bloomreach

BrXM Essentials Workshop

BLOOMREACH

bloomreach

Official documentation

Bloomreach Experience Manager v14

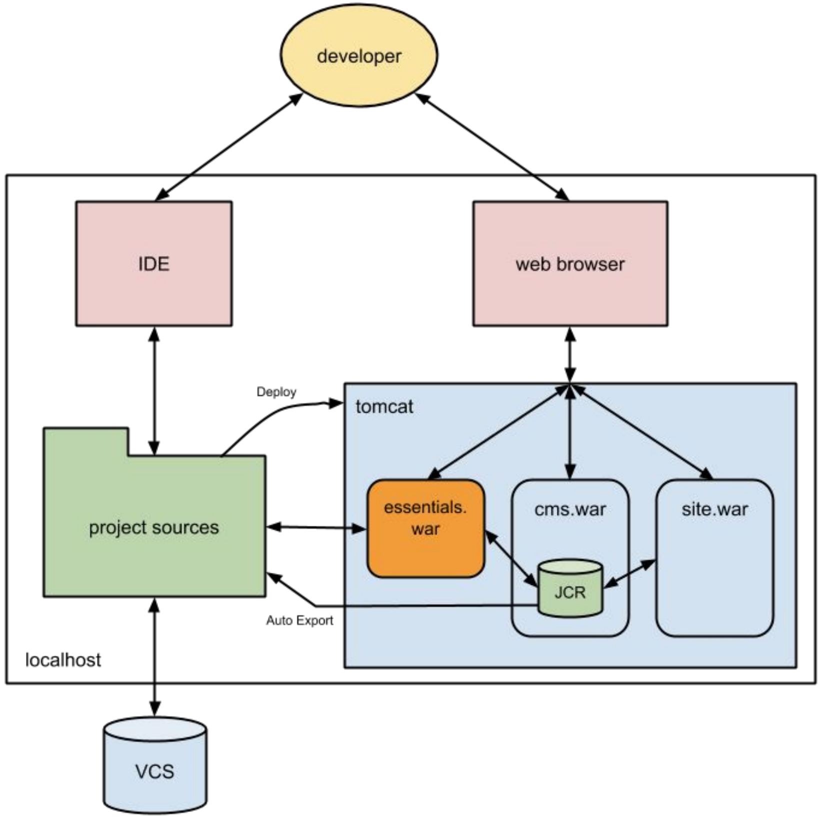
BLOOMREACH

Agenda

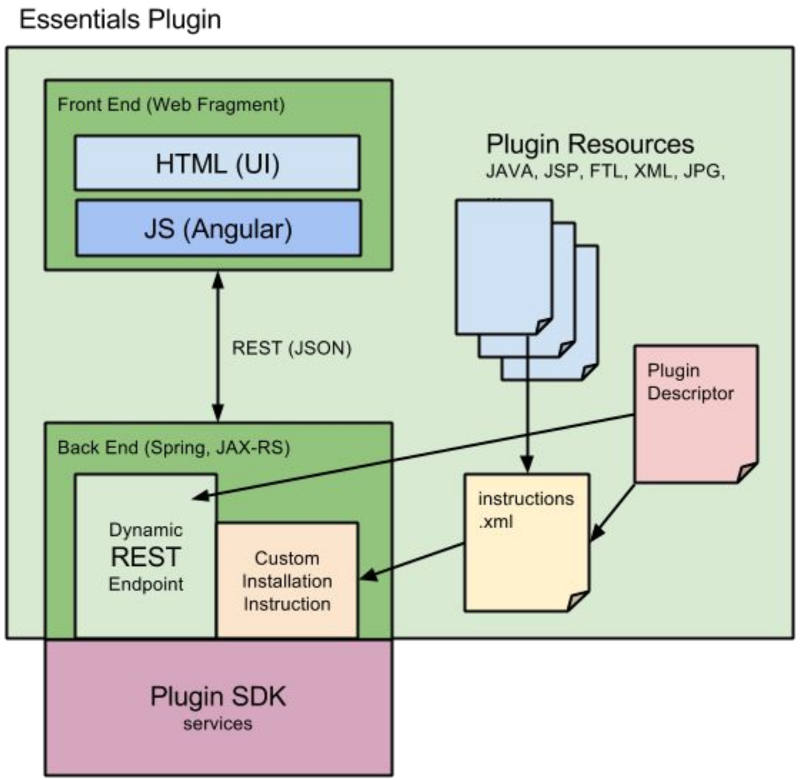
1. Introduction
2. Essentials refresher
3. Get dirty
4. Wrap up

Introduction

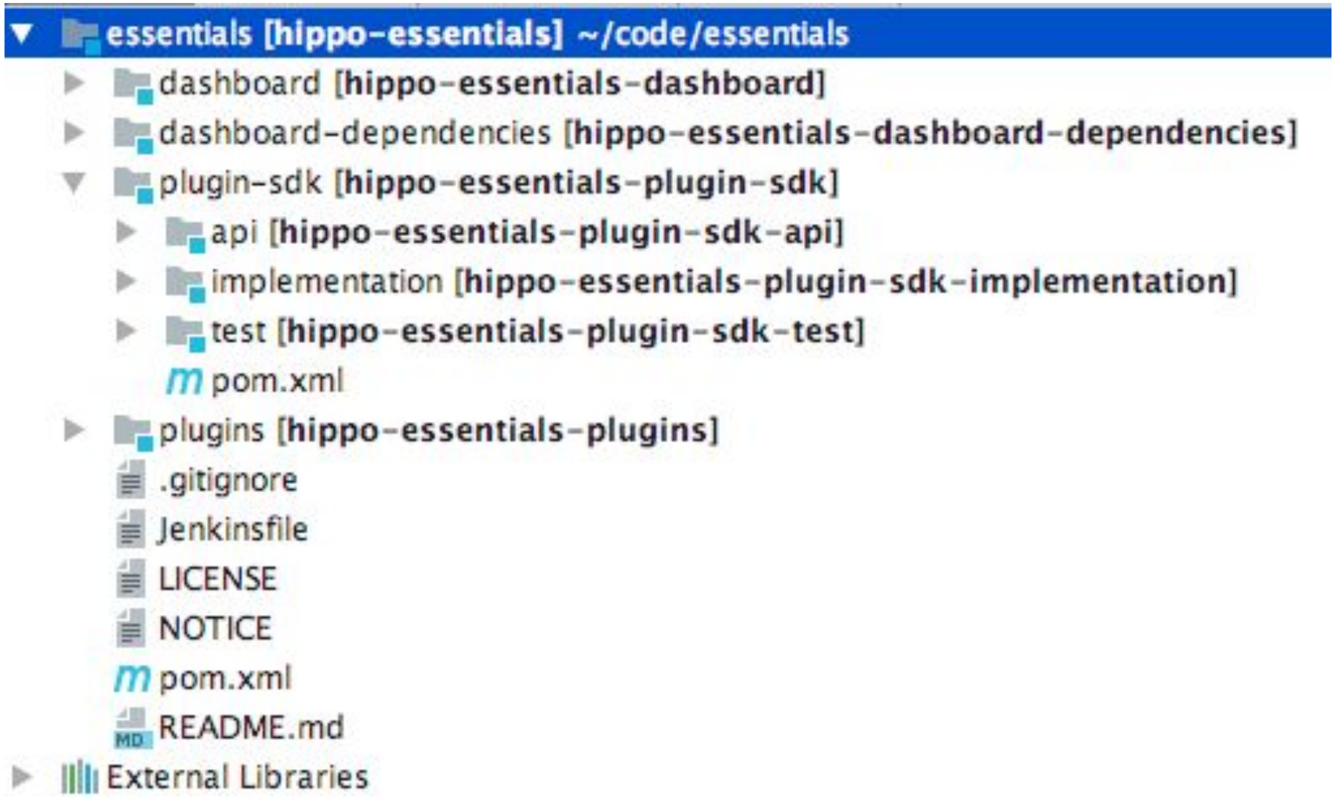
Essentials Architecture



Essentials Plugin Architecture

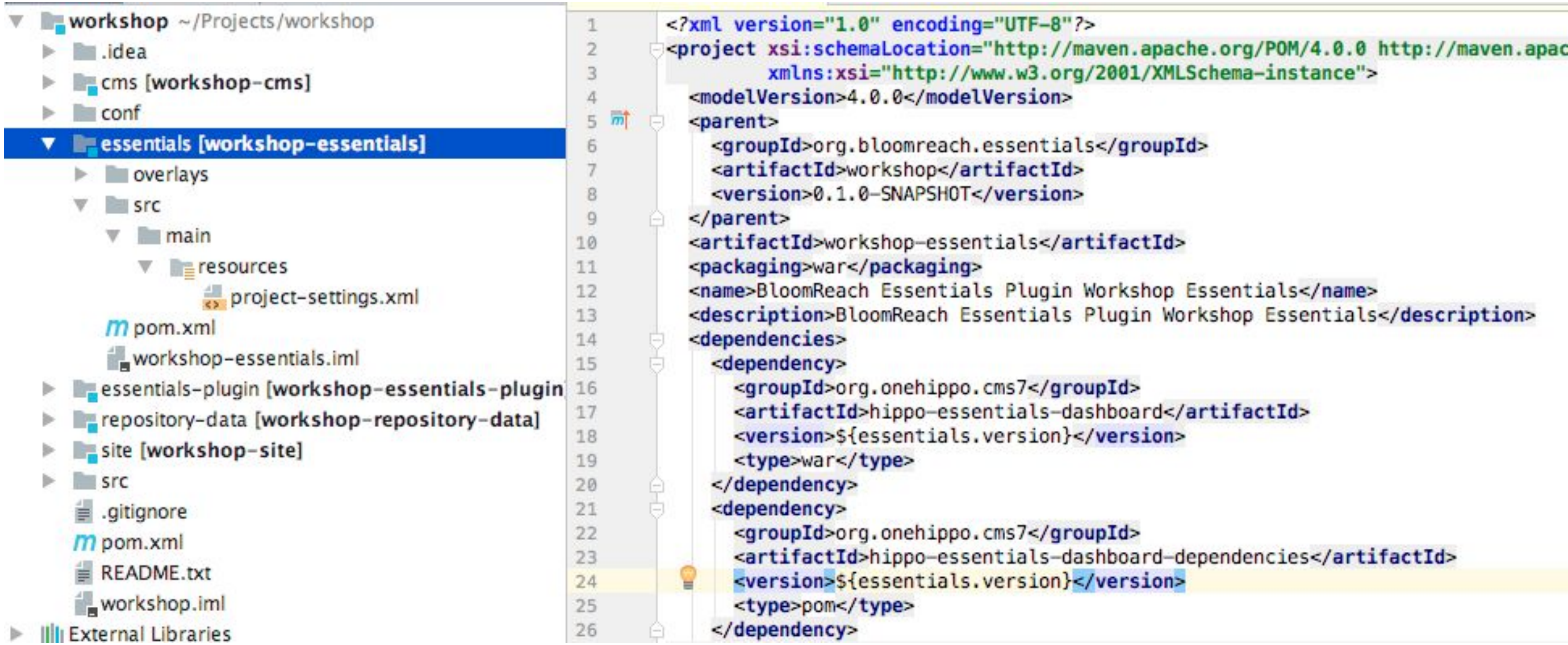


Essentials Project



```
▼ essentials [hippo-essentials] ~/code/essentials
    ▶ dashboard [hippo-essentials-dashboard]
    ▶ dashboard-dependencies [hippo-essentials-dashboard-dependencies]
    ▼ plugin-sdk [hippo-essentials-plugin-sdk]
        ▶ api [hippo-essentials-plugin-sdk-api]
        ▶ implementation [hippo-essentials-plugin-sdk-implementation]
        ▶ test [hippo-essentials-plugin-sdk-test]
          pom.xml
    ▶ plugins [hippo-essentials-plugins]
      .gitignore
      Jenkinsfile
      LICENSE
      NOTICE
      pom.xml
      README.md
▶ External Libraries
```

BrXM Project



```
workshop ~/Projects/workshop
    .idea
    cms [workshop-cms]
    conf
    essentials [workshop-essentials]
        overlays
        src
            main
                resources
                    project-settings.xml
        pom.xml
        workshop-essentials.iml
    essentials-plugin [workshop-essentials-plugin
    repository-data [workshop-repository-data]
    site [workshop-site]
    src
    .gitignore
    pom.xml
    README.txt
    workshop.iml
External Libraries
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apac
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.bloomreach.essentials</groupId>
    <artifactId>workshop</artifactId>
    <version>0.1.0-SNAPSHOT</version>
  </parent>
  <artifactId>workshop-essentials</artifactId>
  <packaging>war</packaging>
  <name>BloomReach Essentials Plugin Workshop Essentials</name>
  <description>BloomReach Essentials Plugin Workshop Essentials</description>
  <dependencies>
    <dependency>
      <groupId>org.onehippo.cms7</groupId>
      <artifactId>hippo-essentials-dashboard</artifactId>
      <version>${essentials.version}</version>
      <type>war</type>
    </dependency>
    <dependency>
      <groupId>org.onehippo.cms7</groupId>
      <artifactId>hippo-essentials-dashboard-dependencies</artifactId>
      <version>${essentials.version}</version>
      <type>pom</type>
    </dependency>
```

Why Create Your Own Plugin?

- Ease installation of your brXM add-on
- Re-use functionality in multiple brXM projects
- Speed up creation of demo / test brXM projects
- "Clean Plugins"

Get Dirty

So, where is that workshop?

Get Exploring

1. if you haven't done so yet, **grab** the workshop project
   https://github.com/bloomreach/xm-essentials-workshop

2. **expand** it, go into the **essentials-plugin** dir, check the **README**

3. **build** and **run** the project

4. **visit** /essentials, **examine** skeleton plugin in Library

5. **install** any available (library) plugin, **inspect** changes to project

6. **stop** and **reset** the project (revert changed files, delete created files)

The Essentials Plugin SDK API

Essentials provides a **Plugin SDK API** JAR.

Essentials plugins define a (Maven) **dependency** on that artifact in order to get access to code and definitions which facilitate the interactions between the plugin and the Essentials web application.

The Plugin SDK API provides an **XSD** for the installation instructions file, the **Instruction** service provider interface (SPI) for building custom instructions, and a **set of injectable services** for plugins to manipulate a brXM project.

The Plugin Descriptor

Each plugin must contain a **plugin descriptor**, represented by the file **plugin-descriptor.json** at the root of the plugin's JAR package.

Essentials scans all packaged JARs, looking for plugin descriptors, and parses them into a list of available plugins.

The plugin descriptor contains all information necessary to represent the plugin in Essentials' Library, such as the name, icon, vendor, description and documentation link.

The plugin descriptor also defines a **unique plugin ID**, and the plugin **type**, i.e. whether the plugin represents a feature or a tool.

Exercise: Fun with the Plugin Descriptor

1. change the **icon** of your plugin ('76x76 px)
2. update the **introduction**
3. add a **description**, providing additional details
4. add a list of **'imageUrls'** to render additional imagery
5. **rebuild** and **restart** your project to see the differences

BLOOMREACH

Feature Installation - Instructions (1/3)

Features are installed from the Essentials Library. The Installation State Machine ensures proper installation, including inter-plugin **dependencies**, requesting **user input** (installation parameters), installation **instructions** and the need to **rebuild** after installation.

Essentials provides a set of **built-in** installation instructions, and you can implement **custom** instructions.

The **packageFile** XML file lists all installation instructions, to be [conditionally] executed during installation. It must have a unique name, by default '[pluginId]_instructions.xml'.

Feature Installation - Placeholders (2/3)

When an instruction pushes code, resources or YAML definitions into a project, you may need to adjust the content or the target location **depending on parameters of the project**, such as its main namespace.

For this, Essentials provides support for interpolating ‘placeholders’, represented by double curly braces, like **{{namespace}}**.

Check out the **PlaceholderService** included in Essentials’ Plugin SDK API for a list of all supported placeholders.

Feature Installation - Other Parameters (3/3)

One feature may depend on other features. This is expressed in the plugin descriptor's **pluginDependencies** list.

Essentials recognises installation parameters **sampleData** and **extraTemplates**, typically controlled through the global project settings. If the settings indicate to "Confirm / override settings on a per-feature basis", Essentials asks for user input, unless the **installWithParameters** field is set to false.

By default, Essentials requires a project rebuild and restart to fully install a feature. Set **rebuildAfterInstallation** to false if your feature doesn't require this.

Exercise: Fun with installation instructions (1)

Checkout branch: **instructions**

find the **instructions XML file** of your plugin, then add instructions to:

1. **add a Maven dependency** to cms/pom.xml
2. **copy an FTL template** into the project's webfiles
3. **import XML** HST configuration for that template into the repository
4. **copy an image** into the site's WEB-INF folder, avoid that Essentials attempts to replace placeholders
5. copy a folder as **'development' sample data** into your project

Note: bootstrapping of this folder **depends** on the administration folder, provided by Essentials' **'skeleton'** plugin.

Exercise: Fun with installation instructions (2)

1. **rebuild and restart** the project
2. **inspect** the “Changes made by this plugin”
3. **install** your plugin
4. **examine** the actual changes to your project
5. **check** if the sample data folder is available in the CMS
6. If you get stuck check the **instructions-solution** branch

Custom Installation Instructions (1/2)

Essentials' OOTB instructions may be too limited for your plugin.

You can implement **custom installation logic** by coding a Java class implementing the **Instruction** SPI.

Use Spring's Dependency Injection mechanism to **work with services** from Essentials' Plugin SDK API, e.g.

```
@javax.inject.Inject private JcrService jcrService;
```

Custom Installation Instructions (2/2)

Your custom Instruction should indicate through its ‘ **change message(s)’** how it affects a project.

In order to execute your custom Instruction during installation, add an **<execute>** instruction to the plugin’s instructions file:

```
<execute class="org.example.ExampleInstruction"/>
```

When executed, Essentials passes ‘installation parameters’ to the #execute method.

BLOOMREACH

Exercise: Crazy with custom instructions

1. **implement** a custom instruction, making use of the injectable services provided through Essentials' Plugin SDK API
2. keep the **change messages** up to date
3. ensure the instruction is **executed** during installation
4. **rebuild** and **restart**, check the change messages
5. **install** your plugin, check the effect of your instruction

Wrap up

What more is there?

BLOOMREACH

Tool Plugins

Tools are not associated with an installation state. They live inside Essentials and typically combine a **REST-y end-point** on the back- end with an extension of the **AngularJS front-end** in order to provide functionality to manage some aspect of a brXM project.

Check the public Essentials Plugin documentation and existing tools in the Essentials project for more information and inspiration.

You can also add **tool-like capabilities to a feature,** and make them available through Essentials once the feature is installed. See the ‘blog’ plugin as an example.

Recommended Reading

Public documentation on creating Essentials plugins

https://documentation.bloomreach.com/14/library/essentials-plugins/overview.html

JavaDoc of Essentials Plugin SDK API

Real-world examples of plugins, included in the Essentials project

https://github.com/bloomreach/brxm/tree/brxm-14.5.0-1/essentials/plugins

Additional Resources

Essentials Plugin Archetype

https://documentation.bloomreach.com/14/library/essentials-plugins/archetype.html

‘Essentializer’: Essentials plugin that allows you to extract parts of a project and package them as an Essentials plugin

https://github.com/bloomreach-forge/essentializer

Essentials Intellij Plugin by Marijan Milicevic

https://github.com/machak/essentials-intellij-plugin

bloomreach

Thank you

BLOOMREACH