Development Manual

Table of Contents

[Change Log 3](#_Toc9244638)

[Environment Setting 3](#_Toc9244639)

[Java 3](#_Toc9244640)

[Tomcat 3](#_Toc9244641)

[Maven 3](#_Toc9244642)

[IDE (Optional) 3](#_Toc9244643)

[Neo4j Community Server 3](#_Toc9244644)

[Code Structure 4](#_Toc9244645)

[Folder Structure 4](#_Toc9244646)

[NameSpace 4](#_Toc9244647)

[Logger 4](#_Toc9244648)

[Library required 5](#_Toc9244649)

[Deployment 5](#_Toc9244650)

[Java Style Guide 5](#_Toc9244651)

[CamelCase for types, camelCase for variables, UPPER\_SNAKE for constants 5](#_Toc9244652)

[No trailing whitespace 5](#_Toc9244653)

[Indent style 5](#_Toc9244654)

[We use the "one true brace style" (1TBS). Indent size is 2 columns. 5](#_Toc9244655)

[Continuation indent is 4 columns. Nested continuations may add 4 columns or 2 at each level. 5](#_Toc9244656)

[Method declaration continuations. 6](#_Toc9244657)

[Chained method calls 6](#_Toc9244658)

[Variable naming 6](#_Toc9244659)

[Extremely short variable names should be reserved for instances like loop indices. 6](#_Toc9244660)

[Include units in variable names 7](#_Toc9244661)

[Don't embed metadata in variable names 7](#_Toc9244662)

[Avoid embedding the field type in the field name. 7](#_Toc9244663)

[Avoid embedding scope 7](#_Toc9244664)

[Space pad operators and equals 7](#_Toc9244665)

[Documentation 8](#_Toc9244666)

[Documenting a class 8](#_Toc9244667)

[Documenting a method 8](#_Toc9244668)

[Import ordering 9](#_Toc9244669)

[Use annotations wisely 9](#_Toc9244670)

[@Nullable 9](#_Toc9244671)

[@VisibleForTesting 10](#_Toc9244672)

[Clean code 10](#_Toc9244673)

[*Disambiguate Favor readability* 10](#_Toc9244674)

[Remove dead code 10](#_Toc9244675)

[Delete unused code 10](#_Toc9244676)

[Use general types 11](#_Toc9244677)

[Avoid typecasting 11](#_Toc9244678)

[TODOs 11](#_Toc9244679)

[Leave TODOs early and often 11](#_Toc9244680)

[Leave no TODO unassigned 11](#_Toc9244681)

[Adopt TODOs 11](#_Toc9244682)

# Change Log

|  |  |  |
| --- | --- | --- |
| Date | Editor | Comment |
| 05/15/2019 | Yizhen Chen | Init Doc |
| 05/16/2019 | Yizhen Chen | Update Java version from Oracle Java to OpenJDK |
| 05/20/2019 | Yizhen Chen | Add deployment instructions |

# Environment Setting

## Java

Version: OpenJDK 11.0.2

Download link : <https://jdk.java.net/archive/>

## Tomcat

Version: Tomcat 9

Download link : <https://tomcat.apache.org/download-90.cgi>

## Maven

Version:3.1

Download link: [https://maven.apache.org/download.cgi#](https://maven.apache.org/download.cgi)

## IDE (Optional)

Spring Tool Suite (STS)

Version:3.9.8

Download link : <https://spring.io/tools3/sts/all>

[[1]](#footnote-1)Note: STS embedded with maven. It not require to install maven separately.

## Neo4j Community Server

Version: Neo4j Community Edition 3.5.5

Download link: <https://neo4j.com/download-center/#releases>

Command line to run server : ./bin/neo4j console

# Code Structure

## Folder Structure

ROOT

|-----src

|----main

|----test

|-----webContent

|----META-INF

|----WEB-INF

|----- doc

|----- pom.xml

|  |  |
| --- | --- |
| Src | Src is the folder where the project's source files are located. |
| main | function implementation |
| test | test script (junit) |
| webContent | WebContent folder is always considered as client side code in Java web projects. It contains like HTML, Css, JavaScript, jQuery, JSP pages etc. |
| META-INF | The META-INF folder is the home for the MANIFEST.MF file. This file contains metadata about the contents of the JAR. |
| Web-INF | The WEB-INF directory is part of the directory structure that defines a particular "Web application". The WEB\_INF directory contains resources pertaining to the Webapplication including a web.xml file. |
| Doc | Project’s documents |
| Pom.xml | A Project Object Model or POM is the fundamental unit of work in Maven.  It is an XML file that contains information about the project and configuration details used by Maven to build the project. It contains default values for most projects. |

## NameSpace

gov.nih.nci.icdc.\*

## Logger

ErrorHandler

# API endpoint design

## Version Control

There are two types of API, REST API and GRAPHQL API.

REST API’s end point will follows http://www.domain.com/{version}/rest/{params}

GRAPHQL API’s end point will follows [http://www.domain.com/{version}/graphql/{params}](http://www.domain.com/%7bversion%7d/graphql/%7bparams%7d)

{version} will be short as v1.v2.v3

REST API for testing :

Based on version : http://www.domain.com/v1/rest/ping

## List of APIs

API For Testing purpose:

GraphQL :

query {

findByName(name:"Keanu Reeves") {

born

}

}

EndPoint : <http://localhost:8080/ICDC/person>

Result Shows Below.

A screenshot of a cell phone

Description automatically generated

## API Documentation with Restocks

# Library required

TBD

# Deployment(05/22/2019)

1. Get Source Code

Create a directory : Mkdir tmp & cd tmp

Download source code :git clone <https://github.com/jonkiky/neo4j_for_testing.git>

1. Build Project

CD project directory: cd project\_name

Maven Build : mvn package

1. ReName the War file

CD compiled files: cd target

Rename : mv RESTFfullDemo-0.0.1 SNAPSHOT.war ./RESTFfullDemo.war

1. Deploy to the tomcat
   1. From GUI
      1. CD tomcat bin : cd \*\*\*\*/apache-tomcat-9.0.19/bin
      2. Start Tomcat : sudo ./startup.sh
      3. Browser to the link : <http://localhost:8080/manager/html>
      4. At War file to deploy section , choose file then click deploy.
      5. Browser to the link : [http://localhost:8080/ RESTFfullDemo](http://localhost:8080/manager/html)
      6. Stop tomcat: sudo ./shutdown.sh
   2. From Command line

TBD

# Java Style Guide

Source: <https://github.com/twitter/commons/blob/master/src/java/com/twitter/common/styleguide.md>

## CamelCase for types, camelCase for variables, UPPER\_SNAKE for constants

## No trailing whitespace

Trailing whitespace characters, while logically benign, add nothing to the program. However, they do serve to frustrate developers when using keyboard shortcuts to navigate code.

## Indent style

### We use the "one true brace style" ([1TBS](http://en.wikipedia.org/wiki/Indent_style#Variant:_1TBS)). Indent size is 2 columns.

:::java

// Good.

if (x < 0) {

negative(x);

} else {

nonnegative(x);

}

### Continuation indent is 4 columns. Nested continuations may add 4 columns or 2 at each level.

// Good.

// - Each component of the message is separate and self-contained.

// - Adding or removing a component of the message requires minimal reformatting.

throw new IllegalStateException("Failed to process"

+ " request " + request.getId()

+ " for user " + user.getId()

+ " query: '" + query.getText() + "'");

Don't break up a statement unnecessarily.

## Method declaration continuations.

:::java

// Sub-optimal since line breaks are arbitrary and only filling lines.

String downloadAnInternet(Internet internet, Tubes tubes,

Blogosphere blogs, Amount<Long, Data> bandwidth) {

tubes.download(internet);

...

}

// Preferred for easy scanning and extra column space.

public String downloadAnInternet(

Internet internet,

Tubes tubes,

Blogosphere blogs,

Amount<Long, Data> bandwidth) {

tubes.download(internet);

...

}

## Chained method calls

:::java

// Bad.

// - Line breaks are based on line length, not logic.

Iterable<Module> modules = ImmutableList.<Module>builder().add(new LifecycleModule())

.add(new AppLauncherModule()).addAll(application.getModules()).build();

// Good.

// - Method calls are isolated to a line.

// - The proper location for a new method call is unambiguous.

Iterable<Module> modules = ImmutableList.<Module>builder()

.add(new LifecycleModule())

.add(new AppLauncherModule())

.addAll(application.getModules())

.build();

## Variable naming

### Extremely short variable names should be reserved for instances like loop indices.

:::java

// Bad.

// - Field names give little insight into what fields are used for.

class User {

private final int a;

private final String m;

...

}

// Good.

class User {

private final int ageInYears;

private final String maidenName;

...

}

### Include units in variable names

:::java

// Bad.

long pollInterval;

int fileSize;

// Better.

// - Unit is built in to the type.

// - The field is easily adaptable between units, readability is high.

Amount<Long, Time> pollInterval;

Amount<Integer, Data> fileSize;

### Don't embed metadata in variable names

A variable name should describe the variable's purpose. Adding extra information like scope and type is generally a sign of a bad variable name.

### Avoid embedding the field type in the field name.

:::java

// Bad.

Map<Integer, User> idToUserMap;

String valueString;

// Good.

Map<Integer, User> usersById;

String value;

Avoid embedding scope information in a variable.

Hierarchy-based naming suggests that a class is too complex and should be broken apart.

:::java

// Bad.

String \_value;

String mValue;

// Good.

String value;

## Space pad operators and equals

:::java

// Bad.

// - This offers poor visual separation of operations.

int foo=a+b+1;

// Good.

int foo = a + b + 1;

Be explicit about operator precedence

Don't make your reader open the [spec](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html) to confirm, if you expect a specific operation ordering, make it obvious with parenthesis.

:::java

// Bad.

return a << 8 \* n + 1 | 0xFF;

// Good.

return (a << (8 \* n) + 1) | 0xFF;

It's even good to be really obvious.

:::java

if ((values != null) && (10 > values.size())) {

...

}

## Documentation

The more visible a piece of code is (and by extension - the farther away consumers might be), the more documentation is needed.

"I'm writing a report about..."

Your elementary school teacher was right - you should never start a statement this way. Likewise, you shouldn't write documentation this way.

:::java

// Bad.

/\*\*

\* This is a class that implements a cache. It does caching for you.

\*/

class Cache {

...

}

// Good.

/\*\*

\* A volatile storage for objects based on a key, which may be invalidated and discarded.

\*/

class Cache {

...

}

### Documenting a class

Documentation for a class may range from a single sentence to paragraphs with code examples. Documentation should serve to disambiguate any conceptual blanks in the API, and make it easier to quickly and correctly use your API. A thorough class doc usually has a one sentence summary and, if necessary, a more detailed explanation.

:::java

/\*\*

\* An RPC equivalent of a unix pipe tee. Any RPC sent to the tee input is guaranteed to have

\* been sent to both tee outputs before the call returns.

\*

\* @param <T> The type of the tee'd service.

\*/

public class RpcTee<T> {

...

}

### Documenting a method

A method doc should tell what the method does. Depending on the argument types, it may also be important to document input format.

:::java

// Bad.

// - The doc tells nothing that the method declaration didn't.

// - This is the 'filler doc'. It would pass style checks, but doesn't help anybody.

/\*\*

\* Splits a string.

\*

\* @param s A string.

\* @return A list of strings.

\*/

List<String> split(String s);

// Great.

// - Covers yet another edge case.

/\*\*

\* Splits a string on whitespace. Repeated whitespace characters are collapsed.

\*

\* @param s The string to split. An {@code null} string is treated as an empty string.

\* @return A list of the whitespace-delimited parts of the input.

\*/

List<String> split(String s);

## Import ordering

Imports are grouped by top-level package, with blank lines separating groups. Static imports are grouped in the same way, in a section below traditional imports.

:::java

import java.\*

import javax.\*

import scala.\*

import com.\*

import net.\*

import org.\*

import com.twitter.\*

import static \*

No wildcard imports

Wildcard imports make the source of an imported class less clear. They also tend to hide a high class [fan-out](http://en.wikipedia.org/wiki/Coupling_(computer_programming)#Module_coupling).  
See also [texas imports](https://github.com/twitter/commons/blob/master/src/java/com/twitter/common/styleguide.md" \l "stay-out-of-texas)

:::java

// Bad.

// - Where did Foo come from?

import com.twitter.baz.foo.\*;

import com.twitter.\*;

interface Bar extends Foo {

...

}

// Good.

import com.twitter.baz.foo.BazFoo;

import com.twitter.Foo;

interface Bar extends Foo {

...

}

## Use annotations wisely

### @Nullable

By default - disallow null. When a variable, parameter, or method return value may be null, be explicit about it by marking [@Nullable](http://code.google.com/p/jsr-305/source/browse/trunk/ri/src/main/java/javax/annotation/Nullable.java?r=24). This is advisable even for fields/methods with private visibility.

:::java

class Database {

@Nullable private Connection connection;

@Nullable

Connection getConnection() {

return connection;

}

void setConnection(@Nullable Connection connection) {

this.connection = connection;

}

}

### @VisibleForTesting

Sometimes it makes sense to hide members and functions in general, but they may still be required for good test coverage. It's usually preferred to make these package-private and tag with [@VisibleForTesting](http://docs.guava-libraries.googlecode.com/git-history/v11.0.2/javadoc/com/google/common/annotations/VisibleForTesting.html) to indicate the purpose for visibility.

Constants are a great example of things that are frequently exposed in this way.

:::java

// Bad.

// - Any adjustments to field names need to be duplicated in the test.

class ConfigReader {

private static final String USER\_FIELD = "user";

Config parseConfig(String configData) {

...

}

}

public class ConfigReaderTest {

@Test

public void testParseConfig() {

...

assertEquals(expectedConfig, reader.parseConfig("{user: bob}"));

}

}

// Good.

// - The test borrows directly from the same constant.

class ConfigReader {

@VisibleForTesting static final String USER\_FIELD = "user";

Config parseConfig(String configData) {

...

}

}

public class ConfigReaderTest {

@Test

public void testParseConfig() {

...

assertEquals(expectedConfig,

reader.parseConfig(String.format("{%s: bob}", ConfigReader.USER\_FIELD)));

}

}

## Clean code

#### Disambiguate Favor readability

if there's an ambiguous and unambiguous route, always favor unambiguous.

:::java

// Bad.

// - Depending on the font, it may be difficult to discern 1001 from 100l.

long count = 100l + n;

// Good.

long count = 100L + n;

### Remove dead code

Delete unused code (imports, fields, parameters, methods, classes). They will only rot.

## Use general types

When declaring fields and methods, it's better to use general types whenever possible. This avoids implementation detail leak via your API, and allows you to change the types used internally without affecting users or peripheral code.

:::java

// Bad.

// - Implementations of Database must match the ArrayList return type.

// - Changing return type to Set<User> or List<User> could break implementations and users.

interface Database {

ArrayList<User> fetchUsers(String query);

}

// Good.

// - Iterable defines the minimal functionality required of the return.

interface Database {

Iterable<User> fetchUsers(String query);

}

## Avoid typecasting

Typecasting is a sign of poor class design, and can often be avoided. An obvious exception here is overriding [equals](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)).

## TODOs

### Leave TODOs early and often

A TODO isn't a bad thing - it's signaling a future developer (possibly yourself) that a consideration was made, but omitted for various reasons. It can also serve as a useful signal when debugging.

### Leave no TODO unassigned

TODOs should have owners, otherwise they are unlikely to ever be resolved.

:::java

// Bad.

// - TODO is unassigned.

// TODO: Implement request backoff.

// Good.

// TODO(George Washington): Implement request backoff.

### Adopt TODOs

You should adopt an orphan if the owner has left the company/project, or if you make modifications to the code directly related to the TODO topic.

1. [↑](#footnote-ref-1)