AMA Project Java Coding Standards

# Introduction

The purpose of this document is to convey the Java coding standards to developers in order to have a more consistent, familiar, maintainable, and extensible code base.

This document includes section titles that match those of [Google’s Java Style Guide](https://google.github.io/styleguide/javaguide.html); deviations and additions are documented herein. Other sections have also been added.

# Source File Basics

[Google Java Style: Source File Basics](https://google.github.io/styleguide/javaguide.html#s2-source-file-basics) documents good practices, which, unless a reason surfaces not to comply, this project should comply with.

# Source File Structure

Abide by [Google Java Style: Source File Structure](https://google.github.io/styleguide/javaguide.html#s3-source-file-structure) rules.

# Formatting

This project has adopted RSI Adaption 01, which is RSI’s adaption of [Google Java Style: Formatting](https://google.github.io/styleguide/javaguide.html#s4-formatting).

The Eclipse project is pre-configured with this code style formatting profile, as well as to reformat the code upon save. This may be confirmed by reviewing the associated Eclipse preferences:

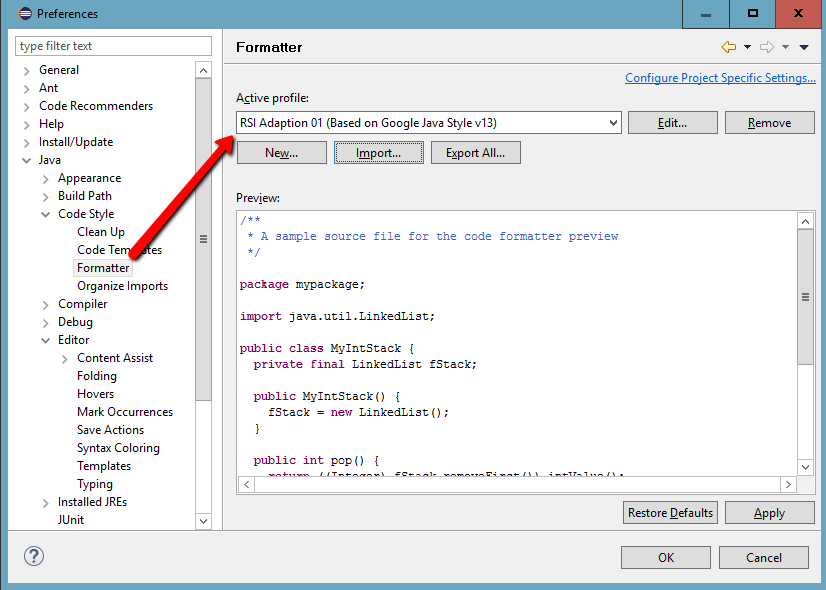
1. Java 🡪 Code Style 🡪 Formatting. See the first image, below.
2. Java 🡪 Editor 🡪 Save Actions. See the second image, below.

Optionally, one may limit these preferences to this project.

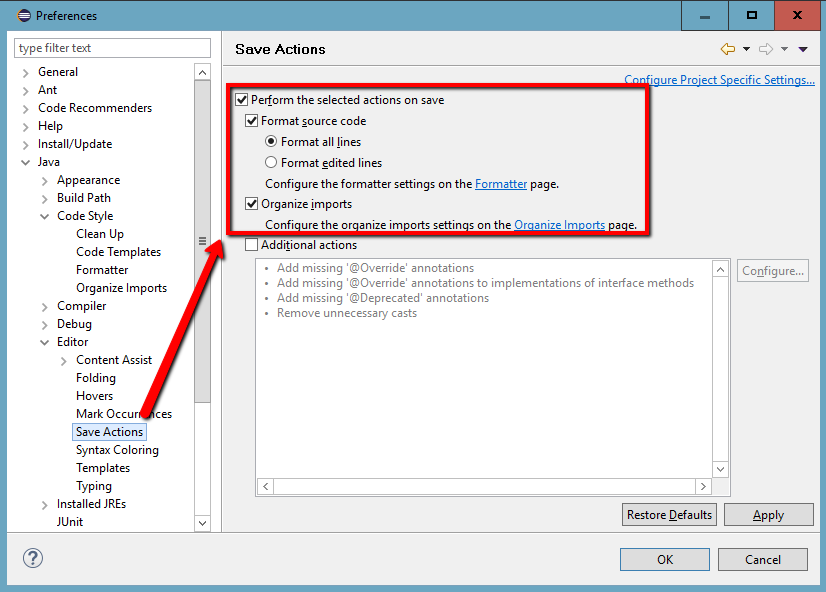
Once configured, the specific formatting settings may be reviewed within Eclipse.

A copy of the profile may be found alongside the build scripts, as eclipse-rsi-adaption-01-google-java-style.xml; however, one may wish to export the project’s current settings to ensure all subsequent variations are accounted for.

**Figure 1: Code Style Formatter Profile**



**Figure 2: Editor Save Actions**



# Naming conventions

The existing Java code aligns well with [Google Java Style: Naming](https://google.github.io/styleguide/javaguide.html#s5-naming).

# Javadoc

Developers are to strive to provide insightful comments that add to the value of the generated documentation. Omission of Javadoc warnings doesn’t mean the Javadoc will be helpful. GAO needs this documentation to help current and future developers learn, maintain, and extend the implementation. The amount of time invested into the Javadoc needs to be a balancing act.

Base direction may be obtained from [Google Java Style: Javadoc](https://google.github.io/styleguide/javaguide.html#s7-javadoc), specifically including where Javadoc should be present.

Noted deviation(s):

1. The selected code style formatter profile formats paragraphs slightly differently than Google’s Javadoc guidance.

The project’s [summary fragments](https://google.github.io/styleguide/javaguide.html#s7.2-summary-fragment) need to become better, and more aligned with Google’s guidance.

# Programming Practices

While brief, [Google Java Style: Programming Practices](https://google.github.io/styleguide/javaguide.html#s6-programming-practices) applies.

TODO: code against interfaces, specifically ContentAssemblyNodeContainer.

TODO: add more

TODO: include best practices herein?

TODO: sort through this…

**Evolving best practices is an important subject to cover.  Examples:**

1. Most performance issues require changes to custom code, and or a configuration change.  Fewer require product enhancements.  Rarely are more system system resources the correct answer.
2. Consolidate RSuite API calls, where you can, such as when setting multiple pieces of LMD on the same object.
3. Embrace MO advisors (global and local)
4. ContentAssemblyNodeContainer is your friend
5. Continue patterns present in RSuite's Java API
6. [Custom RSuite service pattern](https://rsuitecms.zendesk.com/entries/77543413-Pattern-Custom-RSuite-Service)
7. Do not box custom functionality into one RSuite extension point.  Push that implementation into a custom RSuite service or static utility class that any RSuite extension point can reach.
8. Content assembly visitors
9. Operation Options/Result pattern (implemented on the WB, BSI, and USPTO projects)
10. Workflow fault tolerance: design to do no harm when interrupted and replayed from the beginning of any workflow action handler.
11. RXS: convenience functions vs. materialized view
12. Insulation from RSuite's data model

**Nuances of RSuite's APIs and feature set are also helpful for developers to be introduced to, and continue to pickup as they go.  Examples:**

1. CAs vs. CANodes
2. CA references
3. CA types (remember common code driven by CA type enums?)
4. Universal vs. contextual LMD
5. Versioned vs. non-versioned LMD
6. When is it appropriate to write custom XQuery, and what are the drawbacks?

# Design Patterns and Practices

Design pattern objectives:

1. Avoid solving the same problem multiple different ways in the same project.
2. Increase familiarity within and across projects.
3. Employ proven designs.

## Model after RSuite’s Java API

Custom RSuite Java code is to be model after RSuite’s Java API. One reason to do so is to provide consistency between the two for developers.

Examples:

1. RSuite has services and managers (synonymous). Creating custom RSuite services help to continue this pattern, while also making the associated functionality available to more contexts. See the **Custom RSuite Services** and **Liberate Business Logic** sections for more details.
2. By default, method signatures should be as specific as possible (do NOT pass around ExecutionContext, e.g., pass User around instead). This aligns with the RSuite Java API, provides the method with access to RSuite’s services, and enables the method to execute as the specified user. Executing as a non-system user enables RSuite to enforce content and action security.

## Custom RSuite Services

A documented pattern exists for [custom RSuite services](https://rsuitecms.zendesk.com/entries/77543413-Pattern-Custom-RSuite-Service). Since published, it has evolved. The TestingService is a custom RSuite service.

A key benefit to this pattern is providing related functionality to any RSuite extension point. Moreover, any Java code in the same plugin can use these services. This complies with the best practice of *not* implementing business logic *directly* within an RSuite extension point, where it can’t easily be accessed from another context.

Typically web services expose the subset of a custom RSuite service’s functionality required outside the plugin.

The above could also be accomplished with non-singleton utility classes and methods (e.g., ContainerUtils). It’s not always clear when to go with which. Reasons to select a service include having to initialize it, perform service-level operations when the plugin is unloaded, and sharing memory, such as the service’s configuration.

## Liberate Business Logic

Do not box custom functionality into one RSuite extension point.  Push that implementation into a custom RSuite service or static method that any Java code in the same plugin can reach it.

By taking a little extra time up front, zero refactoring will be required should this be needed in another context. When this is how all of a project’s higher-level operations are implemented, it won’t take long for this practice to pay off.

## Separation of Concerns

The OperationOptions and OperationResult classes were introduced to standardize how RSuite extension points use custom operations, features, and the like. This has since broadened to include any of the plugin’s Java code that executes a custom operation. This practice applies when the operation is provided by a custom RSuite service or static method.

The consumer is responsible for providing an instance of OperationOptions, which may be specialized for the given operation. The provider is responsible for returning an instance of OperationResult. Likewise, this may be specialized for the operation. Of the two, more value is being derived from OperationResult.

The consumer needs to know very little about the provider: just what the provider allows to be controlled by the consumer. Once the operation is complete, the consumer is responsible for processing the OperationResult. Especially in the web service and workflow contexts, much of this processing is agnostic of the operation. For instance, all web services are able to give an instance of OperationResult to BaseWebService#getWebServiceResponse(). That one method implements the logic of what to do when the operation experienced failures or warnings. The equivalent may be implemented by a base workflow action handler.

When an instance of OperationResult is in play, exceptions should be added as failures rather than throwing the exception. This does not alleviate consumers from exception handing. Below is an example of adding a failure to an instance of OperationResult rather than throwing an exception. In this case, the operation is not allowed to continue if the user isn’t an author or administrator.

## Custom RSuite Web Services

The pattern for custom web services may be summarized as:

1. Business logic is not implemented in the web service.
2. Get relevant web service parameters to the custom operation. At times, this requires some interpretation, but business logic should be deferred to the custom operation.
3. Invoke the custom operation, passing in an instance of OperationOptions. The custom operation may be accessible via custom RSuite service or static method.
4. Interpret the operation’s OperationResult and return a web service response, putting as much as possible on BaseWebService#getWebServiceResponse().
5. Wrap the entire body of execute() in a try/catch block.

## Utility Classes and Methods

|  |  |
| --- | --- |
| *Related Sections* |  |

Defining logic once and leveraging that definition as many times as necessary is absolutely critical. Organizing the code such that developers can find and become aware of what is available can be equally important, or at least prevent the former from being possible. To that end, there are two main utility packages and a reliance on JavaDoc.

The org.ama\_assn.rsuite.domain package includes utility classes that are specific to this project. Sub-packages could be employed to better isolate the domain-specific utility classes.

The org.ama\_assn.rsuite.utils package is intended to only include utility classes that are not specific to this project. In other words, they could be deployed as an RSuite Java library.

Not all utility methods need to accept an instance of OperationOptions and return an instance of OperationResult. It matters what the method does. For instance, it doesn’t make sense for ContainerUtils.getLayeredMetadataValue() but does for ContainerUtils.deleteContainerAndReferencedContent(). Those that do are one in the same with aforementioned ones that serve as an alternative to methods found in custom RSuite services. Some methods should *accept* an instance of OperationResult such that they can consistently provide info-level messages, warnings, and failures.

Avoid using static methods at all costs as that makes writing unit tests especially harder.

Make methods as specific in terms of their parameters as possible.

## Container Visitors

Container visitors were introduced in RSuite 3.7, when it became possible for one container to reference another container. Upgrades to RSuite 3.7 were more painful because the logic of traversing the contents of a container was mixed with the logic of what to do with the container’s contents. Compounded, the traversal logic was duplicated throughout code bases.

Container visitors are now part of the RSuite Java API, and projects can and should introduce additional, purpose-specific ones, deployed in an RSuite Java library or RSuite plugin.