overview
- the notes in this document apply for all areas
- for additional notes on each area, have a look at the specific document:
- Java: baeldung - for authors - 6 - Java
- Ops: baeldung - for authors - 6 - Ops
- Sql: baeldung - for authors - 6 - SQL

I. basics

getting started - forking the repository
- most of our articles include code examples (or pseudocode/shell commands)
- code examples should also be included in our repository
- we have a dedicated GitHub repository for each area (if it uses code) - have a look at the area-specific doc linked above to find the exact repository
- you won't be able to push directly to our repository, so you have to **fork** it, then work on your fork of the repository
- if you're not familiar with this process, here's a general guide written by one of our editors on his personal blog: How to Fork an Open Source Project

- this video gives an overview of our main repository: *the tutorials project and modules link*

- if you don't have a lot of experience with git, have a look at the resources here as well: for authors - getting started with Git

getting started - IDE setup

- our standard Eclipse formatter is here: link
- our standard IntelliJ formatter is here: link
- make sure you import this in your IDE as you're working, and always use it
- of course, make sure to **format your code before committing it**

- for each XML file, you can open it in GitHub, then click "Raw" to open the raw file, then right click -> Save as to download the XML file without cloning the repository with the formatter files

- for Eclipse, you have to make additional manual changes:
- go to Window -> Preferences -> Java -> Editor -> Save Actions and check the options:
    - Format source code -> Format edited lines
    - Organize imports
    - Additional Actions -> Configure -> Code Style -> Use Blocks in if/while/for/do statements: Always

- additionally, we want to set up a 4-space indentation for XML, HTML, CSS, and any other types of files that we might be handling
- and configure any “deprecated API usage” and any “unused import” to be marked as “Error” by the IDE (sometimes the IDE requires cleaning the Projects first to detect these occurrences):

- we also want to enable the “Turn formatter on/off with markers in code comments” option in the IDE, using “@formatter:off” and “@formatter:on” respectively
- if you want to preserve the manual formatting of a block of code, mark this as excluded from formatting using the tags as shown here: exclude_part_of_code

- naming conventions should follow the Google Java Style Guide:
https://google.github.io/styleguide/javaguide.html#s5-naming

package structure

- each article should have **its own package**, in the module you're using
- for example, let's say the package structure of the module is: *com.baeldung*
- and, let's say your article is about *Defining a DataSource in Spring Boot* (as an example)
- then - your article code should be in: *com.baeldung.boot.data*
- or: *com.baeldung.boot.datasource*

- the main aspect here is - it has its own subpackage, separate from everything else
- that means that it shouldn't use/re-use any code from the other packages either

- **note**: try to make the package readable; so, for example - instead of a single package like: *convertiteratortolist*, it would be better to go with: *convert.iteratortolist*

- relevant video: *basics - existing module, separate package*

about tests

- your code should include tests as needed to verify the behavior added
- all tests should follow the standard naming convention:
  - *FooUnitTest.java / FooUnitTest.kt*
  - *FooIntegrationTest.java* / FooIntegrationTest.k
  - *FooLiveTest.java* / FooLiveTest.kt // tests that require a running component
- that's because unit tests run in the standard build; integration and live tests do not
- the Java and Kotlin builds will fail if a test doesn't follow this convention
- if your tests require accessing a running component (eg: a MySql database), you can rename it to **LiveTest* so it doesn't run as part of the build, which would cause it to fail

- integration tests that require a port allocation should use a random port if possible

- make sure that the resources are closed and any ports released after the test

- tests should not be dependent on each other (enforce the order if really necessary)

- new tests should use JUnit 5 over JUnit 4

- JUnit tests should use package private access (not public, unless it's necessary)

tests - names

- test should follow the **BDD** convention: *givenX_whenY_thenZ*
- the *given* part is optional, but the other two are not
- example: *whenSendingAPost_thenCorrectStatusCode*

- **use domain-focused names** for the tests, rather than code-focused names
- for example, instead of:

  givenHashMap_whenUsingGetOrDefault_thenDefaultReturned
- use:

  givenListOfTasks_whenGetByNonexistentCode_thenDefaultTaskReturned

- this makes the tests easier to read and understand

- also, the delimiter (underline) should only be used between these sections, and not anywhere else
- **for example - this isn't correct**: *whenSomething_andSomethingElse_thenSuccessfull*

- add a new line between the *given*, *when,* and *then* sections of a test

II. advanced

adding the code

- for the most part, you should **use one of the existing modules** we already have
- if you're not sure which module fits - propose a few to your editor and they'll help you pick

- **a module shouldn't have more than 8 articles pointing to it**
- to count the number of articles in a module, go to **the report**: baeldung - content - github modules and backlinks and search for the module in the first column
- the second column contains the list of articles already in that module; the third column shows the total count; if this is >=8, then this means you cannot use this module for your article

- so, if you're thinking of using a module that's already at (or over) the limit - again - talk to your editor about it

- some modules will be in a **sequence** like: libraries-2, libraries-3, libraries-4 etc
- if there is more than 1 module in the sequence, **skip to the last module** (eg: libraries-4) even if other modules before it have fewer articles than the limit; this is because the series are organized by decreasing traffic of articles (or a few other reasons). But if there's a later module in the series already, it means that the previous ones can't be used

when to create a new module

- if you think no module fits - and you need a new one - definitely talk to your editor first
- as a general rule - we should try to introduce a new module if you think that can be used by a few articles (existing or upcoming)
- that means that - if you think the new module you're thinking about has no chance of ever being used by another article (because it's too specific) - then we should try to avoid introducing it (if we can)

new module name

- make sure to choose a good name for the new module
- this can't be too specific, so that no other articles can be added in the future, but not too generic either, to the point where it doesn't say anything about the module
- in the past, we've used the approach of just adding suffixes like -2, -3 to modules if there's no other good name
- however, this should be a last resort, to avoid getting to the point of having modules like libraries-7 or core-kotlin-5
- instead, let's try to narrow the scope of the module, eg: *libraries-concurrency*

how to create a new module

- if the new module is a sub-module, then this should be added to the pom of their parent, instead of the main pom
- for ex, the module core-java-optional is added to the pom of the *core-java-modules* parent

- if the new module is at the root level, make sure you add that module in all the right profiles:
  - *default* and *integration*

- you can read more about the repository and profile structure in the main README.md

Maven info

- the typical **order of the main blocks** should be: *parent, dependencies, build, properties*
- detailed order: *modelVersion, groupId, artifactId, version, packaging, name, description, parent, modules, dependencyManagement, dependencies, build, properties*

- the *artifactId* should be lowercase
- the *name* in the pom is optional, but if we have it, it should be the same as the artifactId
- *finalName* should be same as module name (after doing a site:www.baeldung.com search to ensure *finalName* isn't mentioned anywhere

- as much as possible, all modules should use the parents (in most cases, it should be clear which parent is the right one)

- wherever possible, the project should **use the latest versions** of any dependencies
- the dependencies should define versions using properties (not hardcoded)
- test dependencies should be marked as *<scope>test</scope>*

building the code

- always build your code before opening a PR:
mvn clean install

- to build a module by disabling the incremental build, use the command:
mvn clean install -Dgib.enabled=false

the Pull Request (PR)

- when your code is ready, create a Pull Request (PR) from your fork to our repository - for your editor to review and merge
- make sure to only include the relevant changes that are necessary for your article and nothing else:
    - no formatting changes
    - no IDE artifacts (Eclipse, IntelliJ stuff)
    - basically, nothing that isn't critical to the article

- **note 1:** try to keep the PR **under 10 commits** so that we can merge them quickly
- **note 2:** all PRs should contain the JIRA issue number in the title
- **note 3:** you don't need to add the article link should to the readme, as you don't have the final URL yet

Jenkins and PRs

- when you open a PR or add a new commit, the Jenkins build will start
- the build has to be green; otherwise the PR cannot be merged
- check the status of the build once it finishes; this will have a link to the output of the job
- if you want to re-run the CI build for a PR, you can comment *"retest this please"* on the PR

- **note**: try to **keep the nr of commits to a minimum**, and only create the PR once you're ready; otherwise every new commit will trigger a new build, which delays the queue

logging

- **logging** for the code sample should be: slf4j with logback (that's how most modules are already using)
- use logs instead of sysout unless the article is about console output

- tutorials repository - non-Spring modules:
    - these will use the file *logback-config-global.xml* at the root of the repository by default
    - if you want to use a custom file in a module, add this in the resources folder with the name *logback.xml*, and then add the property: *<logback.configurationFileName>logback.xml</logback.configurationFileName>* to that module's pom.xml

- tutorials repository - Spring modules:
    - these must have their own *logback.xml* file in the resources folder; the global file doesn't work for Spring modules; if you add a new module, you can either copy this from another Spring module or from the global file at the root level

code - assertions

- when creating complex assertions, it's a good idea to use assertj to preserve clarity
- if the assertions are trivial, it's fine to stick to JUnit ones

various small notes

- working with file locations - we usually use the **Linux style** as a reference, and also - whenever possible - we should use files that are local to the project (src/main/resources), and not fully external files
- code should be as **simple** as possible (as long as we're still able to explain the concept)
- for example, when you work with a POJO (Entity, DTO, Resource, etc) - define a simple class: *id* and *name* - if nothing more is needed
- use package-scope access modifier if possible
- prefer Java-based configurations over XML
- **variable names** should be relevant
- for example, use *student* instead of *s1* or *ex1*
- try to avoid using *Thread.sleep* calls in the code, unless necessary, as these may introduce delays in the build

articles that target 2 versions of a library

- the majority of articles will be focused on the latest version of a library
- however, there are (rare) cases when a previous version is still widely in use at the time of writing, especially if the new version is a recent release
- if so, the article should cover both versions, starting with the most recent one, followed by a section on the previous version; example: https://www.baeldung.com/junit-get-name-of-currently-executing-test
- if the 2 versions are not compatible, then the code should be placed in 2 separate modules
- if they are compatible, then the code can be placed in a module dedicated to the new version, which can also contain code for the old one

benchmarking

- if your article requires a performance comparison, make sure to use a benchmarking framework like the JMH for Java: Avoiding Benchmarking Pitfalls on the JVM
- naive benchmarking solutions do not provide reliable results

using lombok

- if the post is not specifically about Lombok, then let's avoid adding it to a project
- this is for consistency and simplicity across articles
- for getters/setters, you can just add the standard comment in the article instead

[optional] how much code?

- Baeldung targets medium to advanced developers and readers
- that means, we should assume the reader understands the basics and should focus - as much as possible - on the main topic of the article
- that also applies to code as well - we have several “intro” level articles that already show how to set up various types of projects, so we can link to these if we have to show some basic setup before reaching our main topic