Lift Microservice Template – Developer’s Guide

# Goal

Provide a guide for better code, architecture, and project quality.

# Scope

* Standards & conventions
  + Style guide
  + Best practices
* File structure
* Testing
* Git

# Assumptions

* Understanding of Scala syntax; basic working and coding
* Basic understanding and usage of Git
* Sbt project folder structure requirement

# Scala Style Guide

Read up here: <http://docs.scala-lang.org/style/>

* Remember, the bottom line is to make the code readable and less error-prone.

# Scala Best Practice

Read up here: <https://github.com/alexandru/scala-best-practices>

Important points:

* Scala code only. No Java code, unless dealing with Java dependencies, which you should really think long and hard before using one.
* NEVER use nulls. Use Option or Lift’s Box or other monads instead.
* Avoid mutability. Use mutable variables only when there’s no other choice. Even then, keep it minimal and contained within a small scope.
* Design for later change/expansion in mind.
* Minimize exceptions and try/catchs. Use appropriate container/abstraction. Throw exceptions only on unexpected occasion.
* Minimize use of implicits. Use only when there’s a good reason.
* Specify return type of methods as much as possible. Leave little to compiler type inference.
* Take time to think of naming meaningful names.

# Package organization

* Package name should reflect its content and intent.
* If there is clear boundary between groups of classes within a package, separate them.
* Conversely, if items from two packages are closely coupled, it may be best to join them.
* Prefer flat package structure over deeply nested structure.
* That said, nest the package when appropriate. Use common sense.
* Declare as nested package scope only when being used very commonly in the package. If unsure, declare as unnested. I.e.,

// default to this

package com.dotography.magic.abracadabra

// use this only when members of `magic` is used very often.

package com.dotography.magic

package abracadabra

# File/Folder Structure and naming

* Match the folder structure to the package structure. Folder names should be the same as the package name. That said, if minor alteration to folder structure makes for better organization, do it.
* File name should follow the topical class/object in the code. Minimize the topic to a single class per file if possible.
* If declaring Arithmetic Data Type (ADT), name the file after the superclass.
* Sometimes it makes sense to put several cohesive classes together. In that case, the filename should start with a lowercase.
* Remember, the point of the file name is to suggest its contents. Better naming results in faster navigation and painless code browsing.
* Package object file should reside within the folder of the package, with the file name package.scala. E.g:

// this should be located at

// src/main/scala/com/dotography/audio/package.scala

package com.dotography

package object audio {

def defaultBps = ...

val defaultEncoder = ...

}

# Testing

* Use Specs2
* Keep ScalaCheck usage to minimum as it increases test time by a lot. Separate into separate test class if possible.
* Minimise mocking and stubbing. If you require their uses often, consider refactoring the code in question.
* Prefer parallel test execution. However, tests with side-effect, e.g. database, would most likely require sequential execution.
* Use Scoverage. Strive for 100% coverage.
* Test Future with .await method, see here: <http://stackoverflow.com/questions/27750244/how-to-test-methods-that-return-future>
* Use aka for better error report message; e.g.

result.size must\_== expectedCount  
// will report something like "0 is not equal to 2" on error  
  
result.size aka "Expected result count" must\_== expectedCount  
// will report "Expected result count is not equal to 2" which is easier to understand

# Git

* Write meaningful commit notes
* Use Git-flow practice. Standard branch naming convention (feature, develop, release, hotfix, master). See here for explanation on Git-flow: <http://nvie.com/posts/a-successful-git-branching-model>
* Using SourceTree application to easily manage Git-flow is recommended.
* Keep the features small. Break large features into small ones.
* Delete the feature branch when completed AND verified.
* Branch within feature if necessary; merging them back before completing recommended.
* Tag the version on master
* Try to push only after the code is stable, i.e. the code compiles and the tests pass. Run a test-quick to double check.