

SUB RDD Technical Reference

5. Mai 2020

`<https://github.com/subugoe/rdd-technical-reference>`

Table of Contents

1 About this Document	4
2 Style Guides	4
2.1 General	4
2.2 Specific for Programming Languages	4
3 Is Your Software Fully Documented?	5
3.1 Doc Sprints	5
3.2 General	5
3.3 Developer Documentation	6
3.3.1 Architecture of the Software	6
3.3.2 API Documentation	6
3.4 Admin Documentation	6
3.5 Server documentation:	7
3.6 User Documentation	7
4 Which version control do you use? You do use version control, do you?	7
5 Do you track your bugs properly?	8
6 Do you test your software?	8
7 Building Code and Continuous Integration	8
7.1 Building Code	8
7.2 Continuous Integration	9
7.2.1 Sample configuration of the GitLab Runner	10
7.2.2 Sample configuration of the Jenkins CI (Multibranch Pipelines)	10
8 Deployment and maintenance	10
8.1 Puppet	10
8.1.1 Monitoring	11
8.1.2 Release Management	11
9 Code quality level for RDD	11
9.1 Code review	11
9.1.1 Sample workflow: Code reviews in SADE	11
9.2 Proof of concept	12
10 Licensing	12
11 Retirement of software	12

1 About this Document

Author: Software Quality Working Group

Audience: Developers of the Research and Development Department of the Göttingen State and University Library.

Purpose: This guideline should help you getting started a new software development project (or improving an existing one!) in the Research and Development Department of the Göttingen State and University Library.

Our goal is to establish better software quality by following standards the developer team has mutually agreed upon. Roughly basing on the [EURL-SE Network Technical Reference](#), these standards are discussed, worked out, and decided in the Software Quality Working Group, which meets biweekly on Tuesdays at 13:00–14:00. However, they aren't cast in stone, so in case you have a good idea for a better standard, feel free to contribute!

Status: This document is a living document and will be extended as soon as the Software Quality Working Group has agreed upon a new standard for software projects in RDD. TODOs and addenda of this document are maintained [here](#).

2 Style Guides

2.1 General

The basic definitions are given by our [EditorConfig](#) file, `.editorconfig`, i.e. Unix line breaks and 2 space indentation.

2.2 Specific for Programming Languages

For the more prominent programming languages we have formatting and general style guides we ask you to follow:

- **Java:** The Java style guide can be found [here](#). It's based on the [Google style guide for Java](#) with some minor RDD specific settings. You can configure Eclipse to use it automatically at *Eclipse > Preferences > Java > Code Style > Formatter*. Just load the [RDD Eclipse Java Google Style](#) in the formatter preferences and use it in your RDD projects.
- **JavaScript:** For JS we use the [Airbnb JavaScript Style Guide](#).
- **HTML/CSS:** For HTML/CSS we agreed upon the [Google HTML/CSS Style Guide](#).
- **XQuery:** We use the [xqdoc style guide](#) with the following addenda:

- use double quotes instead of single quotes (for easy escaping)
- use four spaces for a TAB (because eXide switching the preferences in eXide’s setting isn’t permanent)
- **XSLT**: Since there is no official style guide for XSLT, we decided to write **our own**, resulting from common best practices and own experiences within the department.
- **Python**: For Python **PEP 8** should be used, Django has a style guide based on PEP-8 with some exceptions: **Django-Styleguide**. There are linters and tools like **flake-8** and **pep-8** available as support.
- **SPARQL**: For SPARQL there is not really any official style guide and there is no possibility to simply include any code style automatically using a code style file. We just collect some advice how to format and use SPARQL code **here**.

3 Is Your Software Fully Documented?

3.1 Doc Sprints

To ensure the best documentation of our code we meet on a weekly base for a code sprint to document everything we have coded throughout the week and haven’t been able to document properly yet. The meeting takes place in the meeting room at 1pm. Cookies may be provided.

3.2 General

- Don’t document a language’s specifics, e.g. operators.
Example: Use of => in the XQuery expression `replace("qbc", "q", "b") => substring(1, 2)` MUST NOT be explained in a comment.
- It is best to use a language’s structure to document.
- Write the best documentation you can.
- Documentation and variable language is American English.
- Docs should be as close to the code as possible.
This refers both to the documentation in a repository and in the code itself, e.g. inline documentation.
- Every code repository MUST have:
 - a README.md file according to **this template** that contains
 - * link to original repository (if the software is forked or otherwise based on preexisting software)

- * short introduction on what the repository is about
- * a guide how to get the software running (if makes sense)
- * link to demo instance
- * example or demo installation
- * link to license file
- * contribution guide
- * link to style guide
- * link to bug tracker/project management system
- * known issues
- * badges to CI status
- a LICENSE file

3.3 Developer Documentation

3.3.1 Architecture of the Software

Each software project should be documented using an architecture diagram that helps understanding its basic functionality (even though using tools to generate diagrams such as UML class diagrams doesn't seem to be possible in every case).

Examples:

- [Generating call graphs in eXist-db](#)

Call diagrams can be useful to follow code and service calls and should be existing for every API method.

3.3.2 API Documentation

- The docs should comprise used parameters, author and @since annotations
- [Example for Java](#)
- Links to callers must not be listed in the documentation, because this info will be deprecated soon. It is strongly recommended to use call stacks of tools like Eclipse (Java) and/or Call Graph Module (SADE).
- Document REST-APIs using [openAPI](#) if possible. OpenAPI docs should be located at `/doc/api` on servers.

3.4 Admin Documentation

- The docs should comprise how to install the software, how to run and/or restart it, how to test the installation, ...

3.5 Server documentation:

This type of documentation is provided and maintained in our DARIAH wiki space.

We have a template encompassing all information necessary: To create a wiki page for a new server navigate to the FE Server list, select “...” right beside the “Create” button and search for “FE-Server”.

3.6 User Documentation

- how to use the software and APIs, FAQs, walkthroughs, ...
- guided tour (Bootstrap Tour) as user documentation
 - for SADE portal usage (such as Fontane, BdN, Architrave)
 - for complex Digital Editions
- screencasts

4 Which version control do you use? You do use version control, do you?

We exclusively use git in RDD. Please see <https://git-scm.com/doc> for information on how it works.

We recommend to use the [Git flow Workflow](#) (also consult the [Cheat Sheet](#)). For git flow it is safest to protect your master and develop branch on server side to avoid accidental pushes into these branches. All specific branches working on an issue described in a bug tracker may utilize the following naming scheme:

`[track]/#[ISSUENUMBER] - [KEYWORD]`

e.g. `bugfix/#12-flux-capacitor`.

A GitHub workflow used in DARIAH-DE and related services is described in the [DARIAH-DE Wiki](#).

You could also use the [GitLab flow](#) as a simpler alternative, which can be broken down into [11 Rules](#).

It is also recommended to automatically close issues via commit message; How this works exactly depends on the Git repository server. Issues can also be [referenced across repositories](#).

We use the following Git servers at the moment in RDD:

- Projects (GWDG) -> <https://projects.gwdg.de>

- GitLab (GWDG) -> <https://gitlab.gwdg.de>
- github.org -> <https://github.com/subugoe>

Which one is suitable for you depends on:

- the project you are working on
- existing code
- whether or not you want to use CI/CD or GitLab Runners
- ...

We have got an [RDD team on GitHub](#). Feel free to join us!

Consider mirroring of repos for project visibility (e.g. mirror GitLab/Projects code to Github?)

5 Do you track your bugs properly?

A bug tracking system is obligatory! Please use the respective bug tracking system of your repo and/or project management solution (please see chapter version control)!

6 Do you test your software?

We aim to have a test coverage of **100%** (except for getter and setter methods). This is understood on a component level, which means that every method should have at least one test. Whether you achieve this by Test Driven Development (TDD) or not is specific to your preferred way to work.

Please keep in mind not only to write a test for each of your functions but also to consider all possible outcomes. It is e.g. not sufficient to test if a function creates a file if the written content depends on variables etc.

Examples for writing tests in different programming languages are:

- [XQuery](#)

7 Building Code and Continuous Integration

7.1 Building Code

Ideally, we use build tools to conveniently get a software running. The reason for using a build tool is to be able to build and/or test a code project with one command (after checking out). Another reason is to include dependency management.

Build tools we are using at the moment

- **bash scripting:** (BdNPrint, FontanePrint)
- **eXist:** Ant (SADE)
- **Java:**
 - Maven (TextGrid)
 - gradle (TextGrid)
- **JavaScript:**
 - bower (DARIAH-DE GeoBrowser, tgForms)
 - cake (tgForms)
 - NPM (DARIAH-DE Publikator, tgForms)
 - rake (DARIAH-DE GeoBrowser)
- **Python:**
 - make (Sphinx documentation)
 - PIP (DiscussData)
- **Ruby:** bundler (DARIAH status page)

Build tools we want to evaluate

- gradle

7.2 Continuous Integration

We want to use CI as soon as possible in new projects. Please set up your gitlab-project to show your pipeline-status and test-coverage for your default-branch under Settings/General->Badges. The generic links for all projects are:

- * Pipeline-status - Link: https://gitlab.gwdg.de/{project_path}/commits/{default_branch}
- Badge image URL: https://gitlab.gwdg.de/{project_path}/badges/{default_branch}/pipeline
- * Test-coverage - Link: https://gitlab.gwdg.de/{project_path}/commits/{default_branch}
- Badge image URL: https://gitlab.gwdg.de/{project_path}/badges/{default_branch}/coverage

The workflows we are using currently in Jenkins and GitLab Runner are:

- Code building
- Testing
- Code analyzer (Sonar)

- Packaging (JAR, WAR, DEB, XAR)
- Distribution (Nexus, APTLY repo, eXist repo)
- Release Management (via GitLab Environments and gitflow)

7.2.1 Sample configuration of the GitLab Runner

There is a [full and documented example](#) of how the GitLab Runner is used in SADE.

7.2.2 Sample configuration of the Jenkins CI (Multibranch Pipelines)

- On commit and push to the <https://projects.gwdg.de> gitolite repo (such as <https://projects.gwdg.de/projects/tg-crud/repository>) Jenkins on ci.de.dariah.eu is notified (see projects' gitolite configuration)
- Jenkins then does a checkout and build of configured branches (see Jenkins' project's multibranch pipeline configuration such as <https://ci.de.dariah.eu/jenkins/job/DARIAH-DE-CRUD-Services>).
- Stage *Preparation*: Prepare things
- Stage *Build*: Build, JAR, WAR, and DEB packages from source code, deploy JAR and WAR packages to the [Nexus repo](#). For further information on Nexus cf. [ci's server documentation](#). Jenkins is configured to deploy JARs and WARs via Maven and a Nexus deploy-account.
- Stage *Publish*: Publish DEB packages to the [DARIAH-DE Aptly Repo](#). Jenkins is using a shared library of scripts and publishing is divided into four conditionals: TG version, DH version, SNAPSHOT version, or RELEASE version due to given version suffixes!

8 Deployment and maintenance

8.1 Puppet

For server configuration and setup we use puppet for most servers. The main puppet code is provided in GitLab <https://gitlab.gwdg.de/dariah-de-puppet>. The DARIAH-DE and TextGrid Repository module (dhrep) is contained in Github <https://github.com/DARIAH-DE/puppetmodule-dhrep>.

8.1.1 Monitoring

- Icinga probes for DARIAH-DE services <https://icinga.de.dariah.eu/icinga>
- Metrics for Sever specific monitoring <https://metrics.gwdg.de>

8.1.2 Release Management

9 Code quality level for RDD

9.1 Code review

We want to ensure code review for all major commits, in gitflow for everything that is subject to be merged into `develop`.

For projects with more than one developer in the team it is preferred to have code reviews within the team, in other cases your friendly RDD developer team is on your side.

- idea: invite all developers to a MR, at least 2 approvals needed for MR taking place. this way, everybody gets the chance to have a look at other people's code

9.1.1 Sample workflow: Code reviews in SADE

1. A developer decides to work on a feature. She commits her changes to a separate feature branch. After some time she finishes the feature and wants it to be part of the development branch.
2. The developer creates a merge request and assigns everybody she sees fit to properly review her code to it.
3. To avoid diffusion of responsibility, she also assigns one of the chosen assignees as **MUST**. This means that this person has to approve the MR, otherwise the merge cannot be done. Although GitLab sends notifications to everybody who is newly assigned to a MR, she should notify the **MUST** assignee personally (in case he or she doesn't notice the mail sent by GitLab).
4. The **MUST** assignee reviews the changes according to style, variable naming, understandability, documentation provided, functionality, etc. If everything is to his or her liking, he or she approves the MR. The other assignees are free to review the code as well. **Note:** MRs without docs should not be accepted.
5. After the MR has been (dis)approved, the assignee removes his- or herself from the list of assignees. The **MUST** assignee informs the developer over the review being done.

6. The developer merges her changes into the development branch.

9.2 Proof of concept

When preparing a proof of concept that is always labeled `poc`, a code review is not necessary.

10 Licensing

- clarify software license before programming
- add license to code header

Best practice is to maintain a file listing all third-party packages that are part of the software. This list should hold the following metadata and SHOULD be prepared like the table below, always in alphanumeric order.

name	license	origin
foo	barware	github.com/foo/bar

Maybe the `license-maven` plugin will help you.

11 Retirement of software

- clarify if software is no longer supported

12 Helpful links and references

- EURISE-network technical-reference:
<https://github.com/eurise-network/technical-reference>
- DHTech – An international grass-roots community of Digital Humanities software engineers:
<https://dh-tech.github.io>
- The Software Sustainability Institute, Guidelines and Publications:
<https://www.software.ac.uk>
- The Joel Test: 12 Steps to Better Code:
<https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better>
- Software Quality Guidelines:
<https://github.com/CLARIAH/software-quality-guidelines>

- Software Testing Levels:
<http://softwaretestingfundamentals.com/software-testing-levels>
- Netherlands eScience Center Guide
<https://guide.esciencecenter.nl>