Code Style

Your Rating:  Results:  143 rates

Java

General Style

- Check out Standard Java coding conventions-PDF-version, although it's still valid but please apply with care since the last revision to this document was made on April 20, 1999)
- Google's codestyle conventions are 99% what we do or should do. It's still up-to-date.

Order of methods

We aim for best readability - that's the reason why we (from now on) put methods in the specific order:

- Business methods first, technical/config methods last (e.g. getters, setters, add-methods for collections, ...)
    - U tilize lombok for technical/config (getter/setter, etc.) methods lest our code be flooded with trivial methods
- Methods that logically belong together should be placed next to each other (e.g public method with supporting private methods)

Imports

- We use the following order for import statements ⚠️
    - Static imports
    - info.magnolia.*
    - java.*; javax.*; org.*; com.*;
- .* imports are only allowed for static ones (checked by our checkstyle setting)
    - configure your IDE to use 2 as *Number of static imports needed for .** (e.g. in Eclipse Preferences -> Java -> Codestyle -> Organize Imports)

Indentation

- Tab policy: spaces only ⚠️
- Indentation: 4 spaces for Java-Files, 2 spaces for XML
- No maximum line width, no automated line wrap

Naming conventions

Package

- Package names are all lowercase, with consecutive words simply concatenated together (no underscores)
- Companies use their reversed Internet domain name to begin their package names
- Group the classes of same functionality/concept together
- Use the plural for packages with homogeneous contents and the singular for packages with heterogeneous contents.
    - A package named com.myproject.taskdoes not mean that each contained class is an instance of a task. There might be a TaskHandler, a TaskFactory, etc.
    - A package named com.myproject.tasks, however, would contain different types that are all tasks: TakeOutGarbageTask, DoTheDishesTask, etc.
- Example of good package naming: **info.magnolia.config.converters**

Interface and concrete class

Follow the suggestion from Stephen Colebourne's blog:

- **Foo** - The interface ultimately defines the concept, so it should have the best name.
- **AbstractFoo** - An abstract implementation intended to be used as the base of a hierarchy of classes.
- **BaseFoo** - An implementation intended to be used as the base of a hierarchy of classes, where the base class could be used on its own if necessary.
- **DefaultFoo** - A "default" implementation that would be appropriate for the majority of typical use cases.
- **SimpleFoo** - A "simple" implementation with no unexpected functionality, perhaps as an example or as a mock. A simple POJO would be a good "simple" implementation.
- **{Descriptive}Foo** - Other implementations should describe what makes them unique.

Enum

- In Java, all *enums* are actually classes, and should follow the same naming pattern – **upper camel case.**
- Instances of the enum are constants, and should follow the same naming convention as for final variables – **upper case,** and an underscore as the separator.
- Exception: lowercase/camelCase enum members are perfectly acceptable for 2 reasons:
    - If they are used in configuration (tree), where an ALL_CAPITAL value would look out of place

- If they implement complex types, whereas "regular" constants are usually native types or Strings. It's not unusual to do stuff like `layout.doLayout(someComponent)`, where `layout` is actually one of the values of `MyLayoutEnum`. It reads nicer without the capitals and underscores.

Test code

- Use "**/IT*.java", "**/*IT.java", "**/*ITCase.java" pattern for integration test classes (from maven-failsafe-plugin)
- Use "**/Test*.java", "**/*Test.java", "**/*Tests.java", "**/*TestCase.java" pattern for unit test classes (from maven-surefire-plugin)
- Apply Roy Osherove's naming standard for the name of test case: [**m ethodName_stateUnderTest_expectedBehavior**]
  - Examples:
    - isAdult_ageLessThan18_false
    - withdrawMoney_invalidAccount_exceptionThrown
    - admitStudent_missingMandatoryFields_failToAdmit

If-statements

- Single-line if-statements: we always use curly braces
- Unnecessary else-branch: if the else part is not necessary (e.g. because if-branch always returns or throws an exception), we omit it

Overrides

For optimal refactoring support, we use the *@Override* for all methods that override methods declared in superclasses/implemented interfaces

- Hint: you might want to adapt compiler settings of your IDE to show Errors on missing @Override's (Both Eclipse and IntelliJ IDEA provide that option)

SuppressWarnings

- Never suppress all kinds of warning / everything.
- Don't use @SuppressWarnings("deprecation")
  - Using deprecated code is smelly - we want to be aware of places where we use old stuff
  - Try to replace the deprecated code with its replacement
- @SuppressWarnings is usually used for unchecked, serial cases
- We try to avoid using @SuppressWarnings
  - If it's required, make sure you use it for the smallest scope - e.g. don't add on method level when you only need it for one instruction
    - Example:

```
public Object getReferenceDataValueForId(final ReferenceDataTypes type, final String id,
final String keyToGet) {
    @SuppressWarnings("unchecked")
    final Map<String, Object> referenceData = restTemplate.getForObject(systemConfig.
getReferenceDataLocation(), Map.class, type.name());
    @SuppressWarnings("unchecked")
    final List<Map<String, Object>> values = (List<Map<String, Object>>) referenceData.get
("values");
    final Map<String, Object> types = flattenToMap(values, keyToGet);

    if (types.containsKey(id)) {
        return types.get(id);
    }

    return null;
}
```

serialVersionUID's

We don't explicitly set serialVersionUID's as we don't want to decide ourselves when these IDs should be updated. We rely on the JVM doing that for us (Be aware that this ID can potentially differ from JVM to JVM). If for any (valid) reason, such an ID would be needed for a given class, the reason needs to be documented in the code. Failing that, there's a good chance that the field will be removed by some zealous fellow developer.

$NON-NLS-1$

If you come across a *//$NON-NLS-1$*, please just delete it - that's legacy stuff created by Eclipse - we don't want to keep that!

TODOs & FIXMEs

- These are used for minor issues or hints. As it may be handy to ask the author for details and we don't want to spend too much time on querying the SCM
- It's now strongly recommended to add the author's name:
    - Example: *TODO **Eric H**. - tell the world we'll now always add authors to the TODOs and FIXMEs*

Eclipse setup

Please see Eclipse setup page and attached formatter settings (Magnolia.epf).

Intellij setup

Please see Intellij IDEA page and attached formatter settings (Magnolia-IntelliJ-CodeStyle.xml).

Lombok

We use Lombok sparingly in our own *implementation details*; we try to avoid abusing it for publicly visible code (configured classes).

- `@EqualsAndHashCode` are mostly fine (don't forget *callSuper=true* if extending another class!).
- `@Getter` / `@Setter` should be avoided in classes which are not plain POJOs (usually configured classes). If you need to use it, do consider correct access level.
- `@SneakyThrows` should be avoided in best effort.

Atomic-refs / concurrency in general

- Use local variables => Don't need to worry about synchronization
- Prefer immutable classes => Don't need to worry about their state
- Minimize locking scope => Locking greatly affects performance
- Prefer thread pool executors to work with threads => Built-in, well-tested code for managing threads; why to write your own?
- Prefer synchronization utility over wait/notify => Much easier to implement synchronization than wait/notify
- Prefer concurrent collections over synchronized collections => For performance reason
- Avoid using static variables => Static variables share values among objects => Causing nightmare in concurrency

Empty lines

One blank line should be used :

- Between sections of a source file
- Between methods
- Between the local variables in a method and its first statement
- Between logical sections inside a method to improve readability
- Between static and non-static import

Final variables

- Define constant variables : use **static final** keywords
- Final variables : it should be used as default keyword, unless you have a clear indication that its value is going to be changed later.

Final keyword for variable prevents re-assign, implicitly indicates that the value once assigned can not be changed (only true for primitive type), so developers don't have to track the state/value of the variable, instead, concentrate on other things. It also improves readability.

Example:

```
        protected Node getOrCreate(String tagName) throws RepositoryException {

        tagName = nodeNameHelper.getValidatedName(tagName);

        final Session session = getTagsSession();

        final Node parentNode = NodeUtil.createPath(session.getRootNode(), getParentPath(tagName), NodeTypes.
Folder.NAME, false);

        if (parentNode.hasNode(tagName)) {

            return parentNode.getNode(tagName);

        } else {

            final Node node = parentNode.addNode(tagName, TAG_NODE_TYPE);
```

```
            session.save();

            return node;

        }
        }
```

Functional-style idioms / Stream formatting

- Use Java Stream API (introduced in Java 8) whenever possible to promote fluent API design
    - Streams are preferred over Guava transforms
- Stream code are formatted with at most one operation per line, after the `.stream()` initiator
    - `Import static` all of the standard stream related methods
    - Prefer method references to lambdas
    - Use `IntStream`, `LongStream` and `DoubleStream` when working with primitive types

```
// BAD FORMATTING:
strings.stream().filter(s -> s.length() > 2).sorted()
            .map(s -> s.substring(0, 2)).collect(Collectors.toList());

// GOOD FORMATTING:
strings.stream()
       .filter(s -> s.length() > 2)
       .sorted()
       .map(s -> s.substring(0, 2))
       .collect(Collectors.toList());
```

Null vs Optional

Prefer Optional to Null, except for the following cases:

- Local variables
- A rguments to private methods (`Optional` is a good return type, but not an argument type)
- Performance critical code blocks

`Optional` works well with the `Stream` API for chaining action. It leads to more explicit APIs saying that a (return) value may be present or not.

Example:

```
@Override
public Optional<? extends Form> fetch(Form.Id id) {
    return Optional.ofNullable(formMap.get(id));
}
```

General QA, what do we look for

In Magnolia development process, we have (not only) at least the following steps/process to ensure code quality:

- Code review
- Unit tests
- UI tests
- Integration tests
- Version handler tests

These tests are automatically checked on our Jenkins server. When code review and automated tests passed, manual tests must be executed before integration (pre-integration test aka piQA). After piQA is successfully completed and code is merged, integration tests must be executed manually once again to ensure quality.

In all those automated and manual tests : all edge cases, error cases, happy cases... must be covered.

Log statements, arguments, messaging

- Should apply a pattern that shows the exact date and timezone, and other important information
    - Suggested pattern: **%d{"yyyy.MM.dd HH:mm:ss z"} %-5level [%thread][%logger{0}] %m%n**
- Each logging statement should contain both data and description, should log the input arguments of method and its return value and the context, especially when communicating with external system → very useful for later debugging purpose
- Should use Java supplier to provide data for log statement to avoid unnecessary calculation of log statement
- Should have no or little impact on the application's behavior (like causing exception in log statement), avoid excessive logging that can slow down performance (utilize appropriate log level)

CSS style

Apply BEM methodology

- Block: Standalone entity that is meaningful on its own
- Element: A part of a block that has no standalone meaning and is semantically tied to its block.
- Modifier: A flag on a block or element. Use them to change appearance or behavior.

Syntax:

```
.block__element--modifier {}
```

Example:

```
.form {
  position: relative;
  .form__button {
     height: 20px;
     width: 80px;
     .form__button--red {
       color: red;
     }
     .form__button--green {
       color: green;
     }
   }
}
```

Deprecation

- Use both @Deprecated annotation and the @deprecated JavaDoc tag. The @deprecated JavaDoc tag is used for documentation purposes. The @Deprecated annotation instructs the compiler that the method is deprecated.
- Remember to explain:
    - **Why** is this method no longer recommended. What problems arise when using it. Provide a link to the discussion on the matter if any (A J IRA ticket would help).
    - **When** it will be removed. (let your users know how much they can still rely on this method if they decide to stick to the old way)
    - Provide a **solution** or link to the method you recommend {@link #setPurchasePrice()}
- Example:

```
/**
 * @param basePrice
 *
 * @deprecated  reason this method is deprecated <br/>
 *              {will be removed in next version} <br/>
 *              use {@link #setPurchasePrice()} instead like this:
 *
 * <blockquote>
 *   <pre>
 *     getProduct().setPurchasePrice(200)
 *   </pre>
 * </blockquote>
```

```
 *
 */
@Deprecated
public void setBaseprice(int basePrice) {
}
```

Javadocs

- Don't clutter your code with **Javadoc** comments that have absolutely no value and just duplicates what your code already say
    - The variable name and its type say it all.
    - The method name has clearly indicated what it does
- Provide Javadoc for code with:
    - Succinct method description
    - @param tag for parameter that needs clarification for its usage
    - @throws tag that tells in what condition the exception is thrown
    - Use {@link JavaType} to reference to another JavaType if it is used in our code

```
/**
* Does something valuable for your business.
* @param amountOfMoney - Money that is required to perform some action
* @param {@link Finance} - Important object for financial decision
* @throws FinancialException when something is wrong with the business
*/
public void doBusiness(long amountOfMoney, Finance object) throws FinancialException {
        // TODO
}
```

- Class level Javadoc is often the most useful one

Miscellaneous

Arrays#asList vs. Collections#singletonList

- Arrays.asList(something) is a fixed size List which throws UnsupportedOperationException for adding, removing elements although you can set an element for a particular index
- Collections.singletonList(something) will always return a list that has capacity of 1 and any changes made to the List will result in UnsupportedOperationException
- Prefer Collections.singletonList(something) if we want an immutable list with one element, or use Arrays.asList(something) if we want a fixed-size non-structural change list

Magnolia specific

General UI development guidelines

- How we do custom dialogs, fields, custom definitions at all ?
- How we do custom apps

ModuleVersionHandler test

Writing or doing tests for MVH (Module Version Handler) must ensure all version upgrading tasks and fresh install state relevant tasks are triggered.

YAML & Magnolia definitions

YAML is a highly-prominent part of Magnolia configuration; we recommend the following rules throughout YAML definition:

- Put the `class` property (or `fieldType`) before any other specific property
    - Exception: names within sequences
- Indent sequence values (so they don't appear to be on same level as their parent)

```
class: info.magnolia.foo.MySmartDefinition
components:
  - name: First component
```

```
  class: info.magnolia.foo.MySmartComponent
  description: Some short description of the first component
- name: Second ...
```