Magnolia APIs

Magnolia provides both Java APIs, REST APIs and Definitions for Dialogs, Components etc. This page summarizes the ideas, thoughts and best practices around the Magnolia APIs. The intention is to let developers know about the APIs that are safe to use, and to indicate the internal codebase that are subject to change without notice.

This document is applicable to **Magnolia 5.6 and later.**

Disclaimer

With 5.6 we are actually still in the exploration phase and nothing is either set in stone nor considered the definitive approach for the future. Please consider that when you give the very welcome feedback.

General intention

We want to declare our opinion about what code should be used and extended as explicit as possible. Ideally in a way that leaves no space for mistakes through neglect and a very easy way to validate for compliance. Having said that, there is no intention to either hide internal code nor make it impossible to use. If you have a very unique use case and are aware of the implications when using our internal code you should be able to do so.

The reason why we want to have that boundary between internal code and users is, to keep our internal code flexible so we can innovate quickly without impact on current users. Ideally internal code only consists of code no customers is using anyway, but with making it explicitly internal code we have two huge benefits: We can change it as quickly and heavily as we want without being afraid of potentially breaking a promise to our customers AND we don't spend a lot of effort keeping that code backwards compatible for nobody. Depending on how near that ideal is to reality the transition phase will be shorter or longer.

Best practices

Usually we should consider the following list as potential part of the public API:

- Definitions (by *definition* all of them are public, maybe not all of them are directly accessible but thought the {{*2Bean}} mapping they are directly exposed)
- Actions
- Models
- Everything exposed through a RESTful service

The following list contains candidates to be usually considered private:

- Data bindings
- Pluming code in general
- Code of custom Vaadin components

The API Artifacts/Sub-Modules

If possible the API classes and interfaces of a Magnolia Module should be separated into a submodule (e.g. `magnolia-dam-api`). That module should have no dependencies on other internal submodules but be dependent on by all others. API artifacts should always be postfixed as `-api`.

Example needed

Annotations

There is a distinction between different types of API indicated by the presence of different annotations. For the moment we are only using {{@PublicApi}} but others are conceivable like for SPI or experimental APIs.

@PublicApi

API includes interfaces and classes that Magnolia modules need to use to get things done. Modules can safely use there and we will guarantee binary compatibility. This is true as well if you extend or implement these classes if possible.

@PublicSpi

@Experimental

Example code for Annotations needed

Compatibility Policy

Deprecation Policy

Methods or types that are deprecated will have the `@deprecated` annotation and according Javadoc comment.

The Javadoc will explain how to replace the usage and list the module version in which it was set to deprecated.

APIs marked as deprecated will continue to work in the following minor releases, but will be potentially removed in the next major release.

Further Information

- see the following concept Concept - Open Closed Code Space which outlines several steps beyond public APIs
- As an example have a look at the API documentation from Atlassian (https://developer.atlassian.com/jiradev/jira-apis/java-api-policy-for-jira)
- Presentation about Public API

Open Points / TBD

- [ ] Concrete examples for module structure
- [ ] Concrete examples for annotations

- [ ] Define policies with clear explanation how releases (both magnolia & module) related to deprecation and removal of APIs (see https://developer.atlassian.com/jiradev/jira-apis/java-api-policy-for-jira as an example)
- [ ] Explain good practices to extract public api on existing codebase
- [ ] Describe common pitfalls and how to avoid them
- [ ] Outline process to validate API consistency during development
- [ ] Outline process to assess public API compliancy in customer projects