Clariah

Software and Service Requirements

Maarten van Gompel David de Boer Thomas Vermaut Hayco de Jong
Jaap Blom Femmy Admiraal Hennie Brugman Mario Mieldijk
Enno Meijers Roeland Ordelman Ronald Siebes Menzo Windhouwer

version 1.0.1 (CONCEPT), June 2022

Developed in the CLARIAH-PLUS project, funded by the NWO under grant 184.034.023

Contents

Introduction

This document intends to specify minimal *technical* requirements for CLARIAH software and software services. It is aimed at developers and external evaluators. It provides a minimal list of specific and directly actionable criteria to hold the software against. The aim is to ensure a certain level of software quality and sustainability, and to lay out basic requirements needed for interoperability within the larger CLARIAH infrastructure. Only software that is built and maintained well will get used and thus provide value.

The document is loosely derived from earlier work to establish Software Quality Guidelines within CLARIAH-CORE, but condenses that elaborate work to the most relevant and directly actionable points.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Basic Software Requirements

1. The software's source code *MUST* be stored in a public version control system (VCS).

1. The use of a publicly accessible VCS platforms such as Github, Gitlab, Bitbucket or SourceHut is *REQUIRED*. This may be self-hosted by an institution rather than outsourcing it to a third party.
2. CLARIAH is registered as an organization on github, any CLARIAH software project *MAY* be hosted there.

3. Git is *RECOMMENDED*
4. All content, including documentation, *SHOULD* be stored alongside the code in the VCS.
5. The source code repository *SHOULD* be publicly accessible from day 1 of development so interested third parties can follow development and provide feedback. The source code repository *MUST NOT* be reduced to a release/file-transfer mechanism for completed products. Exceptions are when code is under embargo or privacy-sensitive.

2. A README file *MUST* be provided in the root directory of the VCS

1. This should be a readable plain-text files. Markup formats such as Markdown (`README.md`, or alternatively ReStructuredText, `README.rst`) are *RECOMMENDED*. Other formats such as LaTeX, HTML, Word, etc *MUST NOT* be used for the README.
2. The README *MUST* have a clear description of what the software does (what problems it solves) and whom it is intended for.
3. The README *MUST* make the current status of the software clear: is it ready for production, experimental, or a proof-of-concept? It *MUST* also make clear whether the software is actively maintained or not, and if so, by whom (see point 12). The use of the repostatus vocabulary is *RECOMMENDED* to this end. A simple repostatus badge suffices.
4. The README *MUST* make clear who wrote and currently maintains the software, including a contact link, and acknowledge the funders.
5. The README *MUST* provide (or link to) installation instructions.
6. The README *MUST* provide (or link to) usage instructions for a quick start, explaining how the first task can be performed with the system.
7. The README *SHOULD* make explicit any software and minimal hardware requirements the software relies on. This includes making explicit the software's main dependencies, target OS, and minimally required resources.
8. The README *MUST* make clear (link to) a public support channel (e.g. an issue tracker, mailing list) where people can submit bug reports and feature requests.

3. The software *MUST* be distributed as open source under an OSI approved license

The full license *MUST* be stored in a `LICENSE` (`LICENSE.md`) file in the root of the VCS, the same format recommendations and restrictions apply as to the `README` file.

The maintainer of the software *MUST* ensure the license is not in conflict with the licenses of the dependencies they use. Tools such as the JLA - Compatibility Checker can help identify license conflicts.

The license *MUST* be one of the OSI-approved licenses. The license *SHOULD* be one of the popular licenses mentioned there or EUPL-1.2.

4. The software *MUST* be released periodically with clear version numbers

Periodic releases of the software *SHOULD* be released with a clear version number.

1. The use of semantic versioning is *RECOMMENDED*.
2. Each release *must* be accompanied by a tag in the VCS (e.g `git tag`). This is automatically implied when you use the release mechanism by platforms such a GitHub, GitLab, Bitbucket.
3. Each release *must* be accompanied by release notes describing on a more general level what is new in the release. Again, the release mechanisms by major VCS platforms provide for this. Alternatively you *MAY* use a `NEWS` or a `CHANGELOG` file.

5. The software *MUST* separate code from configuration.

Configuration values (such as database connection strings, API URLs or secrets) *MUST* be parameterized; they *MUST NOT* be hard-coded in the application source.

For services, also see point 15.3.

This is a corollary of Infrastructure Requirement 4.

6. Each release of the software *SHOULD* be installable through a proper package manager

When the software is of a type that fits a certain language ecosystem (e.g. a software library in a particular programming language) or the software targets a very specific distribution/OS, then it *MUST* be packaged for a package manager fitting the language or OS. This should ensure the software *and all of its dependencies* are installable through *one single command*:

- Python libraries and command-line tools *SHOULD* be packaged for and submitted to the Python Package Index, installable through `pip`.
- R libraries should be packaged for and submitted to CRAN.
- Perl libraries *SHOULD* be packaged for and submitted to CPAN.
- Java libraries *SHOULD* be packaged for and submitted to Maven Central, installable through `mvn`.
- Rust libraries and command-line tools *SHOULD* be packaged for and submitted to crates.io, installable through `cargo`.
- NodeJS libraries *SHOULD* be packaged for and submitted to npm, installable through `npm`.
- C/C++ software has no specific ecosystem for packaging. The use of distribution-specific packages as described further below is then *RECOMMENDED* The use of a standardized build system such as the autotools, cmake, or make is *RECOMMENDED*. Static linking *MAY* be an appropriate solution to handle dependencies.
- (this list is not exhaustive)

This step may combine both the building of software (compilation) as well the installation. If any building/compilation is performed, then the build process *MUST* make use of an *automated* and *standardized* build system.

Packaging for a package manager shall only be deviated from if there is no suitable packaging ecosystem for the software (C/C++) or if the software is not a fit for the packaging ecosystem, such as the software being too high-level (e.g. a web application) and consisting of too many components for any single packaging ecosystem. In such a case, the individual components should still be packaged as much as possible, and the ensemble as-a-whole made available through containerisation for ease of distribution and deployment (See point 15)

If software has a high reusability potential and is often used by users of a certain OS/distribution, then it is *RECOMMENDED* to package it for that OS/distribution. Whether this is worth the extra investment is to be assessed individually. Unlike language-specific ecosystems, such a submission process generally acts as an extra quality control as there is a human review stage.

- Alpine Linux (apk)
- Debian Linux (`deb` for use with `apt`)
  - Packages submitted to Debian will eventually also land in Debian-derivates like Ubuntu Linux
- Red Hat Linux (`rpm` for use with `yum`)
- There are many other Linux/BSD distributions which can be potential targets.
- Homebrew for macOS
- Android apps *SHOULD* be submitted to Google Play and alternatively also to F-Droid.
- iOS apps *SHOULD* be submitted to the App Store.

7. The software *SHOULD* have a public support channel

This can be an issue tracker as automatically provided by the major VCS platforms, or even a simple mailing-list, on the precondition that a public archive is available.

Users can turn to this place for reporting any bugs they find in the software, or optionally post requests for new features. The public nature ensures that a public knowledge base is constructed where users can find answers to earlier posted questions, alleviating the burden on both the users as well as the developers. The public support channel also gives an indication of community interest in the project.

The only acceptable deviation to this rule is when the software is explicitly unmaintained and unsupported, which must be clearly indicated in the README (see 2.3).

8. Software *SHOULD* be reusable

To foster reusability, any meaningful reusable component of your software *SHOULD* be split into reusable software libraries/tools rather than be part of an indivisible monolithical whole. This ensures the work can be reused where appropriate.

9. Software *SHOULD* come with automated tests

1. To ensure software quality, software *SHOULD* have a test set consisting of unit and/or integration tests with a fair degree of coverage.
2. Automatic *Continuous Integration* infrastructure such as GitHub Actions, GitLab CI/CD, Jenkins or others *SHOULD* be used to automatically test the software upon any code change.
3. Additionaly, code coverage tests and dependency version checks *MAY* be used.

Deviations from this are only acceptable for initial proof-of-concept or highly experimental software (see 1.3).

10. Software *MUST* define software metadata along with the source code

Software *MUST* adhere to the software metadata requirements set by CLARIAH. This stipulates that software metadata *MUST* be specified alongside the source code in the source repository (e.g. in a `codemeta.json` file) and that we use codemeta and schema.org as metadata vocabulary. Automatic conversion to codemeta from various industry-standard metadata specifications are available. All CLARIAH software *MUST* be registered in the CLARIAH tool source repository.

11. Software *SHOULD* be documented.

Whereas minimal documentation is already *REQUIRED* per point 2, more extensive documentation is *RECOMMENDED*. The sources of which *SHOULD* be stored alongside the source code, ensuring they describe the same version, and the resulting documentation *SHOULD* be served on a website (platforms like readthedocs.io *MAY* be a good solution here).

1. When the software is a library, an API reference *MUST* be provided.
2. When the software is a webservice, a WebAPI reference *MUST* be provided.
3. When the software has a command line interface, usage example should be given in the documentation. Furthermore a `--help` parameter *MUST* be provided in the software itself.
4. When the software has a graphical/web user interface, the documentation *SHOULD* explain how to use it.
5. The use of API documentation as integral part of the code and tools such as doxygen, sphinx is *RECOMMENDED*.

Deviations to this rule are for unsupported/proof-of-concept/experimental software, which must be clearly indicated in the README (see 2.3).

12. Software *MUST* have a clear maintainer

All software *MUST* have a clear maintainer. The only exception is if the software is explicitly unsupported, abandoned or an initial experimental prototype not ready for adoption. The maintainer may be a person (or multiple) or an institution. The maintainer is responsible for responding to user requests, bug reports, security vulnerabilities and for releases and packaging. The maintainer *MUST* be specified in the formal software metadata (point 10) and *SHOULD* also be documented in the README (if not immediately obvious from context), additionally a `CODEOWNERS` file *MAY* be included in the root of your repository.

A project is *RECOMMENDED* to be open to contributions from the open source community (pull requests/merge requests/patches), the maintainer is responsible for reviewing those. It is *RECOMMENDED* to provide a `CONTRIBUTE.md` file in the version control root directory file with guidelines contributors to your software should follow.

The reverse also holds true: It is *RECOMMENDED* to contribute to (third-party) open source software. This is also in line with point 8 regarding reusability. When making changes to existing software that may be to other users' benefit, you *SHOULD* offer those changes back to the upstream maintainer(s) (typically using a pull/merge request, or by mailing a patch). You *SHOULD NOT* create a hard fork of the project unless there are unbridgeable differences with the original maintainer or the direction of the project.

The use of any software that is not or no longer maintained is *NOT RECOMMENDED*.

13. Software *MUST* be developed with attention to security & privacy

Security and privacy *MUST* be a point of attention throughout the software's lifecycle. Malicious actors are ubiquitous and looking for vulnerabilities to exploit, whilst nobody can guarantee to keep out all intruders and bugs are an inevitable part of software development, a best effort *MUST* be made.

1. Software *MUST NOT* store any unhashed user credentials.
2. Software *MUST* guard against common attacks such as SQL injection, Shell injection, Cross-site request forgery, session hijacking attacks, broken access control, and more. For web applications, the OWASP Top 10 is a good resource listing the most critical security risks to look out for in web applications.
3. Components with major known vulnerabilities *MUST NOT* be used (see OWASP A06:2021)
4. Software services *MUST* comply to the GDPR. This corresponds to Point 14 of the Infrastructure Requirements (IR).
5. All outside traffic *MUST* be properly encrypted. (e.g. use HTTPS). This corresponds to Points 11 and 12 of the Infrastructure Requirements (IR).
6. Security issues *SHOULD* be disclosed publicly. The most critical security problems should be kept under embargo first to give affected parties the chance to upgrade.
7. To safeguard the privacy of your users, it is *RECOMMENDED* to limit the amount of cross-site requests to third party content, to only those that are really needed. This is relevant not only from a privacy, but also from a security point of view. Be especially vigilant when incorporating social network 'like' buttons, as those are in essence trackers.

This relates to Point 13 of the Infrastructure Requirements (IR)

Software-as-a-Service Requirements (CLaaS)

14. Services *SHOULD* provide a simple RESTful API

A simple RESTful API is *RECOMMENDED* over more complex solutions such as SOAP. SOAP *SHOULD NOT* be used if it can be avoided. XML-RPC *MAY* be used. CLAM *MAY* be used as a home-grown CLARIAH solution to deliver RESTful webservices around existing tools.

15. Services *MUST* be packaged as containers

All software services *MUST* be packaged as OCI containers (e.g. Docker containers). Containers are self-sufficient (without external dependencies) and uniform (they look the same on the outside). This makes them decoupled from infrastructure specifics (such as the OS used by the infrastructure provider). The same container can be run on any Linux distribution, cloud provider (including AWS, Azure, Google and DigitalOcean) as well as on a developer's local Mac or Windows machine. Like point 6, this ensures that software *and all its dependencies* are installable through one single command.

(This corresponds to point 1 of the Infrastructure Requirements (IR))

1. The container images *MUST* be published in CLARIAH's container registry at each release. This can be automated (see IR7).
2. The Container images *SHOULD* be as small as possible, using multi-stage builds and other best practices for writing Dockerfiles.
3. Configuration values (software parameters) that may vary between deployments *MUST* be parameters to the container. This is implemented through environment variables, as this offers a generic , uniform and OS-agnostic way of specifying key/value pairs; i.e. an abstraction with high granularity. The infrastructure in turn configures all applications through such environment variables (see IR4 and also related to point 7). When a container starts (i.e. this *MUST* be at run-time and not at build-time), these variables are translated into whatever form needed for the application. Aside from the deployment-specific essentials, application providers themselves

decide to what extent their application is configurable at run-time. An important exception is the handling of secrets (usernames, passwords, API tokens) which *MUST NOT* be passed as unencrypted environment variables but for which special facilities (e.g. vaults) are needed (See point 16 of the Infrastructure Requirements (IR))

4. Containers should output all log information to `stdout` so it can be captured by the infrastructure (see IR5 and IR6).
5. Application data (state) that needs to be persistent between runs *MUST* be stored separate from the container (e.g. in a mounted volume) (See point 3 of the Infrastructure Requirements (IR))
6. The container build process *SHOULD* use proper packaging ecosystems for installing its individual components. Providing a container *SHOULD NOT* be considered a substitute for component packaging (see point 6, which is to be considered as complementary to or a prerequisite for this one).

16. Service developers *SHOULD* provide an initial template when multi-container orchestration is needed

If a complex service consists of multiple interacting containers, the developers *SHOULD* provide an initial template in the form of a Docker Compose configuration or a Kubernetes deployment configuration that illustrates how the containers are orchestrated to form the application. This *MUST* be maintained in a VCS repository (infrastructure as code principle). Infrastructure operators can build on this example to deploy the application.

17. Services *MUST* be compatible with CLARIAH's authentication and authorization infrastructure

All services open to end-users and which require some form of user authentication *MUST* be compatible with CLARIAH's authentication and authorization infrastructure. That is, they should be able to communicate with CLARIAH's SATOSA Authentication Provider. It is *RECOMMENDED* to use OpenID Connect for this communication. Instruction can be found here.

18. Services *MUST* expose a public endpoint providing their specification

1. OpenAPI (aka Swagger), WADL, and CLAM are *SUGGESTED* as possible interface description languages.
2. The specification endpoint *SHOULD NOT* be hindered by any authentication barriers.
3. The specification endpoint *SHOULD* be automatically generated.

19. Services *SHOULD* expose a public endpoint providing metadata

This is the corollary of point 10 for software; software as a service *MUST* adhere to the service section of the software metadata requirements set by CLARIAH.

In short, this stipulates that metadata for the service as a whole must be available at a public endpoint. Compliance with point 18 usually automatically implies compliance with this point. All CLARIAH services *MUST* be registered in the CLARIAH tool source repository.

Glossary

- **CLaaS** - CLARIAH as a Service vocabularies.
- **CLAM** - A framework developed in CLARIN/CLARIAH for building RESTful webservices with an additional generic user-interface for human end-users.
- **CodeMeta** - A third-party initiative to describe research software in Linked Open Data, and link to existing
- **Ineo** - The official portal to CLARIAH tools/services and data resources
- **VCS** - A **version control system**, e.g. Git, Mercurial, SVN, etc

License