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, http://o2r.info.

Build version: `2056534`

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**:

- ERC creation
- ERC examination
- ERC template

Are you a **developer** and want to build applications for ERCs? Read **user guides for developers**:

- Developer guide

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 2056534 @ 2017-11-24T07:32:41Z

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 analyses (cf. computational science). It carries technical and conceptual details on how to implement the reproducibility specifications and is as such most suitable **for developers**. **Authors** may feel more comfortable with the *user guides*.

These analyses typically comprise a workspace on a researcher's computer, that contains *data*, *code*, third party software or libraries, and outputs research results such as plots. Code and libraries are required in executable form to re-do a specific analysis. Research is only put into a context by a *textual* publication, 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 read by humans. The latter is often derived, or "rendered" from the former and can be static, visual, or even interactive following a trend towards more interactivity between reader and scientific publication.

Putting all of these elements in a self-contained bundle allows examining, reproducing, transferring, archiving, and formal validation of computational research results. The ERC specification also defines metadata and file structures to support these actions.

Major constituents and design goals

Three major constituents classify user interaction with ERC:

- **Create** means transforming a workspace with data, code and text into an ERC.
- **Examine** means looking at depths of an ERC, scrutinizing its contents.
- **Discover** means searching for content powered by ERC properties, such as text, content metadata, code metadata et cetera.

A core design goal is *simplicity*. This specification should not re-do something which already exists (if it is an open specification or tool). 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.

The final important notion is the one of *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.

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.

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 the user first when he 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 an 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", in the ERC base directory. The name of the archive file MUST be configured in the ERC configuration file in the node `image` under the root-level node `execution`.

The manifest MUST be stored as a text file in the ERC base directory. The name of the manifest file MUST be configured in the ERC configuration file in the node `manifest` under the root-level node `execution`.

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. This SHOULD be a URI (see rfc3986) or a UUID, Version 4.

The main and display file MAY be defined in root-level nodes named `main` and `display` respectively, if they differ from the default file names. 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: the_paper_document.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.

Control statements

The configuration file MUST contain statements to control the runtime container.

These statements MUST be in an array under the root-level node `execution` in the ERC configuration file in the order in which they must be executed.

Implementations SHOULD support a list of bash commands as control statements. These commands are given as a list under the node `cmd` under the root-level node `execution`. If extensions use non-bash commands, they MUST define own nodes under the `execution` node and SHOULD define defaults.

The execution statements MAY ensure the re-computation being independent from the environment, which may be different depending on the host of the execution environment. For example, the time zone could be fixed via an environment variable `TZ=CET`, so output formatting of timestamps does not break checking. This is in addition to ERC authors handling such parameters at a script level.

Example for control statements

```
id: b9b0099e-9f8d-4a33-8acf-cb0c062efaec
spec_version: 1
execution:
  cmd:
    - `./prepare.sh --input my_data`
    - `./execute.sh --output results --iterations 3`
```

License metadata

The file `erc.yml` MUST contain a first level node `licenses` with licensing information for the code, data, and text contained. Each of these three have distinct requirements, hence different licenses need to be applied.

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

Note

There is currently no mechanism to define the licenses of the used libraries, as manual creation would be tedious. Tools for automatic creation of ERC may add such detailed licensing information and define an extension to the ERC

The content of each of these child nodes MUST have one of the following values:

- text string with license identifier or license text. This SHOULD be a standardized identifier of an existing license as defined by the Open Definition Licenses Service, or
- a dictionary of all files or directories and their respective license, each of the values following the previous statement. The node values are the file paths relative to the base directory.

Example for global licenses

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

Example using specific licenses for files

```
---
id: b9b0099e-9f8d-4a33-8acf-cb0c062efaec
spec_version: 1
licenses:
  code:
    others_lib.bin: MIT
    my_code.c: GPL-3.0
  data:
    facts.csv: ODbL-1.0
  text:
    README.md: CC0-1.0
        paper.Rmd: CC-BY-4.0
      ui_bindings: CC0-1.0
      metadata: CC0-1.0
```

Note

It IS NOT possible to assign one license to a directory and override that assignment or a single file within that directory, NOR IS it possible to use globs or regular expressions.

Comprehensive example of erc.yml

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

```
id: b9b0099e-9f8d-4a33-8acf-cb0c062efaec
spec_version: 1
main: paper.rmd
display: paper.html
execution:
  cmd: "Rscript -e 'rmarkdown::render(input = \"paper.Rmd\", output_format = \"html\")'"
licenses:
  code:
    others_lib.bin: MIT
    my_code.c: GPL-3.0
  data:
    facts.csv: ODbL-1.0
  text:
    README.md: CC0-1.0
    paper.Rmd: CC-BY-4.0
  ui_bindings: CC0-1.0
  metadata: CC0-1.0
structure:
  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
```

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

Docker runtime

The ERC uses Docker to define, build, and store the nested runtime environment, i.e. the inner container.

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 *label* of the name `erc` with the ERC's id as value, e.g. `erc=b9b0099e-9f8d-4a33-8acf-cb0c062efaec`.

The image file MAY be compressed.

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

Note

Before exporting the Docker image, first build it from the Dockerfile, including the label which can be used to extract the image identifier, for example:

```
```bash
docker build --label erc=b9b0099e-9f8d-4a33-8acf-cb0c062efaec .
docker images --filter "label=erc=b9b0099e-9f8d-4a33-8acf-cb0c062efaec"
docker save $(docker images --filter "label=erc=1234" -q) > image.tar
# save with compression:
docker save $(docker images --filter "label=erc=1234" -q) | gzip -c >     image.tar.gz
```

Do _not_ use `docker export`, because it is used to create a snapshot of a container, which
```

The output of the image execution can be shown to the user to convey detailed information on progress or errors.

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 SHOULD NOT contain `EXPOSE` instructions.

Docker control statements

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 run-time commands, i.e. `docker load` and `docker run`, using the correct image file name and further parameters (e.g. performance control options).

Example

The following example shows default values for `image` and `manifest` and typical values for `run`.

```yml
id: b9b0099e-9f8d-4a33-8acf-cb0c062efaec
version: 1
execution:
  image: image.tar.gz
  manifest: Dockerfile
  run:
    environment:
      - TZ=CET
```

Note

The Docker CLI commands constructed based on this configuration by an implementing service could be as follows:

```bash
docker load --input image.tar
IMAGE_ID=$(docker images --filter "label=erc=b9b0099e-9f8d-4a33-8acf-cb0c062efaec" -q)
docker run -it --name run_abc123 -e TZ=CET -v /storage/erc/abc123:/erc --label user:o2r $IMA
```

`In this case the implementation uses `-it` to pass stdout streams to the user and adds some`

The only option for `load` is `quiet`, which may be set to Boolean `true` or `false`.

The only option for `run` is `environment` to set environment variables inside containers as defined in docker-compose. Environment variables are defined as a list separated by `=`.

Example for `load` and `run` properties

```
execution:
  load:
    quiet: true
  run:
    environment:
      - DEBUG=1
      - TZ=CET
```

The environment variables SHOULD be used to fix settings out of control of the contained code that can hinder successful ERC checking, e.g. by setting a time zone to avoid issues during checking.

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

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 being relatively small in size, within the container.

Therefore a host directory is mounted into a container at runtime using a data volume.

The Dockerfile SHOULD NOT contain a `COPY` or `ADD` command to include data, code or text from the ERC into the image.

The Dockerfile MUST contain a `VOLUME` instruction to define the mount point of the ERC base directory within the container. This mountpoint SHOULD be `/erc`. Implementations MUST use this value as the default. If the mountpoint is different from `/erc`, the value MUST be defined in `erc.yml` in a node `execution.mount_point`.

Example for mountpoint configuration

```
---
id: b9b0099e-9f8d-4a33-8acf-cb0c062efaec
spec_version: 1
execution:
  mount_point: "/erc"
```

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"]

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

See also: [Best practices for writing Dockerfiles](https://docs.docker.com/engine/userguide/

R workspaces

Structure

The structure within the ERC contents directory are intentionally unspecified. However, the contents 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` within the `structure` 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:

- 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

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
structure:
  convention: https://github.com/ropensci/rrrpkg
```

R Markdown main file

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

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`.

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

```
---
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

```
---
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

```
&amp;#9500;&amp;#9472;&amp;#9472; bag-info.txt
&amp;#9500;&amp;#9472;&amp;#9472; bagit.txt
&amp;#9500;&amp;#9472;&amp;#9472; data
&amp;#9474;&amp;#160;&amp;#160; &amp;#9500;&amp;#9472;&amp;#9472; 2016-07-17-sf2.Rmd
&amp;#9474;&amp;#160;&amp;#160; &amp;#9500;&amp;#9472;&amp;#9472; erc.yml
&amp;#9474;&amp;#160;&amp;#160; &amp;#9500;&amp;#9472;&amp;#9472; metadata.json
&amp;#9474;&amp;#160;&amp;#160; &amp;#9500;&amp;#9472;&amp;#9472; Dockerfile
&amp;#9474;&amp;#160;&amp;#160; &amp;#9492;&amp;#9472;&amp;#9472; image.tar
&amp;#9500;&amp;#9472;&amp;#9472; manifest-md5.txt
&amp;#9492;&amp;#9472;&amp;#9472; tagmanifest-md5.txt
```

BagIt profile

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 could make the requirements of this extension 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":"http://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": [{
        "name": "DataCite Metadata Schema 4.0",
        "name-short": "datacite40",
        "description": "The DataCite Metadata Schema is a list of core metadata properties c
        "schema-version": "4.0",
        "schema-path-local": "erc/schema/datacite40.json ",
        "schema-url": "https://schema.datacite.org/meta/kernel-4.0/metadata.xsd",
        "schema-identifier": "doi:10.5438/0013"
    }, {
        "name": "Zenodo Metadata Schema",
        "name-short": "zenodo",
        "description": "The metadata schema applicable for zenodo 2017.",
        "schema-version": null,
        "schema-path-local": "erc/schema/zenodo.json ",
        "schema-url": null,
        "schema-identifier": null
    }]
}
```


Elements used for each schema / standard used:

- `name`: The name of the schema.
- `name-short`: The abbreviated name.
- `description`: The description of the schema.
- `schema-version`: The version of the schema as stated in the corresponding official schema file.
- `schema-path-local`: The path to the local version of the schema. It may point to a translated version of the original schema, e.g. json file from xml file.
- `schema-url`: The official URL of the schema file
- `schema-identifier`: The persistent identifier for the schema/standard.

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 or platform, and the used format respectively encoding, 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

Other third party standards that will be considered comprise: *CodeMeta*, *EuDat*, *mets/mods*.

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 *under development*

Current JSON dummy to visualise the properties. It SHOULD be filled out as good as possible.

```
{
    "access_right": "open",
    "author": [{
        "name": null,
        "affiliation": [],
        "orcid": null
    }],
    "codefiles": [],
    "community": "o2r",
    "depends": [{
        "identifier": null,
        "version": null,
        "packageSystem": null
    }],
    "description": null,
    "ercIdentifier": 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,
            "uibindings": 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.json`.

Description of metadata properties

Defining explanations on the concept of each metadata element in use.

- `access_right` Modify embargo status, default is `open`.
- `author` Contains a list of authors, each containing author related information.
- `author.affiliation` A list of institutions, organizations or other groups that the creator of the asset is associated with.
- `author.name` The name of the human individual, institution, organization, machine or other entity that acts as creator of the asset.

- `author.orcid` The ORCid of the creator of the asset.
- `codefiles` A list of files, containing programm code (i.e. script files, e.g. .R files) retrieved during the extraction.
- `community` Indicates belonging to a scientific community, e.g. on a repositoy platform.
- `depends` A block for each entity that the software is directly dependent on for execution. The dependency information is designed for the identification of dependent packages within packaging systems. A depends block may describe a transitive dependency.
- `depends.identifier` An identifying name for the depending package.
- `depends.version` The computer software and hardware required to run the software.
- `depends.packageSystem` The package manager system that makes the dependency entity available.
- `description` A text representation conveying the purpose and scope of the asset (the abstract).
- `ercIdentifier` A universally unique character string associated with the asset as *executable research compendium*, provided by the o2r service.
- `file` A block for the main source file for the metadata (e.g. rmd file), generated and used by the o2r service.
- `file.filename` See above
- `file.filepath` See above
- `file.mimetype` See above
- `generatedBy` The entity, person or tool, that created the software.
- `identifier` Contains information related to persitent identifiers for the asset.
- `identifier.doi` The DOI for the asset.
- `identifier.doiurl` The resolving URL for the asset.
- `identifier.reserveddoi` The assigned but inactive DOI for the asset. Might be minted by a repository during publication.
- `inputfiles` A list of files that are loaded as resources by the main or code files of a workspace.
- `interaction` Information on interactive elements in the asset.
- `interaction.interactive` 'TRUE' if interactive elements are already included, otherwise 'FALSE'.
- `interaction.ui_binding` A block for each UI binding - extends a figure by a UI widget, e.g. for manipulation. Final structure depends on purpose.
- `interaction.ui_binding.purpose` What the UI binding is supposed to do.
- `interaction.ui_binding.widget` Which UI widget realizes the purpose.
- `interaction.ui_binding.code` A block containing source-code-specific information required to realize the UI binding.
- `interaction.ui_binding.code.filename` Name of the file including the plot function that creates the figure.
- `interaction.ui_binding.code.function` Name of the function that plots the figure.

- `interaction.ui_binding.code.functionParameter` Parameters required by the shinyInputFunction. Final set of parameters depends on UI widget.
- `interaction.ui_binding.variable` Variable that should be controlled by the UI widget.
- `interaction.ui_binding.code.shinyInputFunction` Function that incorporates the UI widgets, provided by Shiny.
- `interaction.ui_binding.code.shinyRenderFunction` Function that renders the plot after each change, provided by Shiny.
- `keywords` Tags associated with the asset.
- `license` License information for each part of the ERC.
- `license.code` License for the code part of the ERC
- `license.text` License for the text part of the ERC
- `license.data` License for the data part of the ERC
- `license.uibindings` License for the user interface bindings of the ERC
- `license.md` License for the metadata of the ERC
- `paperLanguage` A list of language codes that indicate the language of the asset, e.g. *en*.
- `paperSource` The text document file of the paper.
- `publicationDate` The publication date of the paper publication as ISO8601 string.
- `publication_type` The type of the publication. Default is `other` since the ERC may contain text, data, code and interaction widgets not depictable by other categories.
- `recordDateCreated` The date that this metadata record was created as ISO8601 string.
- `softwarePaperCitation` Related citation information for the asset, e.g. a citation of the related journal article.
- `spatial` Information about the geometric bounding box of the underlying data/software.
- `spatial.files` A Geojson object of the file-wise bounding boxes of the underlying data/software.
- `spatial.union` A Geojson object displaying the spatial properties, e.g. a bounding box of the whole data.
- `temporal` Aggregated information about the relevant time period of the underlying data sets.
- `temporal.begin` The starting point of the relevant time period.
- `temporal.end` The end point of the relevant time period.
- `title` The distinguishing name of the paper publication.
- `upload_type` The zenodo upload type, default is `publication`. This element will be removed, once the target repository is completely configurabe within the o2r shipper micro service.
- `view_file` The main display file.

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*. 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.

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. 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/*
```

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 plain text documents

…

Comparing graphics and binary output

This section outlines possibilities beyond simple comparison and incorporates "harder" to compare files and what to do with them, e.g. plots/figures, PDFs, …

Security considerations

Why are ERC not a security risk?

- the spec prohibits use of `EXPOSE`
- the containers are only executed *without* external network access using `Network: none`, see Docker CLI run documentation