Scala Best Practices

Devopam Mittra

Document deals with the different contexts in which best practices have to be implemented

Contents

[License 2](#_Toc16591861)

[Environment 2](#_Toc16591862)

[IDE 2](#_Toc16591863)

[Compilation 2](#_Toc16591864)

[CICD 2](#_Toc16591865)

[Style 2](#_Toc16591866)

[Documentation 3](#_Toc16591867)

[Coding 3](#_Toc16591868)

# License

* Check each open source component marked for use in project for its license type, liability and fees if any before deploying with no exception
* Java JDK (if not OpenJDK) to be verified licensed before use.
* Validate each external plugin used in project for license implications

# Environment

## IDE

* Either of Eclipse or IntelliJ preferred (or similar feature capability ) due to level of support available in developer communities
* [SBT](https://www.scala-sbt.org/) or Maven build preferred
* Base Java version to be homogeneous for entire build cycle

## Compilation

* Base Java version to be homogeneous for entire build cycle and matching with deployment environment (production)
* Define scalac –X, -Y, -W options in project configuration from beginning. Important parameters for consideration out of the entire [options](https://docs.scala-lang.org/overviews/compiler-options/index.html):
  + -Xlint:nullary-unit
  + -Xlint:inaccessible
  + -Xlint:unused
  + -Xplugin-list
  + -Ywarn-dead-code
  + -Ywarn-octal-literal
  + -Ywarn-value-discard

## CICD

* SVN, Github/Gitlab (or in-house tool with similar capabilities) to be used.
* Code reconciliation to be done no later than 1 working day at each end-of-day and resolved before next working hours.
* Integrate with Jira (or similar capability tool) for bug/feature testing lifecycle management.
* Define major/minor version/build nomenclature upfront and follow strictly.

# Style

* Refer to official Scala-style [guide](https://docs.scala-lang.org/style/index.html) . Use IDE plugin to integrate in project itself. Adhere to standards from inception instead of taking it up as a ‘beautification’ burden towards completion.
* Focus on Naming convention, Indentation style, Comments, Annotations and Grammar in particular
* Fix a suitable linesize and stick to it.

# Documentation

* [Scaladoc](https://docs.scala-lang.org/overviews/scaladoc/overview.html) plugin is available in Eclipse or IntelliJ. Scaladoc can be generated command line, through IDE or through SBT with same result.
* A good commenting style adhering to scaladoc best practices for Packages, Classes, Traits, Objects, annotations and underlying methods and test cases is a necessity for generation of effective scaladoc.
* Scala is dense. Scala is functional programming. These two factors make it imperative that complex business rules implemented in program body be documented in detail else programmer risks losing the context him/herself.
* Do a proof reading of scaladoc before delivering to avoid silly spelling mistakes , grammatically incorrect sentences and rough notes)

# Coding

* Import only what is needed and not the entire package (selective import)
* Never hardcode login credentials to any environment.
* Use a property class to read logon details (host, user and password/token) from a properties file (ConfigFactory in typesafe)
* Configuration should be read automatically based on host environment i.e. dev,test,maint or prod (have an ENV constant for each environment, based on which the corresponding application.conf or system constants shall be looked up automatically
* Constants to be defined at a package (global) or Class level.
* Avoid throwing exceptions to the extreme maximum possible extent. ‘Handle’ the exception condition using suitable try-catch mechanisms and process the exception condition instead.
* Plugin/Build reusable ‘Generic’ [Utility](https://github.com/twitter/util) classes for common functions in a standalone package e.g.:
  + I/O operations (file open, close, copy, with proper handling of exceptions like locks, not found, directory not accessible)
  + DB Connections
  + Error logging (log format: business area-> package-> class , time of error, description, additional data / attributes, logged by)
* Prefer immutable collections
* Use profiler (opensource lightweight like VisualVM or integrated in IDE) to check for identifying performance bottlenecks.
* Understand the uses and difference between def, val and var before you touch code!!
* Do NULL handling suitably always for all data types while processing data (strings, date/timestamp, and numbers to avoid NaN issues).
* Use Option instead of Case as much feasible. It helps with combined usage of flat/maps.
* Use constructors effectively for instantiation and run time (declare default values for args).
* Annotate @tailrec to all recursive functions to check optimization.
* Focus on effective use of concurrency and asynchronous operations as much possible.
* Go through extensive and comprehensive checklists here : [[1](https://github.com/alexandru/scala-best-practices)] & [[2](https://nrinaudo.github.io/scala-best-practices/)]