

# enonic

academy

## WIFI

Network name:

**enonic**

password:

**enonic12**



## XP: Developer 101

# Welcome

- Instructor
- Offices
- Computers
- You?
  - What other CMS have you worked with?
  - Any experience with Enonic XP? (or the old CMS)



# Schedule

- 09:00 - Introduction
- 09:10 - XP, Setup
- 09:30 - XP, creating your first “App”
- 11:30 - Lunch
- 12:00 - XP, extending your app
- 15:20 - Closing words, discussion
- 15:30 - Gives us **feedback** and you can go

# What we'll learn today

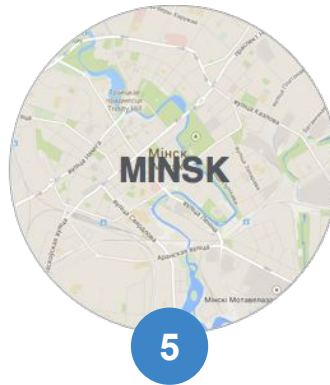
Upon completion of this course, you will be familiar with the following basics of Enonic XP:

- What makes XP awesome
- Install & run Enonic XP
- Create and install apps
  - App structure and MVC
  - Basic Thymeleaf templating
  - Schemas, server-side JavaScript, XP libraries
- Basic use of Content Studio admin tool

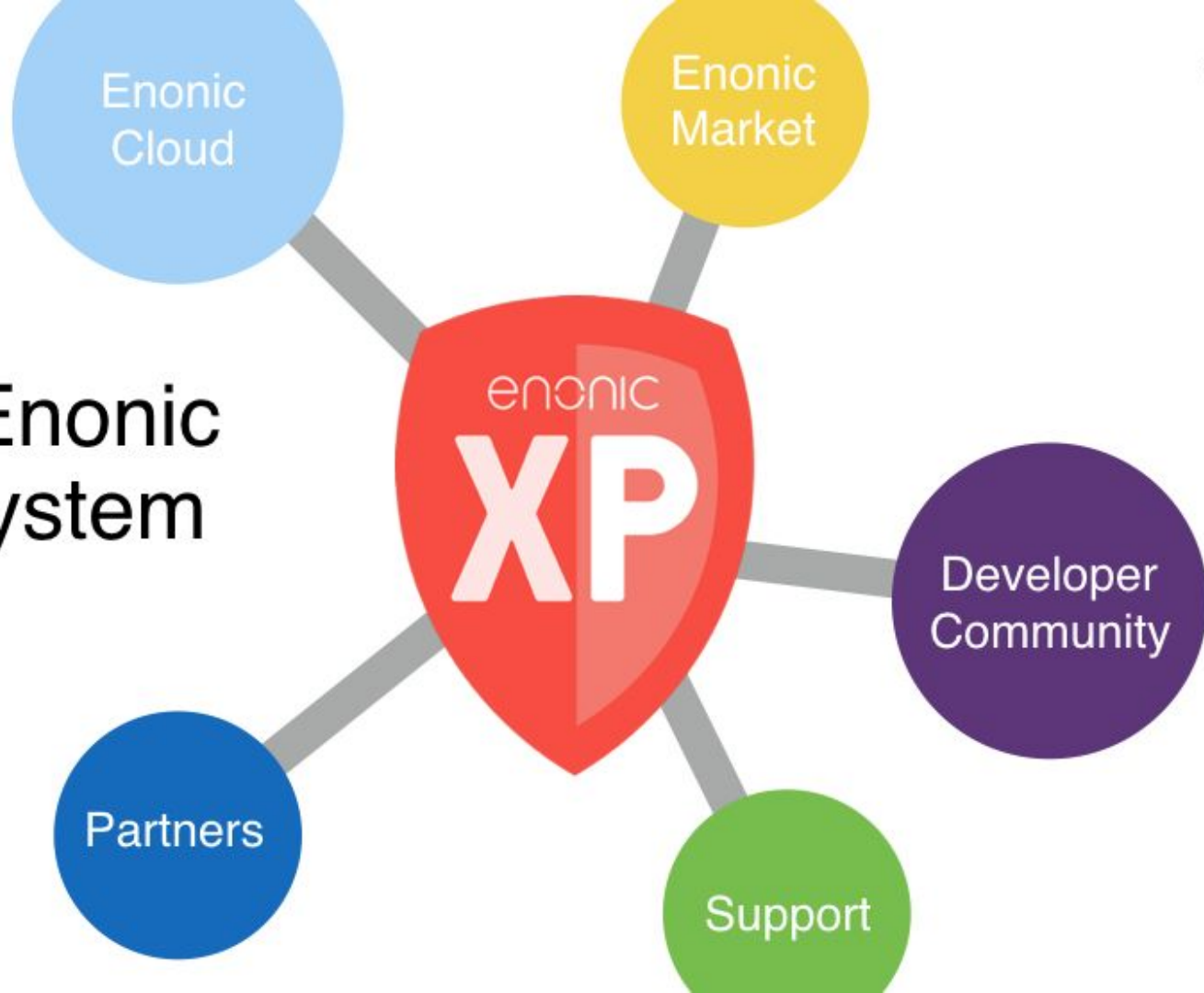
40%



# enonic



# The Enonic Ecosystem

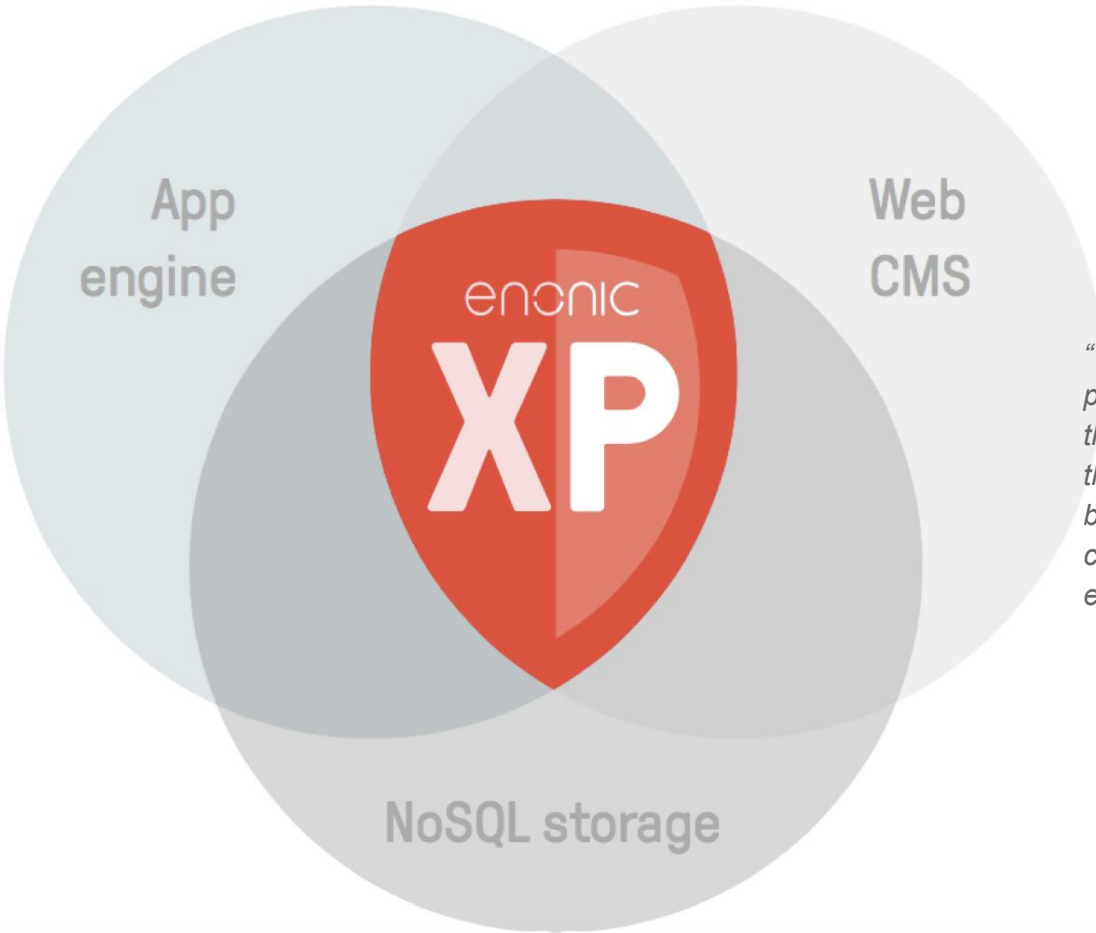






# Architecture

How Enonic XP works

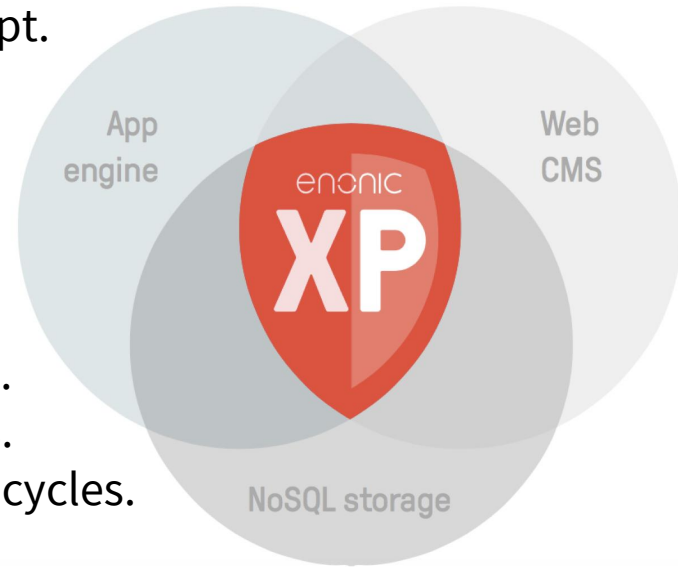


*"15 years of experience with large customers and tech projects has revealed the huge potential in simplifying the technology and infrastructure in use. We have used this knowledge and nearly 100 000 working hours to build a unique platform that replaces traditional components like databases, app servers, search engines and the CMS."*

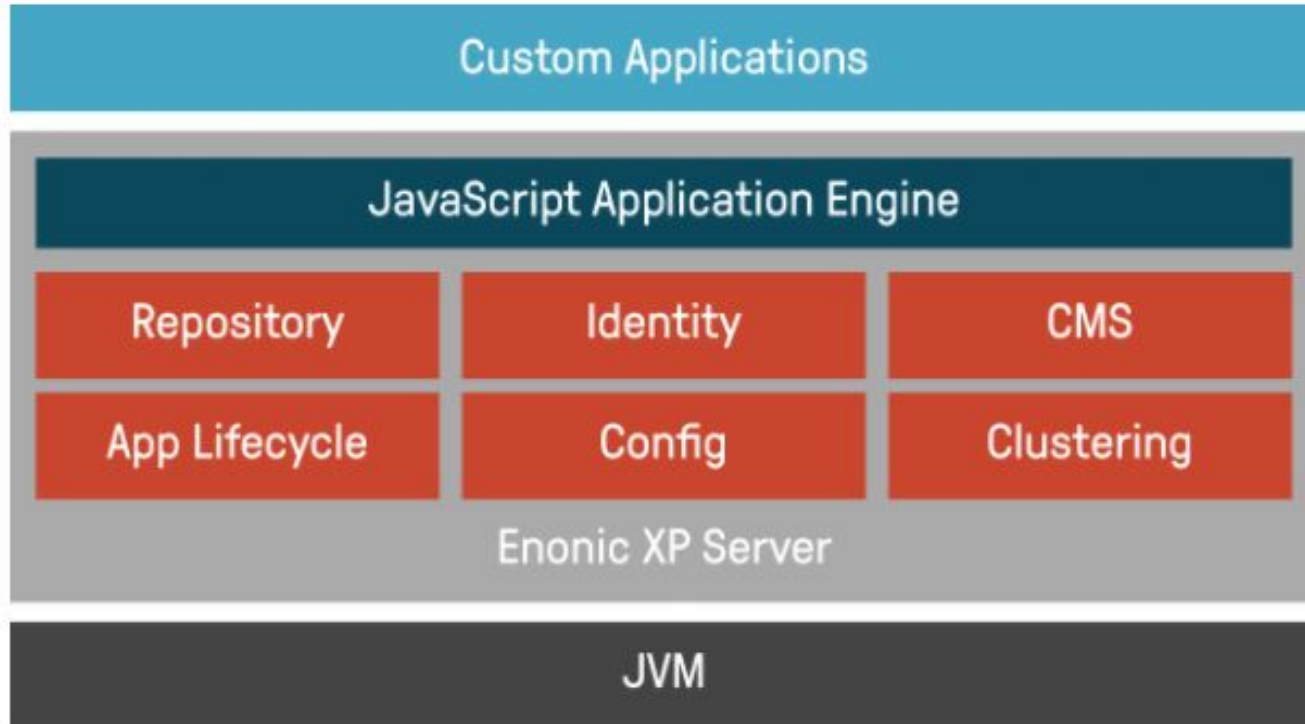
**Thomas Sigdestad - Enonic CTO**

# Enonic XP - technologies

- **NoSQL** - no more struggling with DBs, drivers, etc. XP comes with storage!
- **ElasticSearch** - indexes everything, makes it searchable. Lightning fast!
- **JVM** - runs anywhere Java runs (Mac, Unix, Windows).
- **Nashorn** - build your apps with server side JavaScript.
- **JSON** - data is stored in a JavaScript-friendly way.
- **Java8** - extend your apps to do more, using Java.
- **Jetty** - the app-server running in the back.
- **MVC** - structure your code in your apps.
- **Docker** - ready-to-go code for simple hosting.
- **Thymeleaf** - template engine based on pure HTML5.
- **Gradle** - our build system, develop and deploy apps.
- **OSGi** - dynamic module system that defines app lifecycles.
- **Open Source** - sharing is caring.



# XP Architecture



# The experience

## Developers <3 XP

- Fast!
- Open Source
- Loads of apps
- Training
- Community
- Documentation
- Easy getting started
- “Zero” config
- JavaScript + HTML5
- Extendable
- Any tool/editor
- Any frontend tech

## Ops <3 XP

- Fast!
- Docker containers
- Enonic Cloud
- Community
- Documentation
- Clustering
- Runs “anywhere”
- Configurable
- Secure



**Getting started**

# Current release

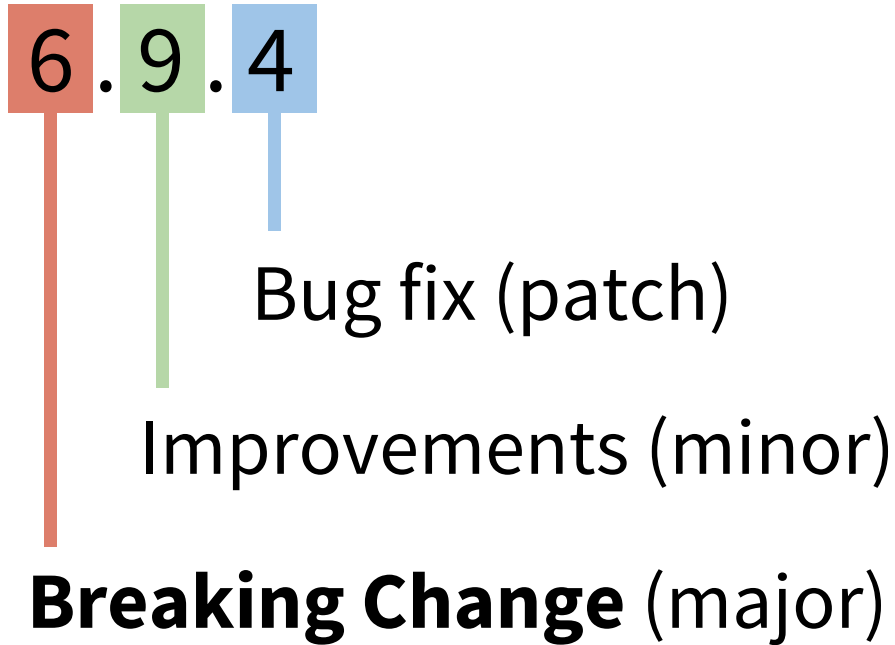
- **6.15.3** - September 2018

Go to: [docs.enonic.com](https://docs.enonic.com)

Find this in menu: **/ Release Notes /**

<http://docs.enonic.com/en/6.15/appendix/release-notes/index.html>

# Semantic Versioning

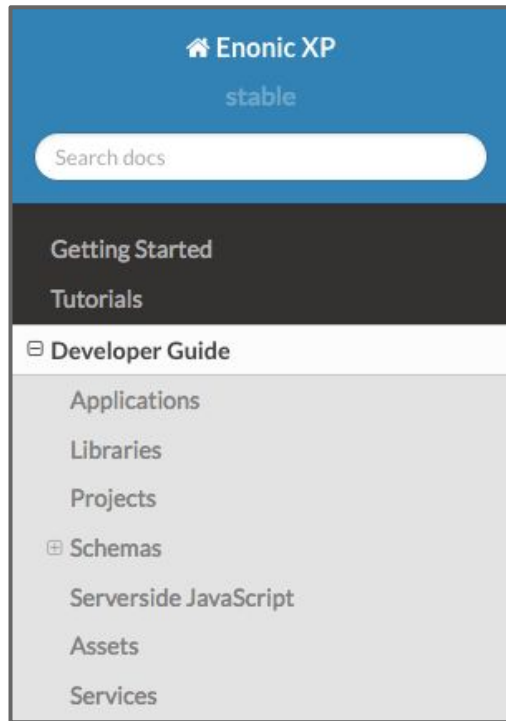




# Docs

Covers all functionality in Enonic XP.

- Versioned! (see bottom panel)
- Tutorials and videos
- Developer Guide
  - Hands-on how to build apps
- Operation Guide (for hosting)
- API and References
  - Details on all JS-functions!



# Help and resources

- **Forum** - [discuss.enonic.com](https://discuss.enonic.com)
- **Docs** - [docs.enonic.com](https://docs.enonic.com)
- **Chat** - [slack.enonic.com](https://slack.enonic.com)
- **Enterprise Support** - [support.enonic.com](https://support.enonic.com)
- **GitHub** - [github.com/enonic](https://github.com/enonic)
- **Enonic Market** - [market.enonic.com](https://market.enonic.com)
- **Thymeleaf** - [www.thymeleaf.org](https://www.thymeleaf.org)
- **JavaScript** - [www.google.com](https://www.google.com)



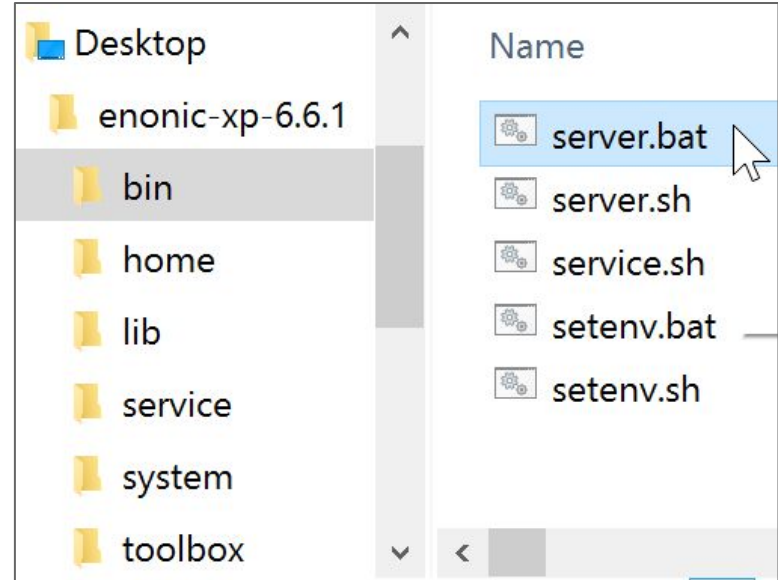
# Task - Download & install XP

You can download Enonic XP in many ways. We have a [Download page](#) with many different packages, our Github page (build it yourself), or from our repo server.

- Visit our Repo server
  - <http://repo.enonic.com/public/com/enonic/xp/distro/>
- Click link for latest stable release
- Download the .zip-file from here
- Unzip to desktop
- Done

# Starting XP

- Go to unzipped folder
  - your XP folder
- Open the /bin/ folder
- Double click “server.bat”
  - Command Line starts
- Keep window open
- Use Ctrl + C to shutdown  
( All changes in admin are stored permanently. )



# Starting XP (cont.)

```
14:57:09.073 INFO org.eclipse.jetty.server.Server - Started @7852ms
14:57:09.315 INFO c.e.x.l.i.framework.FrameworkService - Started Enonic XP in 7652 ms
```

When starting XP you'll find:

- **It starts seriously fast!**
- Server logs:
  - Custom, errors & stack trace
  - XP version number
  - Installed apps
  - **\$XP\_HOME** (more on this later)

# Task - **Login**

- Site found at <http://localhost:8080/>
- **Log in without a user** - since 6.14 you can use XP without a user until a new admin-user is created. Before the default user/password was su/password.
- **Open the Application admin tool**
  - This is where installed applications live (empty now)
  - Keep this tool open!

EASY 

# App

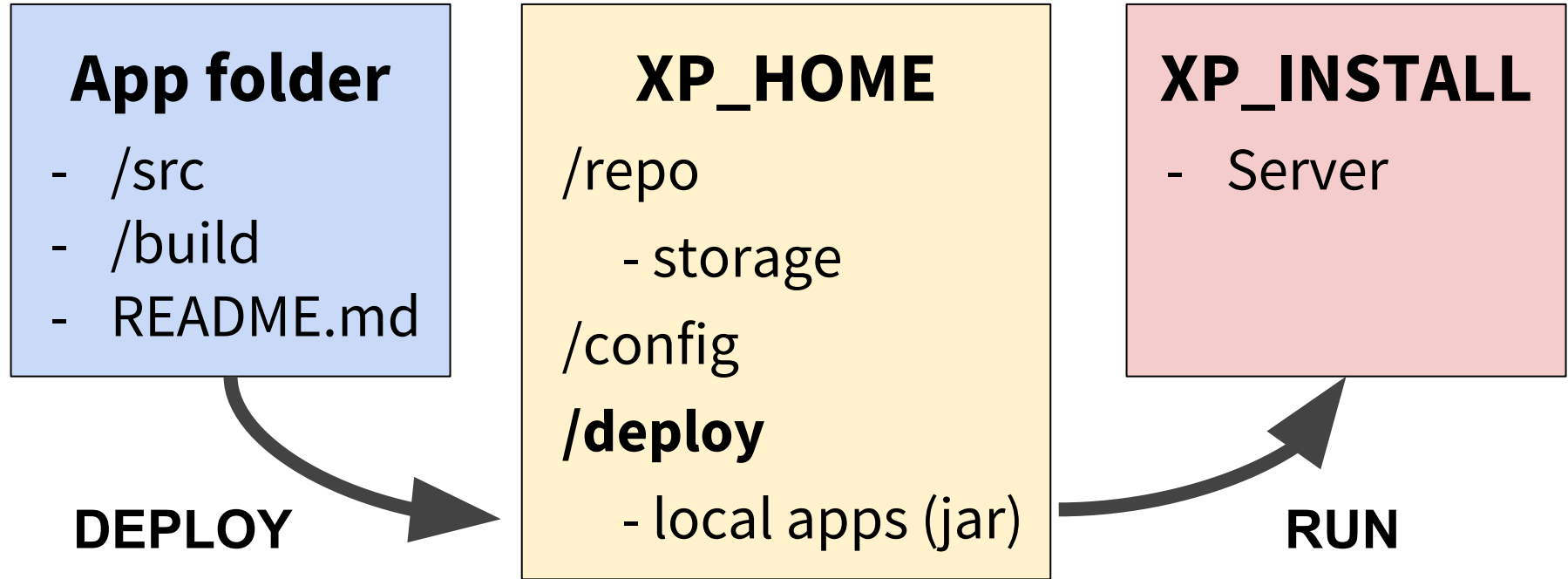
It's all about **Enonic Apps**

# App development basics

- Everything is built as Enonic Apps
  - Themes, websites, intranets, blog, “widgets”
- Rails-like structure of folders
  - Each folder has a specific meaning
- MVC structure of code
  - JavaScript for the logic, Thymeleaf for the view, XML for data models
- Built with Gradle (included)
- Installing your app:
  - Placing the artifact (JAR) file into **\$XP\_HOME/deploy**
    - ( can also be uploaded in Application admin tool )
  - Apps are automatically detected and started
- Check out [Enonic Market](#)



# Local app: 3 important folders



## Task - Setup your first app

Create a new app based on a “starter” (a starter kit, a “skeleton” for an app, a starting point). We’ll be using “**starter-academy**” that we’ve created for training.

- Download the zip from our Github repo
  - [github.com/enonic/starter-academy](https://github.com/enonic/starter-academy)
- Unzip to a folder on your desktop
- Rename this app folder to “dev101”
- Done

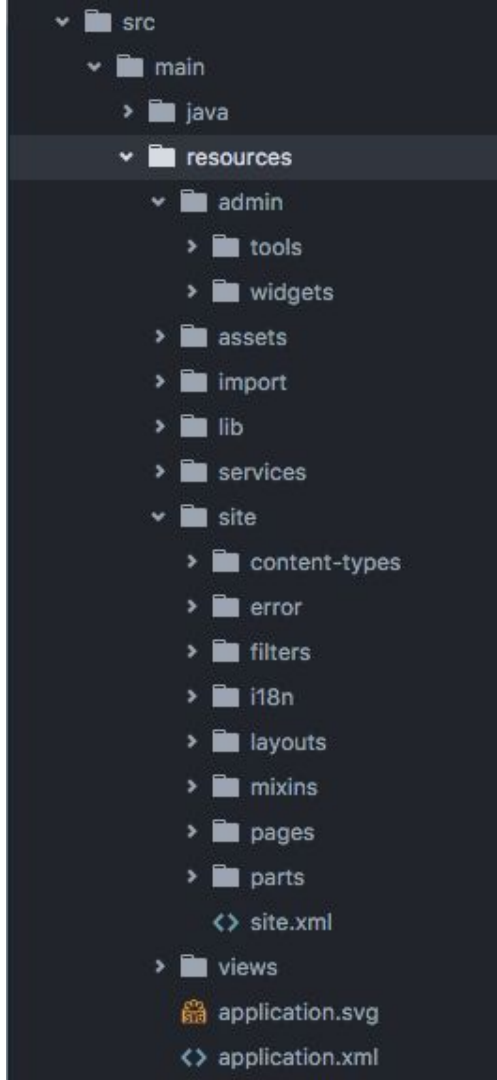
# App folder structure

Inspect the source. The “action” is in here:

`{ $APP }/src/main/resources/*`

Each folder here serves its own special purpose. Naming (even casing) is very important. But you can delete the ones you don't need.

We'll work in `/site/`



# Building

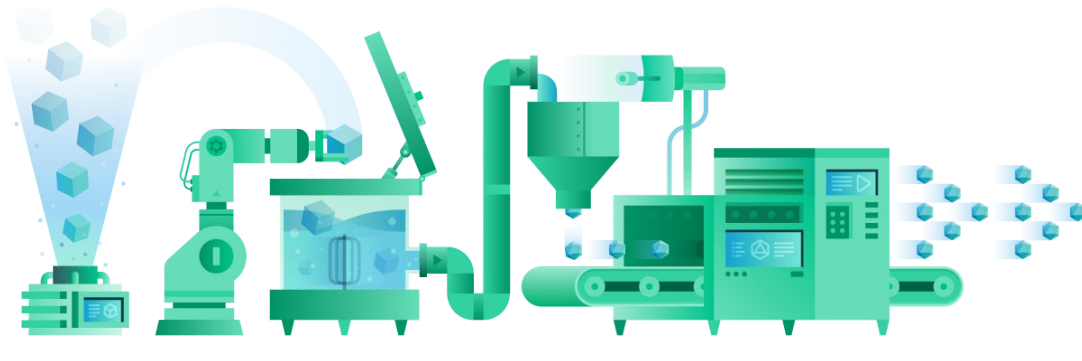
Get your code into XP

# Gradle

Our build-tool of choice, included with all our projects. Gets your code into XP. More info: [gradle.org](https://gradle.org)



JS, HTML, ++



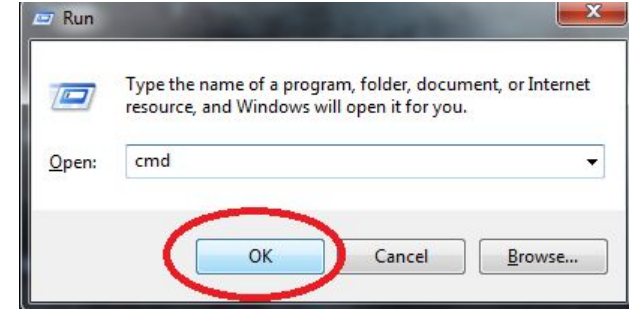
# Building your app

To get an app installed in XP we need to build and deploy a jar-file (containing the app source) to the place we refer to as **\$XP\_HOME**. This is where Gradle comes in.

- Needs some minor configuration/setup
  - The **\$XP\_HOME** system variable needs to be set
  - Gradle's deploy-command needs to be executed
- Gradle will then package and copy the .jar for you!
- Following slides will show the way.

# Command Line

- Use Windows key + R to start the “Run” prompt
- Type “cmd”, hit Enter key
- This starts the **command line terminal**, we’ll build our apps with this.



# Setting \$XP\_HOME

We need two things:

(1) The location of our XP installation **/home/** directory.

(2) Setting a system variable to here - **\$XP\_HOME**.

- Go back to the XP installation folder

- Go into the directory called **/home/**
- Click address bar and **copy** location
  - Something like `C:\Users\[user]\desktop\xp\home\`

- Go back to command line window

- Write “`set XP_HOME=`” and paste the directory path, hit Enter.

Full example: `set XP_HOME=c:\path-to-xp\home`



# Building your app

Use **Command Line** to go to your project's folder:

```
cd c:\Users\Bruker\Desktop\dev101
```

Build the app with the included Gradle (gradle wrapper):

```
gradlew deploy
```

- Wait for Gradle to download (subsequent builds are faster)
- Wait for Gradle to build the jar

(Protip! try continuous mode)

```
gradlew deploy -t
```

# Verify your App

Did the app build properly and get detected by XP?

- Go to your browser and Application admin tool
  - a new app should be there - “Starter Academy”
- Also pay attention to the **server logs**
  - XP will log when apps are installed or updated

```
op.ApplicationServiceImpl - Application [com.enonic.app.bootstrap] installed successfully  
op.ApplicationServiceImpl - Application [com.enonic.app.livetrace] installed successfully
```

- Changes to code are picked up instantly  
(after gradle is done building - pay attention to console)



## Task - Basic code editing

See the name of the Enonic App in Application admin tool, as well as detailed data about it (click to inspect). Let's change it!

- Open the file `/build.gradle` in app source root
- Change variable `displayName` on line 8
  - Invent a new name (your name?)
  - This is displayed in Application admin tool
- Open the file `/src/main/resources/application.xml`
  - Write a nice new description of the app (lorem ipsum?)
- Build changed app with Gradle
  - Verify changes in Application admin



Gradle `-t` cannot detect changes in build-files (stop and start gradle instead).



# Content Studio

XP's main admin interface

# The Content Studio

Use the “Launcher” to start this admin tool.

- Content browser (“The Grid”)
- Instant preview
- Actions: Preview, publish, delete, move, sort
- Create and edit content
- Shortcuts + keyboard support
- ... and more!

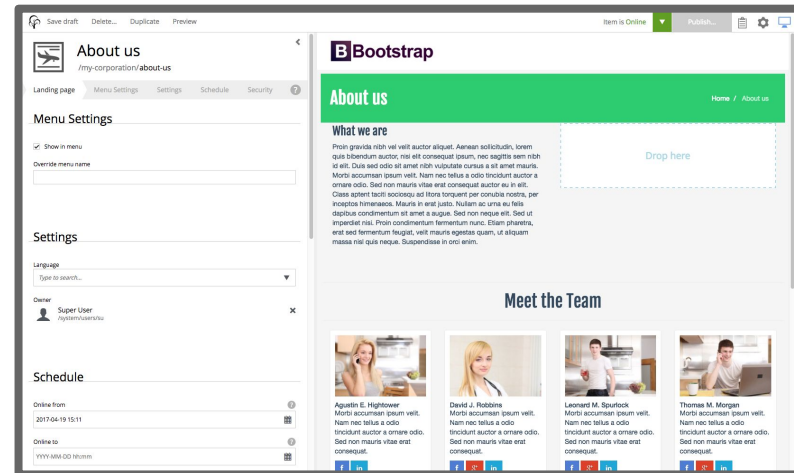
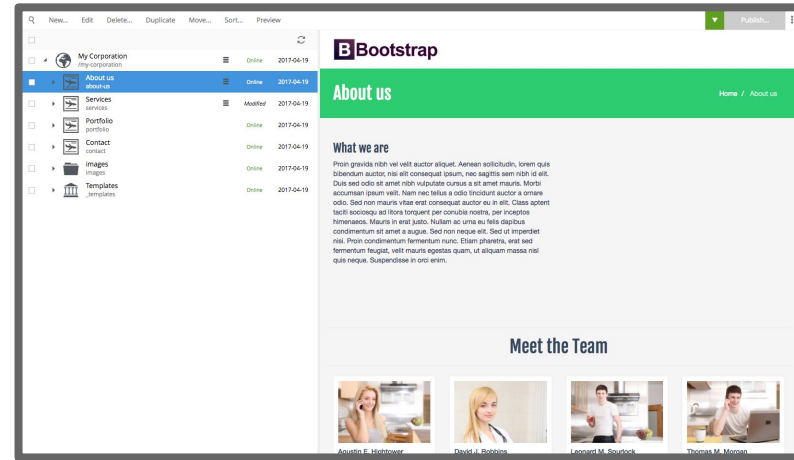
**Protip!** Attend our course “XP: Content 101” for more!

# Views: Browse and Edit

Noticed Content Studio has two views? Looks similar, but have different capabilities.

**Browse:** focus on navigating site, browsing content and tree structure. Static!

**Edit:** opens in new tab, with editing of forms for a page, advance editing of components, and more.



# Content

What's “Content” to Enonic XP?



# Task - Creating a Content

Let's get used to Content Studio by creating content, the simplest task. Use the “New” action in Content Studio to create a new country that we don't have already.

- Make sure this new Country is placed as **a child of the Site!**
  - Protip! Right-click Site and select “New”
- Click “Save Draft” when done
- Preview it from the Grid
- Close the tab when done
- Create a City inside (as a child item) this new country





# Content (instantiated in Content Studio)

- Each content is a separate item in the grid
- Has unique URL (based on hierarchy of content names)
- Can be published, scheduled
- Can be deleted, moved, sorted
- All contents are versioned
- Content have their own display (end-result HTML)
  - More on this later =)

Developer can define types of content that can be created in the Content Studio, these sets of rules are **Content Types**.

# Content Types (defined in code)

Content Types are like a class in a programming language, or tables in traditional databases, but for content in XP. Each content in XP are created out of one content type.

Examples: **articles, news, staff-members, offices, images, pdfs, calendar-entries, cars, videos**

- Anything you want to store data about, in a structured way.
- Dictates storage
- Handles validation
- Generates a form for the end-user
  - Consists of form fields, each uses one **input type**

# Input types

Docs: / Developer Guide / Schemas / **Input Types** /

XP ships with a big set of input types, all with different purposes and behaviour in Admin.

- TextLine, TextArea, HtmlArea
- RadioButton, CheckBox, ComboBox
- ContentSelector, ImageSelector (with upload)
- Long, Date, DateTime, CustomSelector, GeoPoint

Some common config, like <label>, <occurrences>, <default> and <help-text>

Time ...

For ...

Code!





# Task - Extend a Content Type

Docs: / Developer Guide / Schemas / **Input Types** /

We have a few content types in this app, but we need to extend the one for **Country**. Add another input type (a checkbox) to it.

- Add checkbox for “Is EU member”
  - / site / content-types / country / **country.xml**
- Deploy and find new checkbox it in Content Studio



*XP loads schemas when an Edit tab is opened.  
Reload open tabs to refresh schemas!*

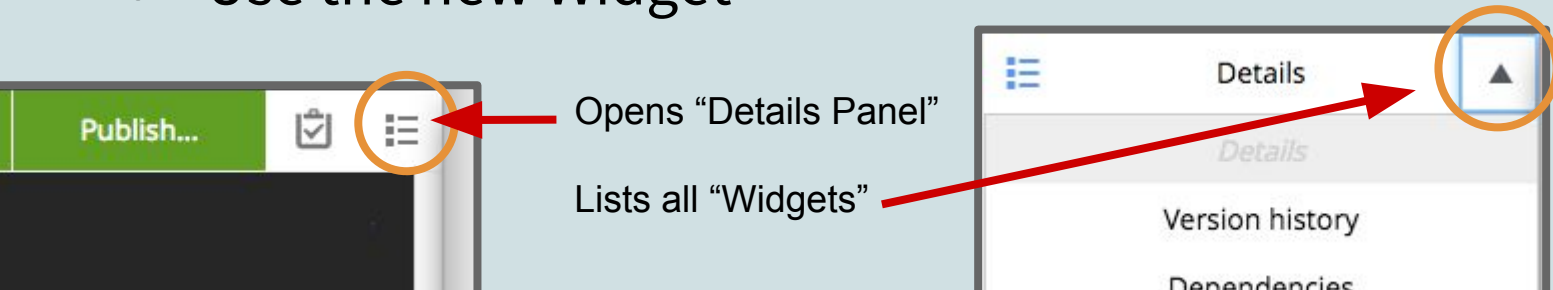
- Check it and save at least one content



## Task - Inspecting stored data

It's important to see how data is stored. Let's install an Enonic App from the Market, using the **Application admin tool**. We'll start with a variant of an App called a Widget.

- Click “Install” in the Application admin tool
- Type to filter list and install “Content Viewer”
- Go back to Content Studio and select a content
- Use the new Widget



EASY 

# How data is stored

The short version is that data is stored in XP as JSON, making JavaScript tasks easier to perform.

```
var heading1 = content.displayName;
```

“Root”-elements are data all content in XP share. All are useful, but of extra interest:

- displayName
- \_path
- \_id
- type

```
{
  "_id": "6de3c69b-324c-4fd0-926a-67e549f59d4e",
  "_name": "colombia",
  "_path": "/enonic-academy/colombia",
  "creator": "user:system:su",
  "modifier": "user:system:su",
  "createdTime": "2015-11-26T15:33:59.116Z",
  "modifiedTime": "2017-05-09T09:01:24.143Z",
  "owner": "user:system:su",
  "type": "com.enonic.starter.academy:country",
  "displayName": "Colombia",
  "hasChildren": false,
  "valid": true,
  "data": {
    "description": "Colombia, at the northern tip of South America, is a country of lush rainforest, towering mountains and coffee plantations. In the high-altitude capital, Bogotá, the Zona Rosa district is known for its restaurants, bars and shopping. Cartagena, on the Caribbean coast, features a walled colonial Old Town, a 16th-century castle and popular beaches. Nearby, culture-rich Barranquilla hosts a massive yearly Carnival.",
    "population": "48.32 million"
  },
  "x": {},
  "page": {},
  "attachments": {},
  "publish": {
    "from": "2017-05-09T09:01:24.143Z",
    "first": "2017-05-09T09:01:24.143Z"
  }
}
```

# A) Add, B) Store, C) Inspect

Previous two tasks form a best-practise pattern for extending and creating content types (and other XP components).

- **Add** fields
  - Make sure you see new fields in Content Studio
- **Store** new data
- **Inspect** data
  - Using tools like the Content Viewer widget



# Components

Dynamic little “modules” you develop

# Components intro

A Component will show either hard-coded things, things based on its configuration, a **content**, or a blend of them all.

Components are a central term in XP. You **create** Components in code, but use Content Studio to:

- Insert, delete, reorder components on pages
- Configure components individually

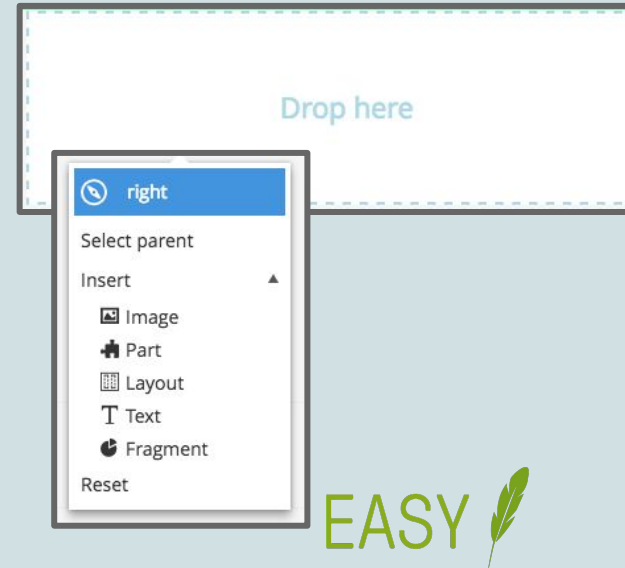
We have different types of components (more on that later).

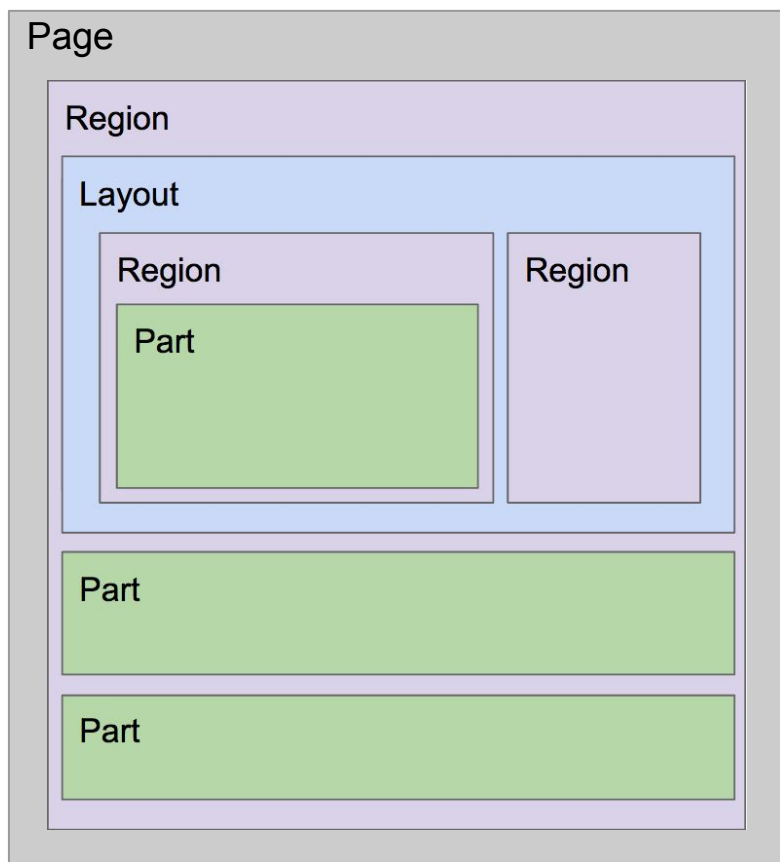
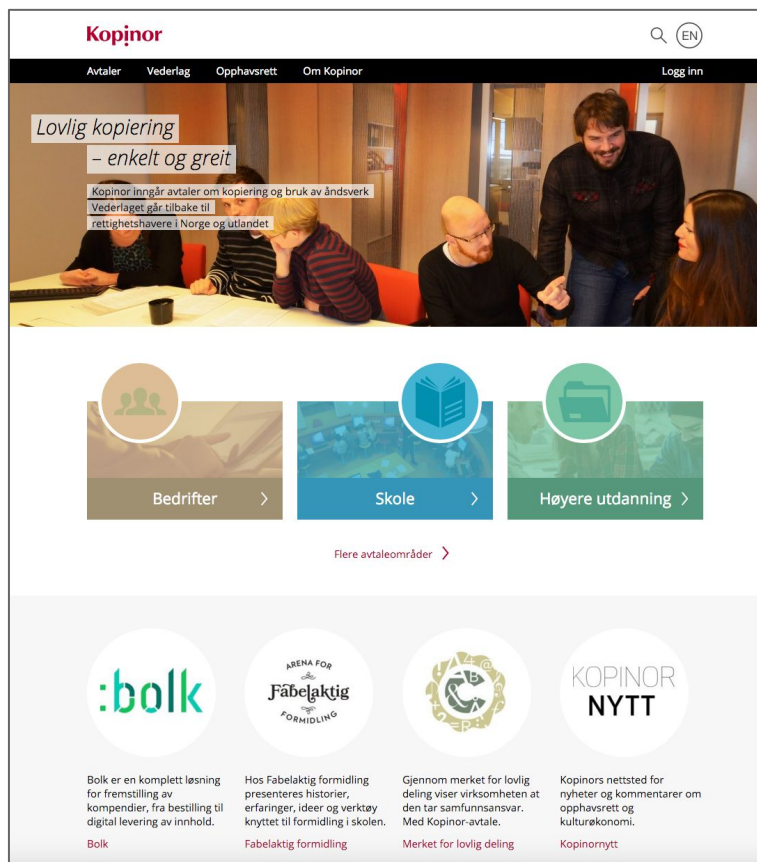


# Task - Adding a Component

There's a **Part component** in this app that is not yet used on the site. Edit the site content and click the right preview area.

- **Right click** on “Drop here”
- Click “Insert” in context menu
  - Click “Part” in the expanded list
- Now type “Banner” **and** select it
- Configure in the panel to the right
  - add image & text
- Save your changes





# Pages

Everything you see on a website built in XP.  
Each content has a URL - this is a **page**.

- Has a “Page Controller”
  - It’s like an **engine**
  - Created in your app code
- Has fixed code
  - Headers, Footers, Menus
  - Metadata
  - JavaScripts, Analytics
- High re-usability

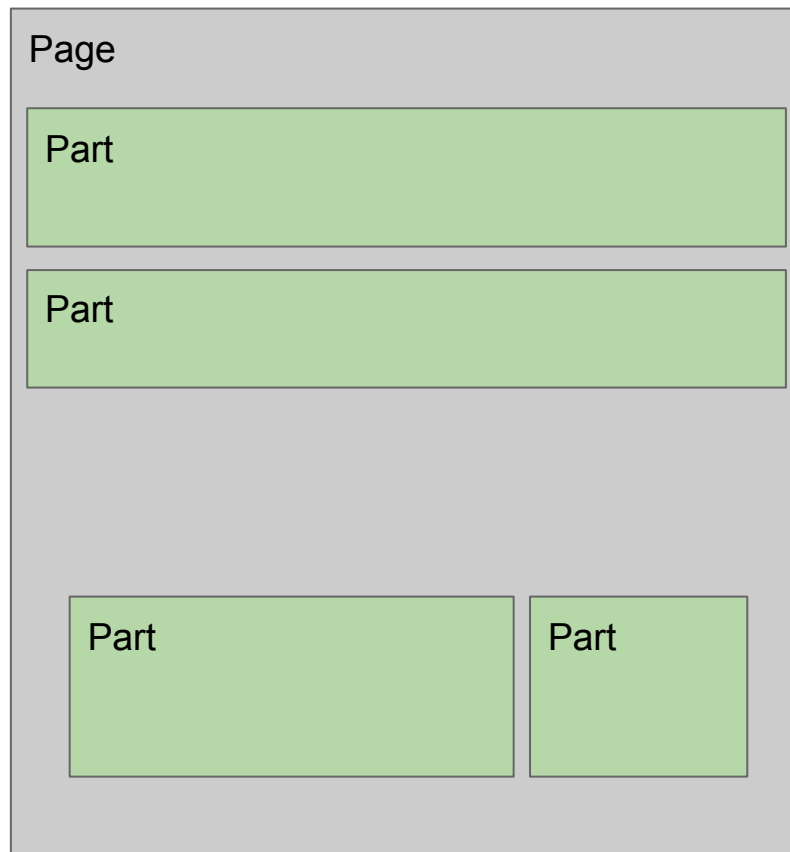
Page



# Parts

Pluggable “modules”. Our key **Component**. Placed from Content Studio.

- Placed on your pages
  - Creates dynamic pages
- You build and define
- Self contained
- Re-usable

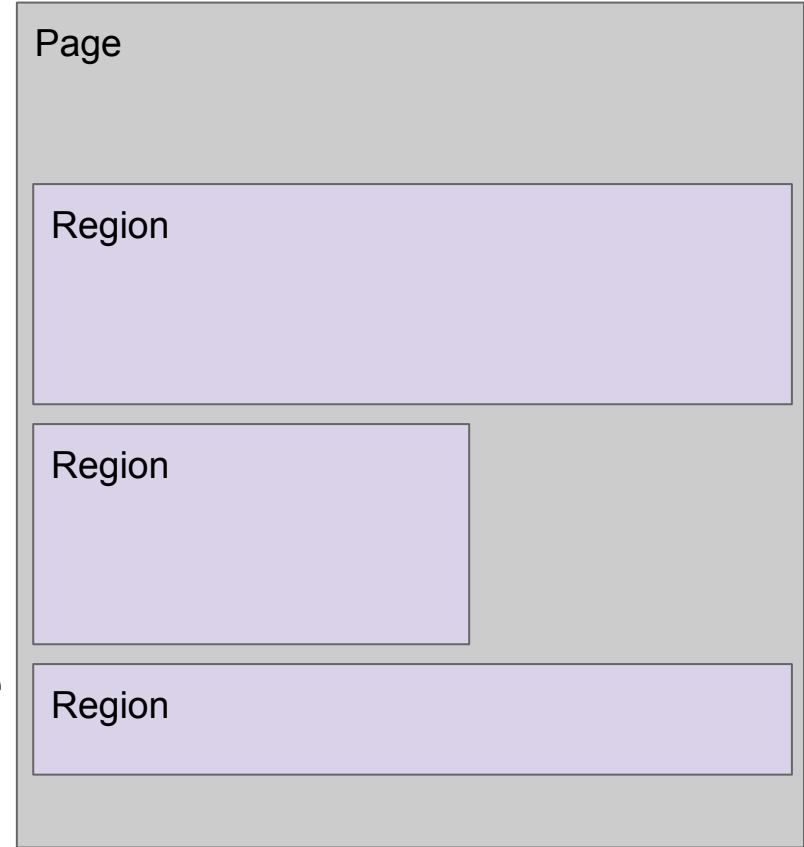


# Pages + Regions

The Page defines Regions.

Page Controllers found in  
`/site/pages/*` in app source.

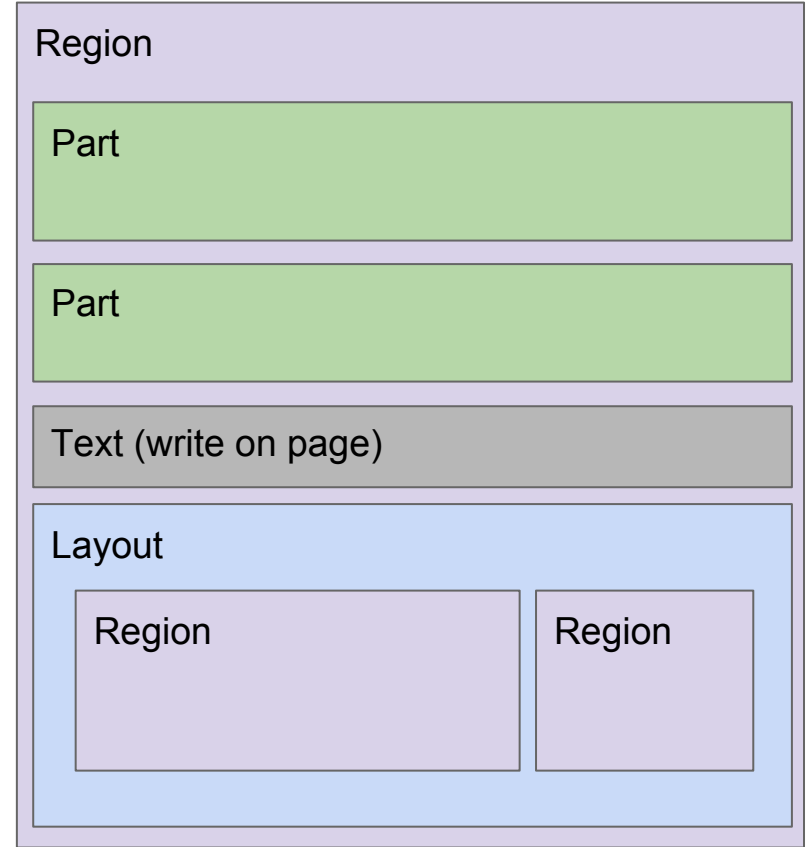
- You create each region
- You define where on page
- A placeholder for components
  - Displays “Drop Here” in CS
- Makes the pages customizable



# Regions

We know they're defined in the Page Controller. Region specs:

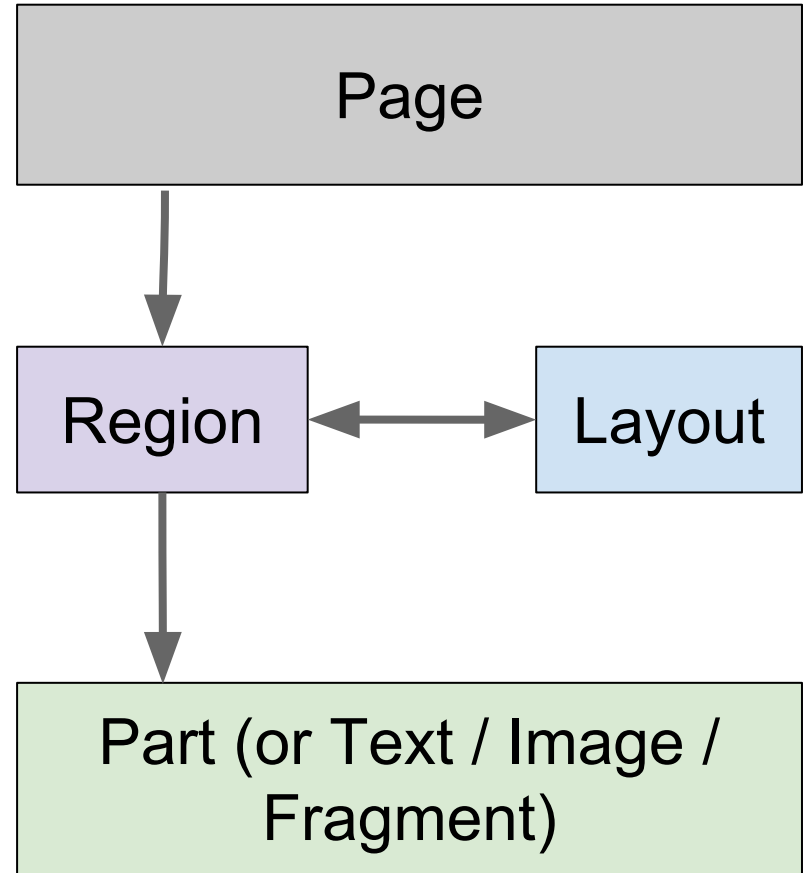
- Has unique name
- No limits on number per page
- No regions inside regions!
- Not supported in Parts
- Can have zero, one or more **regions**
  - 1 region is most common



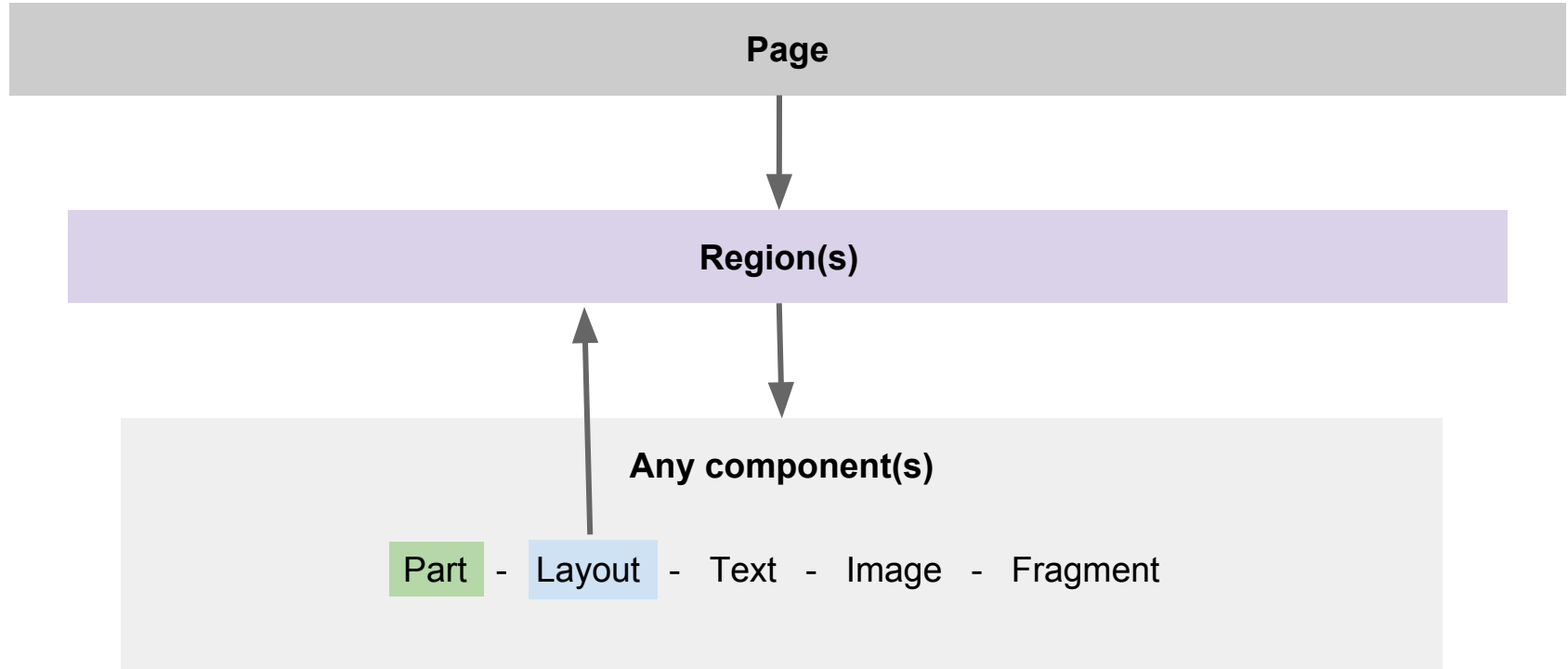


# Hierarchy

- A page contains 1+ regions, in code
- Drop Parts and Layouts (etc) into Regions, in Content Studio
- A layout contains more regions, in code



# Summary



Part

Part

Part

Region

Page

Kopinor

Avtaler

Vederlag

Opphavsrett

Om Kopinor

Logg inn

EN

Lovlig kopiering

- enkelt og greit

Kopinor inngår avtaler om kopiering og bruk av åndsverk

Vederlaget går tilbake til rettighetshavere i Norge og utlandet

Bedrifter

Skole

Høyere utdanning

Flere avtaleområder

:bolk

Bolk er en komplett løsning for fremstilling av kompendier, fra bestilling til digital levering av innhold.

Bolk

ARENA FOR Fabelaktig FORMIDLING

Hos Fabelaktig formidling presenteres historier, erfaringer, ideer og verktøy knyttet til formidling i skolen.

Fabelaktig formidling

Gjennom merket for lovlig deling viser virksomheten at den tar samfunnsansvar. Med Kopinor-avtale.

Merket for lovlig deling

KOPINOR NYTT

Kopinors nettsted for nyheter og kommentarer om opphavsrett og kulturøkonomi.

Kopinornytt

59

# MVC

At the core of all App's components.

# Anatomy of a Component - MVC

## Model

*"What data do we want to store, which format, and how? Our 'database'."*

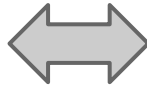
XML Schemas - Content types and Descriptor configurations



## Controller

*"Fetch and manipulate our data"*

JavaScript (server side)



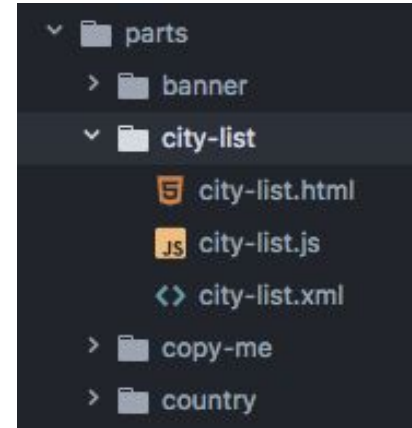
## View

*"How do we want to display our data?" (front-end)*

Thymeleaf, Mustache, XSLT

# Anatomy of a Component

- Placed in specific “root” folder
  - i.e. “/site/parts/” for part components
- One folder per component
  - No spaces, prefer camel- or dash-case
  - “/site/parts/**component-name**/”
- Usually contains up to three files
  - Controller - “**component-name**.js”
  - View - “anything”.html - (not required)
  - Descriptor - “**component-name**.xml” - (not required)



# Model / Descriptor

XML

# The Descriptor in practice

```
<part>
  <display-name>Banner</display-name>
  <config>
    <input type="TextLine" name="header">
      <label>Header</label>
      <occurrences minimum="0" maximum="1"/>
      <default>Awesome land</default>
    </input>
    <input type="TextArea" name="description">
      <label>Description</label>
      <occurrences minimum="0" maximum="1"/>
      <default>Such an amazing country, everyone should vi
    </input>
    <input type="ImageSelector" name="imageKey">
      <label>Image</label>
      <occurrences minimum="0" maximum="1"/>
      <config>
        <allowPath>${site}/*</allowPath>
      </config>
    </input>
  </config>
</part>
```

Name of component, displayed in admin  
(and searchable) - "Insert **Banner**"

Input type (more in next slide)

Label - shown in admin

"Database" name (JSON node name), used in  
your controller when fetching data.

Control how many of these we  
should support. 0 to 1 is the  
normal. 1 to 1 means  
"required", maximum 0 means  
"infinite".



# How data is stored

The “name” field in the Descriptor decides how to store each field value.

```
var population = content.data.population;
```

```
1 <content-type>
2   <display-name>Country</display-name>
3   <super-type>base:structured</super-type>
4   <form>
5     <input type="TextArea" name="description">
6       <label>Description</label>
7       <occurrences minimum="0" maximum="1"/>
8     </input>
9     <input type="TextLine" name="population">
10      <label>Population</label>
11      <occurrences minimum="0" maximum="1"/>
12    </input>
13    <!-- Mixin of the type "inline" goes here -->
14  </form>
15 </content-type>
16
```

```
{
  "_id": "6de3c69b-324c-4fd0-926a-67e549f59d4e",
  "_name": "colombia",
  "_path": "/enonic-academy/colombia",
  "creator": "user:system:su",
  "modifier": "user:system:su",
  "createdTime": "2015-11-26T15:33:59.116Z",
  "modifiedTime": "2017-05-09T09:01:24.143Z",
  "owner": "user:system:su",
  "type": "com.enonic.starter.academy:country",
  "displayName": "Colombia",
  "hasChildren": false,
  "valid": true,
  "data": {
    "description": "Colombia, at the northern tip of South America, is a country of lush rainforest, towering mountains and coffee plantations. In the high-altitude capital, Bogotá, the Zona Rosa district is known for its restaurants, bars and shopping. Cartagena, on the Caribbean coast, features a walled colonial Old Town, a 16th-century castle and popular beaches. Nearby, culture-rich Barranquilla hosts a massive yearly Carnival.",
    "population": "48.32 million"
  },
  "x": {},
  "page": {},
  "attachments": {},
  "publish": {
    "from": "2017-05-09T09:01:24.143Z",
    "first": "2017-05-09T09:01:24.143Z"
  }
}
```

# Controller

Javascript

# The **Controller** in practice

## Global scope

Add libraries here (with require), & global settings.

## Local scope

Multiple “exports.” supported!  
**get**, **post** or **delete**. Or any custom function name.

```
var libs = {  
  portal: require('/lib/xp/portal'), // Import the portal functions  
  thymeleaf: require('/lib/xp/thymeleaf') // Import the Thymeleaf rendering fu  
};  
  
// Specify the view file to use  
var conf = {  
  view: resolve('country.html')  
};  
  
// Handle the GET request  
exports.get = function(req) {  
  // Get the country content and extract the needed data from the JSON  
  var content = libs.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  
  };  
  
  // Return the merged view and model in the response object  
  return {  
    body: libs.thymeleaf.render(conf.view, model)  
  };  
};
```

# The **Controller** in practice (cont.)

Libraries

Components view file

Handle **get**-requests

Query storage, get data

Build the model (js-object)  
to be sent to the view  
(html/thymeleaf)

Return something (use  
thymeleaf to parse this)

```
var libs = {  
  portal: require('/lib/xp/portal'), // Import the portal functions  
  thymeleaf: require('/lib/xp/thymeleaf') // Import the Thymeleaf rendering fu  
};  
  
// Specify the view file to use  
var conf = {  
  view: resolve('country.html')  
};  
  
// Handle the GET request  
exports.get = function(req) {  
  // Get the country content and extract the needed data from the JSON  
  var content = libs.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  
  };  
  
  // Return the merged view and model in the response object  
  return {  
    body: libs.thymeleaf.render(conf.view, model)  
  }  
};
```

# The **Controller** in practice (cont.)

In most cases, these are the only lines you will change for each part, page or layout. The rest will stay mostly the same.

*Path to the view file is unique to each component*

```
var libs = {  
  portal: require('/lib/xp/portal'), // Import the portal functions  
  thymeleaf: require('/lib/xp/thymeleaf') // Import the Thymeleaf rendering fu  
};  
  
// Specify the view file to use  
var conf = {  
  view: resolve('country.html')  
};  
  
// Handle the GET request  
exports.get = function(req) {  
  // Get the country content and extract the needed data from the JSON  
  var content = libs.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  
  };  
  
  // Return the merged view and model in the response object  
  return {  
    body: libs.thymeleaf.render(conf.view, model)  
  }  
};
```

# The **Controller** in practice (cont.)

Remember this  
**Controller pattern:**

- **Collect**
- **Manipulate**
- **Prepare**
- **Return**

```
var libs = {  
  portal: require('/lib/xp/portal'), // Import the portal functions  
  thymeleaf: require('/lib/xp/thymeleaf') // Import the Thymeleaf rendering fu  
};  
  
// Specify the view file to use  
var conf = {  
  view: resolve('country.html')  
};  
  
// Handle the GET request  
exports.get = function(req) {  
  
  // Get the country content and extract the needed data from the JSON  
  var content = libs.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  
  };  
  
  // Return the merged view and model in the response object  
  return {  
    body: libs.thymeleaf.render(conf.view, model)  
  }  
};
```

# View

Thymeleaf, Mustache, XSLT, ...

# Thymeleaf

A natural templating language. You write your HTML5 just as usual, but add “data”-attributes to your elements to do the magic.

data-th:**text**

th:**text**



# The **View** in practice

```
1 <div class="xp-part">
2   ... <h1 data-th-text="${name}"></h1>
3   ... <p data-th-if="${population}" data-th-text="'Population: ' + ${population}"></p>
4   ... <p data-th-if="${description}" data-th-text="${description}"></p>
5 </div>
```

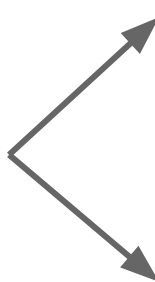
If we find a value in the variable “description” this tag (<p>) will be displayed. Otherwise not.

Fill this html tag (<p>) with text (html is escaped) from the variable “description”.

```
<div class="xp-part">
> <h1>Sweden</h1>
> <p>Population: 9 828 655</p>
> <p>Sweden is a Scandinavian nation of thousands of coastal islands, inland lakes,
</div>
```

# The **View** in practice (cont.)

Note!



```
<div class="xp-part cities">
  <h2>Cities</h2>
  <div class="city" data-th-each="city : ${cities}">
    <h3 data-th-text="${city.name}"></h3>
    <p data-th-if="${city.population}" data-th-text="${city.population}"></p>
  </div>
</div>
```

**Notice** - only **one** root html node in any view file! One - and only one - is allowed. Not two, not zero. *Invalid views won't show and/or work as expected!*

Looping is easy! Send a JSON array from your Controller and use **data-th-each** to iterate over it.

**city : \${cities}** - explained:

- > **city** is the local variable for the loop for each item
- > **\${cities}** is sent from controller as an JS array
- > Access each item with **city.name** (*localName.arrayItemName*)

# The **View** in practice (cont.)

```
<a href="website.html" data-th-text="${newLinkText}" />
<div data-th-if="${variable}">Do I exist?</div>
<div data-th-text="${someText}"><span>I'm <strong>gone</strong></span></div>
```

## Gotchas:

- > data-th-text fills current tag with content (deletes anything already there, even tags!).
- > a false “if” (or null/undefined) removes tag & content!
- > data-th-src overwrites src tag

```



<a href="website.html" data-th-href="${newUrl}" />
<div data-th-remove="tag">Do something in here ...</div>
```

## Useful tags:

data-th-src  
data-th-alt  
data-th-title  
data-th-href

## data-th-remove="tag"

Used to clean up your resulting html. This will remove the wrapping <div> and leave only the text there!

# Build Parts

Let's extend and develop Parts



# Task - Output data from Content

- Collect new EU-data in the part's controller
  - / site / parts / country / **country.js**
  - It's already collected with `portal.getContent()`
  - Attribute "name" on the input field = JS object property  
name *<input name="test" is stored as "data.test"*
- Add it to the js object being returned to view
  - The "model" variable on line 18
- Output in Thymeleaf ("if member then")
  - `<p data-th-if="{member}">Member!</p>`

# getContent() from lib-portal

When viewing a specific content, you need a component to actually extract the data. Add storage first (to the Content Type schema), then extract this in a component of choice (Part).

Use **getContent** to get the data stored for the content you are currently viewing.

(It's like running `SELECT * FROM x WHERE id = 'y'` in SQL)



# Task - My first Part - creation

Let's create a completely new part, from scratch!

- Duplicate the part called “copy-me”
  - Rename it (files and folder) to “first-part”
  - Edit conf.view variable in controller (to first-part.html)
  - Set a good <display-name> in the part descriptor (xml)
- Update the Thymeleaf to display text:  
“<p>This is my First Part!</p>” (all else inside <div> can be replaced)
- Deploy code (without errors in log)
- Use Application admin tool to see part





## Task - **My first Part** - adding to page

- Edit the Site in Content Studio
- Right click in the Page Editor
- Select “Insert” and then “Part”
- Type to search for your part
  - This searches in the XML <display-name>!
- Select your new part in the drop down

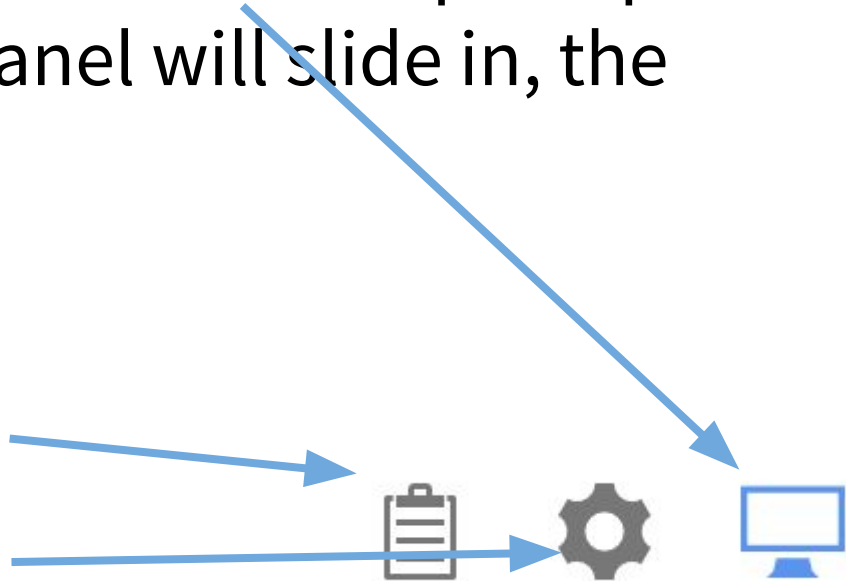
After: instructor will show two other ways!



# Advanced editing

In Edit view, to the right we have the preview. Click this area - called **Page Editor** - to open up advanced editing. A panel will slide in, the **Inspection Panel**.

- **Component View**
- **Inspection Panel**





# Task - Empower your Part (1 of 2)

Docs: / Developer Guide / Schemas / Input Types / **TextLine** /

Let's go back to our “My First Part” and make it better. We'll start by adding new configuration option to it, and test it.

**1. Add config.** 2. Query data. 3. Send to Thymeleaf. 4. Display

- Edit your descriptor - `first-part.xml`
- (1) Add an input type, TextLine
  - (to control your hard-coded text from admin instead)
- **Deploy**
- Find and verify the field in Content Studio





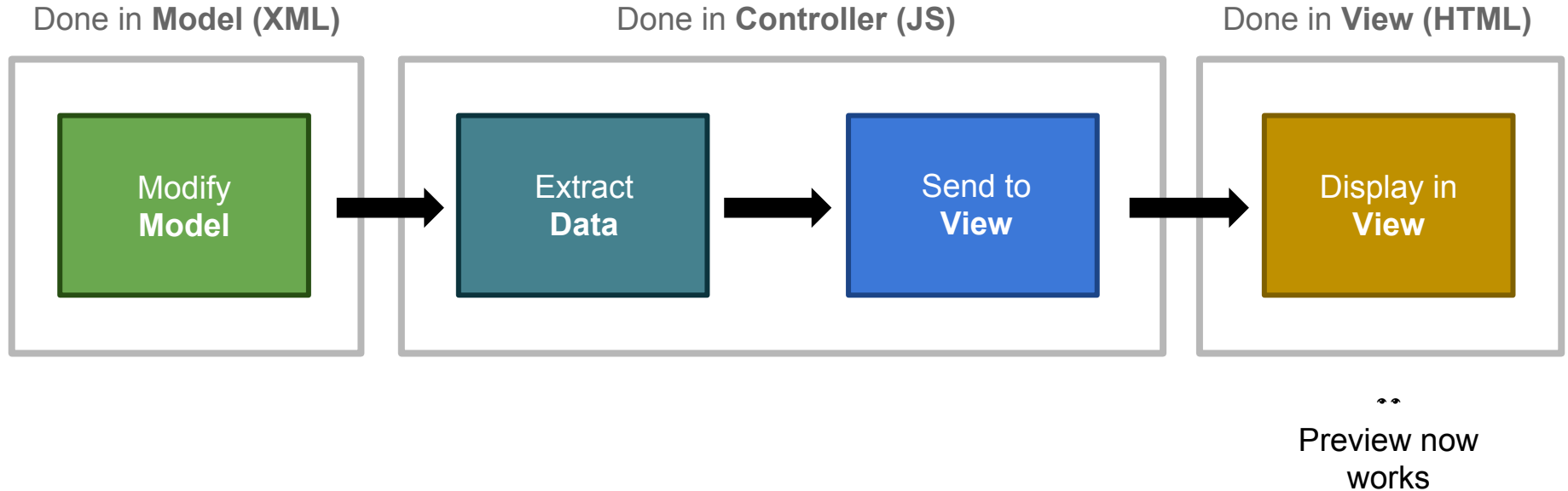
# Task - Empower your Part (2 of 2)

Your part has a new config, let's extract that data!

1. Add config. **2. Query data.** **3. Send to Thymeleaf.** **4. Display**


- In the Controller:
  - (2) Fetch new data with `libs.portal.getComponent();`
    - **Warning!** JS object data format is slightly changed!
  - (3) Store data in the model-object
  - (3) Make sure model-object is sent to View
- (4) View: Output data in Thymeleaf with `data-th-text`
- **Deploy**
- Verify by changing text from Content Studio

# The XP workflow - components



# getComponent() from lib-portal

Whenever you want configuration on any component (Part, Layout, Page), you can add these to the Descriptor (its XML file).

- The descriptor is not a mandatory file.
- Just deploy and reload content in admin to see form
-  *Schemas are only updated on page load!*
- Extract the input with `libs.portal.getComponent()`;
  - **lib-portal** is a library bundled with Enonic XP
- JS object format differs slightly from `getContent()`



**Protip:** To refresh a part with new data, hit “Apply” button

# Layout

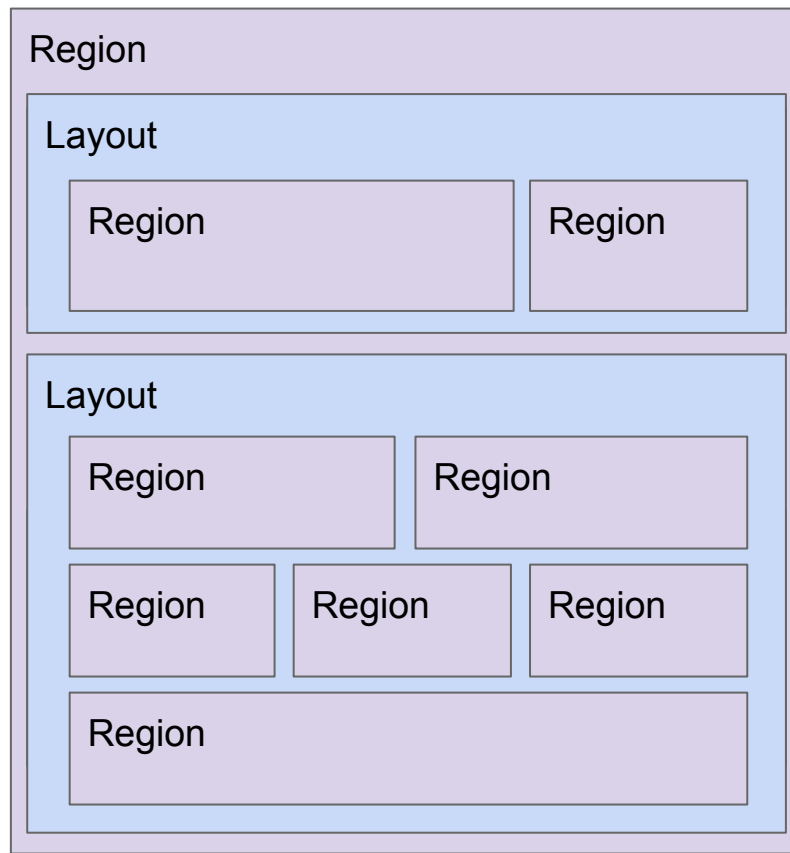
Another type of Component

# Layouts

Remember we said “no regions inside regions”?

Introducing: **Layouts**

- Placed in a **page region**
- Defines new regions
  - Any number
  - Any placement
- Style with CSS



– og det med god samvittighet  
bare Kopinor-avtalen er på plass

Les mer om rettighetsmerket





# Task - Layouts

Docs: / Developer Guide / Sites / **Layout** /

Let's add a Layout component to the front page (the site content is the front page) and play with XP's Text Component.

- Edit the Site content
- Add a Layout component
  - Named "**2 columns**"
- Add a Text Component to each side
- Write some "lorem ipsum" text in both of them

Done? Check app source code to see how layouts work.



# Content, 2

Real-world example: building a content type



# Task - My first Content Type

Docs: / Developer Guide / Sites / Content Types / **Custom..** /

- Create a new content type

- Duplicate folder / site / content-types / **duplicate-me** /
- Rename folder **and** XML-file to “celebrity”
- **Deploy**

- Click “New” in Content Studio



**Protip:** No reload required!

**Protip:** Type to filter in the list

- Create a celebrity inside a country

( We'll extract this data later, now we just create the Type )

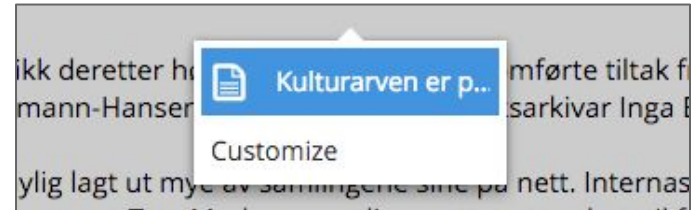
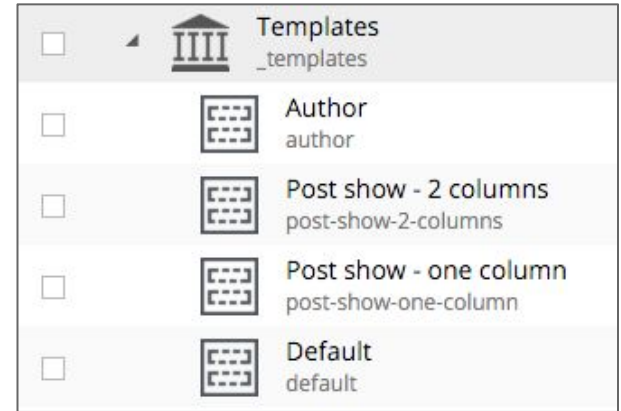
# Templates

A special contenttype

# Templates

Templates are created, stored,  
and published from the Content Studio

- “Special” folder
  - Don’t rename, move or delete it!
- Comes with every site
- Used to automate website display
- Any number of templates
- Content can override this
  - Click “Customize Page”



# Templates (cont.)

- Governed by one Page controller (JavaScript-file)
  - Different templates can use the same page controller
- Select content types it “Supports”
  - Automatically used by XP for previewing content
  - Automatically used for display when creating new content
- Has any selection of components



**Protip:** Aim to support only one content type per template!

Example: View article text - “article show”, adds parts that extracts this data to

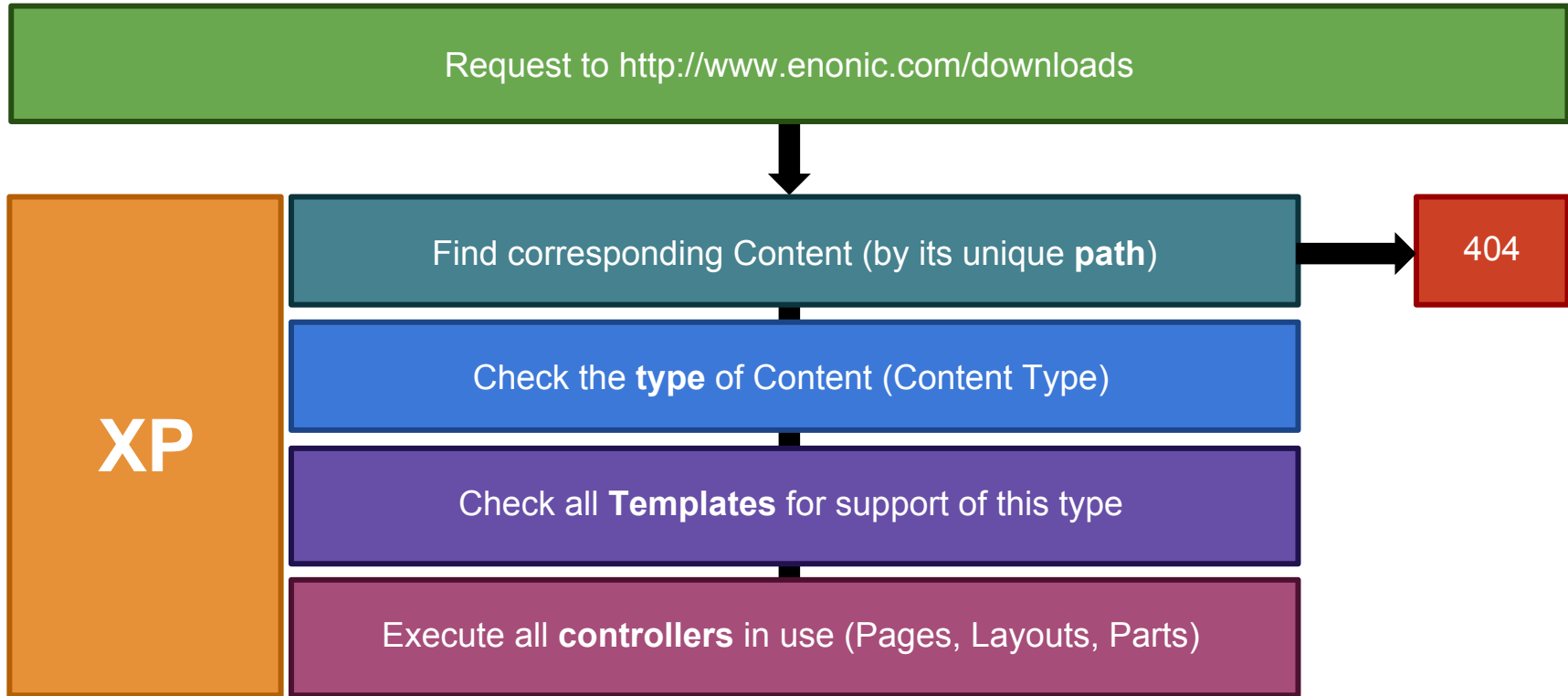


# Task - Your first template

Page Templates are key to render anything in XP. No content knows by itself how it should look. Clicking any City or Celebrity in Content Studio will show blank pages.

- Go back to Content Studio
- Right-click the folder called “Template” in your site
- Select New and then “Template” (the only option)
  - Give it a name - “Celebrity” - (same name as content type is good practise)
  - Let it “support” content type **Celebrity**
  - Use Page Controller called “**Default**”
    - *(Page controller is selected in the blue area to the right)*
  - Write on the page with Text Component
    - Like “This is the Celebrity Template! It will show more soon ...”
- Preview any celebrity content as proof

# A wild visitor appears





# Time to step it up

Medium tasks - images, references, ++



# Task - A Part for your Content (1 of 2)

First we'll create the Part and add it to a template. The code comes afterwards.

- Create a new Part
  - Duplicate folder / site / parts / **copy-me** /
  - Rename folder **and** all containing files to “celebrity”
  - **Don't forget** **conf.view** variable in top of controller file!
  - **Don't forget** to change **<display-name>** in XML file!
- Deploy
- Open the Template “Celebrity” you created earlier
  - Add your new Part to this template
  - Save and preview any Celebrity



## Task - A Part for your Content (2 of 2)

With the Part created and in place, let's code some! We need to extract the stored data from our content type.

- Open the part controller in your code editor
- Make sure `libs.portal.getContent()` is there
  - ( this one fetches the stored data )
- Extract “displayName” from the contents’ JSON
  - ( as a separate variable )
- Store it in the JS object called `model`
- Display this data in Thymeleaf
- **Deploy** and test

# XP fundamental concepts

- Everything is content (*even folders, sites, templates*)
  - Needed to see anything from a site
  - Creates the actual URLs
  - Indexed and searchable instantly
- Creating content requires a Content Type
  - Contains the rules for what to store (and how)
  - XP has built-in types, but you can easily add custom
- Content is created and edited in Content Studio
- Templates helps showing content
  - With essential Components added to them

# The XP workflow

When developing a new feature for an app **ask yourself:** do we need a Content Type?

1. Create storage
2. Create part(s) extracting data
  - a. **x-list** - list some data from all items (if needed)
  - b. **x-single** - view all data for a single item (if needed)
3. Add part(s) to a Template
4. Create content in Content Studio

# Frontend assets (CSS, JS)

A common task for a web developer is adding local assets to an app, like javascripts, CSS, and graphical images.

Items that are not editable from Content Studio, but bundled with the App itself.

Assets are placed in the app's `/assets/` folder.

Their URL needs to be dynamically generated - `portal.assetUrl()`



# Task - `assetUrl` function

Docs: / API and Reference / View Functions / **`assetUrl`** /

- Create a **new** css file in the App's `/assets/`-folder
  - Call it `overwrite.css`
  - Add `html, body { background-color: red !important; }`
- Get the correct URL to it with `assetUrl()` View Function  
( See how we add CSS in “default.html” )
- Add this to your main page controller's view-file  
(`default.html`) and verify

# Content Studio assets

When it comes to images that editors upload and control, from Content Studio, we need another function.

Introducing `portal.imageUrl()`

Requires the image's **key**, will return the full URL.

*All references (like an article's header image) are stored in XP with their keys only. Each content item has an unique key (use Content Viewer). You can reorganize without worry!*



**Protip:** Use Content Viewer widget to see data!





# Task - Working with images

Docs: / Developer / Schemas / Input Types / **ImageSelector** /

- Add an **ImageSelector** to your content type “Celebrity”
  - Label: “Profile picture”, occurrences: min 0, max 1.
- Modify the Part’s controller (/celebrity/celebrity.js)
- Send new data to Thymeleaf
  - Use data for an image’s source with `data-th-src`
- **Discovery** - the image will fail! *Inspect html source*
- Try using `portal.imageUrl()` function in Thymeleaf

```

```

# portal.imageUrl()

Docs: / API and Reference Guide / View Functions / **imageUrl** /

As discovered, actual data stored for images (and all content references) in XP are merely keys. Not usable on their own.

Run them through portal.imageUrl()-function to return a dynamical URL based on context.

Mandatory: “**\_id**” (or “**\_path**”) to image, and a “**\_scale**” setting.  
Also has parameters like “**\_filter**”, “**\_type**”, “**\_quality**”

# Functions: View vs. Controller

`imageUrl()` and `assetUrl()` are something we call **View Functions** - they're used in your View (Thymeleaf/XSLT/Mustache).

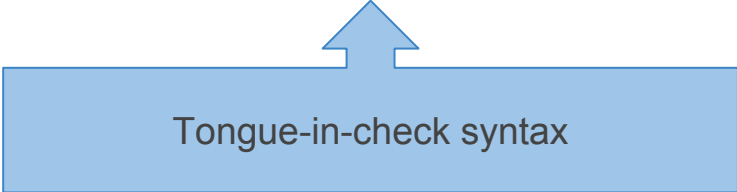
The code behind them is also available directly in your JavaScript, it's then referred to as a **Controller Function** (*documented separately*).

- They work exactly the same (same Java backend)
- Slightly different syntax!

```

```

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



Tongue-in-check syntax

# JavaScript lib docs

Docs: / API and Reference Guide / **JavaScript Libraries** /

We have a **separate** documentation, lib-doc, for all our controller functions!

Detailed information on all our libraries and available functions and their parameters.

- Generated from XP source code
- Versioned with XP
- Use **Ctrl/Cmd + F** to search
- Functions displayed to the left
  - Grouped under containing library

<a href="#">Home</a>
<b>CLASSES</b>
Cache <ul style="list-style-type: none"><li>clear</li><li>get</li><li>getSize</li></ul>
Resource <ul style="list-style-type: none"><li>exists</li><li>getSize</li><li>getStream</li></ul>
<b>MODULES</b>
lib/xp/auth <ul style="list-style-type: none"><li>addMembers</li><li>changePassword</li><li>createGroup</li><li>createUser</li><li>findPrincipals</li><li>findUsers</li><li>generatePassword</li><li>getIdProviderConfig</li></ul>



# Task - Images - controller function

Docs: / API and Reference Guide / View Functions / **imageUrl** /

Docs: / API and Reference Guide / **JavaScript Libraries** /

Head to our Controller Function documentation and find the definition and examples for `portal.imageUrl()`.

- Migrate the function to JavaScript instead!
  - Send the image ID into the function
  - Store in a new variable - imageUrl
  - Make sure to pass it over to Thymeleaf
- Remove the use of `portal.imageUrl()` in Thymeleaf
- Save and deploy
- Verify that the image shows up



# Task - The HTML-editor

Docs: / Developer / Schemas / Input Types / **HtmlArea** /

Using the HtmlArea introduces us to this same issue, image and links are stored as key-references.

Example href="content://kjhsf879856" and src="media://fdghm35552".

- Add a new input type, **HtmlArea**, to **Celebrity** content type
  - Write something in this field in admin (with links and images)
- Extract the data in Controller, and send to Thymeleaf
- Use the **data-th-utext** to output HTML unescaped!
- **Discovery** - No links or images will work!
- Run view-function **portal.processHtml()** on html



# Publishing

Going live in XP

# Publishing, paths, branches



- Branches: **draft & master**
  - End user only sees contents in master
  - Logged in administrator only sees draft ( in Content Studio, Preview )
- You must actively publish items
  - Bulk publishing, and “with children”, supported
  - Content is **copied** into “master” (kept in draft).
- Saving **only** affects “draft” branch





# Task - Go live!

Editors (or developers) decide when and which contents should be available to the outside world. It's called publishing.

- Publish your entire site
  - Agree to include all related items, and all children
  - Wait for publishing to finish
- Click preview on site
- Change URL here
  - **Remove** “/admin” and “/preview”
  - **Change** “/draft” to “/master”

# Publishing, paths, branches (cont.)

- [/admin/portal/preview/draft/enonic-academy/norway](#)
  - Preview mode (**draft**)
  - (must be logged in)
- [/portal/master/enonic-academy/norway](#)
  - Master branch (“live”)
  - Only published content can be accessed
- To get to master:
  - Use preview mode URL
  - Remove “/admin” and “/preview”
  - Change “/draft” to “/master”

# Pro tips, Future, etc

Common errors and solutions, what's next, ++

# Solutions to common problems

- New stuff doesn't appear in Content Studio
  - `gradle -t` sometimes builds mid-way through changes, might result in corrupted jar-file. Stop and start again.
  - Always double-check `$XP_HOME` is correct.
  - Double-check XP and Gradle uses identical `$XP_HOME`
  - Try logging from controller (`log.info`) just to make sure
- Old things appear in Content Studio, no new stuff
  - Make sure you're not inside the `/build/`-folder editing
  - Did gradle suddenly stop with an error?
  - Try cleaning the build-folder using `gradlew deploy clean`

# Database-less

- XP doesn't need external database, it's built-in
  - But we have a lib-sql if you need =)
- Uses Elasticsearch for search - super fast
- All data is replicated to disc
  - Stored as files in **\$XP\_HOME/repo**



Wanna bring your data home today? Copy the entire repo-folder and put on your USB-stick, replace repo-folder at home!

# What's next?

- XP 6.15

Wanna know what's coming in the future?

Go to Discuss and find category called “**PAB**”

<https://discuss.enonic.com/c/pab>

Want latest news? <https://discuss.enonic.com/c/news>

Have ideas? <https://discuss.enonic.com/c/features>

Got bugs? <https://discuss.enonic.com/c/bugs>



# Expand your skills further

Attend the **XP Developer 201** course for:

- Basic roles-theory
- Global app settings
- Mixins, Page Contributions
- Error handling, Filters, and other cool functionality!
- Extend your code with more libraries
- Build your own libraries
- Architecture, best practises

And more!

# Enonic Meetup

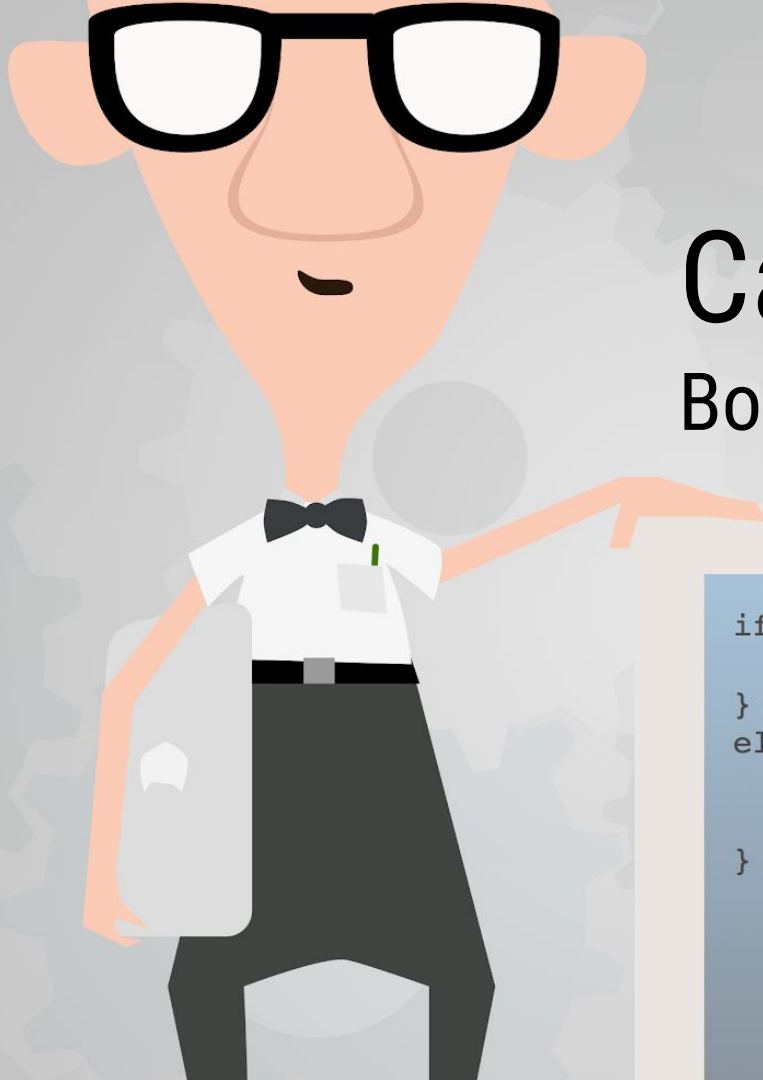
<http://www.meetup.com/enonic-oslo/>





# Feedback

<https://www.surveymonkey.com/r/enonicfeedback>



Can't get enough?  
Bonus assignments!

```
if (idea) {  
    enonicExperiencePlatform();  
}  
else {  
    drinkCoffee();  
    getIda();  
}
```

# Tutorials

Check our video-archive found on our website on the “[Learn](#)” page.

A video describing how to create a part from scratch. It recaps a lot of the things this course covered.

<https://www.youtube.com/watch?v=aeMHLfrc2IA>



# Task - Link between content

Docs: / Developer / Schemas / Input Types / **ContentSelector** /

Docs: / API and Reference / View Functions / **pageUrl** /

A ContentSelector lets you link from one content to another.

- Add a ContentSelector for / site / parts / **banner** /
  - Filter it on Site - `${site}`
  - Allow only Content Type **Country**
- Fetch data in controller
  - Use `portal.pageUrl()` to get URL
- Send to Thymeleaf
  - Create a `<a>` tag saying “Read more” inside a `<p>` tag
  - Clicking this link should open the URL (try `data-th-href`)



# Task - **Fragments - store parts**

The Fragment component is a way for web editor to store specific component configurations as reusable content.

- Open the Site for edit
- Right click the banner you inserted earlier
- Select “Create Fragment”
- Re-use it on any Content
  - Insert component “Fragment”
  - Add to at least two country
  - Update the text on the fragment, see it change



## Task - **getSite** function

Data store on a site (enter edit mode on the site content) can easily be accessed through the **getSite controller function**. It fetches information like site name, path, and description.

- Try Controller Function `libs.portal.getSite();`
  - Output result in server logs
  - Take a look at the output, what does it contain?
- Output site name (`displayName`) to footer!
  - (In the view file for your default page component - `default.html`)



# Task - Extend Content Type (2)

Docs: / Developer Guide / Schemas / **Input Types** /

- Add a new input field to Celebrity CTY
  - A **RadioButton**
  - Label it “Active” with the options yes and no
- Update a content in Admin
- Use content viewer app to see stored data
  - Look at content without this data ...
  - ... and content that you have updated
  - See how it is stored only when updated!
- Output new data in view (celebrities.html)



## Task - **Install another app**

XP Apps stack and their functionality are added cumulatively to sites using them. Try this out by adding more apps to your site content.

- Go to Enonic Market
- Find the **SEO Meta Fields** app
- Install this from the Application admin tool
- Add app to your site
  - (edit site content)
- Play around with it





# Task - Full websites

Want to check out some example website applications built for Enonic XP? See how things are built, and get some more experience using Content Studio!

- Use the Application admin tool
- Install **Superhero Blog** app
  - Play around with the demo site
  - See the video: [youtube.com/watch?v=YBOghIzIHDg](https://youtube.com/watch?v=YBOghIzIHDg)
  - Check our Github: [/enonic/app-superhero-blog](https://enonic/app-superhero-blog)
- Install the **Corporate Theme** app
  - Play around with the demo site
  - Check our Github: [/enonic/app-corporate-theme](https://enonic/app-corporate-theme)