EURISE Network

Technical Reference

Release 0.2-SNAPSHOT

Jun 05, 2020

Contributors:

Carsten Thiel, Michelle Weidling, Yoann Moranville

Contents:

1 Developer Guidelines **3**
- 1.1 Basics . . . 3
- 1.2 The README . . . 4
- 1.3 Documentation . . . 5
- 1.4 Tooling . . . 5
- 1.5 Interoperability . . . 6
- 1.6 Changelog . . . 6

2 Operational Guidelines **7**
- 2.1 Basic Operational Guidelines . . . 7
- 2.2 Infrastructure Documentation . . . 7
- 2.3 Documentation for (virtual) servers . . . 8
- 2.4 Security . . . 8
- 2.5 Incidents and Postmortems . . . 8

3 Policy Recommendations **11**
- 3.1 Code Hosting Policies . . . 11
- 3.2 Release policy . . . 12

4 Software Quality **13**
- 4.1 Software Quality Checklist . . . 13
- 4.2 Small steps to a higher software quality . . . 15

Glossary **17**

Bibliography **19**

This is the common Technical Reference developed by the EURISE Network, a cooperation of CESSDA ERIC, CLARIN ERIC and DARIAH ERIC.

The EURISE Network Technical Reference, is a collection of guidelines and how-tos, mainly aimed at developers contributing to the participating infrastructures. It serves as a combined introduction to many aspects of software engineering, while being neutral with respect to particular choices. This ensures the largest possible intersection of recommendations, while allowing the partners to implement their own specifications on top.

To make use of these guides, implementing an institutional manual based on this is recommended. In many places, possible or required choices are pointed out and should be made for a specific manual.

It is particularly recommended to adopt the Recommendations to encourage best practices in research software whenever starting a new project. Also, consider using the Software Sustainability Institute's Software Management Plan.

CHAPTER 1

Developer Guidelines

1.1 Basics

All development should be made available publicly under open source licences.

1. Use version control right from the beginning of a new project.
   - If in doubt, use Git.
   - Implement a *Code Hosting Policy*.
   - Use meaningful commit messages, cf. [ProGit] Sec. 5.2:
     - Capitalised summary with a maximum of 50 characters followed by a blank line.
     - Detailed but concise explanations in paragraphs or bullet points at 72 characters line length.
     - Explain *what* you do and *why*, but *not how*.
   - Never include any secrets in the code.
2. Use an appropriate OSI approved license.
   - Decide on an appropriate license before you first commit.
   - Ensure the license is compatible with all dependencies.
   - If in doubt, choose APACHE-2.0 or EUPL-1.2.
   - Add the text in a `LICENSE.txt`.
   - Add license statements to code files, consider using *SPDX <https://spdx.org/specifications>*__ identifiers.
3. Maintain a *README*.
4. *Document* your software properly.
5. Use existing *tooling* to support development workflows.
6. Ensure maximal *interoperability*.
7. Ensure your software is usable and accessible.
8. Implement a *release policy* and keep a *changelog*.
9. Add a code of conduct in a `CODE_OF_CONDUCT.md`, like `we do`.

10. Specify contribution policies in a `CONTRIBUTING.md`, like `we do`. Don't ignore non-code contributions. A legitimate policy can be that external contributions are not accepted and merged.

1.2 The README

The first thing users and other developers look at when first making contact with a repository is the `README.md` in the repository's root directory. Its purpose is to give a concise but comprehensive introduction to the project. It should provide links to further (more detailed) documentation, websites or other background information. Depending on the relevant ecosystem, specific guides or templates for READMEs exist.

The following basic information must be provided:

1. The project name
2. A short but meaningful descriptive summary of the repository
3. The maintenance status

The following questions must be answered:

1. What does the software do?
2. Who will use the software?
3. What are alternatives to the software, and how do they differ?
4. How can someone get started?
    - Requirements
    - Binary download location
    - Build instructions with dependencies
    - Installation instructions
    - Quick start examples
    - Link to full user documentation
5. How can others join the development?
    - Coding styles used
    - API design
    - Toolchain and frameworks used
    - Community communication platforms
    - Link to developer documentation
    - Test suite details
    - Contribution guides
6. Release details and versioning
7. Who has contributed to the software?
8. Which license is used? In most cases a link to the *LICENSE.txt* is sufficient.
9. Who has to be acknowledged? I.e. who has played a significant role for the creating of the software? This can e.g. include funding, research communities, or co-workers that are not part of the project but have given advice and/or input to the development process.

1.3 Documentation

Documentation is fundamental to ensure usability and usefulness of the software. It must be stored along the code, ideally in the repository's `docs` folder. Basic documentation should also be included in the *README*.

Documentation is relevant in many forms, each of which should be addressed for different audiences with varying degree of experience and knowledge.

1. **User documentation**: Include a documentation for end users, including e.g.
   - Examples
   - Tutorials
   - How-Tos
   - FAQs
   - Screen-casts
   - API documentation, using OpenAPI standard
2. **Developer documentation**: Provide instructions for developers.
   - How to set up the environment.
   - Dependencies, including
     - Supported operating systems
     - Required libraries
     - External dependencies
   - Requirements, e.g. hardware, architecture, CPU, RAM, disk space and network bandwidth.
   - How to build the code.
   - How to package the code.

   Additionally, inline code documentation should be used as appropriate.
   - Always adhere to the language's standard or well established ones such as the Google Style Guides.
   - Document the *why* and not the *what*, cf. [CleanCode].
3. **Administration documentation**: Provide instructions for installation, configuration and maintenance, in particular when running as a daemon (e.g. a micro service).
   - Configuration instructions
   - Start-up script (e.g. init or systemd)
   - Monitoring setup, ideally through a monitoring endpoint

1.4 Tooling

In order to support an efficient development workflow and ensure a high degree of software quality the use of appropriate tooling is strongly recommended.

This should include:

1. Using an appropriate code editor or IDE.

   While the individual person should be free to choose the solution that best fits his or her need, the editor/IDE should provide standard features such as syntax highlighting and code completion for all relevant languages.

   Use EditorConfig to ensure consistency of code submitted by all developers. In case others can contribute to the software as well, it is recommended to add the respective `.editorconfig` to the software's repository.

2. Code linting within the editor, as pre-commit hooks and part of further automation should be used.
3. Unit testing to improve code quality and simplify future development.
4. Static code analysis to reduce common errors and improve overall quality by adhering to standards and best practices.
5. A Continuous Integration solution that runs on every code commit to verify the test suite, lint code, perform static analysis and more.

   Popular choices for CI are:

   - Travis CI
   - GitLab CI
   - Jenkins

1.5 Interoperability

1. Externalise all configuration to configuration files or environment variables.
2. Internationalisation principles must be applied to ensure future localisation.
3. Localisation of the software must be available in English and should be available in other relevant languages.
4. Provide an API, using versioning for REST APIs.
5. Use federated authentication and/or social login.
6. Provide full configuration examples.
7. Provide endpoints for monitoring (service status) and statistics (e.g. user requests and resource utilisation).
8. Use structured logging, e.g. NSD JSON Schema.

1.6 Changelog

1. Adopt a *release policy*.
2. Maintain a manually curated Changelog.
   - Use a `CHANGELOG.md`, like `ours`.
   - Structure in a descending order by version with release date.
   - Add all changes applied in that version.
   - Have an *unreleased* section at the top for the next release.

CHAPTER 2

Operational Guidelines

For a generic collection of IT Service Management documents, see e.g. [FitSM].

2.1 Basic Operational Guidelines

1. Maintain a proper up-to-date *documentation*.
2. Define and follow proper *security practices*.
3. Establish processes for recurring actions.
4. Automate as much as possible.
5. Implement and verify backups.
6. Monitor the infrastructure.
7. Consider software configuration management.

2.2 Infrastructure Documentation

1. Keep an up to date inventory of all infrastructure components used, including
   - Actual physical hardware (servers, switches, racks, tapes, appliances etc.)
   - *Virtual machines*
   - Cloud and container infrastructure
   - Storage solutions and instances
   - Backup systems
2. Keep an inventory of all services that are offered and their dependencies with each other and the underlying infrastructure.
3. Implement *security*.
4. Keep a record of *incidents*.

2.3 Documentation for (virtual) servers

Documentation for virtual servers should include the following information:

- Where is it?
- Location, e.g. rack number or visualisation infrastructure
- IP address(es)
- DNS name(s)
- What is it?
- Operating system
- Human readable purpose
- Who knows about it?
- Responsible system administrator
- Secondary system administrators
- Project manager
- Which services does it run?

For each service, list the following:

- Service name
- Ports
- Status verification
- Log file
- Configuration file(s)
- Data directories
- Dependencies
- How can it be fixed?

List possible procedures for emergencies, such as

- Whether to try rebooting or remounting something
- How to restore from backup

2.4 Security

Ensure all systems are patched regularly.

Use proper SSL setup, e.g. by testing with the Qualys SSL server test.

Implement a good AAI and adhere to standards such as [Sirtfi] and [Snctfi]

2.5 Incidents and Postmortems

Record all outages including

- Which service was *disrupted*?
- What else was *affected*?
- Who was *in charge* of the recovery?

- *When* was the incident *discovered*?
- *How* and *by whom*?
- When has the incident *begun*?
- When was the incident *mitigated*?
- Who was *informed* and how?
- Has this ever happened *before*?
- Has *sensitive* data, such as user data or secrets, been compromised?

Particular importance should be applied to record all steps taken to mitigate the incident. These should include the person, time and specifics of any action taken.

Security breaches and vulnerability exploits may need to be reported to authorities, in particular if sensitive and/or (legally) protected data was (potentially) affected. Users must be informed appropriately, responsibly and quickly.

Finally, decide upon and implement measures to prevent repetitions.

CHAPTER 3

Policy Recommendations

3.1 Code Hosting Policies

3.1.1 General Rules

Make sure to publish your code in a version control repository.

- There are a number of well-known commercial solutions, such as
  - GitHub
  - GitLab
  - Bitbucket

  They all offer some free options and using them has a number of advantages, e.g.
  - Good and established usability
  - High visibility of you code
  - Low barrier for find-ability and re-use
  - Good integration with other services and solutions

  When using commercial and in particular external services, you must have a backup and data extraction strategy in place, which ensures that you can always move to another solution.
- There are a number of possibilities to host your own solution
  - The commercial solutions above.
  - GitLab Community Edition
  - Gogs
  - Gitea
  - gitolite

3.1.2 Specific Solutions

On Site

- Be sure to implement all relevant operational procedures.

GitHub

When using GitHub, implementing a suitable policy is recommended. This should include the following considerations.

Organisations

GitHub organisations can be used to group all repositories of an institution or a research project. It should be clearly stated who is responsible for an organisation, ideally stated on the organisation's GitHub Page.

Always ensure rights are managed by a sufficient number of people:

- At least two people should be owner of the organisation. It may be appropriate to have two owners from each institution involved in the project.
- Create an institutional account. This account becomes owner of all organisations the institution is involved in and makes sure access is granted to all appropriate individual employees.

Organisations should have a policy on who can be a member as well as how repository maintainers and maintenance status are defined and communicated.

Backup

As GitHub is owned by a commercial company (the Microsoft Corporation), the service provided through GitHub is subject to change through corporate development. It is highly recommended to set up an automated backup system, in order to ensure that a copy of all code and metadata (including issues, wikis etc.) exists.

Features

Use the features provided by GitHub, such as issue labels, issue and pull-request templates etc.

3.2 Release policy

- Use semantic versioning.
- Provide releases as downloads.
- If applicable, provide binaries for all supported platforms.
- Have a roadmap.
- Consider providing releases via research repositories with citable references.

CHAPTER 4

Software Quality

Improving and assessing software quality can be done using many different approaches.

Existing solutions are [SQA-baseline] and [CESSDA-SML].

4.1 Software Quality Checklist

4.1.1 General

- [ ] Does the software have a descriptive name?
- [ ] Is there a short high-level description of the software?
- [ ] Is the purpose of the software clear?
- [ ] Does the software exactly match its requirements?
- [ ] Is the targeted audience of the software clear?
- [ ] Has the software been tested by members of the target audience in respect of its usability?
- [ ] Does the software (and its dependencies) use OSI approved licenses?
- [ ] Is the software under version control?
- [ ] Is there a website for the software?
- [ ] Is the software's website mobile friendly to a certain degree? I.e. can it be accessed on a smartphone or tablet without hiding the most important information and features?
- [ ] Are the user interface design and the software's website mindful of accessibility? I.e. does it consider e.g. a high contrast between colors for colorblind users, are there alternative texts for images that can be read by a screenreader, are texts easily resizeable, . . . ?
- [ ] Does the software have a release mechanism?
- [ ] Is the software available in packaged format or only sources?
- [ ] Are maintainer and development status clear, including up to date and accessible contact information?
- [ ] Are the requirements listed and up to date?
- [ ] Is copyright and authorship clear and accessible?

- [ ] Is there a contribution guide?

4.1.2 Documentation

- [ ] Is there an accessible low-level guide for getting started?
- [ ] Is there an accessible user guide?
- [ ] Is there a full user documentation?
- [ ] Does the user interface link to held references?
- [ ] Are there examples, FAQs and tutorials?
- [ ] Is there information stated about who to ask when a problem is not covered by the FAQ?
- [ ] Are known issues documented and easily accessible for all user groups?
- [ ] Can bugs/issues be reported easily by other developers and users?

4.1.3 Development

- [ ] Is the development setup documented?
- [ ] Is the build mechanism documented?
- [ ] Does the build mechanism use a common single-command system (i.e. Maven)?
- [ ] Is the software API documented?
- [ ] Are all appropriate config options externalised and documented?
- [ ] Does the code allow internationalisation (i18n)?
- [ ] Is the software localised (l10n)? English is mandatory.
- [ ] Is there a test suite?
- [ ] Is test coverage above 80%?
- [ ] Are the tests run on a regular and frequent basis, e.g. on commit/every night/. . . ?
- [ ] Do you have and stick to a policy for security by design?
- [ ] Is the software portable?
- [ ] Has the portability been tested?

4.1.4 Interoperability

- [ ] Are file formats standard compliant and documented?
- [ ] Is the API standard compliant?
- [ ] Does it provide a monitoring endpoint?
- [ ] Does it adhere to an interface style guide?
- [ ] Does it use existing authentication systems (OAuth2/eduGain)?

4.1.5 Administration

- [ ] Are software requirements such as operating system, required libraries and dependencies specified including versions?
- [ ] Are hardware requirements for CPU, RAM, HDD, Network specified?
- [ ] Are there deployment instructions?
- [ ] Is there a comprehensive and fully documented example configuration?
- [ ] Is a startup script provided?
- [ ] Are there troubleshooting guides?

4.2 Small steps to a higher software quality

The following list proposes small habit changes you/your team can easily implement into your day-to-day work. Sticking to just one of these suggestions will increase your software's quality at once without being too time-consuming. Approximate time costs are stated in brackets after each suggestion.

Please note that this list is far from being complete and it is ever developing since technologies change a lot. We're looking forward to your PR to extend this list or discuss its recommendations.

1. Functions and variables
   - Give your functions and variables a descriptive name, e.g. "currentYear" instead of "y". *[0.5min]*
   - One function should be responsible for doing one thing. If it does more than this, split it. *[max. 10min]*
   - Clean as you go: In case you notice something worth changing and it takes less than 2 minutes to fix it, do it at once. *[max. 2 min]*
2. Error handling
   - Write custom error codes with a descriptive error messages for your modules. *[ca. 3min per error code]*
3. Documentation
   - Dedicate on 1h per week to writing/updating your documentation. *[1h]*
   - Organize a weekly doc sprint (approx. 1h) with your colleagues. *[1h]*
4. Usability
   - Engage in hallway usability testing: Once a week let a colleague test some part of your User Interface for 5 minutes. (You'll get the best results if s/he doesn't know what you are working on.) *[7min + Xmin for solving the problems you have found]*
   - When choosing a color scheme for a GUI, turn your screen to gray scale and check if the contrast is sufficient. *[max. 5min]*
   - Use relative font sizes in CSS instead of absolute ones. *[0.5min]*
5. Making developing in teams less painful
   - Commit early. Commit often.
   - Think about adopting a branching model like [git flow](https://nvie.com/posts/a-successful-git-branching-model/).

Glossary

These are some of the most important terms used throughout. See also GitHub Glossary.

- *Collaborator* – Someone with write access to the repository.
- *Contributor* – Someone who submits code to the repository, either directly or via pull-/merge-request.
- *Developer* – Anyone who writes code.
- *Maintainer* – The person responsible for feature development and deciding about contributions.

Bibliography

[CleanCode] Robert C. Marton: **Clean Code: A Handbook of Agile Software Craftsmanship**, Prentice Hall PTR, 2008

[FitSM] ITEMO: **FitSM** – A free standard for lightweight ITSM, http://fitsm.itemo.org/, 2016

[ProGit] Scott Chacon, Ben Straub: **Pro Git**, https://git-scm.com/book/en/v2, Version 2.1.64, 2018-06-01

[Sirtfi] AARC: **Security Incident Response Trust Framework for Federated Identity**, https://aarc-project.eu/policies/sirtfi/, 2015

[Snctfi] AARC: **Scalable Negotiator for a Community Trust Framework in Federated Infrastructures** https://aarc-project.eu/policies/snctfi/, 2017

[SQA-baseline] Orviz, Pablo; López García, Álvaro; Duma, Doina Cristina; Donvito, Giacinto; David, Mario; Gomes, Jorge: **A set of Common Software Quality Assurance Baseline Criteria for Research Projects**, https://indigo-dc.github.io/sqa-baseline/, 2019

[CESSDA-SML] John Shepherdson: **CESSDA Software Maturity Levels**, https://doi.org/10.5281/zenodo.2614050, 2019

[NLeSC-Guide] Netherlands eScience Center: **Guide**, https://guide.esciencecenter.nl/

[CLARIAH-SQG] CLARIAH: **Software Quality Guidelines**, <https://github.com/CLARIAH/software-quality-guidelines>