Executable Research Compendium

This is the technical specification of the Executable Research Compendium (ERC) in PDF format.

The **normative version** is available in Markdown format in the online repository at https://github.com/o2r-project/erc-spec/.

This specification and guides are developed by the members of the DFG-funded project Opening Reproducible Research, https://o2r.info.

Build version: `c2081e4`

Contents

Executable Research Compendium

This is the technical specification of the Executable Research Compendium (ERC).

Read the specification (PDF download**.

Guides

Are you a **scientist** and want to publish your research as an ERC? Read **user guides for authors, reviewers, and readers**:

- ERC creation
- ERC examination
- ERC template

Are you a **developer** or architect and want to build applications using ERCs? Read our developer documentation:

- Developer guide
- o2r System Architecture

Are your a **librarian** or **preservationist** and want to use ERCs for archival of scholarly works? Read **user guides for librarians and preservationists**:

- ERC & OAIS

Credits

This specification and guides are developed by the members of the DFG-funded project Opening Reproducible Research

License

Figure 1: CC-0 Button

Build a112fd8 @ 2018-10-04T14:11:16Z

ERC specification

An Executable Research Compendium (ERC) is a packaging convention for computational research. It provides a well-defined structure for data, code, text, documentation, and user interface controls for a piece of research and is suitable for long-term archival. As such it can also be perceived as a digital object or asset.

Note

This is a draft specification. If you have comments or suggestions please file them in the . If you have explicit changes please fork the and submit a pull request.

Preface

Version

Specification version: `1`

Warning

This version is *under development*!

Notational conventions

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

The key words "unspecified", "undefined", and "implementation-defined" are to be interpreted as described in the rationale for the C99 standard.

Purpose, target audience, and context

This specification defines a structure to transport and execute a computational scientific analysis (cf. computational science). It carries technical and conceptual details on how to implement tools to enhance reproducibility and is most suitable **for developers**. **Authors** may feel more comfortable with the *user guides*.

These analyses typically comprise a digital workspace on a researcher's computer, which contains *data* (born digital, simulated, or other), *code*, third party *software* or libraries, and *outputs* of research such as digital plots or data. Code and libraries are required in executable form to re-do a specific analysis or workflow. Research is only put into a context by a *text*, e.g. a research paper, which is published in scholarly communication. The text comes in two forms: one that is machine readable, and another one that is suitable for being viewed by humans.

The latter is derived, or "rendered", from the former. The viewing experience can be static, textual, visual, or interactive.

Putting all of these elements in a self-contained bundle allows examining, reproducing, transferring, archiving, and formally validating computational research results in a time frame for peer review and collaboration. The ERC specification defines metadata and file structures to support these actions.

Major constituents

Three major constituents group possible user interactions with ERC.

Create Creation is transforming a workspace with data, code and text into an ERC.

Examine Examination is evaluating ERC at different levels, from inspecting contents to creating derived analyses.

Discover Discovery is searching for content powered by ERC properties, such as text, content metadata, code metadata et cetera.

Design principles

Simplicity This specification should not re-do something which already exists (if it is an open specification or tool). The risk of scattering information is mitigated by clear documentation. It must be possible to create a valid and working ERC *manually*, while supporting tools should be able to cover typical use cases with minimal required input by a creating user.

Nested containers We acknowledge well defined standards for packaging a set of files, and different approaches to create an executable code package. Therefore an ERC comprises *one or more containers but is itself subject to being put into a container*. We distinguish these containers into the inner or "runtime" container and the outer container, which is used for transfer of complete ERC and not content-aware validation.

Transparency, Stability, and Openness Plain text files usable by both humans and computers are the backbone to make sure ERCs are acceptable by users from all scientific domains, are understandable today and tomorrow, and are easy to extend. The ERC contains everything needed to execute a workflow.

How to use an ERC

The steps to (re-)run the analysis contained in an ERC as part of an examination are as follows:

- (if compressed first extract then) unpack the ERC's outer container
- execute the runtime container
- compare the output files contained in the outer container with the output files just created by the runtime container

This way an ERC allows computational reproducibility based on the original code and data.

Three questions

[Section inspired by REANA's "Four Questions"]

The ERC helps to make research papers more transparent an reusable by giving minimal structure for contents and context. They help to answer the "Three Questions" both for users, but more importantly for tools and services built around them.

1. **What is your result?**
   - file I should look at to see the description and visualisations
   - the "display file" shown by applications based on ERC
2. **What is your workflow?**
   - file I should look at as a reader when I want to understand your code/analysis/workflow, the steps you took
   - the "main file" used by applications based on ERC for creating ERCs and executing them, which means running the analysis and creating the result
3. **What is your environment?**
   - operating system you used
   - software you used (libraries, your own scripts, ...)
   - can be used by tools to recreate the same environment

ERC structure

Base directory

An ERC MUST has a *base directory*. All paths within this document are relative to this base directory.

The base directory MUST contain an ERC configuration file.

Besides the files mentioned in this specification, the base directory MAY contain any other files and directories.

Main & display file

An ERC MUST have a _main file, i.e. the file which contains the text and instructions being the basis for the scientific publication describing the packaged analysis. An ERC MUST have a *display file*, i.e. the file which is shown to a user first when she opens an ERC in a supporting platform or tool.

Main file and *display file* MUST NOT be the same file.

The *main file* MUST be *executable* in the sense that a software reads it as the input of a process to create the *display file*. The *main file*'s name SHOULD be `main` with an appropriate file extension and media type.

Note

The *main file* thus follows the literate programming paradigm.

Example

If the main file is an R Markdown document, then the file extension should be `.Rmd` and the media type `text/markdown`. A file `main.Rmd` will consequently be automatically identified by an implementation as the ERC's *main file*.

The display file's name SHOULD be `display` with an appropriate file extension and media type.

Example

If the display file is an Hypertext Markup Language (HTML) document, then the file extension should be `.htm` or `.html` and the media type `text/html`. A file `display.html` will consequently be automatically identified by an implementation as the ERC's *display file*.

The ERC MAY use an interactive document with interactive figures and control elements for the packaged computations as the *display file*. The *interactive display file* MUST have `HTML` format and SHOULD be valid HTML5.

Example

Typical examples for the two core documents are R Markdown with HTML output (i.e. `main.Rmd` and `display.html`), or an `R` script creating a PNG file (i.e. `main.R` and `display.png`).

Nested runtime

The embedding of a representation of the original runtime environment, in which the analysis was conducted, is crucial for supporting reproducible computations. Every ERC MUST include two such such representations:

1. an **executable runtime image** of the original analysis environment for re-running the packaged analysis, and
2. a **runtime manifest** documenting the image's contents as a complete, self-consistent recipe of the runtime image's contents which is a machine-readable format that allows a respective tool to create the runtime image.

The image MUST be stored as a file, e.g. a "binary" or "archive", in the ERC base directory.

The manifest MUST be stored as a text file in the ERC base directory.

System environment

The nested runtime encapsulates software, files, and configurations up to a specific level of abstraction. It may not include a complete operating system, for example for better performance or security reasons. While this information is included in the nested runtime, it MUST be accessible without executing the runtime. Hard to obtain information SHOULD be replicated in the configuration file.

If the nested runtime does not include the operating system, then the configuration file MUST include the following data about the environment used to create the ERC:

- *architecture*
- *operating system*
- *kernel* (if applicable)
- *runtime software version*

An implementation SHOULD notify the user if the provided system environment is incompatible with the implementations capabilities.

Tip

A partially incompatible system environment, especially a different kernel version, may still produce the desired result, as breaking changes are very rare. An implementation may utilise semantic versioning to improve its compatibility tests. An incompatible operating system, e.g. `linux` vs. `windows`, and architecture, e.g. `amd64` or `arm/v7`, are likely to fail.

ERC configuration file

The ERC configuration file is the *reproducibility manifest* for an ERC. It defines the main entry points for actions performed on an ERC and core metadata elements.

Name, format, and encoding

The filename MUST be `erc.yml` and it MUST be located in the base directory. The contents MUST be valid YAML 1.2. The file MUST be encoded in `UTF-8` and MUST NOT contain a byte-order mark (BOM).

Basic fields

The first document content of this file MUST contain the following string nodes at the root level.

- `spec_version`: a text string noting the version of the used ERC specification. The appropriate version for an ERC conforming to this version of the specification is `1`.

- `id`: globally unique identifier for a specific ERC. `id` MUST not be empty and MUST only contain lowercase letters, uppercase letters, digits and single separators. Valid separators are period, underscore, or dash. A name component MUST NOT start or end with a separator. An implementation MAY introduce further restrictions on minimum and maximum length of identifiers.

Note

While URIs (see rfc3986) are very common identifiers, not all systems support them as identifiers. For example they cannot be used for Docker image names. A UUID is a valid `id`. A regular expression to validate identifiers is `/^[^-_.][a-zA-Z0-9._-]+[^-_.]$/`.

The main and display file MAY be defined in root-level nodes named `main` and `display` respectively. If they are not defined and multiple documents use the name `main.[ext]` or `display.[ext]`, an implementation SHOULD use the first file in alphabetical order.

Example of ERC configuration file with user-defined main and display files

```
id: b9b0099e-9f8d-4a33-8acf-cb0c062efaec
spec_version: 1
main: workflow.Rmd
display: paper.html
```

Additionally, related resources such as a related publication can be stated with the `relatedIdentifier` element field. A related identifier SHOULD be a globally unique persistent identifier and SHOULD be a URI.

Author and license metadata

The main document MUST include information about the authors. It SHOULD contain this information in a structured way so it can be parsed by tools supporting ERCs.

Note

An example for structured metadata is markup with author names and affiliations in the header of the main document.

The file `erc.yml` MUST contain a first level node `licenses` with licensing information for contained artefacts. Each of these artefacts, e.g. code or data, have distinct requirements so it must be possible to apply different licenses.

The node `licenses` MUST have four child nodes: `text`, `data`, `code`, and `metadata`.

Note

There is currently no mechanism to define the licenses of all the used libraries and software in a structured format. Manual creation would be tedious. Tools for automatic creation of ERC may add such detailed licensing information and define additional metadata elements.

The content of each of these child nodes MUST be a string with one of the following contents:

- *license identifier* as defined by the Open Definition Licenses Service
- *name of file* with either documentation on licensing or a full license text

Example for common licenses

```
id: b9b0099e-9f8d-4a33-8acf-cb0c062efaec
spec_version: 1
licenses:
    code: Apache-2.0
    data: ODbL-1.0
    text: CC0-1.0
    metadata: CC0-1.0
```

Example for non-standard licenses

```
id: b9b0099e-9f8d-4a33-8acf-cb0c062efaec
spec_version: 1
licenses:
    code: Apache-2.0
    data: data-licenses.txt
    text: "Creative Commons Attribution 2.0 Generic (CC BY 2.0)"
    metadata: "see metadata license headers"
```

Runtime manifest and image

The ERC uses Docker to define, build, and store the nested runtime environment.

Runtime image

The *runtime environment or image* MUST be represented by a Docker image `v1.2.0`.

Note

A concrete implementation of ERC may choose to rely on constructing the runtime environment from the manifest when needed, e.g. for export to a repository, while the ERC is constructed.

The base directory MUST contain a tarball.

The image MUST have a tag `erc:<erc identifier>`, for example `erc:b9b0099e-9f8d-4a33-8acf-cb0c062efaec`.

The image file MAY be compressed.

The image archive file name MUST be `image` with an appropriate file extension, such as `.tar`, `.tar.gz` (if a gzip compression is used for the archive) or `.bin`, and have an appropriate mime type, e.g. `application/vnd.oci.image.layer.tar+gzip`.

Note

Before exporting the Docker image, it should be build from the runtime manifest, including the tag which can be used to identify the image, for example:

```
docker build --tag erc:b9b0099e-9f8d .
docker images erc:b9b0099e-9f8d
docker save erc:b9b0099e-9f8d &gt; image.tar
# save with compression:
docker save erc:b9b0099e-9f8d | gzip -c &gt; image.tar.gz
```

Do *not* use `docker export`, because it is used to create a snapshot of a container, which must not match the Dockerfile anymore as it may have been manipulated during a run.

Runtime manifest

The *runtime manifest* MUST be represented by a valid `Dockerfile`, see Docker builder reference.

The file MUST be named `Dockerfile`.

The Dockerfile MUST contain the build instructions for the runtime environment and MUST have been used to create the image saved to the runtime image. The build SHOULD be done with the option `--no-cache=true`.

The Dockerfile MUST NOT use the `latest` tag in the instruction `FROM`.

Note

The "latest" tag is merely a convention to denote the latest available image, so any tag can have undesired results. Nevertheless, using an image tagged "latest" makes it much more likely to change over time. Although there is no guarantee that images tagged differently, e.g. "v1.2.3" might not change as well, using such tags shall be enforced here.

The Dockerfile SHOULD contain the label `maintainer` to provide authorship information.

The Dockerfile MUST have an active instruction `CMD`, or a combination of the instructions `ENTRYPOINT` and `CMD`, which executes the packaged analysis.

The Dockerfile MUST contain a `VOLUME` instruction to define the mount point of the ERC base directory within the container. This mount point MUST be `/erc` and the bind MUST be configured as with *read and write access*. Implementations

SHOULD make sure an execution does not interfere with original uploaded files, but a write access is required to store the created display file outside of the container.

The Dockerfile MUST contain a `WORKDIR` instruction with the value `/erc`.

The Dockerfile SHOULD NOT contain a `COPY` or `ADD` command to include data, code or text from the ERC into the image. These commands MAY be used to copy code or libraries which must be available during the image build.

The Dockerfile SHOULD NOT contain `EXPOSE` instructions.

System environment

The following *system environment configurations* MUST be provided as nodes under the root-level node `execution`:

- (if applicable) *kernel*, node `kernel`

The following *system environment configurations* are available within the runtime image metadata and therefore not be replicated in the ERC configuration file.

- *operating system*, node `os`,
- *architecture*, node `architecture`
- *runtime software version*, node `DockerVersion` in output of `docker inspect` and node `docker_version` in image metadata JSON file (cf. source code).

Accessing system environment configurations from image metadata in a saved image tarball

`manifest.json` contains a list of the layers and the config as the name of the configuration file. The image metadata is in the `<image id>.json` file in the root directory of the tarball. The following commands show how to extract the image metadata file from the tarball and print the relevant properties to the console using the JSON cli tool jq.

```
$ tar -xf image.tar --wildcards --no-anchored '[!manifest]*.json'
$ cat *.json | jq '.architecture, .os, .docker_version'
"amd64"
"linux"
"17.05.0-ce"
```

Together the image metadata and ERC configuration file provide all properties of the underlying system environment. An implementation SHOULD notify the user if the required system environment is incompatible with the implementation's capabilities.

System environment incompatibilities

A partially incompatible system environment, especially a different kernel version, may still produce the desired result, as breaking changes are very rare. An implementation could utilise semantic versioning to improve its compatibility tests. An incompatible operating system, e.g. `linux` vs. `windows`, and architecture, e.g. `amd64` or `arm/v7`, are likely to fail.

Example of ERC configuration file with user-defined kernel and excerpt from runtime image metadata

ERC configuration file

```
id: b9b0099e-9f8d
spec_version: 1
execution:
    kernel: `4.13.0-32-generic`
```

Image metadata (excerpt) (results of an `docker image inspect` call):

```
[
    {
        "Id": "sha256:87362162878143c5e10e94a6ec9b7e925b...",
        "RepoTags": [],
        "RepoDigests": [],
        "Parent": "sha256:a280c143ff833d99274e96bbcfdc86...",
        "Created": "2018-02-15T15:18:42.623467682Z",
        "Container": "840b75b48121012a0847bbae148ed96df7...",
        "ContainerConfig": { ... },
        "DockerVersion": "17.05.0-ce",
        "Author": "&lt;https://o2r.info&gt;",
        "Config": { ... },
        "Architecture": "amd64",
        "Os": "linux",
        [...]
    }
]
```

Image metadata (excerpt) (content of `<image id>.json` from `image.tar`): “‘json { “architecture”: “amd64”, “config”: { . . . , “Labels”: { “maintainer”: “o2r” } }, “container”: “747198d654630530c2a6523abbc19e41d7fcf977833c6854a2a48fb11b8c607c”, “container_config”: { . . . }, “created”: “2018-03-08T15:24:20.164740334Z”, “docker_version”: “17.05.0-ce”, “history”: [. . .], “os”: “linux”, “rootfs”: { “type”: “layers”, “diff_ids”: [“sha256:8568818b1f7f534832b393c531edfcb4a30e7eb40b573e68fdea90358987231f”, “sha256:fccd38ea8016190426aa7ef4baba29b0c92de1ee863c3460a34151695fbcba08”, “sha256:cf52051fff5bb6430c972ef822d435e9b5242117398b43c6d36f1ed71d978a94”, “sha256:5535e4fbfa3ed182d3cc87bfe643f87801c91be6c171535675effb4efc8c1e5a”, “sha256:9d55d57e41e02115f48e428a880d88d7bf0af993a232d0c967cc17f012e2e250”] } }

```
```

Execution

The configuration file MUST provide enough information to for implementations to create the *commands* for execution of the runtime image and to provide access to the data and software in the ERC. Implementations MUST support Docker Engine API `v1.35` (or compatible).

Making data, code, and text available within container

The runtime environment image contains all dependencies and libraries needed by the code in an ERC. Especially for large datasets, it in unfeasible to replicate the complete dataset contained within the ERC in the image. For archival, it can also be confusing to replicate code and text, albeit them potentially being relatively small in size, within the container.

Therefore a host directory MUST be **mounted** (also "bind-mounted") into the compendium container at runtime using a data volume.

Example Dockerfile

In this example we use a *Rocker* base image to reproduce computations made in R.

```
```Dockerfile
FROM rocker/r-ver:3.3.3

RUN apt-get update -qq \
    && apt-get install -y --no-install-recommends \
    ## Packages required by R extension packages
    # required by rmarkdown:
    lmodern \
    pandoc \
    # for devtools (requires git2r, httr):
    libcurl4-openssl-dev \
    libssl-dev \
    git \
    # for udunits:
    libudunits2-0 \
    libudunits2-dev \
    # required when knitting the document
    pandoc-citeproc \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

# install R extension packages
RUN install2.r -r "http://cran.rstudio.com" \
      rmarkdown \
      ggplot2 \
```
```

```
      devtools \
      && rm -rf /tmp/downloaded_packages/ /tmp/*.rd

# Save installed packages to file
RUN dpkg -l > /dpkg-list.txt

LABEL maintainer=o2r \
  description="This is an ERC image." \
    info.o2r.bag.id="123456"

VOLUME ["/erc"]
WORKDIR ["/erc"]

ENTRYPOINT ["sh", "-c"]
CMD ["R --vanilla -e \"rmarkdown::render(input = '/erc/myPaper.rmd', \
    output_dir = '/erc', output_format = rmarkdown::html_document())\""]
```

Main and display file in the container

The fixed mount point have the advantage that users and tools can be sure the main and display files are usually available at `/erc/main.Rmd` and `/erc/display.html` respectively.

Default execution

If no execution information is provided, then the implementation MUST assume an unconfigured Docker control flow for loading and executing the nested runtime environment is sufficient. Unconfigured means that NO configuration besides providing a mount of the compendium files (see previous section) MAY be applied.

The control statements for Docker executions comprise `load`, for importing an image from the archive, and `run` for starting a container of the loaded image. Both control statements MUST be configured by using nodes of the same name under the root-level node `execution` in the ERC configuration file. Based on the configuration, an implementation can construct the respective runtime software's commands, i.e. `docker load` and `docker run`, using the correct image file name and further parameters (e.g. performance control options).

Constructing the execution commands

The Docker CLI commands constructed based on configuration file for ERC with ID `b9b0099e-9f8d` could be as follows. In this case the implementation uses `-it` to pass stdout streams to the user and adds an identifier for the container using `--name`.

```
docker load --input image.tar
```

```
docker run -it --name run_b9b0099e \
    --volume /storage/erc/abc123:/erc \
    erc:b9b0099e-9f8d
```

The output of the container during execution MAY be shown to the user to convey detailed information to users.

Adjusted execution

Two means MAY be used to adjust the execution of a compendium: **environment variables** and **bind mounts**.

Environment variables can be set for containers at runtime. They overwrite variables that are defined within the image and thus SHOULD be used sparsely, for example only when the same configuration can not be achieved within the main file, and only to *increase reproducibility*.

The MUST NOT be used for manipulating the compendium's workflow instead of using UI bindings.

Environment variable use case: Time zone

A possible use case for environment variables can be setting the time zone. When the display file contains text output of times and timestamps, running the analysis on a machine with a different time zone may wrongly cause errors during checking. While a careful author can cover this within the main file via settings or controlling output, she may also be offered during a creation workflow to freeze the timezone. The following command sets the system time zone to `CET`.

```
docker run -it --name run_b9b0099e \
    --volume /storage/erc/abc123:/erc --env TZ=CET \
    erc:b9b0099e-9f8d
```

In addition to the mandatory mount of all compendium files, *bind mounts* MAY be added to replace specific files for substitution.

The mounts MUST be configured in a list node `bind_mounts` under the root-level node `execution` in the ERC configuration file. Implementations SHOULD apply them in the same order as given in the configuration file. Each mount MUST include the following nodes:

- `source`: mount source file or directory.
- `destination`: mount target path within the container; MUST be an absolute path.

The binds used for substitution MUST be configured as read only.

If a list of mounts is configured, it MAY NOT include the mandatory bind mount.

Example: data file replacement with bind mounts

The following example includes an explicit definition of the mandatory mount to `/erc` and an overlay bind mount of a CSV file.

```
id: b9b0099e-9f8d
spec_version: 1
execution:
    bind_mounts:
        - source: '/storage/erc/abc123'
          destination: /erc
        - source: /storage/erc/other/input_data/fixed.csv
          destination: /erc/data.csv
```

It can be translated by an implementation to the following bind string:

```
/storage/compendium/other123/input_data/fixed.csv:/erc/data.csv:ro
```

More on mounts and binds

See Docker API specification section Create a container > `HostConfig` > `Binds`/`Mounts`.

R workspaces

ERC support the R software environment for statistical computing and graphics.

Structure

The structure (file names for data, directories, etc.) within the ERC are intentionally unspecified. However, the content's structure MAY follow conventions or be based on templates for organizing research artifacts.

If a convention is followed then it SHOULD be referenced in the ERC configuration file as a node `convention` section. The node's value can be any text string which uniquely identifies a convention, but a URI or URL to either a human-readable description or formal specification is RECOMMENDED.

A non-exhaustive list of potential conventions and guidelines *for R* is as follows:

- Ben Marwick's `rrtools`
- ROpenSci rrrpkg
- Jeff Hollister's manuscriptPackage
- Carl Boettiger's template
- Francisco Rodriguez-Sanchez's template
- Ben Marwick's template
- Karl Broman's comments on reproducibility
- R package: "Writing R Extensions"

Example for using the ROPenSci `rrrpkg` convention

The convention is identified using the public link on GitHub.

```
id: b9b0099e-9f8d-4a33-8acf-cb0c062efaec
spec_version: 1
convention: https://github.com/ropensci/rrrpkg
```

R Markdown main file

The ERC's *main file* for R-based analyses SHOULD be R Markdown.

If the main file is R Markdown, it SHOULD include basic metadata in its YAML front matter: author(s), title, date, et cetera.

The main document SHOULD NOT contain code that loads pre-computed results from files, but conduct all analyses, even costly ones, during document weaving.

The document MUST NOT use `cache=TRUE` on any of the code chunks (see `knitr` options. While the previously cached files (`.rdb` and `.rdx`) MAY be included, they SHOULD NOT be used during the rendering of the document.

Note

A popular alternative solution is Sweave with the `.Rnw` extension, which is still widely used for vignettes. R Markdown was chosen of LaTex for its simplicity for users who are unfamiliar with LaTeX.

Fixing the environment in code

The time zone MUST be fixed to `UTC` Coordinated Universal Time) to allow validation of output times (potentially broken by different output formats) by using the following code within the RMarkdown document, or other code to that effect.

```
Sys.setenv("TZ" = "UTC")
```

The manifest file (i.e. `Dockerfile`) MUST run a plain R session without loading `.RData` files or profiles at startup, i.e. use `R --vanilla`.

Interactive ERC

Enabling interaction with the contents of an ERC is a crucial goal of this specification (see Preface). Therefore this section defines metadata to support two goals:

- aide inspecting users to identify core functions and parameters of an analysis, and

- allow supporting software tools to create interactive renderings of ERC contents for manipulation.

These goals are manifested in the **UI bindings** as part of the ERC configuration file under the root level property `ui_bindings`. The license of code specifically included to support UI bindings MUST be the same as the `code` license.

An ERC MUST denote if UI bindings are present using the boolean property `interactive`. If the property is missing it defaults to `false`. An implementation MAY use the indicator `interactive: true` to provide other means of displaying the display file.

Example for minimal interaction configuration

```yaml
id: b9b0099e-9f8d-4a33-8acf-cb0c062efaec
spec_version: 1
ui_bindings:
  interactive: true
```

An ERC MAY embed multiple concrete UI bindings. Each UI binding is represented by a YAML dictionary.

It MUST comprise a purpose and a widget using the fields `purpose` respectively `widget` (both of type string). The values of these fields SHOULD use a concept of an ontology to clearly identify their meaning.

A *purpose* defines the user's intention, for example manipulating a variable or inspecting dataset or code. A *widget* realizes the purpose with a concrete interaction paradigm chosen by the author, for example an input slider, a form field, or a button.

For each widget, implementations MAY use the properties `code`, `data`, and `text` to further describe how a specific UI binding acts upon the respective part of the ERC.

Example of two UI bindings

```yaml
id: b9b0099e-9f8d-4a33-8acf-cb0c062efaec
spec_version: 1
ui_bindings:
  interactive: true
  bindings:
    - purpose: http://.../data-inspection
      widget: http://.../tabular-browser
      code: [...]
      data: [...]
      text: [...]
    - purpose: http://.../parameter-manipulation
      widget: http://.../dropdown
```

Preservation of ERC

This section places the ERC in the context of preservation workflows by defining structural information and other metadata that guarantee interpretability and enable the bundling of the complete ERC as a self-contained, archivable digital object.

Archival bundle

For the purpose of transferring and storing a complete ERC, it MUST be packaged using the BagIt File Packaging Format (V0.97) (BagIt) as the outer container. BagIt allows to store and transfer arbitrary content along with minimal metadata as well as checksum based payload validation.

The remainder of this section comprises

- a description of the outer container,
- a BagIt profile,
- a package leaflet, and
- secondary metadata files.

BagIt outer container

The ERC base directory MUST be the BagIt payload directory `data/`. The path to the ERC configuration file subsequently MUST be `<path-to-bag>/data/erc.yml`.

The bag metadata file `bagit.txt` MUST contain the case-sensitive label `Is-Executable-Research-Compendium` with the case-insensitive value `true` to mark the bag as the outer container of an ERC.

Implementations SHOULD use this field to identify an ERC.

Example `bagit.txt`

```
Payload-Oxum: 2172457623.43
Bagging-Date: 2016-02-01
Bag-Size: 2 GB
Is-Executable-Research-Compendium: true
```

Example file tree for a bagged ERC

```
&#9500;&#9472;&#9472; bag-info.txt
&#9500;&#9472;&#9472; bagit.txt
&#9500;&#9472;&#9472; data
&#9474;   &#9500;&#9472;&#9472; 2016-07-17-sf2.Rmd
&#9474;   &#9500;&#9472;&#9472; erc.yml
&#9474;   &#9500;&#9472;&#9472; metadata.json
&#9474;   &#9500;&#9472;&#9472; Dockerfile
&#9474;   &#9492;&#9472;&#9472; image.tar
```

```
&#9500;&#9472;&#9472; manifest-md5.txt
&#9492;&#9472;&#9472; tagmanifest-md5.txt
```

BagIt profile - DRAFT

Note

The elements of the o2r Bagit Profile is yet to be specified. This section is under development. Current BagIt tools do not include an option to add a BagIt Profile automatically.

A BagIt Profile as outlined below would make the requirements more explicit. The BagIt Profiles Specification Draft allows users of BagIt bags to coordinate additional information, attached to bags.

```
{
  "BagIt-Profile-Info":{
  "BagIt-Profile-Identifier":"https://o2r.info/erc-bagit-v1.json",
  "Source-Organization":"o2r.info",
  "Contact-Name":"o2r Team",
  "Contact-Email":"o2r@uni-muenster.de",
  "External-Description":"BagIt profile for packaging
        executable research compendia.",
  "Version":"1"
  },
  "Bag-Info":{
    "Contact-Name":{
       "required":true
    },
    "Contact-Email":{
       "required":true
    },
    "External-Identifier":{
       "required":true
    },
    "Bag-Size":{
       "required":true
    },
    "Payload-Oxum":{
       "required":true
    }
  },
  "Manifests-Required":[
    "md5"
  ],
  "Allow-Fetch.txt":false,
  "Serialization":"optional",
```

```
  "Accept-Serialization":[
     "application/zip"
  ],
  "Tag-Manifests-Required":[
    "md5"
  ],
  "Tag-Files-Required":[
     ".erc/metadata.json",
     "erc.yml"
  ],
  "Accept-BagIt-Version":[
     "0.96"
  ]
}
```

Package leaflet

Each ERC MUST contain a package leaflet, describing the schemas and standards used. Available schema files are supposed to be included with the ERC, if available (licenses for these schemas may apply).

Example package leaflet

```
{
    "standards_used": [
        {
            "o2r": {
                "map_description": "maps raw extracted metadata to
                    o2r schema compliant metadata",
                "mode": "json",
                "name": "o2r",
                "outputfile": "metadata_o2r.json",
                "root": ""
            }
        },
        {
            "zenodo_sandbox": {
                "map_description": "maps o2r schema compliant MD to
                    Zenodo Sandbox for deposition creation",
                "mode": "json",
                "name": "zenodo_sandbox",
                "outputfile": "metadata_zenodo_sandbox.json",
                "root": "metadata"
            }
        }
    ]
```

```
}
```

Elements used for each schema standard used are contributed via the MD mapping files in the o2r meta tool suite.

Secondary metadata files

The ERC as an object can be used in a broad range of cases. For example, it can be an item under review during a journal publication, it can be the actual publication at a workshop or conference or it can be a preserved item in a digital archive. All of these have their own standards and requirements to apply, when it comes to metadata.

These metadata requirements *are not* part of this specification, but the following conventions are made to simplify and coordinate the variety.

Metadata specific to a particular domain or use case MUST replicate the information required for the specific case in an independent file. Domain metadata SHOULD follow domain conventions and standards regarding format and encoding of metadata. Duplicate information is accepted, because it lowers the entry barrier for domain experts and systems, who can simply pick up a metadata copy in a format known to them.

Metadata documents of specific use cases MUST be stored in a directory `.erc`, which is a child-directory of the ERC base directory.

Metadata documents SHOULD be named according to the used standard/model, format/encoding, and version, e.g. `datacite40.xml` or `zenodo_sandbox10.json`, and SHOULD use a suitable mime type.

Requirements of secondary metadata

In order to comply to their governing schemas, secondary metadata must include the mandatory information as set by 3rd party services. While the documentation of this quality is a perpetual task, we have gathered the information most relevant our selection of connected services.

Zenodo

- Accepts metadata as `JSON`.
- Mandatory elements:
  - Upload Type (e.g. Publication)
  - Publication Type
  - Title
  - Creators
  - Description
  - Publication Date
  - Access Right
  - License

DataCite (4.0)

- Accepts metadata as `XML`.
- Mandatory elements:
  - Identifier
  - Creator
  - Title
  - Publisher
  - Publication Year
  - Resource Type

Development bundle

While complete ERCs are focus of this specification, for collaboration and offline inspection it is useful to provide access to parts of the ERC. To support such use cases, a *development bundle* MAY be provided by implementations. This bundle most importantly would not include the *runtime image*, which is potentially a large file.

The *development bundle* SHOULD always include the *main file* and (e.g. by choice of the user, or by an implementing platform) MAY include other relevant files for reproduction or editing purposes outside of the runtime environment, such as input data or the *runtime manifest* for manual environment recreation.

Content metadata

The current JSON dummy file to visualises the properties. These elements SHOULD be filled out as good as possible in the user interface.

```
{
    "access_right": "open",
    "author": [{
        "name": null,
        "affiliation": [],
        "orcid": null
    }],
    "codefiles": [],
    "community": "o2r",
    "depends": [{
        "identifier": null,
        "version": null,
        "packageSystem": null
    }],
    "description": null,
    "file": {
        "filename": null,
        "filepath": null,
```

```
        "mimetype": null
    },
    "generatedBy": null,
    "identifier": {
        "doi": null,
        "doiurl": null,
        "reserveddoi": null
    },
  "inputfiles": [],
    "keywords": [],
    "license": {
        "text": None,
        "data": None,
        "code": None,
        "md": None
    },
    "paperLanguage": [],
    "paperSource": null,
    "publicationDate": null,
    "recordDateCreated": null,
    "softwarePaperCitation": null,
    "spatial": {
        "files": [],
        "union": []
    },
    "temporal": {
        "begin": null,
        "end": null
    },
    "title": null,
    "upload_type": "publication",
    "viewfiles": []
}
```


The path to the o2r metadata file MUST be

`<path-to-bag>/data/metadata_raw.json`

and the refined version `metadata_o2r.json`.

Description of o2r metadata properties

- `access_right` *String.*
- `creators` *Array of objects.*
- `creators.name` *String.*
- `creators.orcid` *String.*
- `creators.affiliation` *String.*

- `codefiles` *Array of strings* List of all files of the recursively parsed workspace that have an extension belonging to a ("R") codefile.
- `communities` *Array of objects* prepared zenodo MD element
- `communities[0].identifier` *String.* Indicating the collection as required in zenodo MD, default "o2r".
- `depends` *Array of objects.*
- `depends.operatingSystem` *String.*
- `depends.identifier` *String.*
- `depends.packageSystem` *String.* URL
- `depends.version` *String.*
- `description` *String.* A text representation conveying the purpose and scope of the asset (the abstract).
- `displayfile` *String.* The suggested file for viewing the text of the workspace, i.e. a rendering of the suggested mainfile.
- `displayfile_candidates` *Array of strings.* An unsorted list of candidates for displayfiles.
- `identifier` *Object.*
- `inputfiles` *Array of strings.* A compiled list of files from the extracted workspace that is called or used in the extracted code of the workspace.
- `interaction` TBD
- `keywords` *Array of strings.* Tags associated with the asset.
- `license` *Object.* License information for the entire ERC.
- `license.code` *String.* License information for the code included.
- `license.data` *String.* License information for the data included.
- `license.md` *String.* License information for the metadata included. Should be cc0 to include in catalogues.
- `license.text` *String.* License information for the text included.
- `mainfile` *String.* The suggested main file of workspace
- `mainfile_candidates` *Array.* Unsorted list of mainfile candidates of the workspace.
- `paperLanguage` *Array of strings.* List of guessed languages for the workspace.
- `publication_date` *String.* The publication date of the paper publication as ISO8601 string.
- `publication_type` *String.*
- `related_identifier` *String.*
- `spatial` *Object.* Spatial information of the workspace.
- `spatial.files` *Array of objects.*
- `spatial.union` *Array of objects.*
- `temporal` *Object.* Aggregated information about the relevant time period of the underlying data sets.
- `temporal.begin`
- `temporal.end`
- `title` The distinguishing name of the paper publication.
- `upload_type` *String.* Zenodo preset. Defaults to "publication".

ERC checking

Procedure

A core feature ERCs are intended to support is comparing the output of an ERC executions with the original outputs. Therefore checking an ERC always comprises two steps: the execution and the comparison.

The files included in the comparison are the *comparison set*. The comparison set MUST include the display file. It MAY include any other files. An implementation MUST communicate the comparison set to the user as part of a check.

Previous to the check, an implementation SHOULD conduct a basic validation of the outer container's integrity, i.e. check the file hashes. The output of the image execution MAY be shown to the user to convey detailed information on progress or errors.

Comparison set file

The ERC MAY contain a file named `.ercignore` in the base directory to define the comparison set.

Its purpose is to provide a way to efficiently exclude files and directories from checking. If this file is present, any files and directories within the outer container which match the patterns within the file `.ercignore` will be excluded from the checking process. The check MUST NOT fail when files listed in `.ercignore` are failing comparison.

The file MUST be UTF-8 (without BOM) encoded. The newline-separated patterns in the file MUST be Unix shell globs and support the prefix `!` (see `man gitignore`). For the purposes of matching, the root of the context is the ERC's base directory.

Lines starting with `#` are treated as comments and MUST be ignored by implementations.

Example `.ercignore` file

```
# comment
.erc
*/temp*
data-old/*
```

Recommended `.ercignore` file comparing only the display file

Assuming the display file is named `display.html`, the following '.ercignore ignores all files except the display file.

```
!display.html
```

Note

If using md5 file hashes for comparison, the set could include plain text files, for example the `text/*` media types (see IANA's full list of media types. Of course the comparison set should include files which contain results of an analysis.

Comparing the display file

Readers make the ultimate decision about the results of a check, but they tools SHOULD assist them as much as possible to compare the display file generated by the original author with the display file generated during a check, manipulation, or substitution.

Tools MAY include other files than the non-display files in a check, but authors SHOULD make sure that the display files contains suitable computational results to judge the outcome of the analysis.

Comparing text output SHOULD utilise established file comparison and difference, or "diff tools", which the text-based HTML format allows very well.

Comparing graphics included in the display file SHOULD also provide visual comparison results, e.g. on a pixel-by-pixel basis or even conceptual differences of images ("perceptual hashes").

Comprehensive example of erc.yml

The following example shows all possible fields of the ERC specification with example values.

```yaml
id: b9b0099e-9f8d-4a33-8acf-cb0c062efaec
spec_version: 1
main: paper.rmd
display: paper.html
execution:
  bind_mounts: ...
licenses:
  code: MIT
  data: ODbL-1.0
  text: "data_licenses_info.pdf"
  metadata: CC0-1.0
convention: https://github.com/ropensci/rrrpkg
ui_bindings:
  interactive: true
  bindings:
    - purpose: http://.../data-inspection
      widget: http://.../tabular-browser
      code: [...]
```

```
    data: [...]
    text: [...]
  - purpose: http://.../parameter-manipulation
    widget: http://.../dropdown
```

Glossary

(Computational) Analysis

A scientific workflow that is to be preserved in an ERC. It conducts a number of operations on data and generates an output (text, numbers, plots).

Bag

See BagIt specification.

> A set of opaque data contained within the structure defined by this specification.

Compendium contents

See ERC contents

Container

A receptacle holding a collection of things ("payload" or "contents"). In the context of this specification, two containers are distinguished: runtime container and outer container.

Check

A subconstituent of *Examine*. Checking an ERC is a syntactical validation, which may be largely automated by a software tool reporting the check result and potential errors. A check comprises (a) the validation of a concrete ERC against the ERC specification, e.g. are required files and metadata fields present, and (b) an execution of the contained analysis. The execution includes a comparison of the result files in the just executed inner container with the result stored in the outer container.

Create

One of the major constituents of ERC interaction. The user can create an ERC by following the technical instructions included in the Specification (ERC Spec) or use the o2r reproducibility service. For more information, see erc-spec/user-guide/creation/.

Discover

One of the major constituents of ERC interaction. Discovery comprises the findability of the ERC as well as the exploration of its features, e.g. time and space driven search operations.

ERC

Executable Research Compendium, see article.

ERC contents

See workspace.

ERC metadata

Schema compliant information about the ERC, its contents and creators.

Examine

One of the major constituents of ERC interaction. It comprises *Check*, *Inspect*, *Manipulate* and *Substitute*. To examine an ERC means to explore its contents in depth, i.e. check the reproduced version, inspect text, code and data, manipulate interactive elements, as well as exchange input data.

Inner container

See runtime container

Inspect

A subconstituent of *Examine*. Inspection includes looking at all the contents of an ERC, such as code or data files, and metadata documents. A user conducting inspection evaluates the meaning of the ERC's artifacts.

Dependency

If software/library `X` is required by software/tool `Y` to function properly, then `Y` has the dependency `X` or `X` is a dependency of `Y`. Collecting all the right dependencies, which work with each other, can be a hard problem, see Dependency hell. Dependencies can be packages of the same language (like R extension package requiring another R extension package) or system dependencies (like a Python library from PyPI requiring a specific library available via the operating system package manager).

Display file

The file in the container that a reader software uses as the first display to a user to read text and explore graphics. The entry point for examination.

Manipulate

A subconstituent of *Examine*. A manipulation comprises interactive changing of selected, pre-defined parameters that influence the computation packaged in an ERC. For example, the number of layers in a neural network, the size/selection method of the training dataset in supervised machine learning, or the variogram model of geostatistical kriging. These parameters are defined via UI bindings.

OAIS

The Open Archival Information System and its reference model.

Outer container

Term used to distinguish the "outer" Bag from the embedded runtime container.

Reproducible, Reproducibility, Replication

See section 2.1 "Definition of Reproducibility".

Runtime container

A Linux container, more specifically a Docker container, which is a special format to package an application and its dependencies. For usage in this specification, the runtime container can be used to provide the computational environment needed for execution of an ERC's workflow. It is a transferable snapshot of the authors computer, but also documents the software used by an ERC.

Runtime manifest

A formal description or recipe for a runtime container, more specifically a Dockerfile.

> *Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.* source

Substitute

A subconstituent of *Examine*. During a substitution, compatible parts of an ERC are exchanged, e.g. similar data sets for a given analysis, or exchanging an analysis script. A substitution process usually creates a new ERC based on two input ERCs: the *base ERC* and the *overlay ERC*. One or several data or code files from the *overlay ERC* replace corresponding files in the *base ERC*, to create a new ERC.

UI bindings

Formal descriptions of parameters and interactions used during *Examine*. The UI bindings are included in the configuration file and may be created manually or with help of a user-friendly wizard.

Workspace

The files created by the author of the original analysis. The workspace is packaged together with ERC metadata, runtime container and runtime manifest in the payload directory of the outer container.