SUB RDD Technical Reference

20. Dezember 2018

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

Table of Contents

1 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 DARIAH Technical Reference, these standards are discussed, worked out, and decided in the Software Quality Working Group, which meets biweekly on Tuesdays at 12:30-13:30. However, they aren't cast in stone, so in case you have a good idea for a better standard, feel free to contribute!

2 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.

3 Guidelines

3.1 Do you stick to our code style guides?

3.1.1 General

The basic definitions are given by our EditorConfig, i.e. unix line breaks and 2 space indentation.

Unfortunately, not all editors support EditorConfig. In case you use **eXide**, the IDE that comes with exist-db, you can set 2 space indentation as default by editing `/db/apps/eXide/src/preferences.js`.

3.1.2 Specific for programming languagues

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 setting. 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. @TODO uv: Tutorials verlinken!

- **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 makes it so)
- **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.
- **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 advices how to format and use SPARQL code.
  - declaration of variables should start with a **?** (and not with a **$**).
  - opening parenthesis **{** should be at the end of the line. Closing parenthesis in a separate line.

"'SELECT * WHERE { ?s ?p ?o . } LIMIT 10"'

- `group concatenations in SELECT command should be in seperate lines.`

"'**TODO** @Max: Provide example"'

3.2 Is your software fully documented?

3.2.1 General issues

- don't document computer language's interna
- best use language structure to document
- write the best documentation you can
- documentation and variable language is American English
- should be as code-near as possible
- every code repo must have
  - a README.md file that contains @TODO mw: provide link to example
    * link to original repository
    * short introduction

    - link to demo instance
    - example or demo installation
    - link to licence file
    - contribution guide
    - link to style guide
    - link to bugtracker/project managemenmt system
    - known issues
    - badges to ci status
  - a LICENCE file

3.2.2 Developer documentation

Architecture of the software Each software project should be documented using an architecture diagram that helps understanding its basic functionality (using tools to generate diagrams such as UML class diagrams seems not to be possible in every case).

Examples: - @TODO mw: call graph - @TODO fu: Looks for JAVA UML things (crud?)

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

API documentation

- used parameters, author and since annotations
- example for Java: `https://lab.sub.uni-goettingen.de/self-updating-docs.html`
- links to callers? who is calling this method, and when?
- meet and write documentation together regularly (documentation sprint)?

3.2.3 Admin Documentation

- how to install the software, how to run and/or restart it, how to test the installation, ...
- server documentation

3.2.4 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

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

We are using GIT in RDD! Nothing else! How it works, please see `https://git-scm.com/doc`.

We recommend to use Gitflow Workflow: `https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow`, Cheat Sheet: `https://danielkummer.github.io/git-flow-cheatsheet`), if possible on server side: use protection of the develop and master branches.

Automatically closing issues via commit message depends on the Git repository server. @TODO mw: References to other issues, etcpp.

Which repo you are using depends on:

- the project
- existing code
- using Gitlab Runners
- ...

We use the following at the moment in RDD:

- Projects (GWDG) –> `https://projects.gwdg.de`
- Gitlab (GWDG) –> `https://gitlab.gwdg.de`
- github.org –> `https://github.com/subugoe`

We have got an RDD team on Github: `https://github.com/orgs/subugoe/teams/fe`

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

3.4 Are you tracking 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)!

3.5 What is your test coverage?

We aim to have a test coverage of **100%** (except for getter and setter methods). Whether you achive 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 different programming languages are:

- **XQuery**: `https://gist.github.com/joewiz/fa32be80749a69fcb8da`

3.6 Code building and continuous integration

- @TODO: Code building (such as Jenkins, Gitlab Runner)
- @TODO: Provide complete examples for Jenkins and GitLab runner.
- @TDOO: Monitoring (such as Icinga, Metrics)

4 Code quality level for RDD

- Evaluate Software maturity Levels from CESSDA: **TODO**: @mw

5 Licencing

6 Retirement of software

7 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 publicartions: `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-code`
- Software Quality Guidelines: `https://github.com/CLARIAH/software-quality-guidelines`
- Software Testing Levels: `http://softwaretestingfundamentals.com/software-testing-levels`