” site editor tab to see the content pane.

When you click on the “Hello World” site content, the preview should look something like this:



### 2.1.4 Add some Countries

In order to make our “World” slightly more interesting, we need some data - or more specifically countries.

To add structured data (such as countries), we need so-called *Content Types*. The content type defines the form (and underlying schema) of items you manage.

1. Create a folder called “country” inside the “content-types” folder of your project.
2. Add the Country content type file below to this folder.

Listing 2.3: Country content type - site/content-types/country/country.xml

```
<content-type>
  <display-name>Country</display-name>
  <super-type>base:structured</super-type>
  <form>
    <input type="TextArea" name="description">
      <label>Description</label>
      <occurrences minimum="0" maximum="1"/>
    </input>
    <input type="TextLine" name="population">
      <label>Population</label>
      <occurrences minimum="0" maximum="1"/>
    </input>
  </form>
</content-type>
```

Each content type can have a custom icon that will be visible in the Content Manager app. Though not required, content icons can be helpful for content editors.

3. Copy the image below to the the same folder (content-types/country) with the name *country.png*.



This content type defines form inputs for **description** and **population**. All content has a built-in field for **Display Name**. When the app is redeployed, this content type will produce the form seen below in the Content Manager app.

The screenshot shows the Content Manager interface for a content type named 'Country'. The top navigation bar includes 'Save draft', 'Delete', 'Duplicate', 'Preview', 'Item is Offline', and 'Publish'. The main content area displays the 'Country' content type with a 'Display Name' field (placeholder: '<Display Name>'), a 'Description' field, and a 'Population' field. The interface also shows a 'Settings' tab and a 'Security' tab.

**Tip:** Each content type must reside in its own folder under the `site/content-types` directory. The name of the content type XML file and the icon PNG file must be the same as the directory that contains them.

## 2.1.5 Create the Country Part

We also need a way to present a country - because every country wants to be seen. This time, rather than just making another page controller, we will create a *Part* component. Parts are reusable components that can be added to pages containing “regions” - more on this later.

1. Create a folder called “country” inside the “parts” folder in your project.
2. Add the part **controller** and **view** files below to the “country” folder:

Listing 2.4: Country part controller - `site/parts/country/country.js`

```
var portal = require('/lib/xp/portal'); // Import the portal functions
var thymeleaf = require('/lib/xp/thymeleaf'); // Import the Thymeleaf rendering function

// Handle the GET request
exports.get = function(req) {

  // Get the country content and extract the needed data from the JSON
  var content = portal.getContent();
```

```

// Prepare the model object with the needed data extracted from the content
var model = {
  name: content.displayName,
  description: content.data.description,
  population: content.data.population
};

// Specify the view file to use
var view = resolve('country.html');

// Return the merged view and model in the response object
return {
  body: thymeleaf.render(view, model)
}
};

```

The part controller file above handles the GET request and passes the country content data to the view file which is shown below.

Listing 2.5: Country part view - site/parts/country/country.html

```

<div>
  <h3 data-th-text="${name}"></h3>
  <div data-th-if="${population}" data-th-text="'Population: ' + ${population}"></div>
  <div data-th-if="${description}" data-th-text="${description}"></div>
</div>

```

## 2.1.6 The Hello Region Page

Parts start to make sense when placed into a *region* - regions are “slots” contained within pages or layouts. Pages and layouts may contain multiple regions, and each region must have a unique name.

Let’s create a new page component with a single region called “Main” - we will later place the “Country” part into this region.

The benefit of a region (see *Regions*) is that a page component can be re-used across multiple different pages by simply adding different parts to them as needed.

1. Create a folder called “hello-region” in your project’s `site/pages/` folder.
2. Add the “Hello region” page descriptor, controller and view files:

Listing 2.6: Page descriptor - site/pages/hello-region/hello-region.xml

```

<page>
  <display-name>Hello Region</display-name>
  <config/>
  <regions>
    <region name="main"/>
  </regions>
</page>

```

The XML file above is a *Descriptor*. Regions and page configurations can be defined here.

Listing 2.7: Page controller - site/pages/hello-region/hello-region.js

```

var portal = require('/lib/xp/portal'); // Import the portal functions
var thymeleaf = require('/lib/xp/thymeleaf'); // Import the Thymeleaf rendering function

// Handle the GET request
exports.get = function(req) {

    // Get the content that is using the page
    var content = portal.getContent();

    // Extract the main region which contains component parts
    var mainRegion = content.page.regions.main;

    // Prepare the model that will be passed to the view
    var model = {
        mainRegion: mainRegion
    }

    // Specify the view file to use
    var view = resolve('hello-region.html');

    // Render the dynamic HTML with values from the model
    var body = thymeleaf.render(view, model);

    // Return the response object
    return {
        body: body
    }
};

```

This page controller uses a portal library (see *lib-portal*) to get the content and extract the “main” region which was defined in the descriptor XML file.

Listing 2.8: Page view - site/pages/hello-region/hello-region.html

```

<!DOCTYPE html>
<html>
<head>
    <title>Hello world</title>
</head>
<body data-portal-component-type="page">
    <h1>Country</h1>
    <div data-portal-region="main">
        <div data-th-if="{mainRegion}" data-th-each="component : {mainRegion.components}" data-th-
            <div data-portal-component="{component.path}" data-th-remove="tag"></div>
        </div>
    </div>
</body>
</html>

```

The view file above defines the place on the page where the region will render parts that are dragged and dropped in Live Edit.

3. When done - redeploy your app once again!


```
./gradlew deploy
```


## 2.1.7 Add your favorite country

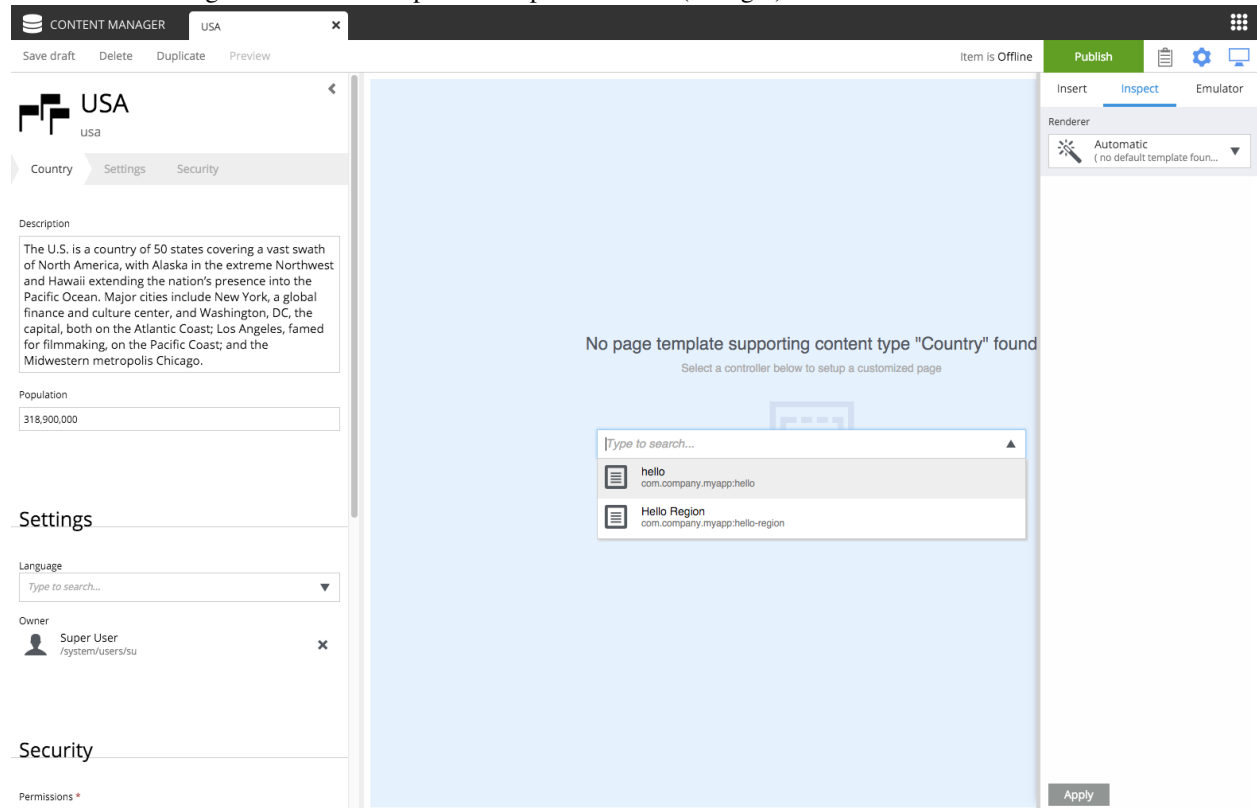
Now that the “Country” content type is installed (and we have a part to display them), we can create new countries using the *Content manager*.

1. Right-click on the “Hello World” site from the navigation tree and select “New”. The “Create Content” dialogue will open.
2. Click “Country” from the list of content types.
3. Fill in the form with the details of your favorite country.

Similar to the site, we must also configure a view for the country

4. In the toolbar, in the top right corner of the page, click the button with the monitor icon  to activate the Page Editor (blue background).

5. Click the cog button  to open the Inspection Panel (far right).





6. In the Page Editor, select “Hello Region” from the template selector dropdown. If the dropdown arrow is not visible, double-click inside the option field or start typing “hello world” in it to see the options.
7. In the Inspection Panel, click the “Insert” tab. This reveals a list of default components that can be placed into regions.
8. Click and drag a “Part” into the box on the page.
9. A new dropdown option will appear. Select the “country” part. (You can start typing “country” in the box or you may need to close the Inspection Panel to see the dropdown.)
10. Save draft and close the content edit tab.

When you click on the country in the content pane, you should see a preview of the rendered page, something like this:

The screenshot shows the Enonic XP Content Manager interface. At the top, there is a dark header with the 'CONTENT MANAGER' logo and a grid icon. Below the header is a toolbar with icons for 'New', 'Edit', 'Delete', 'Duplicate', 'Move', 'Sort', 'Preview', and 'Publish'. A search bar is also present. The main content area is divided into two panes. The left pane, titled 'Select All (3) Clear Selection (1)', contains a list of content items: 'Hello World /hello-world', 'USA usa' (which is selected and highlighted in blue), and 'Templates \_templates'. The right pane displays a preview of the 'Country' page template for the 'USA' content type. The preview shows the title 'Country' and 'USA', followed by the population '318,900,000' and a paragraph of text describing the U.S. as a country of 50 states covering a vast swath of North America, with major cities like New York, Washington, DC, Los Angeles, and Chicago mentioned.

## 2.1.8 The Country Page Template

With our current solution, sadly, we'll have to create a new page for every country we add. As this is not a very effective way of working with large data sets, we will create a page template to simplify the process.

1. Select the Templates item located below the "Hello World" site in the content pane.
2. Click "New" and select "Page Template".
3. Fill in the form as follows:
  - Display Name: "Country"
  - Supports: "Country" (selected from the list of content types)
4. If the blue Page Editor panel is not displayed on the right, click the  button in the toolbar. Then select the "Hello Region" controller with the dropdown.
5. Open the Inspection Panel (activated from the cog button  in the toolbar).
6. Under the "Insert" tab, drag and drop a "Part" into the empty region.
7. Select the "country" part from the dropdown. (You may need to close the Inspection Panel to see the dropdown.)
8. Click "Save draft" in the toolbar and close the tab.

Every "Country" content you create will now use this template by default.

---

**Tip:** The "Support" property is the key. A page template will support rendering of the content types specified here.


---

Try this out by creating a few new countries in your site. Make sure you select the "Hello World" site before clicking "New" in the toolbar. Every content you create will be a child of the content that was selected in the content pane.

## Extra task

### Make your Favorite Country use the page template too!

You might remember that your favorite country was “hardcoded” - so let’s change it to use templates as well.

1. In the Content pane, double click the country content to edit it.
2. Click on the page in the Page Editor. A context menu should appear with the name of the country. If the word “country” appears in the box then you have selected the part. In that case, click the “Parent” button twice.
3. Open the Inspection Panel (cog button ) and select “Automatic” from under the “Renderer” label. (It’s under the “Inspect” tab)
4. Save draft and close the tab.

You can select another *Page template* at any time, or even customize the presentation of a single content.

## 2.1.9 The Country List Part

Each country content can now be viewed on a page. But the site home page is still a bit empty. This section will have you alter the “hello” page controller and view files to list all of the country contents.

1. Edit the “hello” page controller file `site/pages/hello/hello.js` and make the following changes:

```

var thymeleaf = require('/lib/xp/thymeleaf'); // Import the thymeleaf render function
var contentLib = require('/lib/xp/content'); // Import the content service functions
var portal = require('/lib/xp/portal'); // Import the portal functions

// Handle the GET request
exports.get = function(req) {
  var model = {};

  var nearestSite = portal.getSite();

  // Get all the country contents (in the current site)
  var result = contentLib.query({
    start: 0,
    count: 100,
    contentTypes: [
      app.name + ':country'
    ],
    "query": "_path LIKE '/content" + nearestSite._path + "/*'"
  });

  var hits = result.hits;
  var countries = [];

  // Loop through the contents and extract the needed data
  for(var i = 0; i < hits.length; i++) {

    var country = {};
    country.name = hits[i].displayName;
    country.contentUrl = portal.pageUrl({
      id: hits[i]._id
    });
    countries.push(country);
  }
}

```



```

// Add the country data to the model
model.countries = countries;

// Specify the view file to use
var view = resolve('hello.html');

// Compile HTML from the view with dynamic data from the model
var body = thymeleaf.render(view, model);

// Return the response object
return {
    body: body
}
};

```

2. Now edit the “hello” view file `site/pages/hello/hello.html` and make the following changes:

```

<!DOCTYPE html>
<html>
<head>
  <title>Hello world</title>
</head>
<body data-portal-component-type="page">
  <h1>Hello world</h1>
  <h2>Countries</h2>
  <ul>
    <li data-th-each="country : ${countries}">
      <strong>
        <a data-th-href="${country.contentUrl}" data-th-text="${country.name}"></a>
      </strong>
    </li>
  </ul>
</body>
</html>

```

3. Redeploy the app from the command line with `./gradlew deploy`.

Each country that you created is now listed on the home page and the names are also links to the individual content pages.

## 2.1.10 Hello Geo World

Going back to your site, you will now see a list of the countries we have added. To make this even more exciting, we will add a *City* content type with geo-location and a *City list* part with configuration capabilities.

### City content

The next steps will create a content type for adding cities with location coordinates.

1. Create a folder called *city* inside the project’s `site/content-types` folder.
2. Add the content type file below to your project:

Listing 2.9: City content type - site/content-types/city/city.xml

```

<content-type>
  <display-name>City</display-name>
  <super-type>base:structured</super-type>
  <form>
    <input type="GeoPoint" name="location">
      <label>Location</label>
      <occurrences minimum="1" maximum="1"/>
    </input>
    <input type="TextLine" name="population">
      <label>Population</label>
      <occurrences minimum="0" maximum="1"/>
    </input>
  </form>
</content-type>

```

The file above defines a *content type* for cities with a required field for the location in latitude and longitude.

3. Copy the image below and save it in the same folder with the City content type. Name it “city.png”.



### City list part

We need a *part component* to display the city data. It will list the cities and show a Google map of each location.

1. Create a folder called *city-list* inside the project’s *site/parts* folder.
2. Add the part descriptor file.

Listing 2.10: City list part descriptor - site/parts/city-list/city-list.xml

```

<part>
  <display-name>City list</display-name>
  <config>
    <input type="ComboBox" name="mapType">
      <label>Map type</label>
      <occurrences minimum="0" maximum="1"/>
      <config>
        <option value="ROADMAP">ROADMAP</option>
        <option value="SATELLITE">SATELLITE</option>
        <option value="HYBRID">HYBRID</option>
        <option value="TERRAIN">TERRAIN</option>
      </config>
    </input>
    <input type="TextLine" name="zoom">
      <label>Zoom level 1-15</label>
      <occurrences minimum="0" maximum="1"/>
    </input>
  </config>
</part>

```

```
</config>
</part>
```

The part descriptor above has a configuration similar to those found in content types.

3. Add the part controller file.

Listing 2.11: City list part controller - site/parts/city-list/city-list.js

```
var contentLib = require('/lib/xp/content'); // Import the content library functions
var portal = require('/lib/xp/portal'); // Import the portal functions
var thymeleaf = require('/lib/xp/thymeleaf'); // Import the Thymeleaf rendering function

// Handle the GET request
exports.get = function (req) {

    // Get the part configuration for the map
    var config = portal.getComponent().config;
    var zoom = parseInt(config.zoom) && config.zoom <= 15 && config.zoom >= 1 ? config.zoom : 10;
    var mapType = config.mapType || 'ROADMAP';

    // String that will be inserted to the head of the document
    var googleMaps = '<script src="http://maps.googleapis.com/maps/api/js"></script>';

    var countryPath = portal.getContent()._path;

    // Get all the country's cities
    var result = contentLib.query({
        start: 0,
        count: 100,
        contentTypes: [
            app.name + ':city'
        ],
        "query": "_path LIKE '/content" + countryPath + "/*'",
    });

    var hits = result.hits;

    var cities = [];

    if (hits.length > 0) {
        googleMaps += '<script>function initialize() {';

        // Loop through the contents and extract the needed data
        for (var i = 0; i < hits.length; i++) {

            var city = {};
            city.name = hits[i].displayName;
            city.location = hits[i].data.location;
            city.population = hits[i].data.population ? 'Population: ' + hits[i].data.population : null;

            cities.push(city);

            if (city.location) {
                city.mapId = 'googleMap' + i;

                googleMaps += 'var center' + i + ' = new google.maps.LatLng(' + city.location + ');
```

```

        googleMaps += 'var mapProp = {center:center' + i + ', zoom:' + zoom +
            ', mapTypeId:google.maps.MapTypeId.' + mapType + ', scrollwheel: false
            'var map' + i + ' = new google.maps.Map(document.getElementById("google
            'var marker = new google.maps.Marker({ position:center' + i + '}); mar

    }

    }

    googleMaps += ' } google.maps.event.addDomListener(window, "load", initialize);</script>';
}

// Prepare the model object that will be passed to the view file
var model = {
    cities: cities
};

// Specify the view file to use
var view = resolve('city-list.html');

// Return the response object
return {
    body: thymeleaf.render(view, model),
    // Put the maps' javascript into the head of the document
    pageContributions: {
        headEnd: googleMaps
    }
}
};

```

This controller uses *Page Contributions* to put the Google Maps JavaScript into the head of the document.

4. Add the part view file.

Listing 2.12: City list part view - site/parts/city-list/city-list.html

```

<div class="cities" style="min-height:100px;">
  <h3>Cities</h3>
  <div class="city" data-th-each="city : ${cities}">
    <h3 data-th-text="${city.name}"></h3>
    <div data-th-if="${city.population}" data-th-text="${city.population}"></div>
    <div data-th-id="${city.mapId}" data-if="${city.mapId}" style="width:100%;height:300px;"></div>
    <br/><br/>
  </div>
</div>


```

5. Build and deploy your project one final time with `./gradlew deploy`.

All of the project's files are now complete. The rest of the steps will be performed in the Content Manager app.

## 2.1.11 Create Cities

Now let's make use of the new city content type and part component. First we need to add the "City list" part to the "Country" page template.

1. Edit the "Country" page template.
2. Open the Inspect Panel by clicking the cog button  in the toolbar.

3. Under the Insert tab, click and drag a *Part* to the page region below the “country” part. (This may be a bit tricky because the “country” part is small.)
4. Select the “City list” part from the dropdown in the box. You may need to close the Inspection Panel to see it.
5. Save and close the tab.

Now we need to create a few City contents below the countries. (Sample data is available in the table below.)

1. From the content pane, select a country content that you created earlier. It is important that each city content is created under the country.
2. Right-click the country content and select “New”. The “Create content” dialogue will open.
3. Now select “City” from the list of content types.
4. Fill in the city name and location; the population is optional. The location format must be comma separated latitude and longitude with decimals.
5. Save draft.
6. Create several more city contents below each country content by repeating the previous steps. Sample data is provided in the table below.

Country	City	Lat,Long	Population
USA	San Francisco	37.7833,-122.4167	837,442
	Las Vegas	36.1215,-115.1739	603,488
	Washington D.C.	38.9047,-77.0164	658,893
Norway	Oslo	59.9100,10.7500	618,683
	Bergen	60.3894,5.3300	265,857
	Trondheim	63.4297,10.3933	178,021
Colombia	Bogota	4.5981,-74.0758	7,000,000
	Medellin	6.2308,-75.5906	2,440,000
	Barranquilla	10.9639,-74.7964	1,885,500

Each country page will now have a list of the cities you created with a Google map of the location. It should look something like this:

**CONTENT MANAGER**

Search New Edit Delete Duplicate Move Sort Preview Publish

Select All (14) Clear Selection (1)

<input type="checkbox"/>	Hello World /hello-world	Offline	2015-10-28
<input checked="" type="checkbox"/>	USA usa	Offline	2015-10-28
<input type="checkbox"/>	San Francisco san-francisco	Offline	2015-10-28
<input type="checkbox"/>	Las Vegas las-vegas	Offline	2015-10-28
<input type="checkbox"/>	Washington D.C. washington-d-c	Offline	2015-10-28
<input type="checkbox"/>	Norway norway	Offline	2015-10-28
<input type="checkbox"/>	Oslo oslo	Offline	2015-10-28
<input type="checkbox"/>	Bergen bergen	Offline	2015-10-28
<input type="checkbox"/>	Trondheim trondheim	Offline	2015-10-28
<input type="checkbox"/>	Colombia colombia	Offline	2015-10-28
<input type="checkbox"/>	Bogota bogota	Offline	2015-10-28
<input type="checkbox"/>	Medellin medellin	Offline	2015-10-28
<input type="checkbox"/>	Barranquilla barranquilla	Offline	2015-10-28
<input type="checkbox"/>	Templates _templates	Offline	2015-10-28

## Country

### USA

Population: 318,900,000

The U.S. is a country of 50 states covering a vast swath of North America, with Alaska in the extreme Northwest and Hawaii extending the nation's presence into the Pacific Ocean. Major cities include New York, a global finance and culture center, and Washington, DC, the capital, both on the Atlantic Coast; Los Angeles, famed for filmmaking, on the Pacific Coast; and the Midwestern metropolis Chicago.

### Cities

#### San Francisco

Population: 837,442

#### Las Vegas

Population: 603,488

#### Washington D.C.

## 2.1.12 Configure City List

The *City list* part descriptor (<site/parts/city-list/city-list.xml>) has configuration inputs for the map type and zoom level. You can set the default values for these inputs by editing the *City list* part in the *Country* page template.

1. Open the *Country* page template for editing.



2. Open the Inspection Panel by clicking the cog button in the toolbar.
3. Click on the *City list* part in the Page Editor panel. (The *Inspect* tab should open.)
4. Set the Map type to “Hybrid” and Zoom level to 12 with the form inputs in the context panel.
5. Save draft and close the edit tab.

Now all of the countries will show the city maps with the new settings. You can override these defaults for any individual country by editing the Country content and changing its *City list* part configuration.

The screenshot displays the Enonic XP Content Manager interface for editing a 'Country' item. The top navigation bar includes 'CONTENT MANAGER', 'Country', and 'Item is Offline'. The left sidebar shows the 'Country' item details, including its ID 'country', page template, settings, and security. The central preview area shows the 'Country' content with a 'Cities' part highlighted. A context menu is open over the 'Cities' part, showing options like 'City list', 'Select parent', 'Insert', 'Reset', 'Remove', and 'Duplicate'. The right sidebar shows the configuration for the 'City list' part, including a search field for 'Map type' and a 'Zoom level 1-15' set to 12. An 'Apply' button is visible at the bottom right of the preview area.

### 2.1.13 Go Online

Now that your “Hello World” is complete, it’s time to publish.

1. Select the “Hello World” site in the content pane
2. Right-click and select “Publish” from the context menu, or click the “Publish” button in the toolbar
3. The Publishing Wizard will appear. Check the “Include children” checkbox
4. Verify that all your items are listed - click “Publish”!

When publishing content, all the selected items and changes are “cloned” from draft to the master branch (*Repository*).

You will always see the draft version of content in the preview window and the Page Editor of the *Content manager*. If you have placed your site on root level, you can also see your live site at this url: <http://localhost:8080/portal/master/hello-world>.

Well done - you just created your first App for Enonic XP - The Enonic team congratulates you - we look forward to seeing all the brilliant things you will make and are always looking for [feedback](#).

### 2.1.14 Some Pro Tips

#### Handling Multiple projects

A **best practice** for working on multiple projects would involve keeping a separate XP\_HOME folder for each project. The folder structure for such a setup would look something like this:

```
/Users/<name>/development
/Users/<name>/development/software/<xp-install-version>
/Users/<name>/development/xp-homes/<project-name>/home
/Users/<name>/development/projects/<project-name>/<project-source-files>
```

An actual implementation with projects called my-first-app and company-site would look like this:

```
/Users/mla/development/software/enonic-xp-5.3.0
/Users/mla/development/software/enonic-xp-6.0.0
/Users/mla/development/xp-homes/my-first-app/home
/Users/mla/development/xp-homes/company-site/home
/Users/mla/development/projects/my-first-app/...
/Users/mla/development/projects/company-site/...
```

This allows you to have one Enonic XP installation for each version and as many different XP\_HOME folders as you need for your projects. When switching from one project to another, you only have to change the XP\_HOME environment variable and then restart the installation of the Enonic XP version that the project was created for.

Check this [Enonic Labs article](#) for a more in-depth process. It also includes some bash scripting that will help with setting and changing \$XP\_HOME and starting and stopping XP.

#### Logging JSON objects

While developing an app, it can be helpful to see the structure of objects returned by library functions. The best way to do this is to set up a utilities JavaScript file in the project lib folder. Add the following function to the utilities file:

```
site/lib/utilities.js
```

```
exports.log = function (data) {
  log.info('Utilities log %s', JSON.stringify(data, null, 4));
};
```

Call the log function in any controller like the example below and then check the log after refreshing the page.

```
var util = require('utilities');

var content = portal.getContent();
util.log(content);
```

#### Using Gradle continuous-mode

It can be quite time consuming to frequently switch to the terminal to redeploy an app during development. Try using `./gradlew -t deploy` in the terminal (from the project root) to **automatically redeploy** your app every time a change to a file is detected. Gradle 2.7 or newer must be used and the \$XP\_HOME environment variable must be set in the terminal window for this to work.



## 2.1.15 Next Steps

This tutorial only covered the basics of app development. Explore the documentation for more in-depth coverage.

Check our [GitHub](#) page for examples of more advanced apps. The [Xeon](#) app is fairly simple but still much more advanced than this. The [Superhero](#) blog app is also a good place to see more advanced code.

Visit our [YouTube](#) channel for training videos.

Need help? Ask questions on our [forum](#) or answer questions from others.

The screenshot below shows the Content Manager app with content on the left and a site preview on the right.

The screenshot displays the Content Manager interface for an application named 'Hello World'. The interface is split into two main sections: a content list on the left and a site preview on the right.

**Content Manager Interface:**

- Header:** CONTENT MANAGER
- Navigation:** New, Edit, Delete, Duplicate, Move, Sort, Preview, Publish
- Selection:** Select All (15), Clear Selection (1)
- Content List:**

Item Name	Path	Status	Last Modified
USA	usa	Offline	2015-08-28 16:05:58
San Francisco	san-francisco	Offline	2015-08-28 19:11:16
Las Vegas	las-vegas	Offline	2015-08-28 19:10:43
Washington D.C.	washington-d.c	Offline	2015-08-28 19:10:07
Norway	norway	Offline	2015-08-28 19:09:18
Oslo	oslo	Offline	2015-08-28 19:14:00
Bergen	bergen	Offline	2015-08-28 19:12:49
Trondheim	trondheim	Offline	2015-08-28 19:11:53
Colombia	colombia	Offline	2015-08-28 19:06:47
Bogota	bogota	Offline	2015-08-28 19:16:10
Medellin	medellin	Offline	2015-08-28 19:15:18
Barranquilla	barranquilla	Offline	2015-08-28 19:14:42
Templates	_templates	Offline	2015-08-28 15:37:40
Country	country	Offline	2015-08-28 19:04:50

**Site Preview:**

- Page Title:** Hello world
- Content:** Countries
  - Colombia
  - Norway
  - USA
- Buttons:** Preview

The image below is what a page of the site will look like when finished.



```

my-first-app/
  build.gradle
  src/
    main/
      resources/
        site/
          site.xml
          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.

**site/site.xml** The `site.xml` file contains basic information for a site created with the application. Settings for the application can be defined in the `config` element and the values for these settings can be updated in the administration console.

```

<site>
  <x-data mixin="menu-item"/>
  <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>
</site>

```

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

**site/pages/** Page definitions should be placed here. They will be used to create page templates in the repository (see *Page*).

**site/parts/** Part definitions should be placed here. Parts are objects that can be placed on a page (see *Part*).

**site/layouts/** Layout definitions should be placed here. Layouts are definitions that restricts the placement of parts (see *Layout*).

**site/views/** Views can generally be placed anywhere you want, just keep in mind what path to use when resolving them (see *Rendering a View*).

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

**site/content-types/** Content schemas-types are placed here. Used to create structured content (see *Content Types*).

**site/mixins/** Mixin schema-types are placed here. A mixin can be used to add fields to a content-type (see *Mixins*).

**site/relationship-types/** Relationship-types are placed here. They are used to form relations between contents (see *Relationship Types*).

## 2.3 Installing an application

After an application is built with [Gradle](#), it must be deployed to your Enonic XP installation:

To deploy a application, 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 application on changes, you can use [Gradle continuous mode](#). This will watch for changes and run the specified task when something changes. To use this with `deploy` task, you can run the following command:

```
$ gradle --continuous deploy
```

Or the short version:

```
$ gradle -t deploy
```

This will deploy and reload the application on server when something changes and is probably the fastest way to develop your application.

## 2.4 Configuring a site

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

```
<site>
  <x-data mixin="menu-item"/>
  <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>
</site>
```

```

</field-set>
</config>
</site>

```

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

```

var portal = require('/lib/xp/portal');

// Find the site configuration for this app in current site.
var siteConfig = portal.getSiteConfig();

```

## 2.5 Page

A page component is the most basic building block of a site. Each page component must have a JavaScript controller file and optionally an XML descriptor and an HTML view file. These files can define regions in the page where parts and layouts may be added, or they can define a simple page without any compositions. Page components can be added to content individually (via the Content Manager, see *Content manager*) or they can be used to create page templates that automatically render supported content types.

Any number of page templates can be created from a single page component. Thanks to the magic of page templates, even very large sites will typically have very few page components—perhaps one for all the HTML pages and one for RSS pages.

Pages should be placed in the folder `site/pages/[page-name]`

### 2.5.1 Descriptor

The page descriptor is an XML file that is used to define regions and custom input fields for page configuration. If a page does not require regions or configuration options then the descriptor may be omitted.

The file must be named `[page-name].xml`. For example, if a page component is named “default” then the file must reside at `site/pages/default/default.xml`.

```

<page>
  <display-name>My first page</display-name>
  <config>
    <!-- input fields... -->
  </config>
  <regions>
    <region name="top"/>
    <region name="bottom"/>
  </regions>
</page>

```

**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.

### 2.5.2 Controller

A page controller handles requests to the page. The controller is a required file written in JavaScript and must be named `[page-name].js`. A controller exports a method for each type of HTTP request that should be handled.

The handle method has the request object as a parameter and returns the response object (see *Response*).

```
// 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'
  };
};
```

### 2.5.3 Render-view

If you feel like concatenating strings to create an entire web page 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 `[page-name].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.

```
var thymeleaf = require('/lib/xp/thymeleaf');

exports.get = function(req) {

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

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

  // Render a thymeleaf template
  var body = thymeleaf.render(view, model);

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

```
};
};
```

Unlike controllers and descriptors, view files can reside anywhere in your project and have any valid file name. This allows for code reuse as multiple page components can share the same view. If the view file is in the same folder as the page controller then it can be resolved with only the file name `resolve('file-name.html')`. Otherwise, the full path should be used, starting with a `'/'` as in the example above.

## 2.5.4 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](#).

## 2.5.5 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. Each region will be referenced by name.

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

You will also need to handle regions in the controller.

```
var portal = require('/lib/xp/portal');

// Get the current content. It holds the context of the current execution
// session, including information about regions in the page.
var content = 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-region="main">
      <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 into our page.

## 2.6 Part

A part is a building block that can be placed in a *region* on a page or layout. As with pages, each part is composed of a JavaScript controller, an XML descriptor and an HTML view.

The part descriptor and controller files must be placed in the folder `site/parts/[part-name]`

### 2.6.1 Descriptor

The part **descriptor** is where input fields are defined for custom configuration of the part. The descriptor is not required if the part does not need any custom configuration. Parts cannot contain regions.

When used, the descriptor file must have the same name as the *part* folder that contains it `site/parts/[part-name]/[part-name].xml`:

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



## 2.6.2 Controller

To drive this part, we will also need a **controller**. The controller typically uses library functions to get content and/or configurations and prepare data which it passes to the view file for dynamic rendering.

```
site/parts/[part-name]/[part-name].js
```

```
var portal = require('/lib/xp/portal'); // Import the portal functions
var thymeleaf = require('/lib/xp/thymeleaf'); // Import the thymeleaf render function

// Handle GET requests
exports.get = function(portal) {

  // Find the current component from request
  var component = 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('/site/view/my-favorite-things.html');

  // Render a thymeleaf template
  var body = thymeleaf.render(view, model);

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

## 2.6.3 View

The HTML generated by the part must have a single root element. The `things` parameter is basically just JSON data passed from the controller and we can iterate over it easily in Thymeleaf and print its values.

```
<section>
  <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.

## 2.7 Layout

Layouts are used in conjunction with `regions` to organize the structure of 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. Making a layout is similar to making pages and part components. Layouts cannot be nested.

Layout contains - like pages and parts - a descriptor, a controller and a view, and should be placed in the folder `site/layouts/[layout-name]`

### 2.7.1 Descriptor

The layout descriptor defines regions within the layout where parts can be placed with Live Edit. The file must be named `[layout-name].xml`.

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

### 2.7.2 Controller

The layout controller composes the view of the layout based on HTTP requests. The file must be named `[layout-name].js`.

```
var portal = require('/lib/xp/portal');
var thymeleaf = require('/lib/xp/thymeleaf');

exports.get = function(req) {

  // Find the current component.
  var component = 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 = thymeleaf.render(view, model);

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

```
};
};
```

### 2.7.3 View

A layout view defines the markup for the layout component. The sample view below is created in Thymeleaf, but it could be created in any view engine that is supported.

```
<div class="row" data-th-attr="data-portal-component-type=${component.type}">
  <div data-portal-region="left" 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-region="right" 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>
```

### 2.7.4 Styling

For a layout to have any meaning, some styling must be applied to the view. The desired CSS should be placed in the `/assets` folder of the application, and included in the page where the layout should be supported. For example, the view `my-first-page.html` supports 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/boot
</head>
```

## 2.8 Service

With services you can create REST-like services without binding it to some content. Create a folder `site/services/[service-name]` and place a `[service-name].js` file within this folder. This will be accessible with a fixed url:

```
*/_/service/[application]/[service-name]
```

Where `application` is the application name (without version).

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'
};
};
```

## 2.9 Response Filters

Response filters are scripts, similar to controllers, that allow customizing or adapting the response, globally within a site. After the page and component controllers have been processed during rendering, the response filters will be executed.

To add a response filter, create a folder `site/filters` in the application and place a `[filter-name].js` file within this folder. A filter must export a method named `responseFilter`. This method receives the request and response objects as parameters and must return a response object (see *Request* and *Response*).

Here is an example of a `[filter].js` file:

```
exports.responseFilter = function (req, res) {
  var trackingScript = '<script src="http://some.site/js/tracking.js"></script>';

  if (!res.pageContributions.bodyEnd) {
    res.pageContributions.bodyEnd = [];
  }
  res.pageContributions.bodyEnd.push(trackingScript);

  if (req.params.debug === 'true') {
    res.applyFilters = false; // skip other filters
  }

  return res;
};
```

In addition, the filter must be declared in the `site.xml` descriptor by adding a `<response-filter>` tag within the `<filters>` element, with the name and order.

```
<site>
  <filters>
    <response-filter name="filter1" order="10"/>
    <response-filter name="filter2" order="20"/>
  </filters>
</site>
```

Response filters may change any of the values of the response object, that includes: HTTP status code, response body, HTTP headers, cookies and page contributions.

It is also possible to return the response object received without any changes.

## 2.9.1 Execution order

An application can contain multiple filters, declared in `site.xml`. And also multiple applications can be selected for a Site. When a page is rendered, all the filters declared in all the applications selected for the site will be executed. The order in which the filters are executed depends on the `filters order` (as defined in `site.xml`) and the order of the applications configured on the Site.

The filters with a **lower order** will be executed **first**. In case there are several filters with the same `order` number, then the position of the applications (as configured for a Site in Content Manager) determines the order.

The filter-chain execution can be interrupted from either a controller or a filter by setting the `applyFilters` field in the response. When this value is set to `false` all of the remaining filters will be skipped.

## 2.10 Error Page

The error page for a site can be customized by defining one or more error handling scripts.

To add an error script, create a folder `site/error` in the application and place an `error.js` file within this folder. The file follows the same pattern as controllers and filters. If certain methods are implemented and exported, they will be executed in case of errors during rendering.

When there is an error while rendering a page, the system looks for an `error.js` script in all the applications configured for the site, in the order they are set on the site. If an `error.js` script is found, it looks for an exported method named `handleXXX` where `XXX` is the HTTP status-code of the error. If not found, it will try to find the generic error method `handleError` instead.

Here is an example of an `error.js` file:

```
var thymeleaf = require('/lib/xp/thymeleaf');

var view404 = resolve('page-not-found.html');
var viewGeneric = resolve('error.html');

exports.handle404 = function (err) {
  var body = thymeleaf.render(view404, {});
  return {
    contentType: 'text/html',
    body: body
  }
};

exports.handleError = function (err) {
  var debugMode = err.request.params.debug === 'true';
  if (debugMode && err.request.mode === 'preview') {
    return;
  }

  var params = {
    errorCode: err.status
  };
  var body = thymeleaf.render(viewGeneric, params);

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

The input parameter for the `handleXXX` and `handleError` functions is an error JSON object containing the status code, error message, Java Exception object, and the original *Request* object:

```
{
  "status": 404,
  "message": "Some error message",
  "exception": "<the actual exception object in Java>",
  "request": "<original request JSON>"
}
```

The expected returned value for the function is a response object (see *Response*).

The error processing logic will try every handle-function in application order until it can get a result (not `undefined` or `null`). This means that an error function can decide to not handle a specific error and let the next one deal with it. If no result is returned by any function, it will be eventually handled by the internal error page.

Also note that if an error occurs inside the custom-error code, then the internal error page will be rendered.

## 2.11 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 *lib-i18n* library.

### 2.11.1 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 application `site` folder.

Each locale to be localized should be represented by a single resource, e.g this could be a structure for an app 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](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.

### 2.11.2 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

### 2.11.3 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.

## 2.12 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 *lib-thymeleaf*)
- Mustache (see *lib-mustache*)
- Xslt (see *lib-xslt*)

## 2.13 Node Domain

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

### 2.13.1 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

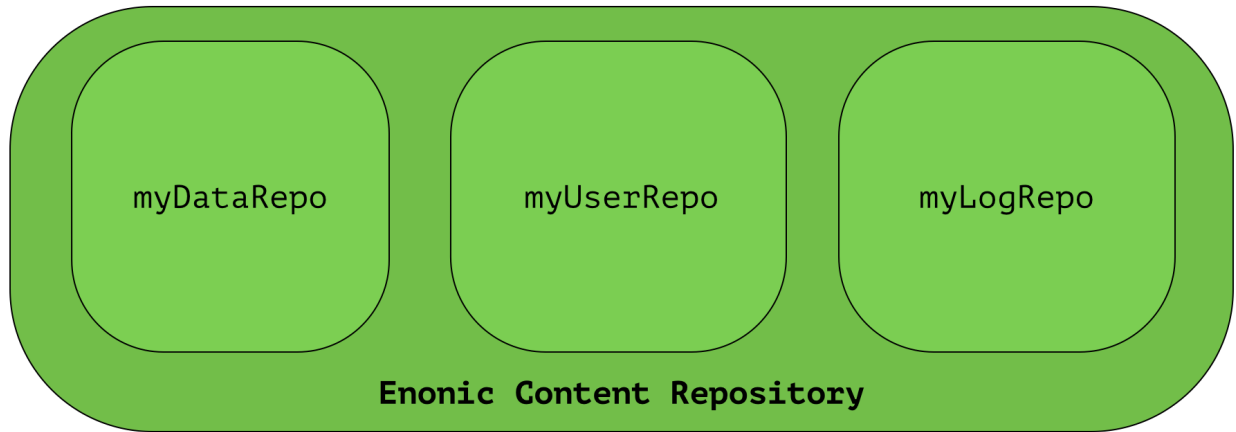
We were unable to find any single solution that was sufficiently simple and included our desired feature set - so we decided to build our own; the Enonic Content Repository.

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.

### 2.13.2 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”:

```
_id = 1001
_name = 'oslo'
_parent = ROOT

displayName = 'Oslo'
data.population = 647676
data.area = 454.03
location = geoPoint('59.9127300,10.7460900')
```

```
_id = 102131
_name = 'enonic'
_parent = '/oslo'

displayName = 'Enonic'
category = 'Software company'
employees = 20
```

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.

### 2.13.3 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")
```

### 2.13.4 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.

### 2.13.5 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.

### 2.13.6 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.

#### Node versions and branches

Consider the ‘Oslo’ and ‘Enonic’ nodes from earlier sections:

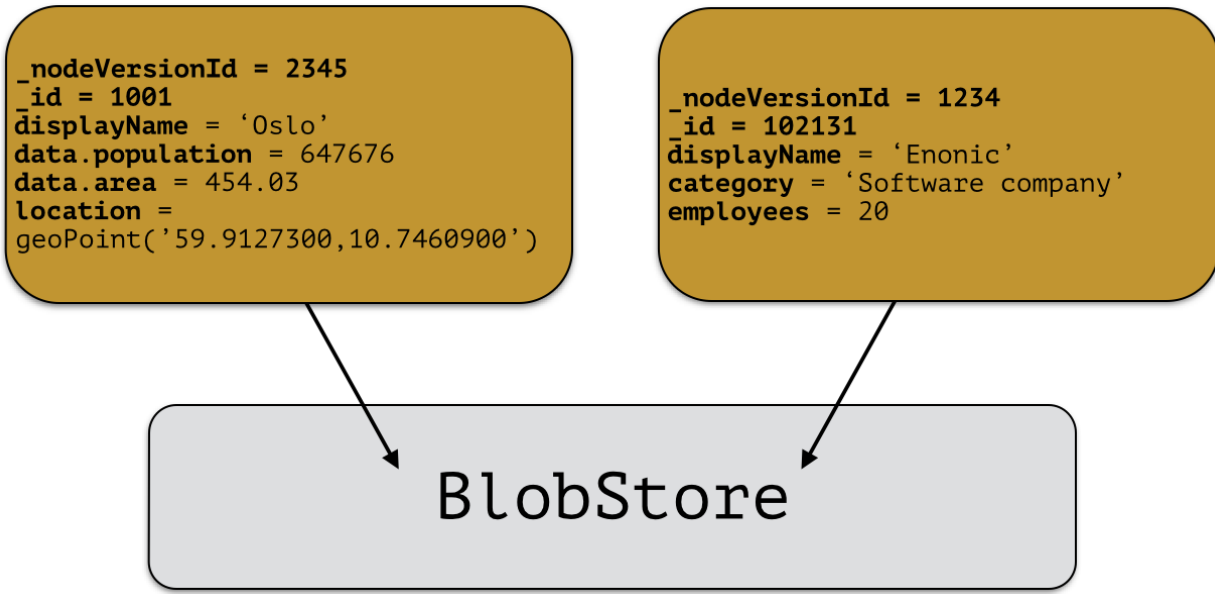
```
_id = 1001
_name = 'oslo'
_parent = ROOT

displayName = 'Oslo'
data.population = 647676
data.area = 454.03
location = geoPoint('59.9127300,10.7460900')
```

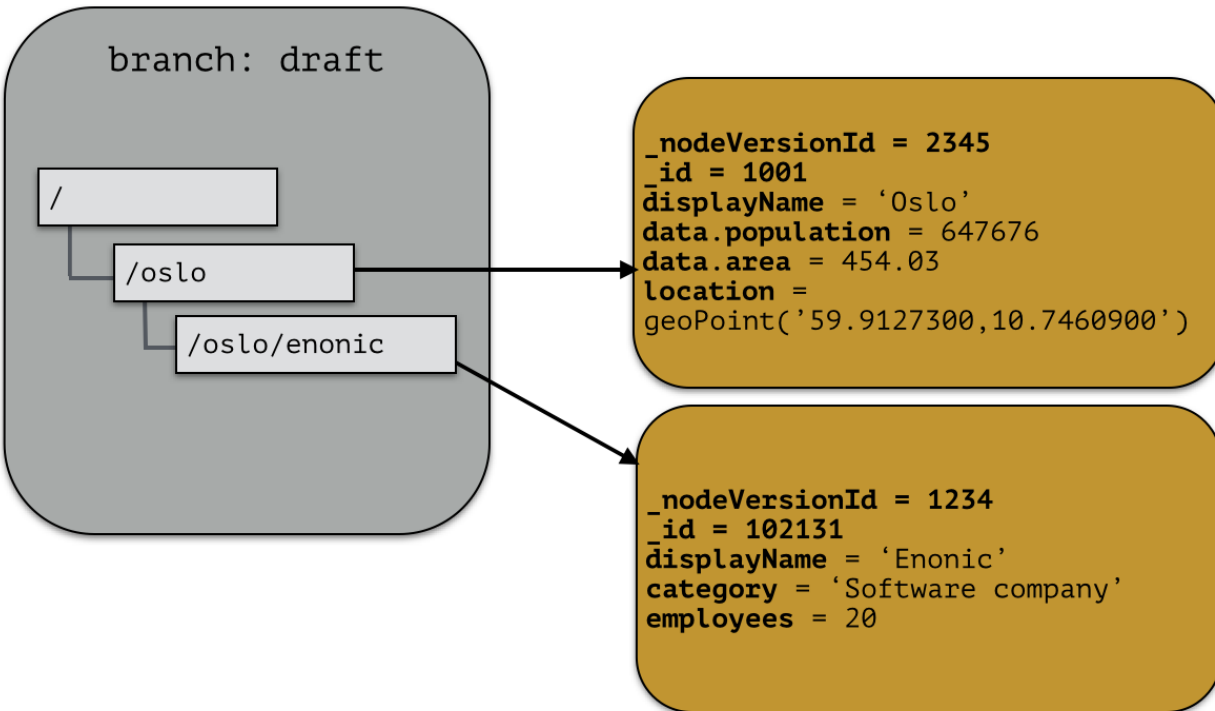
```
_id = 102131
_name = 'enonic'
_parent = '/oslo'

displayName = 'Enonic'
category = 'Software company'
employees = 20
```

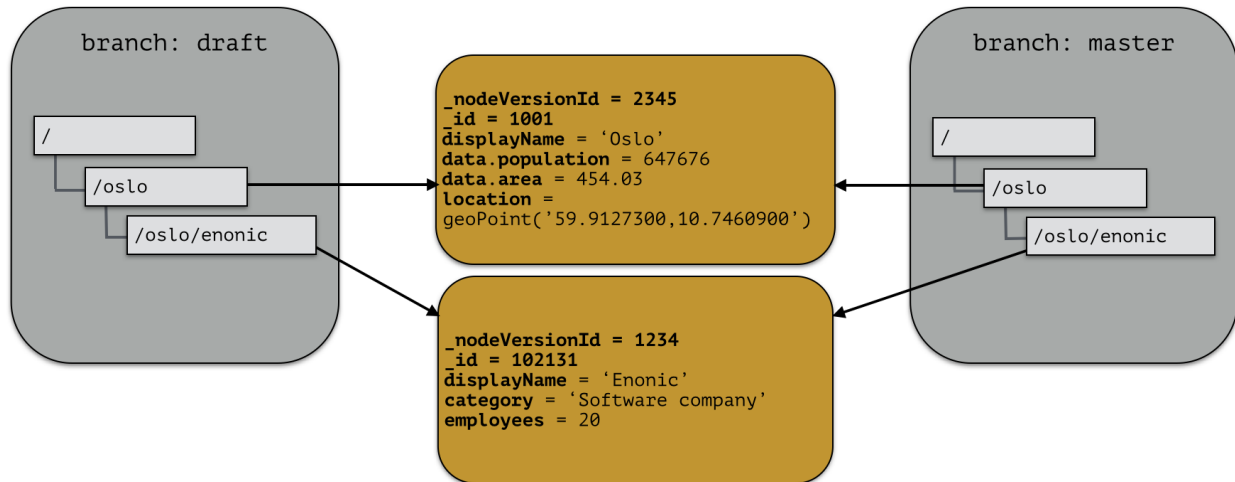
There will be two *node-versions* in the repository stored in the blobstore:



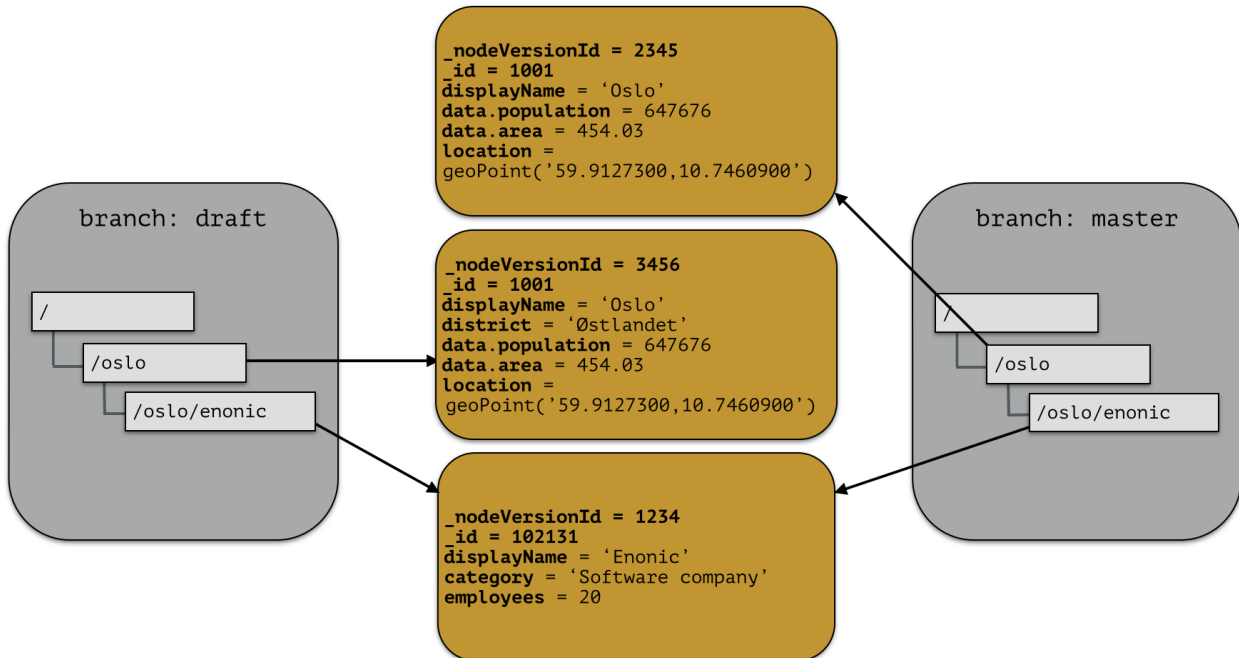
A node-version is a representation of a node's properties. A node-version has no knowledge of name, parent or other meta-data: just the properties of a node. At the same time, the targeted branch (named 'draft' in this example) gets two entries:



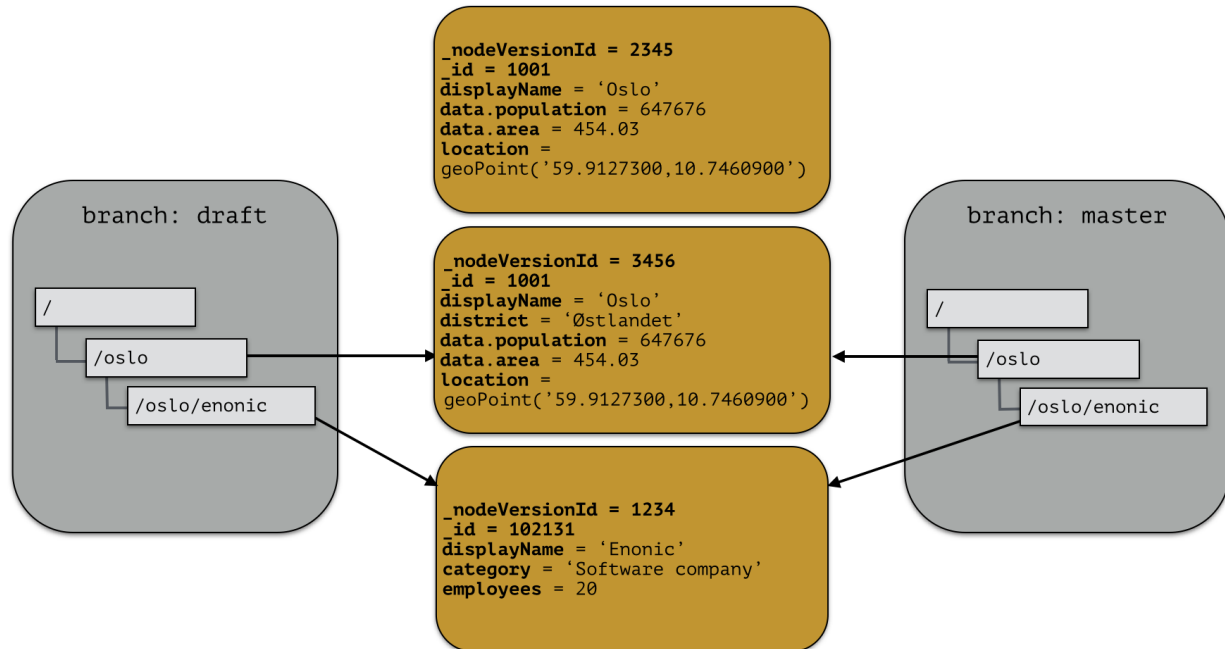
The node-versions are now a part of a tree-structure, based on the node's name and parent. If we *push* the content of branch 'draft' to the default branch 'master', we end up with something like this:



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:



## 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.

### 2.13.7 Blobstore

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

## 2.14 Content Domain

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



## 2.14.1 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.

## 2.14.2 Content manager

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

The screenshot displays the Enonic XP Content Manager interface. At the top, there is a search bar and a navigation menu with options: New, Preview, Edit, Delete, Duplicate, Move, Sort, and Publish. Below the menu, a table lists various content types with their status and last modified dates. The 'Superhero' content type is selected and highlighted in blue.

Content Type	Status	Last Modified
Folder (215)		
Author (2)		
Category (5)		
Comment (5)		
Landing page (1)		
Post (16)		
RSS page (1)		
City (6)		
Audio (1)		
Image (15)		
Page-template (8)		
Site (3)		
Template-folder (3)		
Features /features	New	2015-05-18 13:11:39
Large tree /large-tree	New	2015-05-18 13:11:33
<b>Superhero /superhero</b>	<b>Online</b>	<b>2015-05-18 09:36:06</b>
Search search	Online	2015-04-09 09:09:47
Entries RSS entries-rss	New	2015-03-03 03:12:21
Posts posts	Online	2015-02-28 00:49:37
Comments comments	Online	2015-02-28 00:49:29
Categories categories	Online	2015-02-28 00:49:20
Authors authors	Online	2015-02-28 00:48:59
Templates _templates	Online	2015-02-28 00:46:39
Xslt /xslt	New	2015-05-18 13:11:39

Below the table, a preview of the 'Superhero' WordPress theme is shown. The theme is described as 'Superhero WordPress theme for Enonic XP'. The preview features a large image of a city street with a search bar and a 'Search' button. A post titled 'March Madness' is visible, posted on March 4.

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 left side.

### 2.14.3 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.

### 2.14.4 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

#### Base Content Types

A set of basic content types are provided with the installation.

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

- Content types are named with their application name, i.e. `base:folder`, where “base” is the application - 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 from creating child items on 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 habitat are considered “files”, these are listed below

### Unstructured (base:unstructured)

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

The unstructured content type is a special content type that permits the creation of any property or structure - without actually defining it 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

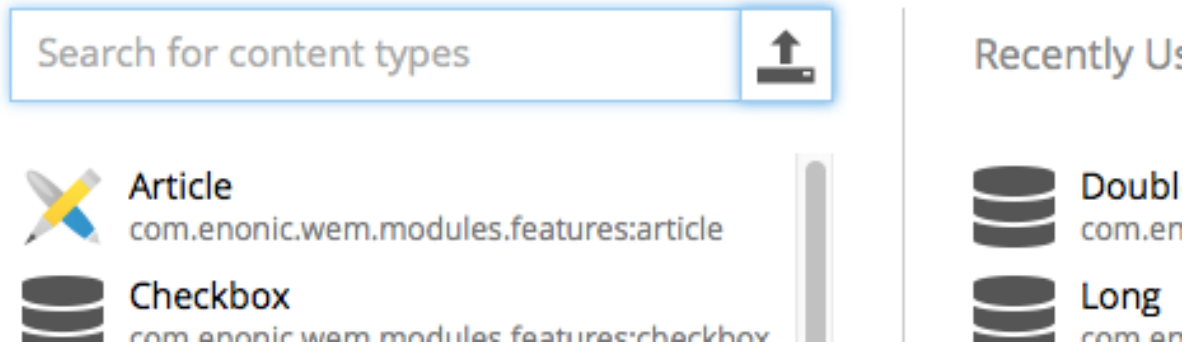
- super-type: base:media
- abstract: false
- final: true
- allow-child-content: false

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).

# Create Content

In: /archive/long-value



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.

## 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 allows creating websites. By creating a content of type Site, it will become the root of a website.

This content type provides an special behavior for the content, allowing to select and configure applications for the website. The types (content types, relationship types and mixins) of the applications selected will be available to be used inside the website content tree.

---

**Note:** The content types of an application can only be used under a content of type Site which has the application selected.

---

### 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`.

## Custom Content Types

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

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

Each folder must then hold a file with the name of the content type and `.xml` extension (e.g. `my-content-type-name.xml`).

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.

---

**Tip:** A content type may optionally have its own specific icon. The icon can be assigned to the content type by adding a PNG file with the same name, in the content type folder, e.g. `site/content-types/my-content-type-name/my-content-type-name.png`

---

## 2.14.5 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"/>
  <config/>
</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 is not currently in use either. 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.

**config** Optional configuration that is used by some of the input-types. The config consists of elements with optional attributes. Each element/attribute name with dashes is automatically camel-cased (`relationship-type` -> `relationshipType`).

## 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.

```
<input name="name" type="CheckBox">
  <label>Required</label>
  <occurrences minimum="0" maximum="1"/>
</input>
```

## ComboBox

A ComboBox needs a list of options.

```
<input name="name" type="ComboBox">
  <label>Required</label>
  <occurrences minimum="1" maximum="1"/>
  <config>
    <option value="one">Option One</option>
    <option value="two">Option Two</option>
  </config>
</input>
```

**option** This element defines the option label. `value` attribute defines the actual value to set when this option is selected. Multiple `option` settings are ordered.

## Date

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

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

```
<config>
  <timezone>true</timezone>
</config>
</input>
```

**timezone** true if timezone information should be used. Default is false.

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

**timezone** true if timezone information should be used. Default is false.

## Double

A double value input-type.

## 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.

## HtmlArea

A field for entering html in a WYSIWYG HTML editor.

## 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>
```



## Long

A simple field for large integers.

## ContentSelector

References to other content are specified by this input type.

```

<input name="name" 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>
    <allow-content-type>my.other.app:quote</allow-content-type>
  </config>
</input>

```

**relationship-type** This setting defines the name of which relationship-type to use. Default is `system:reference`.

**allow-content-type** This is used to limit the content types that may be selected for this input. Use one setting for each content-type.

## RadioButton

An input type for selecting one of several options, defined in the `config` element.

```

<input name="name" type="RadioButton">
  <label>Radio Buttons</label>
  <occurrences minimum="0" maximum="0"/>
  <config>
    <option value="one">Option One</option>
    <option value="two">Option Two</option>
  </config>
</input>

```

**option** This element defines the option label. `value` attribute defines the actual value to set when this option is selected. Multiple `option` settings are ordered.

## Tag

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

## TextArea

A field for inputting multi-line text.

## TextLine

A field for inputting a single line of text.

### Time

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

### 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.

### 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:

```
<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" />
</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.

---

## 2.14.6 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, `site/relationship-types/[name]` and be named `[name].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.

---

**Tip:** A relationship type may optionally have its own specific icon. The icon can be assigned to the relationship type by adding a PNG file with the same name, in the relationship type folder, i.e. `site/relationship-types/[name]/[name].png`

---

### 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”.

## 2.14.7 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 placed in the folder `site/mixins/[name]` and named `[name].xml`. For example, `site/mixins/us-address/us-address.xml`.

```

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

```

**Tip:** A mixin may optionally have its own specific icon. The icon can be assigned to the mixin by adding a PNG file with the same name, in the mixin folder, e.g. `site/mixins/us-address/us-address.png`

## Using a mixin

Below is an example of a simple content type that uses the `us-address` mixin (inline) and the `menu-item` mixin (`x-data`). Notice that the name of the mixin file is used and not the mixin's Display Name.

```

<content-type>
  <display-name>Using mixins</display-name>
  <super-type>base:structured</super-type>
  <form>
    <field-set name="basic">
      <label>Status</label>
      <items>
        <inline mixin="us-address"/>
      </items>
    </field-set>
  </form>
  <x-data mixin="menu-item"/>
</content-type>

```

**inline** When a mixin is added with the `inline` element, the inputs will be included with the content data. Inline mixins can be used in content types, component descriptors, and the `site.xml` file.

**x-data** Mixins can also be added with an `x-data` element in content types and the `site.xml` file. Be aware that any `x-data` inputs added to the `site.xml` file will be applied to all content types in the site.

## 2.14.8 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'
```

## 2.14.9 Content Indexing

All content is indexed when stored. The properties of a content are 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

### 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 is 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. An 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.

## 2.15 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.

### 2.15.1 Overview

When searching in Enonic XP, you are searching for nodes, or content if working in the context of the CMS content API. 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-APIs deal with a number of basic parameters:

- start
- count
- query
- filter
- aggregations

#### Start & count

When searching, the result will contain a number of matching nodes. This number is 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 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.

### 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.

### 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.

### 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.

## 2.15.2 Query Functions

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

### 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 occurrences, 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")
```

### 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")
```

## 2.15.3 Order Functions

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

### 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")
```

## 2.15.4 Aggregations

An aggregation is a function 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 have 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 calculating the average manually, but this would be very inefficient and cumbersome. Luckily, aggregations solve these types of problems easily.

In some API functions it is possible to send in an 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 aggregations:** A metric aggregation 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.

### terms

The ‘terms’ aggregation 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 a result with this structure:

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

## 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 a result with this structure:

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

## dateRange

The `dateRange` aggregation query defines a set of date-ranges that represents a bucket. Only documents with properties of type 'DateTime' will be considered in the `dateRange` 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 `dateRange` aggregation:

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

The above example gives a result with this structure:

```
"price_ranges": {
  "buckets": [{
    "docCount": 2,
```

```
    "key": "a",
    "to": 50
  },
  {
    "docCount": 4,
    "from": 50,
    "key": "b",
    "to": 100
  },
  {
    "docCount": 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
```

### dateHistogram

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 `dateHistogram` 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  $\leq$  `minDocCount`.

Here's an example of the `date_histogram` aggregation:

```

"by_month": {
  "dateHistogram": {
    "field": "init_date",
    "interval": "1M",
    "minDocCount": 0,
    "format": "MM-yyy"
  }
}

```

The above example gives a result with this structure:

```

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

```

## 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 a result with this structure:

```

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

```

## geo-distance

The `geo_distance` aggregation needs a defined range to split the documents into buckets. Only documents with properties of type 'GeoPoint' will be considered in the `geo_distance` aggregation buckets.

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.

**unit (string)** The measurement unit to use for the ranges. Legal values are either the full name or the abbreviation of the following: km (kilometers), m (meters), cm (centimeters), mm (millimeters), mi (miles), yd (yards), ft (feet)



or nmi (nauticalmiles).

**origin (lat: number, lon: number)** The GeoPoint from which the distance is measured.

Here's an example of the range aggregation:

```
"aggregations": {
  "distance": {
    "geo_distance": {
      'field': "data.cityLocation",
      'unit': "km",
      'origin': {
        'lat': "90.0",
        'lon': "0.0"
      },
      'ranges': [ { 'from': 0, 'to': 1200 }, { 'from': 1200, 'to': 4000 }, { 'from': 4000,
```

The above example gives a result with this structure:

```
"aggregations":
  {"distance":
    {"buckets": [{
      "key": "*-1200.0",
      "doc_count": 3,
    },
    {
      "key": "1200.0-4000.0",
      "doc_count": 4,
    },
    {
      "key": "4000.0-12000.0",
      "doc_count": 5,
    },
    {
      "key": "12000.0-*",
      "doc_count": 1,
    }],}}
}
```

NOTE: At the time of writing, there is only one way of find out which result belongs to which bucket: By also sorting the result on geo\_distance, and matching the order to the number of each bucket. In a future version, there will easier ways of doing this.

### 2.15.5 Querying date and time

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

#### 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'
```

### 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.

### 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')
```

## 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')
```

## 2.15.6 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.

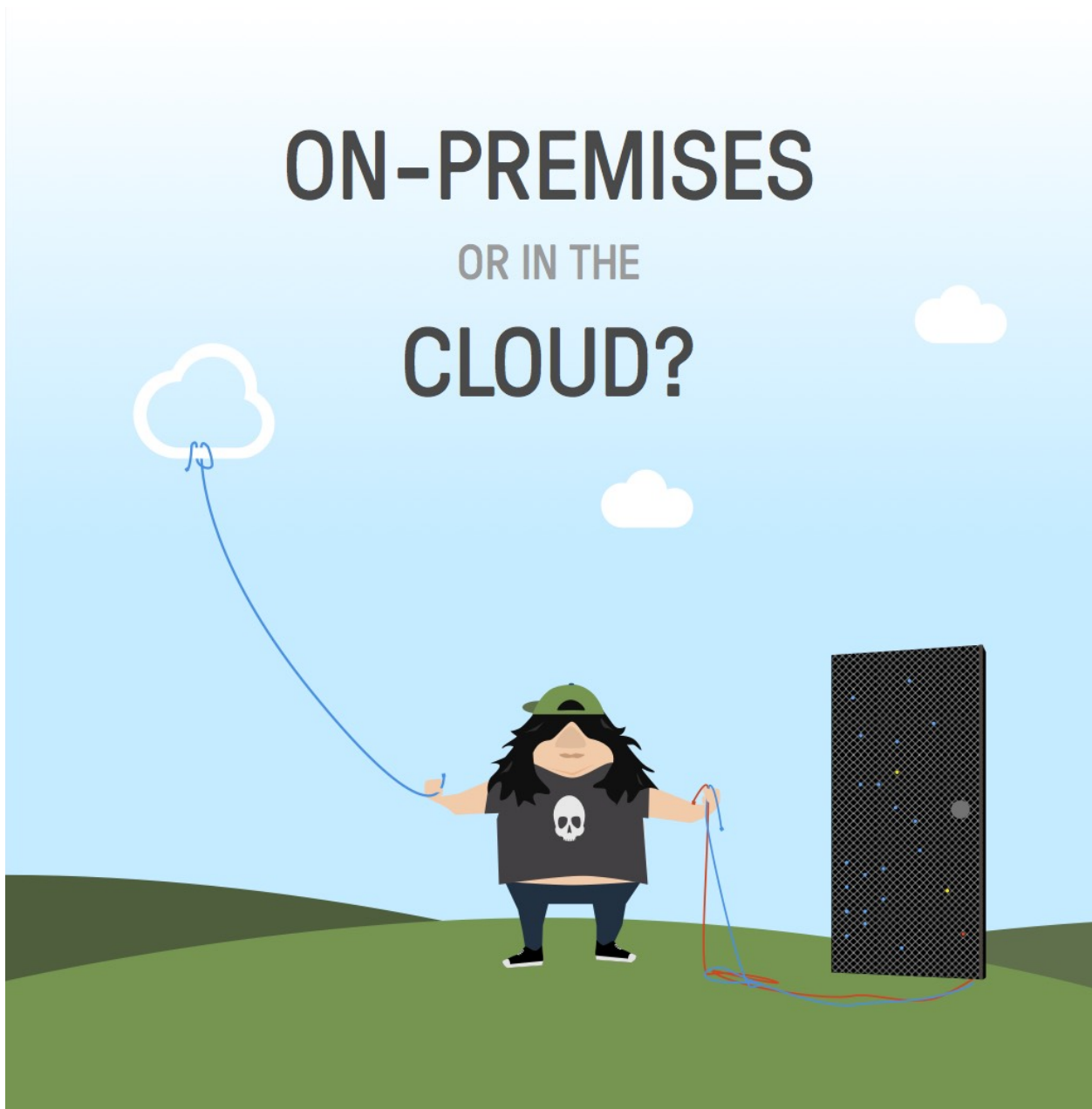
```
_parentPath = '/content/mySite/myCategory'
```

---

## Operations Guide

---

This guide gives you all the gory details on how to tune Enonic XP - if you're looking for installation guides try [Getting Started Guide](#)



### 3.1 Package Structure

The unzipped Enonic XP distribution will have the following structure (folders with a \* will not appear until the installation is started the first time)

```
enonic-xp-[version]
|- bin/
|- home/
  |- config/
  |- data/ *
  |- deploy/
  |- logs/
```

```

|- repo/ *
|- work/ *
|- lib/
|- system/
|- toolbox/
|- work/ *

```

The 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 and setting environment variables.

**home/** Home directory, also called `XP_HOME`. All files for a specific instance of XP reside here. This folder can be copied to other locations for working with multiple projects.

**config/** Configuration files are placed here, including Virtual Host and `system.properties`.

**data/** Additional data like exports, snapshots and dumps. This folder will not appear until certain operations are run.

**deploy/** Hot deploy directory. Applications are automatically installed upon placing their JAR files in this directory.

**logs/** Default location for logs.

**repo/** Repository data (blobs and indexes). This folder will not appear until the installation is started for the first time.

**work/** Cache and generated bundles are stored here. This folder will not appear until the installation is started the first time.

**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*.

**work/** OSGI cache is stored here. This folder will not appear until the installation is started for the first time.

## 3.2 Configuration

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

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

### 3.2.1 System Configuration

The default `system.properties` are listed below.

Listing 3.1: `$XP_HOME/config/system.properties`

```

#
# 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
```

### 3.2.2 Virtual Host Configuration

Virtual hosts have their own configuration file and settings are automatically updated upon changes. A sample virtual host configuration is listed below.

Listing 3.2: `$XP_HOME/config/com.enonic.xp.web.vhost.cfg`

```
enabled = true

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

mapping.intranet.host = enonic.com
mapping.intranet.source = /
mapping.intranet.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 “intranet”, 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`.

### 3.2.3 Mail Configuration

The mail server used for sending email messages using the *lib-mail* API can be configured. A sample mail configuration is listed below.



Listing 3.3: `$XP_HOME/config/com.enonic.xp.mail.cfg`

```
smtpHost=mail.server.com
smtpPort=25
smtpAuth=true
smtpUser=user
smtpPassword=secret
smtpTLS=false
```

**smtpHost** Host name of the SMTP server. Default `localhost`.

**smtpPort** TCP port of the SMTP server. Default `25`.

**smtpAuth** Enable authentication with the SMTP server. Default `false`.

**smtpUser** User to be used during authentication with the SMTP server, if 'smtpAuth' is set to true.

**smtpPassword** Password to be used during authentication with the SMTP server, if 'smtpAuth' is set to true.

**smtpTLS** Turn on Transport Layer Security (TLS) security for SMTP servers that require it. Default `false`.

### 3.2.4 Elasticsearch Configuration

Elasticsearch settings can be configured. When changing `com.enonic.xp.elasticsearch.cfg`, the Elasticsearch node will automatically restart with the new configuration

Listing 3.4: `$XP_HOME/config/com.enonic.xp.elasticsearch.cfg`

```
name = local-node
client = false
data = true
local = true
http.enabled =true
cluster.name = mycluster
network.host = 127.0.0.1
discovery.zen.ping.multicast.enabled = false
cluster.routing.allocation.disk.threshold_enabled = false

path = ${xp.home}/repo/index
path.data = ${path}/data
path.work = ${path}/work
path.conf = ${path}/conf
path.logs = ${path}/logs
path.plugins = ${path}/plugins
```

### 3.2.5 Jetty HTTP Configuration

Jetty HTTP settings can be configured using `com.enonic.xp.web.jetty.cfg` file.

Listing 3.5: `$XP_HOME/config/com.enonic.xp.web.jetty.cfg`

```
# Set this if host name (or ip) needs to be fixed.
host =

# Socket timeout for connections.
timeout = 60000
```

```
# True to send server header.
sendServerHeader = false

# True to enable HTTP connections.
http.enabled = true

# Http port number to use.
http.port = 8080

# Session timeout (when inactive) in minutes.
session.timeout = 60

# Cookie name to use for sessions.
session.cookieName = JSESSIONID

# Enable GZIP compression for responses.
gzip.enabled = true

# Minimum number of bytes in response to consider compressing the response.
gzip.minSize = 16

# True to enable request logging.
log.enabled = false

# Request log file.
log.file = ${xp.home}/logs/jetty-yyyy_mm_dd.request.log

# True to append to the file, or create new one when started.
log.append = true

# True to use extended format.
log.extended = true

# Timezone to display timestamp in.
log.timeZone = GMT

# Number of days to retain the logs.
log.retainDays = 31
```

## 3.3 Backup and Restore

Backing up your data is vital for any installation.

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.

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

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

### 3.3.1 Backup vs Export

The export/import-API enables you to dump your data to a serialized format. The serialized data could then be imported into another instance. This is very useful, but is not optimal for a backup/restore scheme since it requires some work to get things up and running again, especially when working with big installations with a lot of data. The backup/restore-process described below on the other hand, should enable a quick and safe way to get your system back to operation when in a hurry.

See *Export and Import* for more information on export/import.

### 3.3.2 Backing up blobs

The blobs are just files on a filesystem. This should be backed up by your preferred way of doing file-backups.

The folder to backup is:

- `$XP_HOME/repo/blobs`

### 3.3.3 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. To help you out, we have a snapshot-API. A snapshot is exactly that; a snapshot of the indices state at a point of time. There are 4 rest-resources at your disposal.

**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 snapshot or a group of snapshots.

#### 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.

#### 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 is 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.

## 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 *delete-snapshots* tool.

## 3.4 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 tools (*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 same server running Enonic XP.

### 3.4.1 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.

### 3.4.2 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.

### 3.4.3 Export data structure

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

<input type="checkbox"/>		Demo site /demo-site	☰	New
<input type="checkbox"/>		Case studies case-studies		New
<input type="checkbox"/>		Powered by sites powered-by-sites		New
<input type="checkbox"/>		Enonic.com enonic-com		New
<input type="checkbox"/>		A demo case study a-demo-case-study		New
<input type="checkbox"/>		Enonic man.png enonic man.png		New
<input type="checkbox"/>		Enonic.com iphone.png enonic.com iphone.png		New
<input type="checkbox"/>		Enonic.com desktop.png enonic.com desktop.png		New
<input type="checkbox"/>		Contact Enonic contact-enonic		New
<input type="checkbox"/>		Enonic Office enonic-office		New
<input type="checkbox"/>		Enonic in Oslo.jpg enonic in oslo.jpg		New
<input type="checkbox"/>		Templates _templates		New
<input type="checkbox"/>		Case study show case-study-show		New
<input type="checkbox"/>		Landing page landing-page		New

Run the export command:

```
$ ./toolbox.sh export -a su:password -s cms-repo:draft:/ -t myExport
```

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.

### 3.4.4 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:

```
$ ./toolbox.sh import -a 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.

## 3.5 Monitoring

We provide some basic metrics for monitoring easily accessed in a simple JSON format. To access the monitoring JSON feed you can point to the following url:

```
http://localhost:8080/status
```

This will give you information about Enonic XP version, JVM metrics, memory usages and OS information.

```
{
  "xp":{
    "version":"6.0.0-SNAPSHOT",
    "build":{
      "hash":"ea44484449603a138a4df87203433bbc43efa31a",
      "shortHash":"ea44484",
      "branch":"master",
      "timestamp":"2015-08-21T04:41:48-0700"
    },
    "installation":"demo"
  },
  "os":{
    "name":"Mac OS X",
    "version":"10.10.5",
    "arch":"x86_64",
    "cores":8,
    "loadAverage":1.64306640625
  },
  "jvm":{
    "name":"Java HotSpot(TM) 64-Bit Server VM",
    "vendor":"Oracle Corporation",
    "version":"25.51-b03",
    "startTime":1440196338867,
    "upTime":89314660
  },
  "memory":{
    "heap":{
      "init":1073741824,
      "max":1908932608,
      "committed":1397751808,
      "used":414455456
    }
  },
}
```

```
"nonHeap":{
  "init":2555904,
  "max":-1,
  "committed":168714240,
  "used":161517768
},
"gc":{
  "collectionTime":402,
  "collectionCount":16
}
}
```

### 3.6 Troubleshooting

This document is an up-to-date list of known problems (and how to fix them) for our current release and development releases.

#### 3.6.1 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 the `JAVA_HOME` environment variable 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.

Check the `troubleshooting_java` page for more Java help.

#### 3.6.2 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.

If shutting down other software that uses port 8080 is not an option, you may set a different port for Enonic XP. See *Configuration*.

#### 3.6.3 Unexpected behavior

While frequently redeploying an app during development, some instability or unexpected behavior may be noticed. This can be caused by certain changes to the app files. For example, changing the app name in the `build.gradle` file, or deleting content import `node.xml` files. When this occurs, the project may need a clean build `gradle clean`



build. Sometimes the app JAR file may need to be deleted from the \$XP\_HOME/deploy directory as well, and then replaced with the clean build JAR file.

### 3.6.4 Cannot login after install

There could be a problem with file permissions on Windows if Enonic XP was unzipped and started from within the “My Documents” folder. This may allow XP to start, but the users cannot log in. The solution would be to unzip the Enonic XP distribution outside of the “My Documents” folder, or to manually change the file permissions.

### 3.6.5 Sending email with lib-mail not working

Check the installation’s *Mail Configuration*.





API and Reference Guide



Every *Page*, *Part*, *Layout* and *Service* must have a controller which consists of a JavaScript file named `[component-name].js` where *component-name* is also the name of the folder that the controller resides in. For example, the controller for a part named “things” would be `site/parts/things/things.js`.

### 4.1.1 App

The globally available `app` object holds information about the contextual app it was delivered from.

**app.name** Name of the application.

**app.version** Version of the application.

Examples:

```
// Get application name
var name = app.name;

// Get application version
var version = app.version;
```

### 4.1.2 Log

This globally available `log` 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 an 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 an 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');

// Log using the built-in JSON converter
log.debug('My JSON %s', object );
```

### 4.1.3 Resolve()

The globally available 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('/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');
```

### 4.1.4 Require()

The globally available function will load a JavaScript file and return the exports as objects. 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('/lib/mylib.js');

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

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

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

If the path is relative then it will start looking for the file from the local directory. If the file is not found there, it will start looking from the *site/lib* directory. The file extension *.js* is not required.

### 4.1.5 Exports

The globally available `exports` keyword is used to expose functionality from a given Javascript file (controllers, libraries etc). This is part of the `require.js` spec.

Simply use the `exports` keyword to expose functionality from any javascript file.

#### Controller Methods

Javascript controllers specifically are invoked from the portal by exporting functions matching the desired HTTP Method it implements. As such, any controller must explicitly declare one or more export in order to handle requests: GET, POST, DELETE are examples of such methods.

For every request sent to the controller, the appropriate function will be called.

Example usage

```
// 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'
  };
};
```

### 4.1.6 Request

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

```
{
  "method": "GET",
  "scheme": "http",
  "host": "enonic.com",
```

```
"port": "80",
"path": "/my/page",
"url": "http://enonic.com/my/page?debug=true",
"mode": "edit",
"branch": "master",
"params": {
  "debug": "true"
},
"headers": {
  "Language": "en",
  "Cookies": "mycookie=123; other=abc;"
},
"cookies": {
  "mycookie": "123",
  "other": "abc"
}
}
```

**method** HTTP method of the request.

**scheme** Scheme of the request.

**host** Host of the request.

**port** Port of the request.

**path** Path of the request.

**url** URL of the request.

**mode** Portal rendering mode, one of: edit, preview, live.

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

**body** Optional text value

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

**headers** Name/value pairs with the HTTP request headers.

**cookies** Name/value pairs with the HTTP request cookies.

### 4.1.7 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"
  },
  "cookies": {},
  "redirect": "/another/page",
  "pageContributions": {},
  "postProcess": true,
  "applyFilters": true
}
```

**status** HTTP response status code (default is 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.

**cookies** HTTP cookies to be added to the response. Will be described in a later section.

**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.

**postProcess** Whether or not execute the post-processing step after rendering. Post-processing is necessary for page contributions and rendering components in a page. (See also *Page Contributions*) (default is `true`).

**applyFilters** Whether or not execute the filters after rendering. Set to `false` to skip execution of filters. (See also *Response Filters*) (default is `true`).

### 4.1.8 Cookies

Http Cookie values can be set in responses using two different ways.

- If the value is just a string, then the cookie is created using default settings.
- If the value is an object, then it will try to set the settings. Every field is optional except value.

Here’s an example of how the cookies should be set:

```
return {
  status: 200,
  body: "Hello World",
  headers: {
    "header1": "value1"
  },
  cookies: {
    "plain": "value",
    "complex": {
      value: "value",
      path: "/valid/path",
      domain: "enonic.com",
      comment: "Some cookie comments",
      maxAge: 2000,
      secure: false,
      httpOnly: false
    }
  }
};
```

### 4.1.9 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 with 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 four positions where contributed content can be inserted in the page:

- `headBegin`: After the `<head>` opening tag.
- `headEnd`: Before 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 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 two 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 in both places.
- 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.

## 4.2 Javascript Libraries

This section describes the various standard libraries shipped with Enonic XP. The libraries are included in your application through the Gradle build script.

### 4.2.1 lib-auth

This library implements methods for dealing with authentication and the currently authenticated user. Add the following into your `build.gradle` file in the `dependencies` section:

```
include 'com.enonic.xp:lib-auth:6.2.0'
```

To use this library in your JavaScript files, you can require the library like this:

```
var auth = require('/lib/xp/auth');
```

The methods implemented in this library are listed below.

## login

This function verifies the credentials for a user and associates it with the current session.

**login** (*params*)

### Arguments

- **params** (*object*) – JSON with the parameters below.

**Returns** whether the credentials were valid and the authenticated user.

### Parameters:

**user** (*string*) The username or email of the user to be authenticated.

**password** (*string*) The password of the user to be authenticated.

**userStore** (*string*) The user store or list of user stores where to look the user up. Optional. If not specified, the user will be looked up in all existing user stores.

### Examples:

```
var auth = require('/lib/xp/auth');

// attempt to login with a explicit user store
var result = auth.login({
  user: 'user1@enonic.com',
  password: 'secret',
  userStore: 'enonic'
});

if (result.authenticated) {
  log.info('User logged in: %s', result.user.displayName);
}
```

```
var auth = require('/lib/xp/auth');

// attempt to login to any of the user stores, in sequence
var result = auth.login({
  user: 'user1@enonic.com',
  password: 'secret',
  userStore: ['enonic', 'vip']
});
```

```
var auth = require('/lib/xp/auth');

// attempt to login to any of the existing user stores
var result = auth.login({
  user: 'user1@enonic.com',
  password: 'secret'
});
```

### Result:

```
{
  authenticated: true,
  user: {
    disabled: false,
    displayName: 'User 1',
    email: 'user1@enonic.com',
    key: 'user:enonic:user1',
    login: 'user1',
    userStore: 'enonic',
    modifiedTime: '2015-01-10T00:00:00Z'
  }
}
```

### logout

This function logs the current user out.

**logout()**

#### Example:

```
var auth = require('/lib/xp/auth');
auth.logout();
```

### getUser

This function gets the current logged in user.

**getUser()**

**Returns** the user logged in, or *null* if there is no user currently logged in.

#### Example:

```
var auth = require('/lib/xp/auth');
var user = auth.getUser();

if (user) {
  log.info('User logged in: %s', user.displayName);
}
```

### Result:

```
{
  disabled: false,
  displayName: 'User 1',
  email: 'user1@enonic.com',
  key: 'user:enonic:user1',
  login: 'user1',
  userStore: 'enonic',
  modifiedTime: '2015-01-10T00:00:00Z'
}
```

## hasRole

This function checks if the current logged in user has a given role.

**hasRole** (*roleKey*)

### Arguments

- **roleKey** (*string*) – role to be checked.

**Returns** whether the current user has the role specified. Or *false* if no user is logged in.

### Example:

```
var auth = require('/lib/xp/auth');
var hasAdminAccess = auth.hasRole('system.admin.login');
log.info('The user ' + (hasAdminAccess ? 'has' : 'does not have') + ' access to the admin console.');
```

## generatePassword

This function generates a strong password.

**generatePassword** ()

**Returns** a generated password.

### Example:

```
var auth = require('/lib/xp/auth');
var pwd = auth.generatePassword();
log.info('New password: %s', pwd);
```

## changePassword

This function changes the password for a user.

**changePassword** (*params*)

### Arguments

- **params** (*object*) – JSON with the parameters below.

### Parameters:

**userKey** (*string*) Key for the user to change password for.

**password** (*string*) New password to change to.

### Example:

```
var auth = require('/lib/xp/auth');
var user = auth.getUser();

auth.changePassword({
  userKey: user.key,
  password: 'new-secret-password'
});
```

## 4.2.2 lib-content

This library implements JavaScript bindings for standard content-related functionality. Add the following into your `build.gradle` file in the `dependencies` section:

```
include 'com.enonic.xp:lib-content:6.1.0'
```

To use this library in your JavaScript files, you can require the library like this:

```
var contentLib = require('/lib/xp/content')
```

The methods implemented in this library are listed below.

### get

This function fetches a content.

**get** (*params*)

#### Arguments

- **params** (*object*) – JSON with the parameters below.

**Returns** the content fetched from the repository.

#### Parameters:

**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 contentLib = require('/lib/xp/content');

var result = contentLib.get({
  key: '/features/js-libraries/mycontent',
  branch: 'draft'
});

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

#### Result:

```
{
  "_id": "233f668f-103d-4c4f-86ac-e1ce7a166749",
  "_name": "mycontent",
  "_path": "/features/js-libraries/mycontent",
  "creator": "user:system:su",
  "modifier": "user:system:su",
  "createdTime": "2015-10-05T13:27:10.873Z",
  "modifiedTime": "2015-10-05T13:27:10.873Z",
  "type": "com.enonic.app.features:all-input-types",
  "displayName": "My Content",
  "hasChildren": false,
  "language": "no",
```

```

"valid": true,
"data": {
  "myCheckbox": true,
  "myComboBox": "option1",
  "myDate": "1970-01-01",
  "myDateTime": "1970-01-01T10:00",
  "myDouble": 3.14,
  "myGeoPoint": "59.91,10.75",
  "myHtmlArea": "<p>htmlAreaContent</p>",
  "myImageSelector": "5a5fc786-a4e6-4a4d-a21a-19ac6fd4784b",
  "myLong": 123,
  "myRelationship": "features",
  "myRadioButtons": "option1",
  "myTag": "aTag",
  "myTextArea": "textAreaContent",
  "myTextLine": "textLineContent",
  "myTime": "10:00",
  "myTextAreas": [
    "textAreaContent1",
    "textAreaContent2"
  ],
  "myItemSet": {
    "textLine": "textLineContent",
    "long": 123
  }
},
"x": {
  "com-enonic-app-features": {
    "menu-item": {
      "menuItem": true
    }
  }
},
"page": {}
}

```

## getChildren

This function fetches children of a content.

**getChildren** (*params*)

### Arguments

- **params** (*object*) – JSON with the parameters below.

**Returns** the content fetched from the repository.

### Parameters:

**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 contentLib = require('/lib/xp/content');

var result = contentLib.getChildren({
  key: '/features/js-libraries/houses',
  start: 0,
  count: 2,
  sort: '_modifiedTime ASC',
  branch: 'draft'
});

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

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

```

**Result:**

```

{
  "total": 6,
  "count": 2,
  "hits": [
    {
      "_id": "9922a270-f881-4bf8-be35-189e9a72a4f1",
      "_name": "house6",
      "_path": "/features/js-libraries/houses/house6",
      "creator": "user:system:su",
      "modifier": "user:system:su",
      "createdTime": "2015-10-05T12:09:41.998Z",
      "modifiedTime": "2015-10-05T12:25:42.469Z",
      "type": "com.enonic.app.features:house",
      "displayName": "House6",
      "hasChildren": false,
      "valid": true,
      "data": {
        "city": "Oslo",
        "location": "59.92,10.74",
        "price": 3400000,
        "number_floor": 3,
        "description": "House with garden",
        "publish_date": "2015-10-01"
      },
      "x": {
        "com-enonic-app-features": {
          "menu-item": {
            "menuItem": false
          }
        }
      },
      "page": {}
    },
    {
      "_id": "7c1006a0-5ca8-473e-b8dc-79f87c28e9be",
      "_name": "house4",
      "_path": "/features/js-libraries/houses/house4",
      "creator": "user:system:su",

```



```

"modifier": "user:system:su",
"createdTime": "2015-10-05T12:09:41.998Z",
"modifiedTime": "2015-10-05T12:25:28.596Z",
"type": "com.enonic.app.features:house",
"displayName": "House4",
"hasChildren": false,
"valid": true,
"data": {
  "city": "Oslo",
  "location": "59.91,10.78",
  "price": 2900000,
  "number_floor": 3,
  "description": "House with garden",
  "publish_date": "2015-10-03"
},
"x": {
  "com-enonic-app-features": {
    "menu-item": {
      "menuItem": false
    }
  }
},
"page": {}
}
]
}

```

## query

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

**query** (*params*)

### Arguments

- **params** (*object*) – JSON with the parameters below.

**Returns** the content found as JSON.

### Parameters:

**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 contentLib = require('/lib/xp/content');

var result = contentLib.query({
  start: 0,

```

```
count: 2,
sort: "modifiedTime DESC, geoDistance('data.location', '59.91,10.75')",
query: "data.city = 'Oslo' AND fulltext('data.description', 'garden', 'AND') ",
branch: "draft",
contentTypes: [
  app.name + ":house",
  app.name + ":apartment"
],
aggregations: {
  floors: {
    terms: {
      field: "data.number_floor",
      order: "_count asc"
    },
    aggregations: {
      prices: {
        histogram: {
          field: "data.price",
          interval: 1000000,
          extendedBoundMin: 1000000,
          extendedBoundMax: 3000000,
          minDocCount: 0,
          order: "_key desc"
        }
      }
    }
  },
  by_month: {
    dateHistogram: {
      field: "data.publish_date",
      interval: "1M",
      minDocCount: 0,
      format: "MM-yyyy"
    }
  },
  price_ranges: {
    range: {
      field: "data.price",
      ranges: [
        {to: 2000000},
        {from: 2000000, to: 3000000},
        {from: 3000000}
      ]
    }
  },
  my_date_range: {
    dateRange: {
      field: "data.publish_date",
      format: "MM-yyyy",
      ranges: [
        {to: "now-10M/M"},
        {from: "now-10M/M"}
      ]
    }
  },
  price_stats: {
    stats: {
      field: "data.price"
    }
  }
}
```

```

    }
  }
});

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

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

```

**Result:**

```

{
  "total": 6,
  "count": 2,
  "hits": [
    {
      "_id": "9922a270-f881-4bf8-be35-189e9a72a4f1",
      "_name": "house6",
      "_path": "/features/js-libraries/houses/house6",
      "creator": "user:system:su",
      "modifier": "user:system:su",
      "createdTime": "2015-10-05T12:09:41.998Z",
      "modifiedTime": "2015-10-05T12:25:42.469Z",
      "type": "com.enonic.app.features:house",
      "displayName": "House6",
      "hasChildren": false,
      "valid": true,
      "data": {
        "city": "Oslo",
        "location": "59.92,10.74",
        "price": 3400000,
        "number_floor": 3,
        "description": "House with garden",
        "publish_date": "2015-10-01"
      },
      "x": {
        "com-enonic-app-features": {
          "menu-item": {
            "menuItem": false
          }
        }
      },
      "page": {}
    },
    {
      "_id": "7c1006a0-5ca8-473e-b8dc-79f87c28e9be",
      "_name": "house4",
      "_path": "/features/js-libraries/houses/house4",
      "creator": "user:system:su",
      "modifier": "user:system:su",
      "createdTime": "2015-10-05T12:09:41.998Z",
      "modifiedTime": "2015-10-05T12:25:28.596Z",
      "type": "com.enonic.app.features:house",
      "displayName": "House4",
      "hasChildren": false,
      "valid": true,

```

```
"data": {
  "city": "Oslo",
  "location": "59.91,10.78",
  "price": 2900000,
  "number_floor": 3,
  "description": "House with garden",
  "publish_date": "2015-10-03"
},
"x": {
  "com-enonic-app-features": {
    "menu-item": {
      "menuItem": false
    }
  }
},
"page": {}
},
],
"aggregations": {
  "price_ranges": {
    "buckets": [
      {
        "key": "*-2000000.0",
        "docCount": 0,
        "from": null,
        "to": 2000000
      },
      {
        "key": "2000000.0-3000000.0",
        "docCount": 3,
        "from": 2000000,
        "to": 3000000
      },
      {
        "key": "3000000.0-*",
        "docCount": 3,
        "from": 3000000,
        "to": null
      }
    ]
  }
},
"floors": {
  "buckets": [
    {
      "key": "2",
      "docCount": 3,
      "prices": {
        "buckets": [
          {
            "key": "3000000",
            "docCount": 1
          },
          {
            "key": "2000000",
            "docCount": 2
          },
          {
            "key": "1000000",
```

```

        "docCount": 0
      }
    ]
  },
  {
    "key": "3",
    "docCount": 3,
    "prices": {
      "buckets": [
        {
          "key": "3000000",
          "docCount": 2
        },
        {
          "key": "2000000",
          "docCount": 1
        },
        {
          "key": "1000000",
          "docCount": 0
        }
      ]
    }
  }
]
},
"my_date_range": {
  "buckets": [
    {
      "key": "*-01-2015",
      "docCount": 0,
      "to": "2015-01-01T00:00:00Z"
    },
    {
      "key": "01-2015-*",
      "docCount": 6,
      "from": "2015-01-01T00:00:00Z"
    }
  ]
},
"by_month": {
  "buckets": [
    {
      "key": "09-2015",
      "docCount": 2
    },
    {
      "key": "10-2015",
      "docCount": 4
    }
  ]
},
"price_stats": {
  "count": 6,
  "min": 2100000,
  "max": 3400000,
  "avg": 2800000,

```

```
    "sum": 16800000
  }
}
```

## delete

This function deletes a content.

**delete** (*params*)

### Arguments

- **params** (*object*) – JSON with the parameters below.

**Returns** true if deleted, false otherwise.

### Parameters:

**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 contentLib = require('/lib/xp/content');

var result = contentLib.delete({
  key: '/features/js-libraries/mycontent',
  branch: 'draft'
});

if (result) {
  log.info('Content deleted');
} else {
  log.info('Content was not found');
}
```

## create

This function creates a content.

**create** (*params*)

### Arguments

- **params** (*object*) – JSON with the parameters below.

**Returns** the content created as JSON.

### Parameters:

**name** (*string*) Name of content. Optional, see below.

**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.

**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.

**language** (*string*) The language tag representing the content’s locale. This parameter is optional.

**data** (*object*) Actual content data.

**x** (*object*) eXtra data to use.

The parameter name is optional, but if it is not set then `displayName` must be specified. When name is not set, the system will auto-generate a name based on the `displayName`, by lower-casing and replacing certain characters. If there is already a content with the auto-generated name, a suffix will be added to the name in order to make it unique.

**Tip:** To create a content where the name is not important and there could be multiple instances under the same parent content, skip the `name` parameter and specify a `displayName` .

### Example:

```
var contentLib = require('/lib/xp/content');

var result = contentLib.create({
  name: 'mycontent',
  parentPath: '/features/js-libraries',
  displayName: 'My Content',
  requireValid: true,
  contentType: app.name + ':all-input-types',
  branch: 'draft',
  language: 'no',
  data: {
    myCheckbox: true,
    myComboBox: 'option1',
    myDate: '1970-01-01',
    myDateTime: '1970-01-01T10:00',
    myDouble: 3.14,
    myGeoPoint: '59.91,10.75',
    myHtmlArea: '<p>htmlAreaContent</p>',
    myImageSelector: '5a5fc786-a4e6-4a4d-a21a-19ac6fd4784b',
    myLong: 123,
    myRelationship: 'features',
    myRadioButtons: 'option1',
    myTag: 'aTag',
    myTextArea: 'textAreaContent',
    myTextLine: 'textLineContent',
    myTime: '10:00',
    myTextAreas: [
      'textAreaContent1',
      'textAreaContent2'
    ],
    myItemSet: {
      'textLine': 'textLineContent',
      'long': 123
    }
  },
  x: {
    "com-enonic-app-features": {
      "menu-item": {
        "menuItem": true
      }
    }
  }
});
```

```

    }
  }
});

log.info('Content created with id ' + result._id);

```

**Result:**

```

{
  "_id": "233f668f-103d-4c4f-86ac-elce7a166749",
  "_name": "mycontent",
  "_path": "/features/js-libraries/mycontent",
  "creator": "user:system:su",
  "modifier": "user:system:su",
  "createdTime": "2015-10-05T13:27:10.873Z",
  "modifiedTime": "2015-10-05T13:27:10.873Z",
  "type": "com.enonic.app.features:all-input-types",
  "displayName": "My Content",
  "hasChildren": false,
  "language": "no",
  "valid": true,
  "data": {
    "myCheckbox": true,
    "myComboBox": "option1",
    "myDate": "1970-01-01",
    "myDateTime": "1970-01-01T10:00",
    "myDouble": 3.14,
    "myGeoPoint": "59.91,10.75",
    "myHtmlArea": "<p>htmlAreaContent</p>",
    "myImageSelector": "5a5fc786-a4e6-4a4d-a21a-19ac6fd4784b",
    "myLong": 123,
    "myRelationship": "features",
    "myRadioButtons": "option1",
    "myTag": "aTag",
    "myTextArea": "textAreaContent",
    "myTextLine": "textLineContent",
    "myTime": "10:00",
    "myTextAreas": [
      "textAreaContent1",
      "textAreaContent2"
    ],
    "myItemSet": {
      "textLine": "textLineContent",
      "long": 123
    }
  },
  "x": {
    "com-enonic-app-features": {
      "menu-item": {
        "menuItem": true
      }
    }
  },
  "page": {}
}

```

If the content cannot be created because there is already a content with the specified path and name, an exception will be thrown. This case can be handled in JavaScript by wrapping the call with a `try catch` and checking if the error code is `contentAlreadyExists`, as in the example below:



**Example:**

```

var contentLib = require('/lib/xp/content');

try {
  var result = contentLib.create({
    name: 'content_name',
    parentPath: '/some/content/path',
    displayName: 'My content',
    contentType: 'base:unstructured',
    data: {}
  });

  log.info('Content created with id ' + result._id);
} catch (e) {
  if (e.code == 'contentAlreadyExists') {
    errorMsg = 'There is already a content with that name';
  } else {
    errorMsg = 'Unexpected error: ' + e.message;
  }
}

```

**modify**

This function modifies a content.

**modify** (*params*)

**Arguments**

- **params** (*object*) – JSON with the parameters below.

**Returns** the modified content as JSON.

**Parameters:**

**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.language = 'en';
  c.data.myCheckbox = false;
  c.data["myTime"] = "11:00";
  return c;
}

var contentLib = require('/lib/xp/content');

var result = contentLib.modify({
  key: '/features/js-libraries/mycontent',
  editor: editor
});

```

```
if (result) {
  log.info('Content modified. New title is ' + result.displayName);
} else {
  log.info('Content not found');
}
```

**Result:**

```
{
  "_id": "233f668f-103d-4c4f-86ac-elce7a166749",
  "_name": "mycontent",
  "_path": "/features/js-libraries/mycontent",
  "creator": "user:system:su",
  "modifier": "user:system:su",
  "createdTime": "2015-10-05T13:27:10.873Z",
  "modifiedTime": "2015-10-05T13:27:10.947Z",
  "type": "com.enonic.app.features:all-input-types",
  "displayName": "Modified",
  "hasChildren": false,
  "language": "en",
  "valid": true,
  "data": {
    "myCheckbox": false,
    "myComboBox": "option1",
    "myDate": "1970-01-01",
    "myDateTime": "1970-01-01T10:00",
    "myDouble": 3.14,
    "myGeoPoint": "59.91,10.75",
    "myHtmlArea": "<p>htmlAreaContent</p>",
    "myImageSelector": "5a5fc786-a4e6-4a4d-a21a-19ac6fd4784b",
    "myLong": 123,
    "myRelationship": "features",
    "myRadioButtons": "option1",
    "myTag": "aTag",
    "myTextArea": "textAreaContent",
    "myTextLine": "textLineContent",
    "myTime": "11:00",
    "myTextAreas": [
      "textAreaContent1",
      "textAreaContent2"
    ],
    "myItemSet": {
      "textLine": "textLineContent",
      "long": 123
    }
  },
  "x": {
    "com-enonic-app-features": {
      "menu-item": {
        "menuItem": true
      }
    }
  },
  "page": {}
}
```

### 4.2.3 lib-i18n

This library implements JavaScript bindings for standard i18n-related functionality. Add the following into your `build.gradle` file in the `dependencies` section:

```
include 'com.enonic.xp:lib-i18n:6.1.0'
```

To use this library in your JavaScript files, you can require the library like this:

```
var i18n = require('/lib/xp/i18n')
```

The methods implemented in this library are listed below.

#### localize

This function localizes a phrase.

**localize** (*params*)

##### Arguments

- **params** (*object*) – JSON with the parameters below.

**Returns** the localized string.

##### Parameters:

**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 i18n = require('/lib/xp/i18n');

var complex_message = i18n.localize({
  key: 'complex_message'
});

var message_multi_placeholder = i18n.localize({
  key: 'message_multi_placeholder',
  locale: "no",
  values: ["John", "London"]
});
```

### 4.2.4 lib-mail

This library implements methods for mail operations. Set the [Mail Configuration](#) and add the following into your `build.gradle` file in the `dependencies` section:

```
include 'com.enonic.xp:lib-mail:6.1.0'
```

To use this library in your JavaScript files, you can require the library like this:

```
var mail = require('/lib/xp/mail')
```

The methods implemented in this library are listed below.

## send

This function sends an email message using the mail server configured in the *Mail Configuration*.

**send** (*params*)

### Arguments

- **params** (*object*) – JSON with the parameters below.

**Returns** *true* if the message was sent successfully, *false* otherwise.

### Parameters:

**from** (*string*) The email address, and optionally name of the sender of the message.

**to** (*string[]*) The email address(es), and optionally name(s) of the primary message's recipient(s).

**cc** (*string[]*) The carbon copy email address(es). Optional.

**bcc** (*string[]*) The blind carbon copy email address(es). Optional.

**replyTo** (*string*) The email address that should be used to reply to the message. Optional.

**subject** (*string*) The subject line of the message.

**body** (*string*) The text content of the message.

**contentType** (*string*) Content type of the message body. Optional.

**headers** (*object*) Custom headers in the form of name-value pairs. Optional.

The address values can be either a simple email address (e.g. `'name@domain.org'`) or an address with a display name. In the latter case the email will be enclosed with angle brackets (e.g. `'Some Name <name@domain.org>'`).

The parameters `to`, `cc` and `bcc` can be passed as a single string or as an array of strings, if there are multiple addresses to specify.

---

**Tip:** The content-type of the email can be specified by using the *contentType* parameter. See example below for sending a message with an HTML body.

---

### Example:

```
var mail = require('/lib/xp/mail');

var sendResult = mail.send({
  from: 'Sales Department <sales@enonic.com>',
  to: 'user@somewhere.org',
  subject: 'Email Test from Enonic XP',
  cc: 'other@example.org',
  bcc: ['support@enonic.com', 'other@enonic.com'],
  replyTo: 'support@enonic.com',
  body: 'Welcome to Enonic XP!' + '\r\n\r\n' + '- The Dev Team',
  headers: {
    'Disposition-Notification-To': 'me@enonic.com'
  }
});

var sendResultHtml = mail.send({
  from: 'me@enonic.com',
  to: 'user@somewhere.org',
  subject: 'HTML email test',
  body: '<h1>HTML Email!</h1><p>You can use the contentType parameter for HTML messages.</p>',
```

```
contentType: 'text/html; charset="UTF-8"'
});
```

### 4.2.5 lib-mustache

This library implements JavaScript bindings for Mustache template processing. Add the following into your `build.gradle` file in the `dependencies` section:

```
include 'com.enonic.xp:lib-mustache:6.1.0'
```

To use this library in your JavaScript files, you can require the library like this:

```
var mustache = require('/lib/xp/mustache')
```

The methods implemented in this library are listed below.

#### render

This function renders a view using mustache.

**render** (*view*, *model*)

##### Arguments

- **view** (*ref*) – Location of the view.
- **model** (*object*) – View model.

**Returns** the rendered output.

##### Parameters:

**view** (*ref*) 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 model = {
  fruits: [
    {
      name: 'Apple',
      color: 'Red'
    },
    {
      name: 'Pear',
      color: 'Green'
    }
  ]
};

var mustache = require('/lib/xp/mustache');
var result = mustache.render(view, model);
```

## 4.2.6 lib-portal

This library implements JavaScript bindings for portal related functions Add the following into your `build.gradle` file in the `dependencies` section:

```
include 'com.enonic.xp:lib-portal:6.1.0'
```

To use this library in your JavaScript files, you can require the library like this:

```
var portal = require('/lib/xp/portal')
```

The methods implemented in this library are listed below.

### assetUrl

This function generates a URL pointing to a static file.

**assetUrl** (*params*)

#### Arguments

- **params** (*object*) – Input parameters.

**Returns** the generated URL.

#### Parameters:

**path** (*string*) Path to the asset.

**application** (*string*) Other application to reference to. Defaults to current application.

**type** (*string*) URL type. Either “server” (server-relative URL) or “absolute”. Default is “server”

**params** (*object*) Custom parameters to append to the url.

#### Example:

```
var portal = require('/lib/xp/portal');  
  
var url = portal.assetUrl({  
  path: 'styles/main.css'  
});
```

### attachmentUrl

This function generates a URL pointing to an attachment.

**attachmentUrl** (*params*)

#### Arguments

- **params** (*object*) – Input parameters.

**Returns** the generated URL.

#### Parameters:

**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`.

**type** (*string*) URL type. Either “server” (server-relative URL) or “absolute”. Default is “server”

**params** (*object*) Custom parameters to append to the url.

**Example:**

```
var portal = require('/lib/xp/portal');

var url = portal.attachmentUrl({
  download: true
});
```

## componentUrl

This function generates a URL pointing to a component.

**componentUrl** (*params*)

### Arguments

- **params** (*object*) – Input parameters.

**Returns** the generated URL.

**Parameters:**

**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.

**type** (*string*) URL type. Either “server” (server-relative URL) or “absolute”. Default is “server”

**params** (*object*) Custom parameters to append to the url.

**Example:**

```
var portal = require('/lib/xp/portal');

var url = portal.componentUrl({
  component: 'main/0'
});
```

## imageUrl

This function generates a URL pointing to an image.

**imageUrl** (*params*)

### Arguments

- **params** (*object*) – Input parameters.

**Returns** the generated URL.

**Parameters:**

**id** (*string*) ID of the image content.

**scale** (*string*) Required. Options are width(px), height(px), block(width,height) and square(px). See the *Scaling* page for more information.

**filter** (*string*) A number of filters are available to alter the image appearance, for example, `blur(3)`, `grayscale()`, `rounded(5)`, etc. All filter options are listed on the [Styling](#) page.

**type** (*string*) URL type. Either “server” (server-relative URL) or “absolute”. Default is “server”

**params** (*object*) Custom parameters to append to the url.

### Example:

```
var portal = require('/lib/xp/portal');

var url = portal.imageUrl({
  id: '1234',
  scale: 'block(1024,768)',
  filter: 'rounded(5);sharpen()'
});
```

## pageUrl

This function generates a URL pointing to a page.

**pageUrl** (*params*)

### Arguments

- **params** (*object*) – Input parameters.

**Returns** the generated URL.

### Parameters:

**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.

**type** (*string*) URL type. Either “server” (server-relative URL) or “absolute”. Default is “server”

**params** (*object*) Custom parameters to append to the url.

### Example:

```
var portal = require('/lib/xp/portal');

var url = portal.pageUrl({
  path: '/my/page',
  params: {
    a: 1,
    b: [1, 2]
  }
});
```

## serviceUrl

This function generates a URL pointing to a service.

**serviceUrl** (*params*)

### Arguments

- **params** (*object*) – Input parameters.

**Returns** the generated URL.



**Parameters:**

**service** (*string*) Name of the service.

**application** (*string*) Other application to reference to. Default is current application.

**type** (*string*) URL type. Either “server” (server-relative URL) or “absolute”. Default is “server”

**params** (*object*) Custom parameters to append to the url.

**Example:**

```
var portal = require('/lib/xp/portal');

var url = portal.serviceUrl({
  service: 'myservice',
  params: {
    a: 1,
    b: 2
  }
});
```

**processHtml**

This function replaces abstract internal links contained in an HTML text by generated URLs.

**processHtml** (*params*)

**Arguments**

- **params** (*object*) – Input parameters.

**Returns** the processed HTML.

**Parameters:**

**value** (*string*) Html value string to process.

**type** (*string*) URL type. Either “server” (server-relative URL) or “absolute”. Default is “server”

**Example:**

```
var portal = require('/lib/xp/portal');

var processedHtml = 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="">I
<a href="/admin/portal/preview/draft/features/content/_/attachment/download/123/image.jpg" target="">
```

**Note:** When outputting processed HTML in Thymeleaf, use attribute `data-th-utext="{processedHtml}"`.

## getComponent

This function returns the component corresponding to the current execution context. It is meant to be called from a layout or part controller.

**getComponent** (*params*)

### Arguments

- **params** (*object*) – Input parameters.

**Returns** the component as JSON.

### Example:

```
var portal = require('/lib/xp/portal');
var result = 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"
}
```

## getContent

This function returns the content corresponding to the current execution context. It is meant to be called from a page, layout or part controller.

**getContent** (*params*)

### Arguments

- **params** (*object*) – Input parameters.

**Returns** the content as JSON.

### Example:

```

var portal = require('/lib/xp/portal');
var result = portal.getContent();

log.info('Current content path = ' + result._path);

```

**Result:**

```

{
  "_id": "123456",
  "_name": "mycontent",
  "_parentPath" : "/a/b",
  "_path": "/a/b/mycontent",
  "createdTime": "1970-01-01T00:00:00Z",
  "creator": "user:system:admin",
  "data": {
    "binaryReference": "abc",
    "boolean": [true, true, false],
    "c": {
      "d": true,
      "e": ["3", "4", "5"],
      "f": 2
    },
    "double": 2.2,
    "doubles": [1.1, 2.2, 3.3],
    "geoPoint": "1.1,-1.1",
    "geoPoints": ["1.1,-1.1", "2.2,-2.2"],
    "instant": "+1000000000-12-31T23:59:59.999999999Z",
    "link": "/my/content",
    "localDate": "2014-01-31",
    "localDateTime": "2014-01-31T10:30:05",
    "long": 1,
    "longs": [1, 2, 3],
    "set": {
      "property": "value"
    },
    "string": "a",
    "stringEmpty": "",
    "strings": ["a", "b", "c"],
    "xml": "<xml><my-xml hello='world'/></xml>"
  },
  "displayName": "My Content",
  "hasChildren": false,
  "language": "en",
  "modifiedTime": "1970-01-01T00:00:00Z",
  "modifier": "user:system:admin",
  "page": {
    "config": {
      "a": "1"
    },
    "controller": "myapplication:mycontroller",
    "regions": {
      "top": {
        "components": [{
          "config": {
            "a": "1"
          },
          "descriptor": "myapplication:mypart",
          "name": "mypart",

```



**Result:**

```
{
  "_id": "100123",
  "_name": "my-content",
  "_parentPath": "/",
  "_path": "/my-content",
  "data": {
    "siteConfig": {
      "config": {
        "Field": 42
      },
      "applicationKey": "myapplication"
    }
  },
  "hasChildren": false,
  "page": {},
  "type": "base:unstructured",
  "valid": false,
  "x": {}
}
```

**getSiteConfig**

This function returns the site configuration for this app in 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.

**getSiteConfig()**

**Returns** the site configuration for current application as JSON.

**Example:**

```
var portal = require('/lib/xp/portal');
var result = portal.getSiteConfig();

log.info('Field value for the current site config = ' + result.Field);
```

**Result:**

```
{
  "Field": [42]
}
```

**4.2.7 lib-thymeleaf**

This library implements JavaScript bindings for Thymeleaf template processing. Add the following into your `build.gradle` file in the dependencies section:

```
include 'com.enonic.xp:lib-thymeleaf:6.1.0'
```

To use this library in your JavaScript files, you can require the library like this:

```
var thymeleaf = require('/lib/xp/thymeleaf')
```

Thymeleaf also supports a set of *View Functions*. The methods implemented in this library are listed below.

## render

This function renders a view using thymeleaf.

**render** (*view*, *model*)

### Arguments

- **view** (*ref*) – Location of the view.
- **model** (*object*) – View model.

**Returns** the rendered output.

### Parameters:

**view** (*ref*) 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 model = {
  fruits: [
    {
      name: 'Apple',
      color: 'Red'
    },
    {
      name: 'Pear',
      color: 'Green'
    }
  ]
};

var thymeleaf = require('/lib/xp/thymeleaf');
var result = thymeleaf.render(view, model);
```

## 4.2.8 lib-xslt

This library implements JavaScript bindings for XSLT template processing. Add the following into your `build.gradle` file in the `dependencies` section:

```
include 'com.enonic.xp:lib-xslt:6.1.0'
```

To use this library in your JavaScript files, you can require the library like this:

```
var xslt = require('/lib/xp/xslt')
```

XSLT also supports a set of *View Functions*. The methods implemented in this library are listed below.

## render

This function renders a view using XSLT.

**render** (*view*, *model*)

### Arguments

- **view** (*ref*) – Location of the view.

- **model** (*object*) – View model.

**Returns** the rendered output.

**Parameters:**

**view** (*ref*) 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.xsl');
var model = {
  fruits: [
    {
      name: 'Apple',
      color: 'Red'
    },
    {
      name: 'Pear',
      color: 'Green'
    }
  ]
};

var xslt = require('/lib/xp/xslt');
var result = xslt.render(view, model);
```

## 4.3 View Functions

Some view technologies support a set of view functions. The list of view functions below are supported out-of-the-box:

### 4.3.1 assetUrl

This generates a URL pointing to a static file in the site/assets folder, such as CSS, background images, etc.

**Parameters:**

**\_path** Path to the asset.

**\_application** Use this when the asset referenced is in another application. Defaults to current application. Use the app name, for example, `com.enonic.blog.superhero`.

**\_type** URL type. Either `server` (server-relative URL) or `absolute`. Default is `server`.

**everything else** Custom parameters to append to the url.

**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>
```

### 4.3.2 attachmentUrl

This generates a URL pointing to an attachment.

**Parameters:**

**\_id** Id to the content holding the attachment.

**\_path** Path to the content holding the attachment.

**\_name** Name to the attachment.

**\_label** Label of the attachment. Default is `source`.

**\_download** Set to true if the disposition header should be set to attachment. Default is `false`.

**\_type** URL type. Either `server` (server-relative URL) or `absolute`. Default is `server`.

*everything else* Custom parameters to append to the url.

**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>
```

### 4.3.3 componentUrl

This generates a URL pointing to a component.

**Parameters:**

**\_id** Id to the page.

**\_path** Path to the page.

**\_component** Path to the component. If not set, the current path is set.

**\_type** URL type. Either `server` (server-relative URL) or `absolute`. Default is `server`.

*everything else* Custom parameters to append to the url.

**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>
```

**4.3.4 imageUrl**

This generates a URL pointing to an image.

**Parameters:**

**\_id** Id to the image.

**\_path** Path to the image. If **\_id** is specified, this parameter is not used.

**\_format** Format of the image.

**\_scale** Resize and crop the image to fit the available area. See: *Scaling*

**\_quality** Quality for JPEG images.

**\_background** Background color.

**\_filter** Styling filters to use on the image. More than one filter may be combined with a semicolon. See: *Styling*

**\_type** URL type. Either *server* (server-relative URL) or *absolute*. Default is *server*.

*everything else* Custom parameters to append to the url.

**Usage in Thymeleaf:**

```


```

**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', 'scale=width(200)')"/>
    <xsl:value-of select="portal:imageUrl('_path=test', 'scale=width(200)')"/>
  </xsl:template>

</xsl:stylesheet>
```

**4.3.5 pageUrl**

This generates a URL pointing to a page.

**Parameters:**

**\_id** Id to the page. If **id** is set, then **path** is not used.

**\_path** Path to the page. Relative paths is resolved using the context page.

**\_type** URL type. Either `server` (server-relative URL) or `absolute`. Default is `server`.

*everything else* Custom parameters to append to the url.

### 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>
```

## 4.3.6 serviceUrl

This generates a URL pointing to a service.

### Parameters:

**\_service** Name of the service.

**\_application** Other application to reference to. Default is current application.

**\_type** URL type. Either `server` (server-relative URL) or `absolute`. Default is `server`.

*everything else* Custom parameters to append to the url.

### 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>
```

## 4.3.7 imagePlaceholder

This command generates a URL to an image placeholder.

### Parameters:

**width** Width of image.

**height** Height of image.

**Usage in Thymeleaf:**

```

```

**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>
```

### 4.3.8 localize

This localizes a phrase.

**Parameters:**

**\_key** The property key.

**\_locale** A string-representation of a locale. If the locale is not set, the site language is used.

**\_values** Optional placeholder values (comma separated).

**Usage in Thymeleaf:**

```
<div data-th-text="{portal.localize({'_key=mystring','_locale=en'})}">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>
```

### 4.3.9 processHtml

This function replaces abstract internal links contained in an HTML text by generated URLs.

**Parameters:**

**\_value** Html value string to process.

**\_type** URL type. Either “server” (server-relative URL) or “absolute”. Default is “server”

**Usage in Thymeleaf:**

```
<div data-th-text="{portal.processHtml({'_value=some text'})}">Text</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:processHtml('_value=some text')"/>
  </xsl:template>

</xsl:stylesheet>

```

## 4.4 Query Language

When finding nodes and content you will be using our query language. It is based on SQL and looks very similar.

### 4.4.1 queryExpr

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

```

### 4.4.2 constraintExpr

Grammar:

```

constraintExpr = compareExpr
                | logicalExpr
                | dynamicConstraint
                | notExpr ;

```

### 4.4.3 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')

```

#### 4.4.4 logicalExpr

Grammar:

```

logicalExpr = constraintExpr operator constraintExpr ;
operator    = 'AND' | 'OR' ;

```

Examples:

```

myCategory = "articles" AND myPriority > 10
myCategory IN ("articles", "documents") OR myPriority <= 10

```

#### 4.4.5 dynamicConstraint

Grammar:

```

dynamicConstraint = functionExpr ;

```

Examples:

```

fulltext('myCategory', 'Searching for fish', 'AND')
ngram('description', 'fish boat', 'AND')

```

#### 4.4.6 notExpr

Grammar:

```

notExpr = 'NOT' constraintExpr ;

```

Examples:

```

NOT myCategory = 'article'

```

### 4.4.7 orderExpr

Grammar:

```
orderExpr = 'ORDER BY' ( fieldOrderExpr | dynamicOrderExpr )
           ( ',' ( fieldOrderExpr | dynamicOrderExpr ) )* ;
```

### 4.4.8 fieldOrderExpr

Grammar:

```
fieldOrderExpr = propertyPath [ direction ] ;
direction>      = 'ASC' | 'DESC' ;
```

Examples:

```
_name ASC
_timestamp DESC
title DESC
data.myProperty
```

### 4.4.9 dynamicOrderExpr

Grammar:

```
dynamicOrderExpr = functionExpr [ direction ] ;
direction         = 'ASC' | 'DESC' ;
```

Examples:

```
geoDistance('59.9127300,10.746090')
```

### 4.4.10 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
```

---

### 4.4.11 functionExpr

Grammar:

```
functionExpr = functionName '(' arguments ')';
```

### 4.4.12 Examples

Find all documents where property 'myCategory' is populated with a value, and the value does not equal '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('data.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('data.location', '59.9127300,10.7460900') ASC
```

## 4.5 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 data from every repository.
  export            Export data for a specified path.
  help              Display help information
  import            Import data from a named export.
  init-project      Initiates an Enonic XP application project structure.
  list-snapshots    Returns a list of existing snapshots with name and status.
  load              Import data from a dump.
  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.
  upgrade           Upgrade a dump to the current version. The upgraded files will be written to <

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:

### 4.5.1 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
```

### 4.5.2 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

```

### 4.5.3 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

```

### 4.5.4 delete-snapshots

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
```

### 4.5.5 export

Extract data for a given repository, branch and content path. The result will be stored in the `$XP_HOME/data/export` directory.

**Usage:**

```
NAME
  toolbox export - Export data for a specified path.

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
```

### 4.5.6 import

Import 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:/
```

## 4.5.7 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
```

## 4.5.8 dump

Export data from every repository. This is used to backup the entire repository when doing an upgrade. The result will be stored in the `$XP_HOME/data/dump` directory.

**Usage:**

```
NAME
    toolbox dump - Export data from every 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
```

## 4.5.9 load

Load data from a named dump and load it into Enonic XP. The dump read has to be stored in the `$XP_HOME/data/dump` directory.

**Usage:**

```
NAME
    toolbox load - Import data from a dump.

SYNOPSIS
    toolbox load -a <auth> [-h <host>] [-p <port>] -s <source>

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 <source>
    Dump name.
```

**Example:**

```
$ ./toolbox.sh load -a su:password -s myDump
```

## 4.5.10 upgrade

Upgrade a data dump from a previous version to the current version. The output of the upgrade will be placed alongside the dump that is being upgraded and will have the name <dump-name>\_upgraded\_<new-version> unless a target location is specified with -t.

The current version XP installation must be running with the upgraded app deployed.

**Usage:**

```
NAME
    toolbox upgrade - Upgrade a dump.

SYNOPSIS
    toolbox upgrade -d dump-path -t target-location

OPTIONS
    -d <dump>
        Directory for dump.
```

**Example:**

```
$ ./toolbox.sh upgrade -d ./data/dump/5.3.1-dump
```

The output would appear as:

```
/data/dump/5.3.1_upgraded_6.0.0/
```

## 4.5.11 init-project

The init-project tool initializes a new application project structure by retrieving a Git repository, removing all references to the Git repository, and adapting its build file properties (gradle.properties).

**Usage:**

```
NAME
    toolbox init-project - Initiates an Enonic XP application project

SYNOPSIS
    toolbox init-project [-a <authentication>]
        [(-d <destination> | --destination <destination>)]
        (-n <name> | --name <name>)
        (-r <repository> | --repository <repository>)
        [(-v <version> | --version <version>)]

OPTIONS
    -a <authentication>
        Optional authentication token for basic authentication (user:password)

    -d <destination>, --destination <destination>
```

```
Optional destination path to create your project, if not specified current directory will be used.

-n <name>, --name <name>
    Unique qualifying name that will be given to your application i.e. com.company.myapp. NOT

-v <version>, --version <version>
    Optional version number that will be set in your application project, if not used 1.0.0-SNAPSHOT

-r <repository>, --repository <repository>
    Git repository you wish to use as starting point. Supports both full urls to any standard
```

### Examples:

```
$ toolbox.sh init-project -d ~/Dev/xp/apps/myApp -n com.company.myapp -v 0.9.0 -r https://github.com
```

```
$ toolbox.sh init-project -n com.company.myapp -v 1.0.0-SNAPSHOT -r enonic/starter-base
```

```
$ toolbox.sh init-project -n com.company.myapp -r starter-base
```

## 4.6 Image Processor

Enonic XP includes a number of image processing commands that may be used to set the size or add style to the images. The commands are appended on the image URLs. To automatically create the URLs, use the *imageUrl* view function.

### 4.6.1 Scaling

#### Scale Max

Scales the image proportionally, so the longest edge has the given number of pixels.

*Arguments:*

**size** The length of the longest edge. Required

**Example:**

```
max(600)
```

#### Scale Wide

Scales the image to fit the given width of the picture. If the image is taller than the given height, it is cropped on top and bottom, based on the focal point position.

*Arguments:*

**width** Width in pixels

**height** maximum height in pixels

**Example:**

```
wide(600,200)
```

### Scale Block

Scales the image, while keeping the aspect ratio, so it fills the rectangle specified by width and height. Then crops the overflowing axis based on the focal point position. The result of a call to this method will be an image that always has the exact size of the specified input.

*Arguments:*

**width** Width in pixels

**height** Height in pixels

**Example:**

```
block(600,200)
```

### Scale Square

Scales the image proportionally to match the shortest edge. The longest edge will be cropped based on the focal point position.

*Arguments:*

**size** The length of both sides in pixels

**Example:**

```
square(600)
```

### Scale Height

Scales the image proportionally to match the given height.

*Arguments:*

**height** Height in pixels

**Example:**

```
height(600)
```

### Scale Width

Scales the image proportionally to match the given width.

*Arguments:*

**width** Width in pixels

**Example:**

```
width(600)
```

## 4.6.2 Styling

### Block

Pixelates the image creating a mosaic like effect.

*Arguments:*

**size** The number of pixels squared, that should be combined to one block. Default: 2

**Example:**

```
block(5)
```



### Blur

Applies a blur effect.

*Arguments:*

**radius** How much blur to apply. Default: 2

**Example:**

```
blur(8)
```





### Border

Applies a rectangular border around the image.

*Arguments:*

**width** The width of the border in pixels. Default: 2

**color** The color of the border as a decimal or hexadecimal number. Default: 0 / 0x000000 (black)

**Example:**

```
border (5)  
border (4, 0x777777)
```



### Emboss

Applies an embossing effect on the image.

*No arguments*

**Example:**

```
emboss ()
```



**Grayscale**

Creates a grayscale variant of the image.

*No arguments*

**Example:**

```
grayscale ()
```



**Invert**

Inverts the colors in the image.

*No arguments*

**Example:**

```
invert ()
```

**Rounded**

Rounds the corners of the image, with an option of adding a border around the rounded image.

*Arguments:*

**radius** The number of pixels from each corner where the rounding starts. Default: 10

**borderSize** The width of the border in pixels. Default: 0

**borderColor** The color of the border as a decimal or hexadecimal number. Default: 0 / 0x000000 (black)

**Example:**

```
rounded ()  
rounded (15)  
rounded (10, 1)  
rounded (8, 4, 0x777777)
```



## Sharpen

Applies a sharpening filter to the image.

*No arguments*

### Example:

```
sharpen ()
```



## RGB Adjust

Adjust the red, green and blue levels in the image.

*Arguments:*

**red** The adjusted red level for the image. Default: 0

**green** The adjusted green level for the image. Default: 0

**blue** The adjusted blue level for the image. Default: 0

### Example:

```
rgbadjust (2.0, 0.25, -1.75)
```



### HSB Adjust

Adjust the hue, saturation and brightness levels in the image.

*Arguments:*

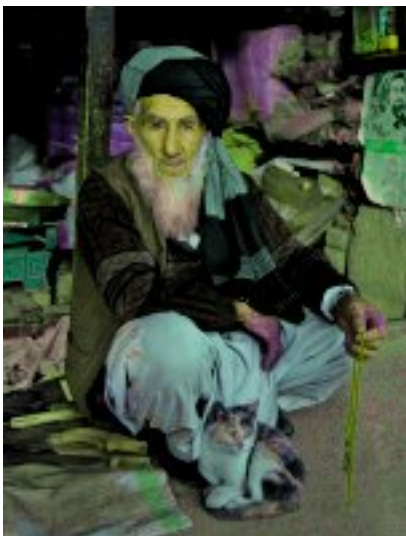
**hue** Value from -1 to 1, of how far around the color wheel to move the hue of the image. Default: 0

**saturation** Value from -1 to 1 to adjust the intensity of the colors in the image. Default: 0

**brightness** Value from -1 to 1 to adjust the brightness of the image. Default: 0

**Example:**

```
hsbadjust (0.5, -0.1)
hsbadjust (-0.15, 0.2, -0.2)
```



### Edge

Creates an abstract image by brightening every edge and darkening every even surface of the image.

*No arguments*

**Example:**

```
edge ()
```



## Bump

Creates a 3D looking texture, based on darkening and lighting each side of edges in the image.

*No arguments*

**Example:**

```
bump ()
```



## Sepia

Creates a grayscale image with a yellow-reddish tint to make it look like an old photograph.



*Arguments:*

**depth** The brightness of the tint. Default: 20

**Example:**

```
sepia ()  
sepia (25)
```



### Rotate 90

Rotates an image 90 degrees

*No arguments*

**Example:**

```
rotate90 ()
```



### Rotate 180

Rotates an image 180 degrees

*No arguments*

**Example:**

```
rotate180()
```



**Rotate 270**

Rotates an image 270 degrees

*No arguments*

**Example:**

```
rotate270()
```



**Flip horizontal**

Flips an image horizontally

*No arguments*

**Example:**

```
fliph()
```





### Flip vertically

Flips an image vertically

*No arguments*

**Example:**

```
flipv()
```



### Colorize

Makes a grayscale image, then applies a tint, based on the specified color.

*Arguments:*

**red** Red boost value. Default: 1

**green** Green boost value. Default: 1

**blue** Blue boost value. Default: 1

**Example:**

```
colorize(3,1,1.5)
```



### HSB Colorize

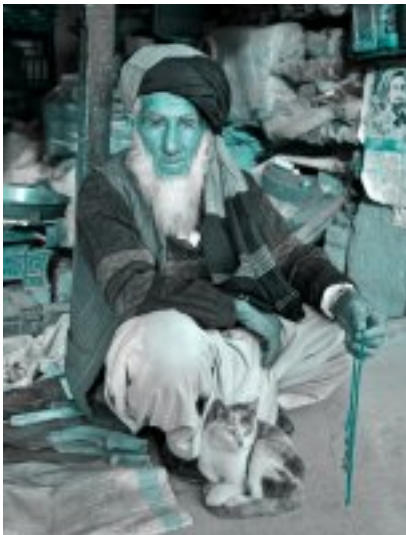
Makes a grayscale image, then applies a tint, based on the specified color.

*Arguments:*

**color** The tint color as a decimal or hexadecimal number. Default: 0xFFFFFFFF

**Example:**

```
hsbcolorize(0x00AAAA)
```



## 4.7 JavaDoc API

You can either download the JavaDoc as a [zip](#) or view it [directly in your browser](#).



---

## Release Notes

---

Enonic XP 6.2 is a minor release, with several new feature and fixes - there are no breaking changes

### 5.1 Custom error-pages

You can now easily create error pages in javascript, handling any http errors. It's as simple as creating an error.js file, exporting for instance `handle404()` or `handleError()` functions.

See *Error Page* for more details

### 5.2 Response Filters

You can now create response filters using Javascript. This is a powerful feature that enable developers to wire into any request for a site - and optionally manipulate it. The response filters will work across different applications when installed on a site.

Example use-cases: \* Create application that automatically adds script elements to all pages \* Replace a certain text string for any page \* Automatically add custom headers to specific pages

See *Response Filters* for more details

### 5.3 Perfect caching

Asset, image and attachment served over HTTP now implement so-called perfect cache. In short, this means that these resources have header tags that enable “infinite” caching, so proxies and browsers can cache them locally - resulting in improved performance for the end user.

The downside of this is normally associated with problems with old items hanging around in the various clients. The perfect cache implementation however always creates new urls if an item is changed, effectively preventing this problem.

Images and attachments that are not publicly accessible (restricted access) are served with “private” headers, effectively preventing proxies from caching them, and exposing the data to the wrong users.

No configuration or changes to your code is required

## 5.4 HTTP Compression

All items served over HTTP are now automatically Gzipped to optimize performance and network bandwidth.

No configuration or changes to your code is required

## 5.5 Minor improvements

- Enabled request log configuration
- Listing of attachments in Content Detail panel
- Lib-auth - New function generatePassword()
- Lib-auth - New function changePassword()
- Content API - name value is now optional
- Init-project - Projects are created even if target folder contains files
- Added “Settings” button to context menu
- Media content’s attachment name does not change if modified
- Simple modular JSON status API now available
- Styled component placeholders with errors in Live Edit
- Fixed layout of “More” and “Actions” fold buttons in Admin
- Tighter integration with bundled HTTP server - Jetty

## 5.6 Bugfixes

- Page Component View - State of context menu’s dropdown icon doesn’t change
- Page Component View - Toggle state of the “Inspect panel” button in toolbar when the context window is open
- Publish a content with an invalid reference fails with exception
- Component View not displayed if a descriptor is missing or has an error
- Content wizard toolbar - “Component view” and “Inspection panel” buttons are visible when LiveEdit is hidden
- Page Component View - Item context menu is shown after the view was closed
- Hide icon for collapsing content wizard when LiveEdit is inactive
- Detail panel - Missing display name for unnamed content
- Details Panel is empty when content have been selected and published
- ContentDeleteDialog - Wrong number of items to delete
- Attachment type for the same attachment differs depending of upload method
- Fix delete confirmation message for unnamed content
- Details panel has empty content on first open with selected item
- Multiple duplicated requests when selecting content in Content Manager
- Page Component View - Context menu should be closed when pressing Application Home

- LiveEdit - Error when opening Superhero for edit in XP admin
- Impossible to delete not valid content that has “modified” status
- Controller get/post without a return giving 405
- Component View - Fix presentation of page in tree
- Grid not refreshed when content deleted
- Enable Apply button in Edit Permissions dialog when overwrite child permissions is selected
- Incorrect appearance of Spinner when site-wizard was opened
- ContentVersionFactory - Incorrect ContentVersionId
- Three requests to WidgetDescriptorService made upon app page loaded
- ContentBrowse-toolbar - ‘More’ and ‘Refresh’ buttons location problem
- Image name and path remains visible after unselecting item in ImageSelector
- Created content not displayed after the page redirection
- Site Wizard - Preview-button on toolbar should be disabled when site has no templates
- App gradle-plugin not working for Gradle 2.8
- Content with invalid fields is marked as valid
- Content still selected after clicking the remove button in the browse panel
- Detail panel - Contents disappear after modifying an item





---

## Upgrading to 6.2

---

**Warning:** In order to upgrade, you must already have upgraded to Enonic XP 6.0.0 or newer

Enonic XP 6.2 download: <https://repo.enonic.com/public/com/enonic/xp/distro/6.2.0/distro-6.2.0.zip>

There are no breaking changes in this release, but due to internal changes in repository you will have to dump your data and load them into the new version:

1. Download and install the new release on your chosen infrastructure
2. Backup your existing data
3. Dump the repository, see *dump* reference
4. Stop the server
5. Move or link (however this is configured) your `$XP_HOME` (configuration and deployments and data) to the new Enonic XP installation
6. Optionally update your system configuration with mail server (See *Mail Configuration*) - introduced in 6.1
7. Delete the `$XP_HOME/repo` - folder
8. Start the new server
9. Load the repository-dump you created, see *load* reference

**Warning:** Remember to update any startup scripts you might have to launch your new installation given a server restart



---

## Frequently Asked Questions

---

### 7.1 What's the latest release?

The latest official release is 6.2. More information is available on the [latest release changelog](#).

### 7.2 Where can I get the source code?

All source code for Enonic XP is published on our [GitHub project page](#).

### 7.3 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 coming, you can go to our [GitHub wiki page](#).

### 7.4 What is \$XP\_INSTALL?

`$XP_INSTALL` and `$XP_HOME` are referenced frequently in the documentation and it is important to understand the difference. `$XP_INSTALL` is the top level directory of the XP installation and it contains the directories *bin*, *home*, *lib*, *toolbox* and others.

### 7.5 What is \$XP\_HOME?

`$XP_HOME`, by default, is the location of the `$XP_INSTALL/home` folder which contains the *config*, *deploy*, *repo* and other directories specific to a single XP instance. The home folder can be copied to multiple locations for developers working on more than one project.

There are two situations where the `$XP_HOME` environment variable must be set:

1. When developers are working on an application and intend to use `./gradlew deploy`.
2. When a *home* folder other than `$XP_INSTALL/home` is to be used.

## 7.6 Where can I get help?

The [community forum](#) would be a good place to start. We also offer [formal training courses](#).

---

## Glossary

---

**\$XP\_HOME** By default, this is the location of the \$XP\_INSTALL/home folder and it contains directories specific to a single XP instance. The home folder can be copied to multiple locations for developers working on multiple isolated projects. The \$XP\_HOME environment variable should be set to the home folder of the project to be run.

**\$XP\_INSTALL** The the location of the unzipped XP download

---

**Tip:** The source code for this documentation source is [available on GitHub](#) .

---



## Symbols

\$XP\_HOME, 177

\$XP\_INSTALL, 177

## A

assetUrl() (built-in function), 130

attachmentUrl() (built-in function), 130

## C

changePassword() (built-in function), 113

componentUrl() (built-in function), 131

create() (built-in function), 122

## D

delete() (built-in function), 122

## G

generatePassword() (built-in function), 113

get() (built-in function), 114

getChildren() (built-in function), 115

getComponent() (built-in function), 134

getContent() (built-in function), 134

getSite() (built-in function), 136

getSiteConfig() (built-in function), 137

getUser() (built-in function), 112

## H

hasRole() (built-in function), 113

## I

imageUrl() (built-in function), 131

## L

localize() (built-in function), 127

log.debug() (log method), 105

log.error() (log method), 105

log.info() (log method), 105

log.warning() (log method), 105

login() (built-in function), 111

logout() (built-in function), 112

## M

modify() (built-in function), 125

## P

pageUrl() (built-in function), 132

processHtml() (built-in function), 133

## Q

query() (built-in function), 117

## R

render() (built-in function), 129, 138

require() (built-in function), 106

resolve() (built-in function), 106

## S

send() (built-in function), 128

serviceUrl() (built-in function), 132