Module Development Guidelines and Best Practices

Developer Guide

Version: Kaltura MediaSpace 5

Kaltura Business Headquarters

5 Union Square West, Suite 602, New York, NY, 10003, USA
Tel.: +1 800 871 5224

Contents

SECTION 1

Understanding MediaSpace Modules

Kaltura MediaSpace (also referred to as KMS) enables community, collaboration and social activities by leveraging the power of online video by using an out-of-the-box video portal.

KMS is a Kaltura API based application. KMS uses its associated Kaltura Account to store content, metadata and user information and does not have a local database of its own.

MediaSpace is a highly customizable application. Administrators can configure and manage the application through an administration interface and developers can extend its functionality and add new features through its modular architecture of Model View Controller (MVC) based modules.

KMS Architecture – Core and Modules

This section describes the KMS architecture.

Core Architecture

KMS is implemented as a Zend-Framework application and as such, the main architectural feature to consider is the MVC based modules.

MediaSpace's core is divided into two main parts:

- Application MVC – the core set of models, controllers and views. The main features of KMS, which are considered core, are implemented in this part of the system.
- KMS library – a set of classes that the application uses to manage different flows and resources.

The KMS library also implements the generic infrastructure that allows extending MediaSpace via modules and themes.

NOTE: Custom themes are not covered in this document.

MediaSpace Modules

MediaSpace modules are code bundles that are deployed in a MediaSpace installation.

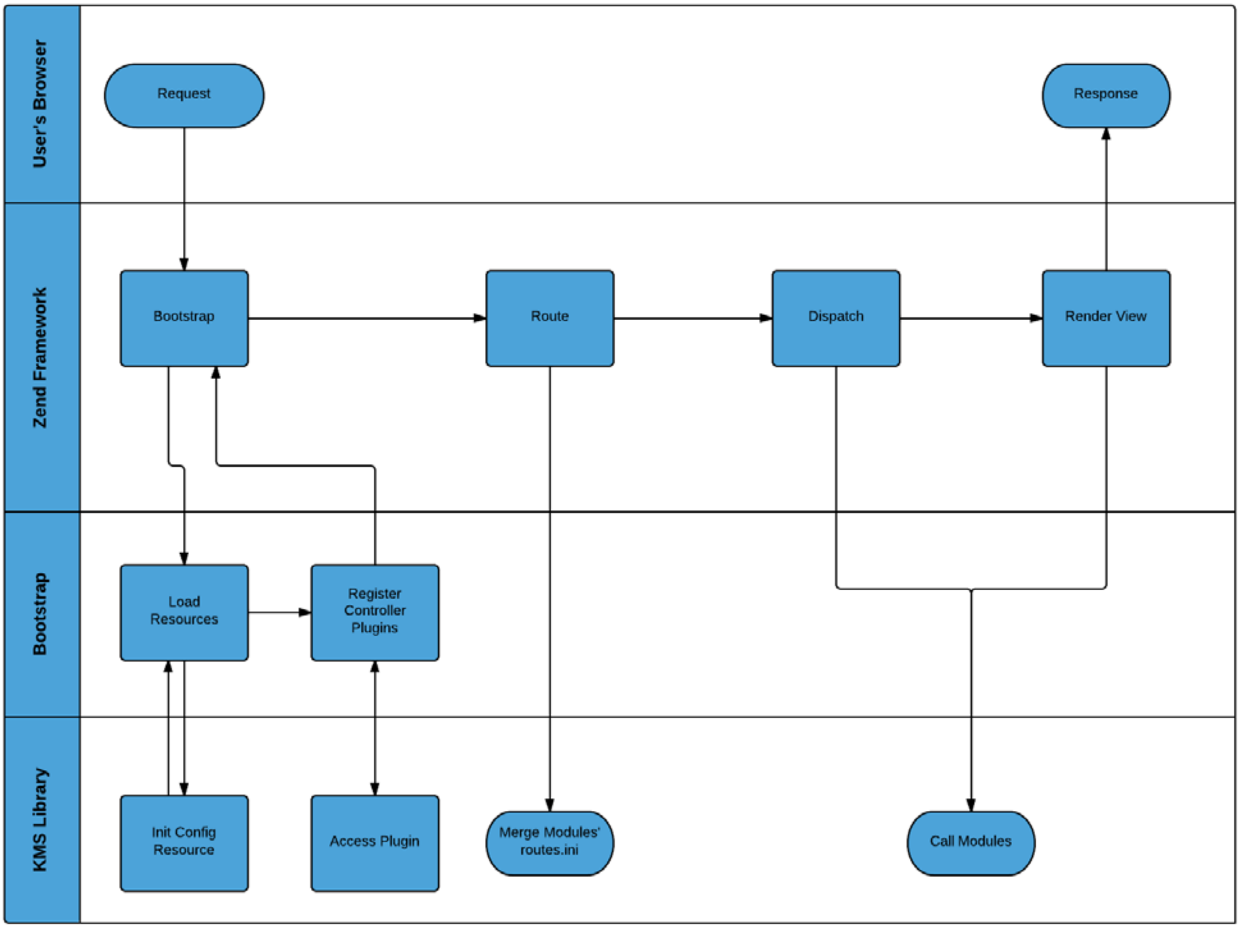
Modules are used to add new capabilities and features to the core functionality of KMS, and even affect or change the way KMS was designed to behave.

To keep the system's behavior cohesive, unified and supported, modules are also expected to use the MVC pattern and be implemented according to the guidelines described in this guide.

SECTION 2

MediaSpace Application Workflow

The following diagram describes the KMS application flow.



Most of the flow actually represents processes that are managed by the Zend Framework (such as Bootstrap and dispatch).

The interesting and important part of the diagram is the bottom swim lane – controlled by the KMS Library. These are the main entry points where KMS interacts with modules.

How Are Modules Initiated?

During different phases, KMS “reaches out” to all enabled modules, through different predefined class interfaces, to allow modules to participate in the application flows.

For example, during the initialization of the Access Plugin, KMS loads all the modules that implement an interface called “*Kms_Interface_Access*” and calls the *getAccessRules()* method that these modules are obliged to implement.

Any KMS module would implement the list of interfaces, which define the entry points that the module would like to be called upon.

Why Class Interfaces?

The primary reason for choosing to use class interfaces to implement the KMS modularity is to be able to maintain backwards compatibility.

Interfaces should not change between KMS versions, unless absolutely necessary.
During development we cautiously consider any changes before applying a change to an existing interface.

The secondary reason for choosing to use class interfaces to implement the KMS modularity, is to be able to easily determine which module should be called for what entry point.

SECTION 3

Module Development Guidelines

Except for modules implemented for private use, there are three distribution channels for custom KMS modules:

Distribution Channels

- Bundled within the Kaltura MediaSpace download package (as a default module, for self-hosted KMS installations)
- Deployed on Kaltura's MediaSpace SaaS Environment
- Downloadable via Kaltura Exchange to be used in self-hosted KMS installations

In any of these cases (including modules implemented for private use only) we encourage you to consider the recommendations in this section.

The goal of these recommendations is to ensure that KMS modules are developed and distributed as high quality software.

Coding Standards

KMS follows Zend Framework coding standards – MediaSpace modules are required to follow these standards as well.

Module Naming Convention

It is recommended to prefix the module name with the company or service name to avoid conflicts with other modules.

Refrain from using the customer name as a module name.

General Do's and Don'ts

- Stick to what KMS allows you to do.
  - Implement the available interfaces (under library/Kms/Interfaces/).
  - Use the asset-delivery action for providing your modules JS/CSS/image (or other static) files. Remember to build your URLs via the *cdnUrl* ViewHelper.
  - Avoid implementing your module through hacks based on Zend Framework behaviour and capabilities.

> **NOTE:** Keeping these recommendations will ensure maximum forward compatibility of your module, thus lowering the chances of significant update work when upgrading your module to future versions of KMS.

  - Your module should NEVER modify KMS core or rely on hacks or patches applied to core files of KMS.

> **NOTE:** Failing to comply with these recommendations or modifying core code makes the KMS upgrade process more difficult and complex.

  - Write safe code so errors or misconfigurations in your module will not crash the entire application.

> **NOTE:** Keeping this recommendation ensures that even if your module is malfunctioning, it does not stop the application from working, thus allowing more flexibility in support.

  - Modules should not modify or override KMC core/default CSS rules or JavaScript code.

> **NOTE:**
>
> - Keeping this recommendation should assist keeping highest forward compatibility.
> - It is OK for your module to provide a CSS for its own element, however, you are recommended to use available CSS definitions from Twitter Bootstrap (utilized by KMS).
> - It is OK for your module to provide JS for its own functionality.

  - Keep your code organized correctly, according to an MVC pattern.
    - Implement logic and any API (for example, Kaltura API) calls in the ***model***
    - Call model methods from controller-action and prepare view data
    - Keep only simple loop and if logic in view while rendering HTML
    - Implement ViewHelper if more complicated (or repeatable) logic is required within a view.
  - Use KMS cache infrastructure when possible and relevant.
  - Do not leave calls to *Kms_Log::trace()* in your final code.
  - Choose smartly when to use available public methods from existing models (for implementing API calls to Kaltura) or implement your own code for making API calls.
  - When using existing KMS models you should usually get a model's instance through *Kms_Resource_Models::get{ModelName}()*. For example:

```
$entryModel = Kms_Resource_Models::getEntry();
```

> **NOTE:** Refrain from instantiating model objects like:
>
> $entryModel = new Application_Model_Entry();
>
> If you do choose instantiate model objects, be extra careful as doing so may have global implications on KMS behavior.

- If your module depends on another module, remember to implement the *Kms_Interface_Model_Dependency* interface.

Security Considerations

Avoid doing anything you would not do if you were to host the code on your own server.

General Security Guidelines

- Make sure your code is safe against OWASP Top 10 vulnerabilities.
- Avoid implementing file uploads directly to KMS servers. Use Kaltura's API and widgets to upload content to Kaltura.

- Do not use PHP’s *eval()* function to run user-input-based code.
- Never output sensitive information to the browser (for example, in case of errors).
- Avoid using *ini_set* in your module’s code.
- Do not use execute shell commands (for example, functions such as: *exec*, *system*, or *passthru*).

Specific Security Guidelines

- If you implement a form of your own to update data, protect against CSRF by having your form class extend the *Application_Form_Base* that handles CSRF protection for you.
- If you perform any redirects in your actions, make sure you only allow internal redirects or redirects to URLs that are valid and accepted. In other words – sanitize user-input URLs before redirecting to such.
- If your module writes cookie with sensitive data, set the httpOnly flag on your cookie so it cannot be read by malicious client-side code.
- Sanitize any user input to protect against XSS.

Expected Documentation

The following documentation is expected when developing a custom module:

Module Description

You are expected to provide a file with your module so that it is clear to the KMS Administrators and other developers what your module does.

KMS modules are expected to include a file called “module.info” at the root of the module folder.

The “module.info” file is expected to be in INI format.

Please use the “description” property in this file and provide a short description of the module’s added functionality.

Code Comments

Provide meaningful code comments in your module, specifically block-comments for classes and methods, so whoever does code review for certification purposes will be able to easily understand what each and every line/piece of code is responsible for.

Release Notes

You are expected to publish release notes for every version release of your module, listing available features, known issues and important notes if there are such.

Release notes should also include support contact information.

Remember that although your module may be bundled with KMS you are still the owner of the code for support purposes.

Setup Guide or Manual

When relevant, provide a setup guide or user manual for your module for users to be able to manage and use it.

Certification

For all distribution channels a module must go through certification process and be certified before made available and recommended to Kaltura MediaSpace customers.

FAQ

Who will be performing the module certification process?

A member of the Kaltura MediaSpace R&D team should perform the certification process.

What does the certification process include?

The process includes code review and sanity QA.

> **NOTE:** code review may also include the use of automatic code-analysis tools if Kaltura finds the need for it.

What do I do if my module failed the certification process?

Fix the issues reported by Kaltura and resubmit the module for an additional certification.

How do I release new versions or apply bug patches to my KMS module?

Depending on the fix, Kaltura may require that the module go through another certification process before being deployed/bundled again. Generally, you should follow the same guidelines in every release of your module, and the Kaltura MediaSpace R&D team will review your code before it is deployed to production.

What would cause my module certification to fail?

The following will fail the certification of a module (not ordered by significance):

- Critical or major bugs found during certification.
- Exposing of sensitive information to user or sending such information to external systems. Examples for sensitive information: Kaltura account information like admin secret, information about the webserver.
- Inability to perform QA due to missing documentation.
- Violation of any item from the Specific Security Guidelines section.
- Violation of any item from the General Security Guidelines section, under the discretion of the reviewer.
- Modification of KMS core code for the module to be operable.
- Exploiting ZF behaviour/capabilities instead of (or in addition to) using dedicated KMS interfaces – under the discretion of the reviewer.
- Trace logs left in the code.
- Not implementing the dependency interface while using another module’s code or decisions.
- Any attempt to obscure what your module is doing or parts of its code.
  For example, if you want to use a minified JS it is OK but you must include a non-minified

version of the JS code in the module as well.

- Loading static assets not via *cdnUrl* ViewHelper.

A module’s certification may fail for additional reasons not listed above.

In case of certification failure, developers will be notified about the reasons for failure so that you can fix and re-submit the module for a second certification.

SECTION 4

Other Recommendations

This section lists miscellaneous recommendations for developing custom modules for KMS.

UI Considerations

The core purpose of your custom module is to provide an integrated experience that is part of MediaSpace and enables additional functionality while following existing user experience and user flows.

If your custom module introduces a user interface that is vastly different from MediaSpace, you are forcing the user to learn a completely new interface and make it harder for them to embrace it. Make sure that the user interface elements are consistent with the MediaSpace interface and HTML tags to support the CSS file, so that your user interface does not "stand out".