Development Guidelines

This topic provides guidelines and conventions for plugin development. The intention of these guidelines is to improve the overall quality of the plugin and to ensure that they are well designed, well-documented, consistent, compatible, Internationalization ready and migration ready. This topic can be used to validate plugins for plugin certification. A certified plugin must comply with many of the guidelines and conventions described. For each guideline or convention it is specified whether it is required eor recommended for a particular certification level

In This Topic

Terminology

This topic uses the following acronyms. The meaning of each one is described below.

Term	Description
POJO	A Plain Old Java Object. This refers to a basic Java class not implementing or extending any special class(es).
MVC	Model-View-Controller. A design pattern used to separate the business model, business logic and UI logic.
Spring MVC	The web application framework used by XperienCentral to render the Editor.
DAO	Data Access Object. An object that encapsulates all access to a particular data source. It is used to prevent a business object from having dependencies with a particular persistence implementation.
DTAP	Development Test Acceptance and Production.
JCR	Java Content Repository.
RBAC	Role Based Access Control.
FBO	A `FormBackingObject`, which is an object that represents the data contained by an HTML form.
Plugin audit	The process of verifying a plugin against the plugin development guidelines and granting a particular certification level.
Plugin certification level	A level that indicates the extent to which the plugin conforms to the plugin development guidelines. The higher the certification level, the more the plugin conforms to the defined guidelines.
Plugin certification range	The range from the first certified plugin version to the first major version of the plugin for which the current plugin certification level will not be valid anymore and a new plugin audit will be required.
Guideline scope	The software component(s) or component type(s) to which the guideline applies.
Pre-audit	The act of running the plugin guideline audit tool to verify your plugin against a basic subset of guidelines.

Back to top

The Certification Process

The goal of certifying plugins is to improve the overall quality of all plugins, which makes it convenient for the customer, the developer of the plugin and any developer not familiar with the plugin who has to perform maintenance on it.

A summary of the plugin certification:

- Guarantees that the plugin will not conflict with other certified plugins.
- Guarantees a certain quality level in the sense of stability, compatibility, upgradability, security, performance and many other themes.
- Increases the ease of maintenance because the plugin's source files are structured in a consistent way.

Certification Levels

For each guideline described in this document, a plugin certification level is defined. Plugin certification comes in three levels:

- **Level 1 - Project specific**. The plugin is fully functional for a specific XperienCentral installation but may lack documentation, localization and may require specific hardware or software. The plugin may not be suitable for running on other XperienCentral installations.
- **Level 2 - Reusable**. The plugin is fully functional, provides proper documentation but may require specific hardware or software as indicated in the packaged `readme.txt`. It may lack support for localization and documentation in US English. The plugin is suitable to run on any XperienCentral installation of the version the certificate is valid for.
- **Level 3 - Product ready**. The plugin is fully functional, provides proper documentation and supports at least the hardware and software required by the XperienCentral platform itself. It supports localization and all documentation is available in at least US English. The plugin is of such high quality that it could be incorporated into the XperienCentral platform as is.

The table in the Guidelines Summary specifies for each guideline the plugin certification level for which the guideline is mandatory. The levels are cumulative. For certification of a particular level it is required that the plugin also conforms to all guidelines required for lower certification levels.

Guideline Versions

These plugin development guidelines are subject to change. Occasionally, new guidelines are added, old guidelines are removed or existing guidelines are updated for various reasons. Updates to these guidelines are always effective as of the next available version of XperienCentral. The guidelines are fixed for XperienCentral certification. A certificate will always mention the XperienCentral version for which the certificate has been granted, indicating the exact version of the guidelines against which it was verified. Note that the guidelines are also occasionally updated with textual improvements.

Plugin Version Updates

A plugin audit is performed on a particular version of a plugin, for example the 1.1.0 version. This version is referred to as the "plugin audit version". After the plugin audit, new versions of the plugin may become available.In a micro version update (for example from version 1.1.0 to 1.1.1), it is unlikely that a code change could lead to a guideline violation. If it did, that it would be a guideline violation since micro version updates should only contain micro software changes. The same partially applies to minor version updates. It would still be rather unlikely that a minor version update would violate a guideline. A major version update, however, could contain major refactoring or major functional enhancements which could mean that the plugin has to be recertified. For that reason GX Software stipulates that a plugin certification level is only valid until the next major version of the plugin . For example, if version 1.1.4 of a plugin has been certified at level 2, then each 1.1.x version, where x >= 4, as well as each 1.y.z version, where y >= 2, will also be certified at level 2. We refer to the range between the first certified version and the version from which the certificate will expire the "plugin certification range.

Preaudit

A plugin audit tool is available that automates and supports plugin audits. Running this tool to verify a plugin is called a pre-audit. Running a pre-audit is required before a plugin audit is requested. If the pre-audit results in one or more guideline violations which the developer of the plugin believes to be invalid, the developer should indicate this in the audit request.

Plugin audit report

A plugin audit report contains four important results:

- The version of the plugin that has been audited.
- The assigned certification level (if applicable) for the plugin (the plugin certification level).
- The XperienCentral version for which the certificate is valid.
- The range of plugin versions for which the certificate is valid (the plugin certification range).

For example:

- Version of the audited plugin : 1.1.4
- Assigned plugin certification level: 2
- Certified on XperienCentral version: 10.8
- Plugin certification range: 1.1.4 to 2.0 (but not including 2.0)

Back to top

Guidelines Overview

This document describes the guidelines and conventions for plugin development. These are divided into the following themes:

1. **GUI guidelines for the Editor** - Guidelines to design consistent and user-friendly User Interfaces for the XperienCentral Editor. These guidelines do not apply to the GUI of the website environment.
2. **Architecture** - The architectural design patterns that the plugin should comply with. Design patterns contribute to software quality in many ways; future compatibility, plug ability, extendibility and synoptic code.
3. **Migration** - These guidelines deal with software upgrades. Conforming to these guidelines will guarantee a smooth migration from older to newer versions of the plugin.
4. **Internationalization** - All plugins must be ready for internationalization. This must be taken into account when designing the software. The XperienCentral platform offers support for internationalization.
5. **Documentation** - Since plugins are small software components that may be reused in other projects than they were originally intended for, it is important that proper user documentation is available.
6. **Distribution** - These guidelines define how a plugin should be packaged and distributed. They cover the contents of a plugin release and how they should be packaged.
7. **Quality** - These guidelines cover the stability and compatibility of the plugin with web browsers, web servers, application servers and databases.
8. **DTAP support** - The plugin must be designed to support the DTAP model.
9. **Security** - The security guidelines define rules that must be followed in order to ensure that the plugin is a secure web application. This prevents common security errors like the enabling of SQL injection and cross-site scripting.
10. **API** - A plugin may expose an API in the form of implementation classes, interfaces and constants. These guidelines describe how such an API should be exposed by a plugin.
11. **Coding conventions** - Conforming to coding conventions contributes to readability of the software code. This makes it easy to read, easy to learn and will prevent bugs.
12. **Javadoc** - A proper Javadoc is very important for software developers coding against the API exposed by the plugin. If a plugin is extendable, it is very important that its API is well documented.
13. **Naming conventions -** For readability and maintainability of the software code, it is always a good idea to define the conventions used to name particular methods and classes.

Guidelines versus Conventions

Conventions are a specific type of guidelines: they describe proper naming or the way software code or documentation should be written. A convention is very specific and easy to validate against. A guideline is a more abstract definition and harder to validate.

Guideline Overview

This part contains an overview of all guidelines and conventions defined in this topic. The tables below define which guidelines and conventions are required for which plugin certification levels. The table also indicates a scope. The scope indicates to which part of the plugin the guideline applies. This may be only one particular component, a set of components or the plugin as a whole.

The following abbreviations are used in the table summary:

Scope	Description
A	The guideline applies to all component types or to the plugin or WCA as a whole.
P	The guideline applies to panel components.
E	The guideline applies to element components.
M	The guideline applies to media item components.
C	The guideline applies to service components.
D	The guideline applies to page metadata components.
L	The guideline applies to servlet components.
R	The guideline applies to presentation components.
F	The guideline applies to form components.
V	The guideline applies to profile provider components.

Back to top

Guidelines Summary

The table below summarizes all the guidelines. Because the guidelines have evolved over time, some IDs have been removed, therefore the sequential order skips numbers in some places. If the column "Certification Level" is blank, this means that the guideline is not required for any certification level and is only a recommendation. Most of the guidelines are discussed in greater length following the table below.

Guideline ID	Description	Certification Level	Scope
G001	The HTML rendering the UI conforms to the basic layout using a table of class `widget grid`.	1	PEMDV
G002	The HTML rendering the UI uses only the defined CSS classes.	3	PEMDV
G003	The UI conforms to the basic guidelines.	2	PEMDV
G004	The UI uses the widgets supported by the XperienCentral platform.	2	PEMDV
G005	The JSPs that render the HTML use the JSTL tags and JSP tags offered by the XperienCentral platform as often as possible.	2	PEMDV
G006	The UI conforms to the defined UI interaction patterns.	2	PEMDV
G007	The business object is implemented as a POJO and does not contain any reference to a controller, form backing object or DAO.	1	PEMDV
G008	The business object does not contain properties or logic whose sole purpose is the view.	1	PDV
G009	The `FormBackingObject` is implemented as a POJO and does not have any reference to a controller, DAO or business object unless the business object is a POJO itself.	1	PEMDV
G010	The `FormBackingObject` reflects the properties and logic to be rendered by the view and not the properties and logic of the business object from which it retrieves values.	1	PEMDV
G011	The `FormBackingObject` and Business Object is not one and the same object.	1	PEMDV
G012	The `copyProperties()` method of `org.springframework.beans.BeanUtils` is used to transfer values from the `FormBackingObject` to the business object (or vice versa).	3	PMDV
G013	The controller is a separate class and implements all controller logic.	1	PEMD
G014	Persistence logic is not contained by the business object but implemented in a separate DAO.	1	PEMDV
G015	JSPs do not contain SQL statements or other persistence implementation-specific logic unless they are contained by a separate JSP tag.	1	PEMDRV
G016	The plugin contains all resources not provided by dependencies or other presentation plugins and is capable of being deployed and functioning properly on any XperienCentral installation, as long as all its defined dependencies are available.	1	A
G017	The version number of a plugin conforms to the syntax "major.minor.micro".	1	A
G019	The version numbers of the plugin are independent of the XperienCentral release they were developed for.	1	A
G020	If the data model of the plugin has been changed in a newer version of the plugin , the plugin must properly handle data model updates.	2	PEMDV
G021	The API that XperienCentral provides is used to access the data model XperienCentral exposes.	1	A
G022	XperienCentral features and API functions are used where possible instead of implementing custom functions.	1	A
G023	When the plugin uses a service, a dependency with that service is defined in the component definition in order to retrieve a reference to the service.	1	A
G024	Text, images and other GUI components are suitable for translation.	3	PEMCDV
G025	Multilingual content that can be created using the plugin is translatable.		PEMCDLFV
G026	Language labels are defined in language resource files conforming to JavaI18N and the filename meets the syntax `messages_<language>_<country>_<variant>.properties`.	1	PEMCDLFV
G027	Label IDs in the language resource files use only lower case letters.	1	PEMCDLFV
G028	Language labels in language resource files are grouped per component and prefixed with at least the ID of the component.		PEMCDLFV
G034	Documentation is available in at least the US English language.	3	PEMDFV
G035	A plugin is distributed as a single ZIP file known as a WCA (WebManager Component Archive) which can also contain other related plugins.	1	A

G036	The contents of the WCA conforms to the defined directory structure.	2	A
G037	The WCA contains a `readme.txt` and a `changelog.txt`.	1	A
G038	The layout of the `readme.txt` and `changelog.txt` follow the defined templates. Click the links to the left to download the templates.	1	A
G039	The plugin JAR file containing the software follows the defined directory structure.	1	A
G042	The HTML generated by the JSPs for rendering the Editor is XHTML 1.0 transitional compliant.	3	PEMDV
G045	All content generated by the plugin is stored in the JCR or in an external database if there is an obvious need for it.	1	PEMCDLFV
G046	The configuration management service is used to store configuration options. No hard-coded web IDs, paths or URLs are used in the software.	1	PEMCDLFV
G047	The preferences service is used to store all preferences.	1	PEMCDLFV
G048	The `output-html-encoded-quotes` XSLT template is used to properly escape HTML strings in the output when using XSLT.	1	PEMDRV
G049	SQL-prepared statements are used for all SQL queries.	1	A
G050	At least one RBAC category is defined for each component which has a GUI representation which contains one or more RBAC permissions for the component.	2	PEDV
G051	A permission category defines and implements at least the RBAC permissions as defined.	2	PEV
G052	The implementation of RBAC permission handling is mainly programmed in a controller or service, not in the business object itself. The check is only performed in the business object itself if the permission defines authorization to retrieve or update that particular property only.	2	PEMDV
G053	The definition of all RBAC permissions is positive. Permissions are defined in a way that assigning the permission to a role grants the role particular rights and it never denies rights.	1	PEMDV
G055	**Coding conventions**: the coding conventions that Sun publishes as the standard for the Java programming language are followed.		A
G056	**Coding conventions**: Java language features are used where applicable (Java version 5 and higher).	3	A
G057	**Coding conventions**: `Java Util Logging` is used where applicable and proper log levels are used (Java version 5 and higher).	1	A
G058	**Coding conventions**: Java concurrency utilities are used where applicable (Java version 5 and higher).	3	A
G059	**Coding conventions**: old collections like `Hashtable`, `Vector` or `Dictionary` are not used if it can be avoided.	1	A
G060	**Coding conventions**: the `@override` annotation is used if a method is overridden from a super class.		A
G061	**Coding conventions**: The basic variant of the Hungarian notation is used.		A
G063	**Coding conventions**: The source code does not contain blank spaces before and after method arguments, however, spaces are used after Java keywords and commas.		A
G064	**Coding conventions**: Declare variables and methods in the following order: Class (static) variables, instance variables, constructors, methods.		A
G067	The FIXME comment is used to indicate that code snippets are incorrect during development and do not appear in a released plugin.		A
G068	The TODO comment is used to indicate that code snippets are incomplete during development and do not appear in a released plugin.		A
G069	**Coding conventions**: Spaces are used instead of tabs for trailing white space in the source code.		A
G070	**Coding conventions**: The size of one line in the software code is limited to 120 characters.		A
G071	**Coding conventions**: Start brackets are added on the same line as the statement to which they apply but ending brackets appear on a new line.		A
G072	**Coding conventions**: Brackets are used in all cases, even for single line statements.		A
G073	**Coding conventions**: The source code conforms to Javadoc conventions defined by Sun.	2	A
G074	**Coding conventions**: Public and protected classes, interfaces, variables and methods are tagged with Javadoc.	2	A
G075	The Javadoc does not contain references to documents that might not be accessible by external developers.	3	A
G076	General plugin classes conform to the defined naming conventions.	1	A
G077	Element component classes conform to the defined naming conventions.	1	E
G078	Element component classes conform to the defined hierarchy.	1	E
G079	Media Item component classes conform to the defined naming conventions.	1	M
G080	Media Item component classes conform to the defined hierarchy.	1	M
G081	Panel component classes conform to the defined naming conventions.	1	P

G082	Panel component classes conform to the defined hierarchy.	1	P
G083	The domain used for naming is returned by `ComponentBundleDefinition.getDomain()`.	1	A
G084	The plugin ID is returned by `ComponentBundleDefinition.getWCBId()`.	1	A
G087	The prefix equals the plugin ID.	1	A
G089	Package names used in the source code of the plugin conform to the package naming guidelines as defined by the Sun coding conventions.	1	A
G090	Top level Java package names follow the syntax `<domain>.<plugin ID>`.	1	A
G095	The artifact ID in the `pom.xml` equals the plugin ID.	1	A
G096	The group ID in the `pom.xml` equals the domain.	1	A
G097	The ID of the component bundle definition defined in the activator of the plugin conforms to the defined syntax.	1	A
G098	The ID of each component definition contained by the plugin must be prefixed with the component bundle definition ID followed by an ID that is unique within the plugin, matches the component name and consists of lower case alphanumeric characters in the range [a-z]. This does not include the media item component definition.	1	A
G099	All IDs and properties defined use only lower case letters.	1	A
G101	The name of the RBAC category conforms to the syntax `<top level domain>.<plugin ID>.<Component ID>`.	1	PEMCDLFV
G102	The technical name of each RBAC permission is prefixed with the technical name of the RBAC category, followed by a dot.	1	PEMCDLFV
G103	Technical names of categories and permissions are in lower case, do not contain spaces and separate words are separated by a dot.	1	PEMCDLFV
G104	For CRUD actions, the defined naming conventions are used.	2	PEMCDLFV
G105	In user documentation, the defined naming conventions are used for XperienCentral assets.	3	A
G106	All names of labels, Java classes, methods, properties, etc. are in US English unless they represent translatable labels presented to the end user.	1	A
G107	The CMU-SEI (https://en.wikipedia.org/wiki/Cyclomatic_complexity) does not exceed 15.	3	PEMCDLFV
G108	Online help covering the visible components contained by the plugin is available.	2	PEMDV
G109	An API is exposed by a domain object interface when it consists of getters and setters for properties of that domain object.	1	PEMCDLFV
G110	Implementations of a domain object interface is postfixed with "Impl".	2	PEMCDLFV
G111	An API is exposed as a service when the API creates or deletes domain objects or operates on multiple domain objects.	2	PEMCDLFV
G112	An API service managing entities is preferably postfixed with "ManagementService".	3	CV
G113	A method handles an exception internally if it is recoverable.	2	A
G114	A method throws a checked exception if it is unrecoverable and occurs in an area outside the immediate control of the program.		A
G116	Identifiers used by a plugin are defined in one class as public static final fields of that class.	3	A
G117	A plugin exposes its identifiers by exposing the class as defined by guideline G116.	3	A
G118	Any class or interface that is exposed by a plugin as API should be contained by a package called "api" directly under the plugin's root package, or by a sub package of this package.	2	A
G119	If content is stored in an internal relational database, the SQL scripts to create those database tables should be contained by the plugin in a `/sqlscripts` directory.	2	A
G120	Each plugin should come with a unit test or test bundle that has a code coverage of at least 10%.	3	PEMCDLFV
G121	The plugin ID only contains alphanumeric characters in the range [a-z].	1	A
G122	The domain only contains alphanumeric characters in the range [a-z] separated by dots.	1	A
G123	In the JSPs, properly escaped strings in the HTML are output using standard JSTL functions.	1	PEMDRV
G124	The `contenttype` of a media item component which is defined by the `@ContentType` annotation must equal the plugin ID or be prefixed by the plugin ID and may only contain alphanumeric characters in the range [a-z].	1	M
G125	Use the Entity Manager as the DAO implementation to store entities in the JCR when possible.		PEMCDLF
G126	Creating and removing resources on which a plugin depends should be done automatically upon installation and purging the plugin if possible.	1	A

G127	Framework labels are grouped separately in the language resource files.		PEMCDLFV
G128	Language files are at least available in US English.	2	PEMCDLFV
G129	A method should not catch an unchecked exception if it is unrecoverable and occurs in an area inside the immediate control of the program.	1	A
G131	A new checked exception type is used as the wrapper for multiple checked exceptions if the amount of thrown checked exception exceeds five.	2	A
G132	The Javadoc clearly explains how the class or interface should be used, preferably by providing code examples.	2	A
G133	For each package contained by the plugin, a `package.html` is provided that clarifies the purpose and contents of the package.		A
G134	HTML generated by the JSPs for rendering the website environment should be XHTML 1.0 transitional compliant.	3	R
G135	If a method of a class implements a method of an interface, refer to the Javadoc of the interface using the `{@inheritDoc}` annotation.	2	A
G136	The `bundleSymbolicName` in the `pom.xml` matches the syntax `<domain>.<plugin ID>`.	1	A
G137	The presentation plugin is self-sufficient and does not depend on any resource provided by another plugin unless it is another presentation plugin .	1	R
G138	JSPs copied from the original XperienCentral platform presentation are copied to the specified directory.	1	R
G139	The value of the name attribute in the descriptor of a JSP is prefixed by the plugin ID.	1	R
G140	Static files used by the presentation plugin are located in a subdirectory that equals the plugin ID.	1	R
G141	The software does not contain code snippets that are commented out, not used or that duplicates of other code snippets in the same plugin .	2	A
G142	Media item JSPs that display the content of a media item contain a check to see whether the media item has not already been rendered before within the same request.	1	MR
G143	Use the `Link.linkAttributes()` method to build links in a JSP.	1	PEMDRV
G144	Implement caching properly by providing SSIs.	1	PEMDRV
G145	A plugin contains only those components that logically belong to each other.	1	R
G147	The plugin does not contain the empty online help that was generated by the archetype.	1	A
G148	Do not use public or protected instance variables. Use private instance variables with getters and setters instead.	1	A
G149	The plugin ID and domain are approved by GX and the plugin (of this version) is registered.	1	A
G151	The name of a scheduled job is prefixed by the plugin ID.	1	C
G152	The ID of each configuration set defined by the plugin is prefixed by or equals the plugin ID.	1	A
G153	The plugin contains an example frontend presentation if it requires such a presentation to work properly.	2	EMDFV
G154	The plugin supports the software and hardware as described in Hardware and Software Requirements.	2	A

Back to top

GUI Guidelines for the Editor

This part provides in an-depth overview of all GUI guidelines. From these GUI guidelines we define the following guidelines:

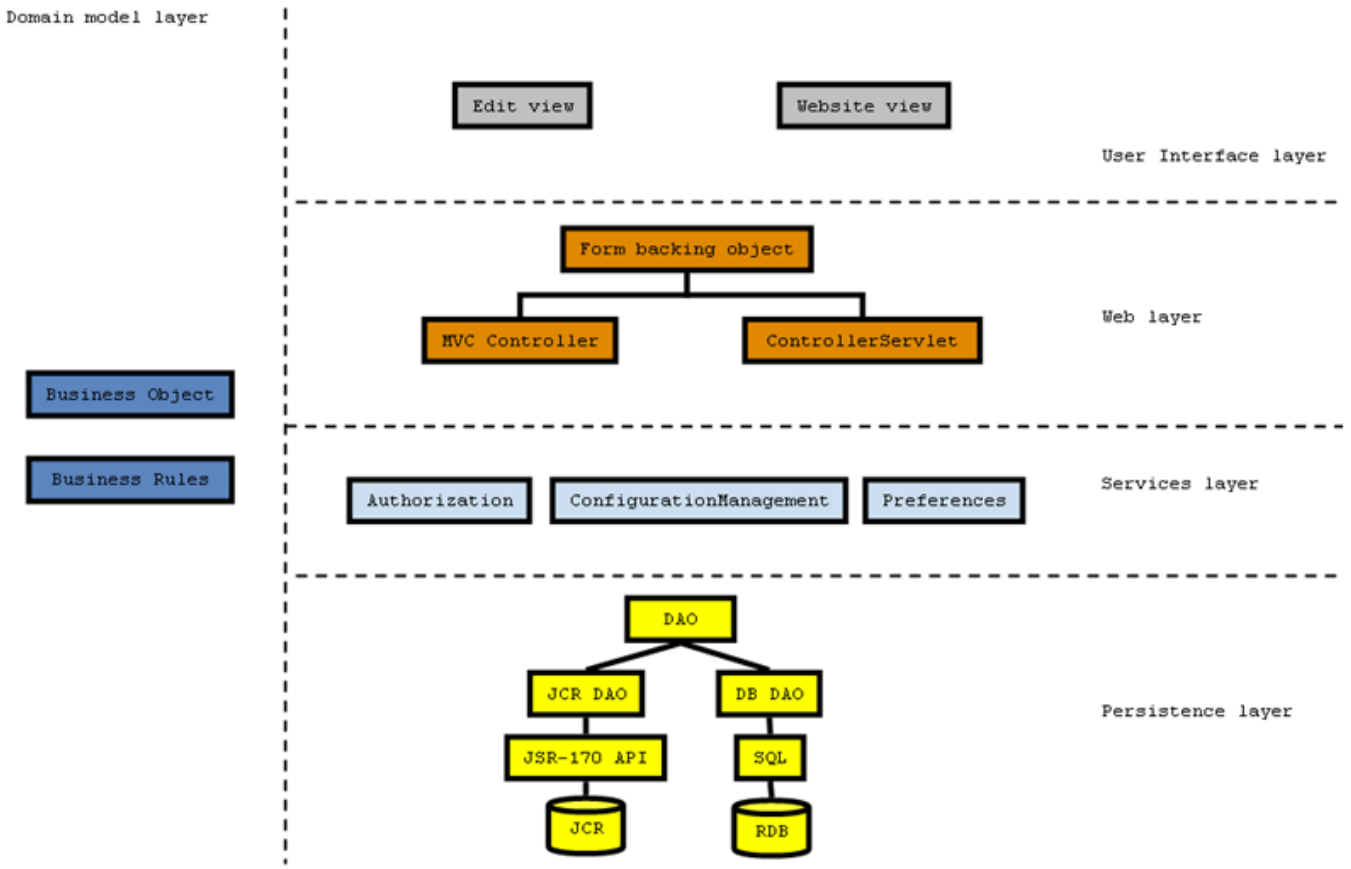
Architectural Guidelines

A design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished technical design that can be transformed directly into software code, instead it is a description or template for how to solve a problem that can be used in many different situations. For plugin development we will provide a few design patterns that will contribute to a proper plugin architecture.

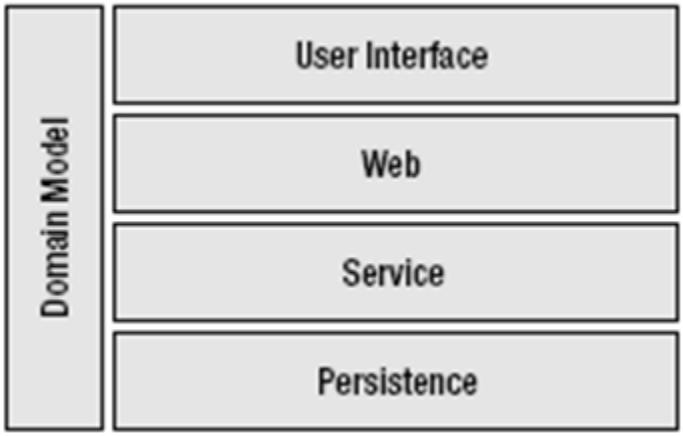
Abstraction Layers

Separating the application in several abstraction layers is a good design principle which prevents tight coupling of the software. Loose coupling by introducing abstraction layers makes the software very flexible, pluggable and extendable, which is very important for a component-based application like XperienCentral.

Developed plugins should apply this design pattern to be flexible, pluggable and extendable themselves. The figure below shows the abstraction layers in XperienCentral:



In the approach depicted above, there is a very clean separation of responsibilities by the several application layers as also depicted below:



The layers also conform to the MVC pattern in which a controller determines the model and returns the view to render the model.

User Interface Layer

The user interface layer defines different views for displaying the model that is provided by its controller. Such a view may be an edit view for editing an element in the Editor or a website view for rendering the element on the website. The views are created and maintained by the controller. In Spring MVC, such a view can be a JSTL view (JSP), redirect view, download view, or any other view since views are agnostic in Spring MVC.

Web Layer

The web layer is responsible for all communication between the model and the view. It retrieves the HTTP requests, generates the view and returns the result. The controller uses a data transfer object, called the `FormBackingObject`, to hold the data that must be rendered by the view or to hold data submitted from an HTML form by an end user. The controller handles the transfer of property values between the `FormBackingObject` and the business objects it represents. The `FormBackingObject` is closely related to one or more business objects in the domain model layer but is not necessarily the same. For the Editor, the controller is a Spring MVC controller - for the Website this is the `ControllerServlet`.

The controller is the spider in the web; it is responsible for handling all requests and returning the proper results. The Web layer typically has dependencies with all other layers.

Services Layer

The services layer exposes several services to software components like authorization, configuration, preferences, session management and transaction management. Services provide business logic that involves multiple business objects. Business objects themselves usually only contain logic to update or retrieve its own properties.

Persistence Layer

The persistence layer is responsible for persisting the business objects invoked by the controller. It is good practice to define a DAO in order to prevent the business object from containing dependencies with a particular persistence implementation like the JCR or JDBC calls.

Domain Model Layer

The domain model layer contains all business logic, business objects and business rules. The domain model objects are implemented as POJOs and do not contain references to any of the other layers. The several other layers however may refer to objects in the domain model layer. The business objects usually contain only business logic to update and retrieve their own properties. When business logic involves multiple business objects, it is recommended that you use a service instead.

Abstraction Layers in a Plugin

To identify the several abstraction layers clearly in a plugin, the following guidelines should be followed:

- The business object is implemented as a POJO and does not contain any reference to a controller, form backing object or DAO. However, It may contain service references or references to other business objects [**G007**].
- The business object does not contain properties or logic whose only purpose is the view [**G008**]. An example violating this guideline would be defining a property like "selectedBook" that is not persisted but only temporarily stores the user selection in the view.
- The `FormBackingObject` is implemented as a POJO and does not have any reference to a controller, DAO or business object, unless the business object is a POJO itself. It may contain references to other `FormBackingObjects`. [**G009**].
- The `FormBackingObject` reflects the properties and logic to be rendered by the view and not the properties and logic of the business object from which it retrieves values. [**G010**].
- The `FormBackingObject` and the business object are not one and the same object [**G011**].
- Use the `copyProperties()` method of `org.springframework.beans.BeanUtils` to transfer values from the `FormBackingObject` to the business object and vice versa [**G012**].
- The controller is a separate class (not mixed with business object and/or `FormBackingObject`) and it implements all controller logic such as creating, updating and deleting `FormBackingObjects`, business objects and the views [**G013**].
- Persistence logic is not contained by the business object but is instead implemented in a separate DAO [**G014**]. It is recommended that you use the Entity Manager (a DAO implementation that currently supports only the JCR) available from XperienCentral.10.3 and later [**G125**]. Note that the current XperienCentral API does not yet support this for element business objects. For all other cases, this is the preferred way.
- JSPs do not contain SQL statements or other persistence implementation specific logic, except for those with scope `Query` or `DatabaseEntity`. If you have to use SQL in a JSP, separate it by including the SQL statement in a separate JSP tag [**G015**].

Plugin Completeness

A plugin is intended to be an individual component that contains all resources (like images, Java code and XML files) that contribute to the functionality of that plugin. If the proper working of a plugin requires, for example, a change to the `web.xml` of the web application, this breaks the one-click deployment concept of a plugin. A plugin must be capable of being deployed on any XperienCentral installation and work properly as long as all its dependencies are available [**G016**]. Note that the plugin may depend on presentation files contained by another plugin for level 1 certification, but for level 2 certification, it is required you add to add at least a working example presentation to the plugin [**G153**].

Dependencies may be service dependencies, dependencies with other plugins but also dependencies on other resources that require user interaction to make those available like database tables or changes in the `web.xml`. The changes necessary should be performed by the plugin itself automatically upon installation of the plugin as much as possible and those resources should be removed when the plugin is purged [**G126**]. For example; a configuration set should automatically be created by the plugin upon installation and removed automatically when the plugin is purged. Another example is the need for a manual execution of an SQL script to install required database tables. A bad example is the need for modification of the `web.xml` which would require a server restart. All dependencies other then service dependencies or plugin dependencies should be documented.

Code Complexity

Writing readable code contributes to the maintainability of the software code and makes the software less error-prone since it easy to understand what the code does and/or is supposed to do. For determining an objective measurement of software complexity the so called Cyclomatic Complexity Number can be determined. A definition of the CMU-SEI interpretation of this number can be found at https://en.wikipedia.org/wiki/Cyclomatic_complexity. It is recommended that the CMU-SEI of the software does not exceed 15 [**G107**].

Back to top

Migration Guidelines

Versioning

Each plugin has a version number that consists of three digits with the following meaning [**G017**]:

- **Major version -** A major software update of the plugin. Such an update may change exposed APIs which may cause incompatibility with plugins that depend on it.
- **Minor version** - A backwards compatible plugin update. The exposed API will not be changed and pluginss that depend on this plugin will thus be compatible with this version update. The API might be extended with additional functions.
- **Micro version** - Update without any change on interface of published services.

The version numbers should be independent of the XperienCentral release they were developed on [**G019**]. Therefore, the first official release should be 1.0.0.

Content Migration

If the datamodel of the plugin has been changed in a newer version of the plugin, the plugin must properly handle datamodel updates [**G020**]. There are two options for doing so:

- The plugin supports automatic data model conversion.
- The plugin states specifically in the `readme.txt` that the data model has been changed and that the new version of the plugin is not compatible with the previous version.

Data Model Access

To ensure that a plugin is backwards compatible with updates of the XperienCentral platform, the API that XperienCentral provides to access the data model XperienCentral exposes must be used [**G021**]. Bypassing the API by writing custom SQL and XPATH queries is not recommended since it may break backwards compatibility.

Custom XPATH and SQL queries for accessing the data model of objects contained by the plugin itself are appropriate.

XperienCentral Features and API Usage

XperienCentral and the XperienCentral API offers a standard set of features and functions that guarantees backwards compatibility when used. Therefore, it is recommended that you use these features and API functions as much as possible rather than implementing custom functions [**G022**].

Service Dependencies

Plugins that use a service must define a service dependency with this service in the component definition in order to retrieve a reference to the service [**G023**]. For example:

```
ComponentDependencyImpl serviceDependency = new ComponentDependencyImpl();
serviceDependency.setServiceName(BookService.class.getName());
serviceDependency.setRequired(true);
elementDefinition.setDependencies(
  new ComponentDependency[]{serviceDependency});
```

Back to top

Internationalization Guidelines

This part covers guidelines which guarantee that content and presentation generated by a plugin is prepared to deal with multiple locales.

Translatability

All text, images and other GUI components should be suitable for translation [**G024**]. Text labels should not be hard-coded: use `<fmt:message …/>` for the Editor JSPs or `<wm:text …/>` for website environment JSPs), and images or other GUI components should not be country or language specific. A well known example of bad translatability is the **B** image button used in Microsoft Word; the image is the same for all languages although the **B** refers to **B**old in the English language. On the contrary, OpenOffice provides a better example by translating this image in other languages (like **V** of "Vet" in the Dutch language). Furthermore, content created by the plugin should be translatable [**G025**]. This means that an editor must be able to edit the content in multiple languages. Note that this is automatically supported for element components, since elements are contained by pages that themselves can have different versions for each language.

Language Files

Language labels should be defined in language resource files conforming to JavaI18N. The name of these language files conforms to the syntax [**G026**]:

```
messages_<language>_<country>_<variant>.properties
```

The language file contains the language labels used throughout the entire plugin but its scope is limited to that plugin. Using the same language labels in different plugins will not cause conflicts. It is recommended that you use only lower case letters for the label ID [**G027**] and that you group the language labels per component and prefix them with at least the ID of the component [**G028**]. All labels with an ID prefixed by panel.button are reserved by the framework for retrieving language labels for panel buttons - it is recommended that you group these labels separately [**G127**].

For example, if a plugin has three components with IDs `reviewelement`, `reviewmediaitem` and `maintenancepanel`, the US English resource file might look like this:

```
// Framework labels
panel.button.apply=Apply
panel.button.close=Close
panel.button.ok=OK

// Labels for review element
reviewelement.menuitem=Books review element
reviewelement.headertitle=Books review element
reviewelement.title=Title
reviewelement.frontcover=Front cover

// Labels for review media item
reviewmediaitem.bookreview=Book review
reviewmediaitem.metadata=Metadata
reviewmediaitem.quote=Review quote
reviewmediaitem.select=Select

// Labels for maintenance panel
maintenancepanel.books=Books
maintenancepanel.reviewers=Reviewers
maintenancepanel.genres=Genres
```

Back to top

Documentation Guidelines

This part provides guidelines for documentation to be distributed among with the plugin.

Contents

For each plugin, the following documentation must be available:

- Javadoc describing all services and objects that the plugin exposes [**G074**].
- Online help covering the visible components contained by the plugin [**G108**].

The plugin should never contain the empty online help generated by the archetype [**G147**].

The user documentation should take the target audience into consideration. XperienCentral users are divided into the following categories:

- Casual user
- Editor
- Main Editor
- Application manager
- Developer

Document Language

Documentation should be available in at least US English [**G034**].

Back to top

Distribution Guidelines

A plugin is distributed as a single ZIP file, called a WCA (WebManager Component Archive) which can also contains other related plugins [**G035**]. The WCA contains the compiled plugins as JAR files, corresponding documentation and (optionally) the source code. The WCA contains the following directory structure [**G036**]:

`/bin` - plugin jar files.

`/src` - Source code of the plugins. The source of each plugin is located in a separate folder that is the same as the plugin ID. Including the source in the WCA is optional.

`/doc` - Contains all documentation (Javadoc and user manual(s)).

`/content` - Contains all content needed by the plugin.

`changelog.txt` - A text file without markup that contains a full history of all issues that have been fixed after the first release of the WCA (stored in the root of the WCA [**G035**]) It should be properly filled in..

`readme.txt` - A text file without markup that contains list of new features in the latest version of the WCA, list of hardware & software requirements, license statement and known issues (stored in the root of the WCA [**G035**]) It should be properly filled in.

The plugin JAR file containing the actual software component contains the following directory structure [**G039**]:

`/<package name>` - Contains the compiled Java classes.

`/editpresentation` - Contains presentation JSPs for rendering the Editor.

`/showpresentation` - Contains presentation JSPs for rendering the frontend of the website.

`/help` - Contains the online help files.

`/messages` - Contains the message language files.

`/static` - Contains static files like images and JavaScript..

`/tags` - Contains custom JSP tags.

`/sqlscripts` - Contains all SQL creation scripts needed by the plugin at runtime.

`/content` - Contains all content import ZIP file needed by the plugin at runtime.

`/configuration` - Contains configuration files like metatype files for Configuration Management.

A plugin should contain only components that logically belong to each other [**G145**].

Back to top

Quality Guidelines

In order for GX Software to be able to effectively monitor adherence to the quality guidelines, a quality assurance report must accompany a plugin that is submitted to GX Software for certification. This report must detail various subjects in order for certification to be able to take place. Please note that the discussion of the plugin quality guidelines will be kept in relatively general terms because an exhaustive treatment thereof would not be practical given the goal of this part. Similarly, the methods for validating adherence to the quality guidelines will also only be discussed in general terms.

Functionality Guidelines

Based upon the functional requirements, a set of test cases is executed. The test cases should represent the most important business processes and also the different user groups (casual user, editor, main editor, application manager and developer). Preferably, the test cases should be designed before test execution. The system is then installed in a (production-like) test environment. By executing the test cases, the correct working of the system is verified and validated.

All defects found are assessed with a certain severity (critical, major, minor).

Examples of important user test cases are:

- For element components, have they been tested on page versions, media items, and page sections?

Other test cases could be:

- Is a proper boundary test performed on the properties of objects managed by the plugin?
- Are objects managed by the plugin and deleted by the editor also removed from the JCR?
- Does the plugin clean up its created content properly? Does the purge really remove all objects that were created by the plugin?

Performance Guidelines

The performance of the plugin under specific conditions is stated unambiguously in the requirements. An indication of the performance can be given by collecting performance statistics during the test execution of the functional test cases. The test environment should be production-like. A more serious approach is the execution of a dedicated performance test. Based upon the performance requirements, the test environment is created and it corresponds exactly with the production environment. The predicted system load is translated into a set of test cases in which the different user groups (including system functions) simulate the intended production use. The database size should approach the production situation. By executing the test cases, the performance of the system is compared to the performance requirements and any defects are found.

Installability Guidelines

For the installability of the plugin, the plugin is tested by following the development lifecycle of the plugin in different test scenarios. Following a test scenario for the installation, a check is performed to ensure the end state of the installation process corresponds with the expected end state. Examples of important test scenarios are: Install, Start, Stop, Uninstall and Purge, combined into several updates, re-installations. Scenarios can also be defined for installation of a clean site and for an upgrade of an existing site.

Stability Guidelines

Generally speaking, the level of stability is measured by methodically, i.e. systematically, running the software during a specified duration, preferably through the means of a balanced set of representative test cases and then registering the number of defects and/or the mean time in-between the occurrences of defects. An indication for the stability is the Mean Time Between Failures (MTBF), the average interval in-between defects that occur during one or more measurement sessions.

Compatibility Guidelines

Software and Hardware Compatibility

For each XperienCentral release, the Hard and Software Requirements is updated which describes the hardware and software on which XperienCentral is supported. The plugin should be compliant with these requirements [**G154**].

XHTML Compliance

The HTML generated by the JSPs for rendering the plugin in the Editor must be XHTML 1.0 transitional compliant [**G042**]. HTML generated by the JSPs for rendering the website environment should be XHTML 1.0 transitional compliant [**G134**].

Scalability Guidelines

Examples of important test scenarios with regard to scalability:

- Do the components of the plugin work correctly even if thousands of object instances are created and managed by the plugin, or when thousands of objects are created that are relevant to the plugin?
- When files are written to disk by the plugin, are these files written to the correct location, are they deleted when the corresponding object is deleted and does the plugin consider the fact that one directory should contain no more than about 30.000 files?

Quality Report

In order to be able to effectively assess adherence to the quality guidelines, a quality assurance report may be used.

Unit Tests

Each plugin should come with a unit test or test bundle that has a code coverage of at least 10% [**G120**]**.**

Back to top

DTAP Guidelines

DTAP is an abbreviation of Development, Test, Acceptance and Production. These terms refer to the environments on which the software will be installed, starting on a development environment it will be installed next onto the test environment. When development and bug fixing is finished it is installed onto an acceptance environment to be tested by the customer. Finally if the software component is accepted, it will be installed onto the production environment.

The XperienCentral platform provides an API that helps the software component to take DTAP into consideration. Guidelines are therefore about proper usage of this API.

Export and Import

An important aspect of a DTAP model is that content created on one environment must be exported so that it can be imported into another environment. To enable this, all content should be stored in the JCR by default [**G045**]. Only if there is a good reason to do so, content may be stored in an internal relational database but the SQL scripts to create those database tables should be contained by in the plugin JAR file (see G039) in a `/sqlscripts` directory [**G119**].

Configuration

For persisting configuration options for a plugin use the Configuration Management Service offered by the XperienCentral platform [**G046**]. This ensures a proper distribution and maintenance of configuration options in multiple environments as applicable for the DTAP model. Note that configuration options are options that must be editable by the end-user are read but not modified by the software. Avoid hard-coded web IDs, paths, context paths and URLs - use the Configuration Management Service instead. The ID of each configuration set defined by the plugin must be prefixed by or equal the plugin ID [**G152**].

Preferences

For persisting preferences for a plugin, use the Preferences Service offered by the XperienCentral platform [**G047**]. This ensures a proper distribution and maintenance of preferences in multiple environments as applicable for the DTAP model. Note that preferences are properties that are maintained by the software and are read-only for the end-user.

Back to top

Security Guidelines

Web applications such as XperienCentral operate in a hostile environment. Over the last few years many powerful and extremely common types of security flaws have been found in web applications. It is an unfortunate truth that security flaws are very easy to introduce into an application. This part provides some basic guidelines for avoiding some common security errors when developing plugins. It is advised that every web developer at least be aware of the Open Web Application Security Project (OWASP) and its top ten list of security vulnerabilities for web applications. The list is updated each year and is located here: https://www.owasp.org/index.php/OWASP_Top_10.

More complete security guidelines are explained in Security Guidelines. The topics in this section are only visible to certified GX Software partners and customers who are logged in to the GX Software domain.

Escape Input from the User

Outputting user supplied data without correctly escaping it for HTML or XML can be exploited for Cross Site Scripting (XSS) attacks. Use the output-html-encoded-quotes XSLT template to properly HTML escape strings in the output when using XSLT [**G048**].

```
<xsl:call-template name="output-html-encoded-quotes">
<xsl:with-param name="text" select="$mystring"/>
</xsl:call-template>
```

In JSP, use standard JSTL functions (like `escapeXML()`) to properly escape strings in the HTML output [**G123**]:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
${fn:escapeXml(user.name)}
```

Use SQL Prepared Statements

SQL injection is a way to abuse poorly programmed SQL statements in order to execute statements that the developer did not foresee. Consider the following example of a badly programmed SQL query:

```
// Bad SQL query vulnerable to SQL injection
String query = "select count(*) from users where name=' + name + "' and password='" + password + "'";
```

If the name and password are directly passed from the posted form onto this query, the query will be vulnerable to SQL injection. If the user would enter username and password:

```
username=’ or 1=1 or name='
password=’ or 1=1 or password='
```

The query would always return 1. SQL injection can be easily prevented by using prepared statements instead. Prepared statements prevent SQL injection by automatically escaping the input. For this reason it is recommended that you use prepared statements in all cases. See `java.sql.PreparedStatement` [**G049**].

Use of RBAC Permissions

The XperienCentral Editor supports basic authorization in the sense that XperienCentral functionality can only be accessed by users that are logged in and elements can only be updated if the editor has permission to update the page. This is basically what the XperienCentral platform supports concerning authorization. Any additional authorization has to be programmed within the plugin itself.

For this purpose the XperienCentral platform supports role-based RBAC security. Each component in the plugin that has a GUI must define at least one RBAC category and RBAC permissions within that category [**G050**]. A category may contain permissions used by all components of the plugin, but it may also define a separate category for each component. Note that a component can only be disabled individually if it is contained by its own permission category.

A permission category should at least define (and implement) the following RBAC permissions [**G051**]:

- For elements: permission to create, edit and delete the element.
- For panels: permission to open the panel

The implementation of RBAC permission handling is programmed mainly in a controller or service not in the business object itself [**G052**]. The reason for this is that authorization defines whether a particular user within a particular context is allowed to perform a particular action. A controller or a service handles that particular action and may invoke multiple methods on multiple business objects to complete the action rather then invoking one method of one particular business object. Authorization check in a business object itself is only allowed if the permission covers only the retrieval or update of the corresponding property. The names of the RBAC category and permissions should follow the guidelines. The definition of an RBAC permission must always be positive and must be defined in a way that assigning the permission to a role always grants the role particular rights and never denies rights [**G05**]. See https://en.wikipedia.org/wiki/Role-based_access_control for the complete definition of the fundamental RBAC standard.

Serving Files

All content must be stored in the JCR [**G045**]. This of course also applies to files. Files can be stored in the JCR in two ways; public and non-public. Public files are copied to all web servers in the production environment and are thus accessible by anyone who visits the website. Non-public files are handled by the application server only and support custom authorization on file downloads.

Back to top

API

A plugin may expose an API containing interfaces, implementation classes, constants, etc. To ensure consistency over all APIs exposed by all plugins, this part describes which guidelines to which such an API should conform.

Exposing an API of a Domain Object

There are two ways to define an API that exposes the business logic of a domain object:

- Implement the API as a service.
- Implement and expose a domain object interface.

The following guidelines determine which method the API should be exposed:

[**G109**] Expose and implement a domain object interface if the API consists of getters and setters that get and set properties of that domain object only. For example; expose a `Book` interface with methods `getTitle()` and `setTitle()`. Preferably the class that implements this interface is post fixed with `Impl` , `BookImpl` for example. [**G110**].

[**G111**] Implement the API as a service if the API creates or deletes domain objects or operates on multiple domain objects. Preferably the service is postfixed with `ManagementService` [**G112**] For example, expose a `BookManagementService` that provides methods `createBook()`, `deleteBook()` and `resetBookTitles()`.

Exception Throwing

There is an important difference between unchecked and checked exceptions. An unchecked exception reflects errors that cannot be recovered from at run time and a checked exception reflects invalid conditions in areas outside the immediate control of the program. Unchecked exceptions derive from `Run timeException` and are usually not thrown explicitly in the method signature and not caught in a try/catch block. Checked exceptions, on the other hand, are thrown explicitly in the method signature and should thus be surrounded by a try/catch block by the invoker of the method.

There are three possible ways to deal with exceptions in the method of an API: throw a checked exception, throw an unchecked exception or handle the exception internally by logging an error message. The following guidelines apply on throwing exceptions.

[**G113**] Do not throw an exception but handle the error internally if it is known how to deal with it and/or what the invoker of that method wants to do in that case, that is, if the exception is recoverable. For example; an exception that occurs when null is provided in a method `getBook(String title)` can be handled internally by logging a warning message and returning null.

[**G129**] Do not catch an unchecked exception if it is unknown how to deal with it and/or what the invoker of that method wants to do when such an exception occurs, that is, if the exception is unrecoverable and the exception occurs in areas inside the immediate control of the program. For example, an unexpected and unrecoverable `NullPointerException` that occurs somewhere deep inside `getBook(String title)` within the immediate control of the program should not be caught.

[**G114**] Throw a checked exception if it is unknown how to deal with it and/or what the invoker of that method wants to do when such an exception occurs that is, if the exception is unrecoverable and the exception occurs in areas outside the immediate control of the program. For example, an SQL exception that occurs inside `getBook(String title)` for an unknown reason outside the immediate control of the program should be thrown.

Remember that any checked exception an API method throws must be caught separately by the invoker of that method. So, if an API method throws five or more different exceptions, the invoker must add the same amount of catch statements to deal with those. In such a case, it is recommended to create a new checked exception type as wrapper for those exceptions (do not forget to expose this exception too) [**G131**].

Exposing RBAC Permissions

Since a plugin may perform authorization checks on RBAC permissions defined by other plugins, a plugin should expose the RBAC permission values. The RBAC permission values should be defined as `public static final String` attributes contained by a Java class as described by G116.

Defining and Exposing Constants

It is generally a good idea to collect IDs like Plugin ID, namespace URI, namespace prefix, etc., in one class that defines these using public static final fields [**G116**]. Since other plugins may also depend on these identifiers, these static fields should be contained by a class that is exposed in the API [**G117**].

Package Name of the Exposed API

Any class or interface that is exposed by a plugin as API should be contained by a package called "api" directly under the plugins root package or by a sub package of this package [**G118**]. Because of guideline [**G090**], the full package name of this API package will be:

```
<domain>.<plugin ID>.api
```

Back to top

Presentation Plugins

Presentation plugins differ in in many ways from other plugins. They contain the actual representation of the website as a whole and thus presentations (in JSP) of several components deployed from several plugins. They overrule presentations defined by the plugins themselves, which should be considered example presentations. A presentation plugin usually is customer-specific - it contains the presentation of that customer's website. For that reason, separate guidelines are applicable for presentation plugins. A presentation plugin must be self-sufficient - it should not depend on any resource (JSP, CSS, etc.) provided by another plugin unless this presentation is a plugin itself [**G137**]. If a presentation plugin depends on presentation JSPs that are part of the original XperienCentral platform, these JSPs should be copied to the presentation plugin in the directory `/src/main/resources/presentationtype/jsp/wm` [**G138**]. Note that the guidelines mentioned in the remainder of this part do not apply to these JSPs.

The value of the name attribute in the descriptor of a JSP should be prefixed by the plugin ID [**G139**]. Static files used by a presentation plugin should be located in a subdirectory that equals the plugin ID (`/src/main/resources/presentationtype/static/<plugin ID>`) [**G140**]. A common problem in the presentation JSP of media item content is recursion. A media collection may display the contents of a media item while the media item itself contains the same media collection element. In that case, the media item will be rendered recursively causing an endless loop and bring the server down. For that reason, a JSP that displays the content of a media item should check whether it has not already been rendered within the same request [**G142**].

Use the `Link.linkAttributes` method to build links in a JSP instead of constructing the href attributes manually [**G143**].

Finally, implement caching properly by providing SSIs [**G144**].

Back to top

Coding Conventions

This part describes the recommended coding conventions for XperienCentral development. These conventions are based on Sun's coding conventions but additional XperienCentral specific conventions apply. G055 covers Sun's coding conventions, all other conventions in this chapter are XperienCentral specific.

Conform to Coding Conventions Defined by Sun [G055]

Developers are encouraged to follow the code conventions that Sun publishes as the standard for the Java Programming Language. In addition to these standard conventions GX maintains a small set of custom conventions used in XperienCentral projects. These conventions are described in the other parts below.

Use of Java Language Features [G056]

Most language features introduced in Java 5 and higher either make code easier to write and read or provide compile time type safety (or both). Use generics and the enhanced for-loop where applicable.

Use of Java Util Logging [G057]

All XperienCentral logging is standardized through the Java Logging API. Log levels used to publish log entries conform to the following rules:

- FINE, FINER and FINEST are used to display information that is usually not of interest by a system administrator. Its only intention is to provide additional debug information which could help solving a problem in this area.
- INFO is used to display information that is potentially of interest for a system administrator. The message is purely informational and does not imply a potential problem.
- WARNING is used to indicate a problem or potential problem. A log entry of this level is by definition of interest to a system administrator or developer and should be investigated further if it appears.
- SEVERE is used to indicate a (serious) problem. A log entry of this level is of high interest to a system administrator or developer and should be investigated whenever it appears as soon as possible.

Use of Java Util Concurrent [G058]

When guarding thread safety, use the Java concurrency utilities for optimal performance and scalability.

Use of Collections [G059]

Do not use old collections like `Hashtable`, `Vector` or `Dictionary` when it can be avoided. They are known to be heavy and over-synchronized. Use `ArrayList` instead of `Vector`, `HashMap` instead of `Hashtable` or `Dictionary`.

Use of @override [G060]

The use of the `@override` annotation is mandatory if a method is overridden from a super class. If the annotation is used, the compiler checks whether the overriding method signature is compliant with the overridden method thus preventing any errors due to misspecification.

Hungarian Notation [G061]

The use of a basic variant of the Hungarian notation is encouraged. It is recommended that you prefix instance variables with “my”. For example:

```
// Correct
private String myTitle;

// Incorrect
private String title;
```

Blank Spaces [G063]

Use of blank space is discouraged before and after a method argument but it is mandatory after Java keywords and commas. For example:

```
// Correct
public void makeItSo(String action, String expression){
    if (check(expression)) {
        makeItSo(action);
    }
}

// Incorrect
public void makeItSo( String action , String expression ) {
    if(check(expression)) {
        makeItSo(action);
    }
}
```

Order of Methods and Variables [G064]

It is recommended that class and instance members be declared in the following order:

1. **Class (static) variables** - Declare first the public class variables, then the protected, then package level (no access modifier), and then the private.
2. **Instance variables** - First the public class variables, then the protected, then package level (no access modifier), and then the private.
3. **Constructors**
4. **Methods**

Use of FIXME [G067]

It is recommended that you use the comment “FIXME” in the source code to indicate that the implementation is incorrect. It may be used, for example, to indicate that the code snippet fails in some cases or that it is a workaround for another issue. It is also recommended that you add the name of the developer that added the FIXME to the code. For example:

```
// FIXME (ivol): this won't work if myNumber equals 0
public String getProperty();
    return 1/myNumber;
}
```

⚠ FIXME's should only be used during development and therefore be removed in the release.

Use of TODO [G068]

It is recommended that you use a TODO in the code if the code snippet works but is not yet finalized. It is also recommended that you add the name of the developer that added the TODO to the code. For example:

```
public int doComplexCalculation();
    // TODO (ivol): must be implemented, return 0 for now
    return 0;
}
```

⚠ TODO's should only be used during development and therefore be removed in the release.

Commented Out, Duplicate or Unused Code [G141]

The software should not contain code snippets that are commented out or not used anymore. Nor should it contain duplicated code within the same plugin. Use JSP tags, services or separate methods to prevent duplicate code.

Indentation [G069]

Do not use tabs in the source code - use spaces. Tabs are interpreted differently in different editors which can make the source code hard to read. There are plugins available that convert tabs to spaces automatically.

Maximum Line Length [G070]

It is recommended that you limit the size of one line in the software code to 120 characters.

Brackets on the Same Line [G071]

It is recommended that you add start brackets on the same line as the statement to which they apply and to add ending bracket on a new line. For example:

```
// Correct
public void makeItSo() {
    if (isTrue) {
        doSomething();
    }
}

// Incorrect
Public void makeItSo()
{
    if (isTrue)
    {
        doSomething();
    }
}
```

Brackets for Single Line Statements [G072]

It is recommended that you use brackets in all cases, even for single line statements. For example:

```
// Correct
if (expression) {
    doSomething();
}

// Incorrect
if (expression) doSomething();
if (expression)
    doSomething();
```

Instance Variable Access [G148]

Instance variables should always be private and provided by accessor methods (getter and setter). Using private instead of public and protected instance variables prevents tight code coupling between the class and classes from other plugins that use this class.

Back to top

Javadoc Conventions

This part describes the recommended Javadoc conventions for XperienCentral development. These conventions are based on Sun's Javadoc conventions. Additional XperienCentral specific conventions are also explained below.

Conform to Javadoc Conventions Defined by Sun [G073]

Developers are encouraged to follow the Javadoc guidelines that Sun publishes as the standard for the Java Programming Language.

Javadoc on Classes, Interfaces, Variables and Methods [G074]

Public and protected classes, interfaces, variables and methods should be tagged with Javadoc notation. For private class and instance variables, a short non-Javadoc comment behind the variable is sufficient.

References [G075]

The Javadoc should not contain references to documents that might not be accessible by external developers. References to a local Wiki or other document must be avoided at all times.

Content

The Javadoc should clearly explain how the class or interface should be used, preferably by using code examples [**G132**]. For each package contained by the plugin, a `package.html` should be provided that clarifies the purpose and contents of the package [**G133**].

Back to top

Class Name Conventions

For consistency of the source code of all plugins, it is important to define naming conventions. This makes the plugin easy to read and maintain for developers since the same classes will be named the same way in each plugin. This part defines naming conventions for Java classes.

Generic Plugin Classes

For general classes (classes that are not directly related to a particular component implementation) used in plugins, the following naming convention for the Java classes are recommended [**G076**]. Note that `<name>` indicates a custom name defined by the developer.

Java class name	Description
`Activator.java`	The activator class of the plugin.
`<name>ComponentDefinitionImpl.java`	The implementation of the specific component definition.
`<name>Editor.java`	The property editor(s) used to convert `String` to a complex object type and vice versa. The class must implement the `PropertyEditor` interface.
`<name>Validator.java`	The validator used to validate user input. The class must implement the `Validator` interface.

Element Components

For element components, the following naming convention for the Java classes are recommended [**G077**]. Note that <name> indicates a custom name defined by the developer like ‘review’ or ‘webshop’.

Java class name	Description
`<name>Element.java`	The interface of the element’s business logic.
`<name>ElementImpl.java`	The implementation class of the element containing the actual business logic.
`<name>ElementFBO.java`	The form backing object for the element. For element components usually there is only one form backing object.
`<name>ElementController.java`	The controller for the element component. An element component can have only one controller.
`<name>ElementComponent.java`	The custom implementation of the element component. Usually the `SimpleElementComponent` can be used instead.

To properly use the API for element components offered by the XperienCentral platform, it is recommended that you use the following hierarchy for the classes contained by an element component **[G078]**:

Java class	Extends	Implements
`<name>Element`	`Element`	
`<name>ElementImpl`	`ElementBase`	`<name>Element`
`<name>ElementFBO`	`ElementFBO`	`<name>Element`
`<name>ElementController`	`ElementComponentController`	
`<name>ElementComponent`	`SimpleElementComponent`	

Media Item Components

For media item components, the following naming convention for the Java classes are recommended [**G079**]. Note that the difference between a media item and a media item version is that a media item has one or more versions. The implementation of the media item itself is not contained by a media item component but by the implementation of the media item version. Again, <name> indicates a custom name defined by the developer like "article" or "movie".

Java class name	Description
`<name>MediaItemVersion.java`	The interface of the media item’s business logic.
`<name>MediaItemVersionImpl.java`	The implementation class of the media item version containing the actual business logic.

`<name>MediaItemVersionFBO.java`	The form backing object for the media item version. For media item components there is usually only one form backing object.
`<name>MediaItemController.java`	The controller for the media item component. A media item component can have only one controller.
`<name>MediaItemComponent.java`	The custom implementation of the media item component. Usually the `SimpleMediaItemComponent` can be used instead.

To properly use the API for element components offered by the XperienCentral platform, it is recommended that you use the following hierarchy for the classes contained by a media item component [**G080**]:

Java class name	Extends	Implements
`<name>MediaItemVersion`	`MediaItemVersion` or `MediaItemArticleVersion`	
`<name>MediaItemVersionImpl`	`MediaItemVersionImpl` or `MediaItemArticleVersionImpl`	`<name>MediaItemVersion`
`<name>MediaItemVersionFBO`	`MediaItemVersionFBO`	`<name>MediaItemVersion`
`<name>MediaItemController`	`MediaItemComponentController`	
`<name>MediaItemComponent`	`SimpleMediaItemComponent`	

Panel Components

For panel components, the following naming convention for the Java classes are recommended [**G081**]. In the table below, `<name>` indicates the name of the panel as a whole while `<tab>` indicates the name of one particular (horizontal, vertical or sub) tab within that panel.

Java class name	Description
`<name>Panel.java`	The implementation of the controller logic for the panel. This is in fact a helper class for the panel component controller.
`<name>PanelComponent.java`	The custom implementation of the panel component - usually the `SimplePanelComponent` can be used instead.
`<name>PanelController.java`	The controller for the panel component. Each panel component has exactly one controller, even if it has several horizontal and vertical tabs.
`<tab>TabFBO.java`	The form backing object for this particular tab of the panel. Usually there is one form backing object for each horizontal, vertical or sub tab.
`<tab>TabController.java`	The controller for this particular tab of the panel. Usually there is one controller for each horizontal, vertical or sub tab.

To properly use the API for element components offered by the XperienCentral platform, it is recommended that you use the following hierarchy for the classes contained by a panel component [**G082**]:

Java class name	Extends	Implements
`<name>Panel`	`PanelBase`	
`<name>PanelComponent`	`SimplePanelComponent`	
`<name>PanelController`	`PanelComponentController`	

Back to top

ID Naming Conventions

Because the ecosystem of plugins will cause a rapidly increasing number of available plugins developed by many different companies, unique identifiers are necessary to prevent conflicts between plugins. This part describes the guidelines for which identifiers defined by a plugin should conform to in order to prevent such conflicts.

Domain and Plugin ID

The domain is a unique identifier which usually equals the top level domain and name of a company, separated by a dot (for example `com.gx`). This domain must be returned by the implementation of `componentBundleDefinition.getDomain()` [**G083**]. The same domain will be used for all plugins developed by that company. The domain may only contain alphanumeric characters in the range [a-z] separated by dots [**G122**]**.** For each plugin, an additional unique identifier should be defined, called the plugin ID, that is returned by `ComponentBundleDefinition.getWCBId()` [**G084**]. This unique identifier will prevent all kinds of collisions that could occur otherwise. The plugin ID may only contain alphanumeric characters in the range [a-z] [**G121**] - spaces, dots, numbers and capital letters are not allowed.

Both the plugin ID and domain must be approved upon by GX and registered. Uploading the version of the plugin to be certified is required for certification [**G149**]. Note that it is not required to actually upload the WCA of that version - only registration of the version is required.

Component IDs

A plugin consists of one or more components. The activator of a plugin defines what component a plugin contains by defining one component bundle definition which registers one or more component definitions. The ID of the component bundle definition defined in the activator of the plugin must follow the following syntax [**G097**]:

```
<domain>.<plugin ID>
```

The ID of each component definition contained by the plugin must be prefixed with the component bundle definition ID followed by an ID that is unique within the plugin and must match the component name and consist of lowercase alphanumeric characters in the range [a-z] [**G098**]. We refer to this unique ID within the plugin as the component ID and the component definition ID conforms to the syntax:

```
<domain>.<plugin ID>.<Component ID>
```

For media Item components, the content type (defined by the `@ContentType` annotation in the media Item version implementation class) must equal the plugin ID or be prefixed by the plugin ID and may only contain alphanumeric characters in the range [a-z] [**G124**].

Package Conventions

The package names used by the plugin must conform to the Java standard as also stated in the Java coding conventions by SUN [**G089**]. The top level package name must meet the following syntax [**G090**]:

```
<domain>.<plugin ID>
```

Artifact and Group IDs

For the artifact ID in the `pom.xml` it is recommended that you use the plugin ID [**G095**]. For the group ID it is recommended that you use the domain [**G096**]. The `bundleSymbolicName` in the `pom.xml` matches the syntax `<domain>.<plugin ID>` **[G136]**.

Scheduled Jobs

If a plugin creates scheduled jobs, the name of those scheduled jobs must be prefixed by the plugin ID in order to prevent collisions with jobs created by other plugins [**G151**]. The scheduler service assumes that a job name is unique.

Summary

This part provides an overview of the recommended naming conventions as described above. An additional guideline is that all these properties may use only lowercase letters [**G099**]:

Property	Example
Domain name	`<domain name>`
Plugin ID	`<plugin ID>`
Namespace prefix	`<plugin ID>`
Component bundle definition ID	`<domain name>.<plugin ID>`
Component definition ID	`<domain name>.<plugin ID>.<Component ID>`
Package of the plugin	`<domain name>.<plugin ID>`
Package of component	`<domain name>.<plugin ID>.<Component ID>`
Artifact ID	`<plugin ID>`
Group ID	`<domain name>`
bundleSymbolicName	`<domain name>.<plugin ID>`
Content Type	`<plugin ID>`

RBAC Naming

Each plugin defines at least one RBAC category which may contain one or more RBAC permissions. The technical naming of all RBAC categories and permissions must be unique in order to avoid collisions across multiple plugins. For this reason, the technical name of each RBAC category must conform to the syntax `<domain>.<plugin ID>.<Component ID>` [**G101**]. Furthermore, the technical name of each RBAC permission must be prefixed with the technical name of the RBAC category, followed by a dot [**G102**]. All technical names of categories and permissions must be lowercase, may not contain spaces and separate words must be separated by a dot [**G103**].

RBAC Naming Example

This part provides an example for using proper naming conventions:

Property	Example
Domain name	`com.libris4you`
Plugin ID	`books`
Namespace prefix	`books`
Component bundle definition ID	`com.libris4you.books`
Component ID	`maintenancepanel`
Component definition ID	`com.libris4you.books.maintenancepanel`
Plugin package	`com.libris4you.books`
Component package	`com.libris4you.books.maintenancepanel`
Artifact ID	`books`
Group ID	`com.libris4you`
Content Type	`Books`
RBAC category maintenance panel component	`com.libris4you.books.maintenancepanel`
RBAC permission add book	`com.libris4you.books.maintenancepanel.add_book`

CRUD Naming

It is recommended that you use the following keywords for common create, read, update and delete actions [**G104**]:

- **create** - create an object

- **add** - create a reference to an existing object
- **insert** - insert a reference to an existing object into a list
- **move** - move an object
- **delete** - delete an object and any references to that object
- **remove** - remove a reference to an existing object
- **update** - update the properties of an object with new values

XperienCentral Naming

When writing documentation or online help for plugins, it is recommended that you use the following guidelines regarding naming of XperienCentral assets [**G105**]:

Property	Example
Editor	The XperienCentral application in which editors maintain the content to be published on the website or another medium.
Website environment	A presentation of (a part of) the content maintained by XperienCentral, usually a website.
Standalone	The mode in which XperienCentral content is served from only one application server.
Clustered	The mode in which XperienCentral content is served from at least two application servers.
Read/write	The application server on which the Editor can be used to maintain the content. In standalone mode, there is only one read/write node. In clustered mode, there is one read/write node and one or more read-only nodes.
Read-only	The application server which only serves content (the website environment) but cannot be used to maintain the content because it is read-only. In standalone mode, there is no read-only. In clustered mode, there are one or more read-only nodes.
External server name	The public hostname of the application server accessible from the internet. The website environment is accessible from this hostname.
Internal server name	The hostname of the application server that exists only in the local network. The Editor is accessible only from this host name.

Language

The language to be used for all names, codes, Java classes, methods, properties, etc. is US English [**G106**]. A different language may only be used if, and only if, it represents a piece of text that is displayed to the end user and may require translation.

Back to top