

FAIR Data Publishing

Publishing a dataset as an R data package with washr

Table of contents

Welcome	4
Three ways to follow the workflow	4
What the chapters cover	4
1 Creating the package and its repository	6
1.1 Create the package	6
1.2 Put it under version control	7
1.3 Create the repository on GitHub	7
1.4 Add the check that runs on every push	9
1.5 Committing and pushing from RStudio	9
2 Preparing the data	10
2.1 Bring in the raw data	11
2.2 Clean and process the raw data	11
2.3 Write the data dictionary	11
2.4 When the data changes later	12
3 Documenting the data	13
3.1 Write the help pages	13
3.2 Complete the DESCRIPTION file	14
3.2.1 Authors	14
3.2.2 The fields washr writes	14
3.2.3 Keywords, coverage and source	15
3.3 Generate the metadata	15
4 Publishing the data	17
4.1 Write the README	17
4.1.1 Create it from the template	17
4.1.2 Write the text	17
4.1.3 Build it	18
4.2 Build the website	18
4.2.1 Add the deployment workflow	18
4.2.2 Write the configuration and build the site	19
4.2.3 Apply the brand	19
4.2.4 Check, commit and publish	20

5	Creating a release and a DOI	21
5.1	Write the citation files	21
5.2	Check that the package is ready	21
5.3	Connect the repository to Zenodo	22
5.4	Create the release	22
5.5	Bring the DOI into the package	22
5.6	Later versions	23
5.7	ETH Research Collection submission	23
6	Troubleshooting	24
6.1	Start with the readiness report	24
6.2	setup_dictionary() stops because the dictionary exists	24
6.3	setup_website() stops because README.md is missing	24
6.4	pkgdown reports “URLs not ok”	24
6.5	Git fails when pushing large files	25
6.6	R CMD check	25

Welcome

A dataset that sits as a spreadsheet in an online archive is available in principle, but hard to find, hard to understand and hard to cite. The FAIR principles ask for more. Data should be findable, accessible, interoperable and reusable. This guide walks through one way to get there. You turn a cleaned dataset into an R data package with a data dictionary, help pages, a README, a website, citation files and machine readable metadata, and you publish it with a DOI.

The guide is written for researchers at [Global Health Engineering](#) at ETH Zurich and for the [openwashdata](#) community, and it works for anyone who publishes a dataset under their own GitHub organisation. It builds on [washr](#), the R package that writes each file of the package from what you have already written. The code in this guide was last checked against washr version 1.2.0, the version recorded in the `WASHR_VERSION` file of the [source repository](#).

Three ways to follow the workflow

The same steps exist in three forms, and each form has one job.

- This guide is the narrative. It explains why each step exists and covers the parts that happen outside R, such as creating the GitHub repository and the Zenodo release. It needs no AI tools.
- The washr vignette [Get started with washr](#) is the condensed reference, with one section per function in the same order as the chapters here.
- The washr skill in the [openwashdata plugin for Claude Code](#) is the AI assisted path. It runs the same functions and drafts the descriptions, titles and README text for you to confirm.

What the chapters cover

Chapter [1](#) creates the package and connects it to GitHub. Chapter [2](#) brings in the raw data, cleans it into a tidy dataset and writes the data dictionary. Chapter [3](#) writes the help pages, completes the package metadata and generates the machine readable description. Chapter [4](#) writes the README and builds the website. Chapter [5](#) creates the release and the DOI. Chapter [6](#) collects the problems we have seen and what fixed them.

Every washr function reads what is there, merges its changes and writes only what changed, so you can run a step again after you have edited the files it wrote.

This guide is a work in progress. As more data is published, we learn more, and these insights flow back into the guide. If something is unclear or incomplete, please open an issue in the [source repository](#).

1 Creating the package and its repository

Everything you publish lives in one R package, and the package lives in one Git repository on GitHub. This chapter creates both and connects them. You need R, RStudio, Git and a GitHub account, and you need to be a member of the GitHub organisation you publish under.

Note

openwashdata packages live in the [openwashdata](#) organisation, and Global Health Engineering packages in [Global-Health-Engineering](#). If you are not a member yet, write to the team of the organisation and ask for an invitation.

Before you start, check that the [usethis](#), [devtools](#) and [washr](#) packages are installed. All three are on CRAN:

```
install.packages(c("usethis", "devtools", "washr"))
```

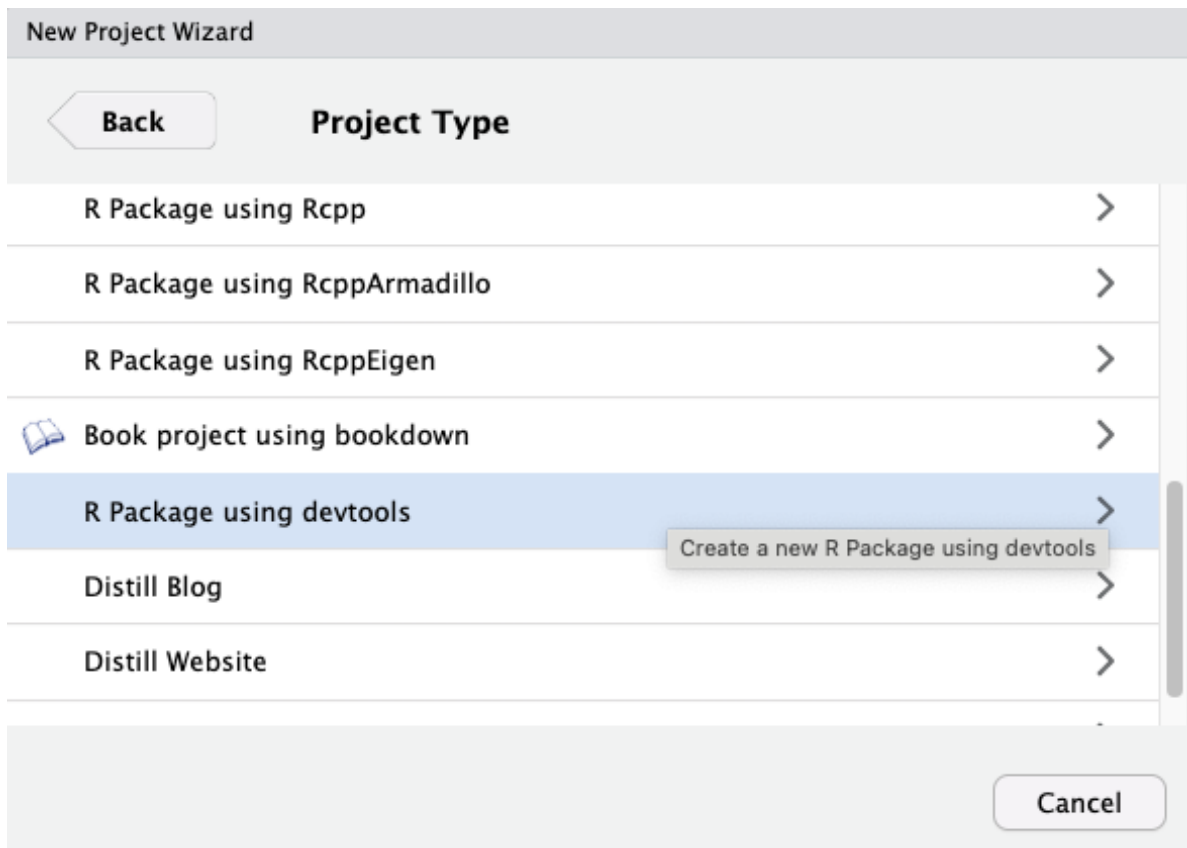
Git needs your name and email, and usethis needs to know your GitHub account. `usethis::git_sitrep()` reports what is set and what is missing, and the usethis article [Managing Git\(Hub\) credentials](#) explains the setup.

1.1 Create the package

The package name is the name of the dataset, in lower case, with no dots or underscores. Choose it with care, because it becomes the repository name, the name people install and the name in every citation. In the R console, create the package and open it as a project:

```
usethis::create_package("~/path/to/yourpackagename")
```

RStudio opens the new project. The folder holds a **DESCRIPTION** file, an empty **R/** folder and a **NAMESPACE** file. Everything else is added in the next chapters. The RStudio menu *File > New Project > New Directory > R Package using devtools* does the same.



1.2 Put it under version control

The project needs to be a Git repository with at least one commit before it can go to GitHub. Run:

```
usethis::use_git()
```

usethis asks whether it may commit the files. Accept. It then asks to restart RStudio so that the Git tab appears. Accept this as well and wait for the restart.

1.3 Create the repository on GitHub

With the project under version control, one call creates the repository in your organisation, connects it to your project and pushes the first commit:

```
usethis::use_github(organisation = "openwashdata")
```

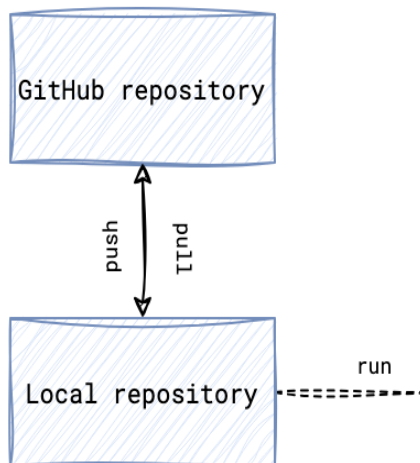
Replace the organisation with the one you publish under. The repository is public, which is right for data that holds nothing confidential or personal. Chapter 2 covers that check before any data is pushed. The call needs a GitHub token with permission to create repositories in the organisation. `usethis::gh_token_help()` explains how to get one.

If the call fails, create the repository by hand. Go to the organisation page on GitHub, click “New repository”, give it the package name, keep it empty (no README, no .gitignore file, no license, they are added later) and create it. Then switch to the Terminal tab in RStudio, next to the Console, and connect the two, with `ORGANISATION` and `REPO` replaced:

```
git remote add origin "https://github.com/ORGANISATION/REPO.git"  
git branch -M main  
git push -u origin main
```

Afterwards, run `usethis::use_github_links()` once, so that `DESCRIPTION` names the repository. `use_github()` does this on its own.

Either way, your local repository on your computer is now connected to the remote repository on GitHub, so you can push and pull changes and invite collaborators.



1.4 Add the check that runs on every push

washr writes a GitHub Actions workflow that runs R CMD check on every push and pull request, on macOS, Windows and three versions of R on Linux. The openwashdata review standard requires it, so add it now and the package meets that part of the standard from its first commit:

```
library(washr)
setup_ci()
```

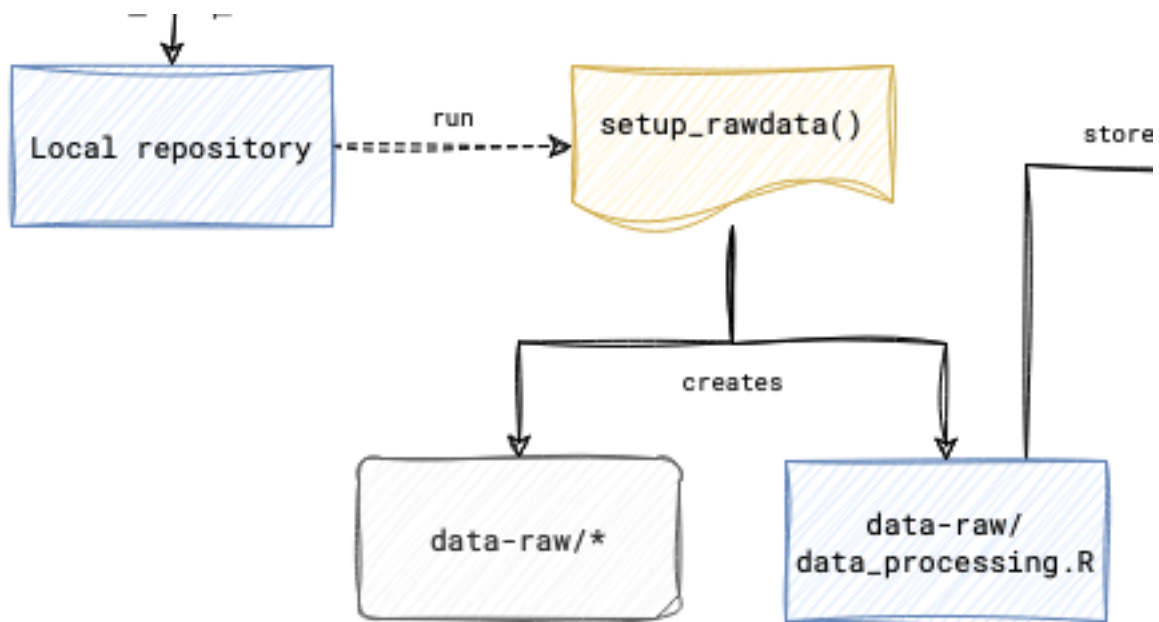
The workflow lands in `.github/workflows/R-CMD-check.yaml`. Commit it and push.

1.5 Committing and pushing from RStudio

The chapters that follow end each step with a commit and a push. In RStudio, open the Git tab in the top right panel. Click “Pull” first, so that you have the latest changes. Tick the boxes next to the files you added or changed and click “Commit”. A window opens. Write a short message that says what changed, for example “Add raw data files”, and click “Commit”. Then click “Push” to upload the commit to GitHub.

2 Preparing the data

Everything starts with a dataset that you want to publish. You might have several files, some on your computer, others on an external drive or in the cloud. This chapter brings them into the package, cleans them into one or more tidy datasets and describes every variable in a data dictionary.



⚠ Check for personal data first

Once data is in a public repository, deleting it later does not undo the exposure, because copies and caches remain, and the git history keeps every version of a file. Before the first commit that contains data, look at every column and a sample of the values in every file. Does a row name or locate a person? Names, phone numbers, email addresses, ID numbers, household coordinates and data at the household or person level all count. The pkgreview [contributor guidebook](#) describes the check in full. If you find any of this, remove or aggregate it before you continue, or ask the openwashdata team for help.

2.1 Bring in the raw data

In the R console, from the package project, run:

```
library(washr)
setup_rawdata()
```

This creates the folder `data-raw/` and a processing script, `data-raw/data_processing.R`, with the steps the script should follow. Copy or move your raw files into `data-raw/`. The raw files are kept exactly as you received them and are never edited by hand. Every change happens in the script, so anyone can trace the published data back to the source.

Commit the raw files and push.

2.2 Clean and process the raw data

Open `data-raw/data_processing.R`. The script reads the raw files, cleans them into one or more tidy data frames and saves each one in two forms. `usethis::use_data()` writes an `.rda` file into `data/`, which is what the package loads. `readr::write_csv()` and `openxlsx::write.xlsx()` write a CSV and an XLSX file into `inst/extdata/`, which is what the website and the download table of the README link to. The template already holds the reading, saving and export blocks for one data frame named after the package. Add your cleaning steps in between, and keep that name for your main dataset or change it in all three export calls.

A package can hold more than one dataset. In that case, repeat the saving and export block for each data frame, so that each one lands in `data/` and in `inst/extdata/`.

When the script is finished, run it from top to bottom with “Source” in the editor. Check the results in `data/` and `inst/extdata/`, then commit the script and the generated files and push.

2.3 Write the data dictionary

The dictionary is the one place where variables are described. The help pages and the metadata are generated from it. Run:

```
setup_dictionary()
```

This reads every dataset in `data/` and writes `data-raw/dictionary.csv` with one row per variable. The columns are the folder, the file the variable belongs to, the variable name, its type and an empty description.

Open the file in RStudio or in a spreadsheet program and write a description for every variable. The description is the most important text of the whole package. A reader who has never seen your project should understand what a variable means from the description alone.

- Write one sentence in plain language.
- Say what was measured or recorded.
- Give the unit when there is one.
- Name the possible values when there are few.

Two descriptions that work:

- “Distance in meters from the household to the nearest water point.”
- “Whether the water point was working on the day of the visit (yes or no).”

Two that do not, because they repeat the variable name or stay vague:

- “The distance variable.”
- “Status.”

Leave no description empty. Keep the five columns washr wrote, in their order, and do not edit the type column, which washr fills from the data. If you edit the file in Excel, save it in the “CSV UTF-8” format, because the plain CSV format of Excel on Windows is not UTF-8.

Commit the dictionary and push.

2.4 When the data changes later

When the processing script changes and a variable is added, removed or changes its type, run:

```
update_dictionary()
```

It adds a row for each new variable, removes the row of a variable that no longer exists, refreshes the types and keeps every description you wrote. Running `setup_dictionary()` again would stop, because the dictionary exists. After an update, run `setup_roxygen()` from the next chapter again, so that the help pages follow.

Your repository now holds the raw files, the processing script, the tidy datasets with their exports, and the dictionary, all under version control.

3 Documenting the data

The dictionary describes the variables. This chapter turns it into help pages, completes the description of the package as a whole and generates the machine readable metadata that dataset search engines read.

3.1 Write the help pages

Every dataset in an R package has a help page. `washr` writes its source from the dictionary:

```
setup_roxygen()
```

This creates one file per dataset under `R/`, with a title placeholder, a description placeholder and the variable table from the dictionary, written as roxygen comments (lines that start with `#'`).

Open each file and replace the two placeholders. The title is a human readable name of the dataset, on one line. The description is a brief statement of what the dataset contains and what it is for. Then add a `@source` line above the quoted dataset name at the end of the file. It names who collected the data, a URL or a reference, and the date you accessed it. For a data package, the source is the most important field of the help page after the description.

The lines from `@format` to the closing brace are generated from the dictionary. Do not edit them. When a description changes, change it in the dictionary and run `setup_roxygen()` again. It regenerates only the variable table and keeps the title, the description and the source you wrote.

Then build the help pages, check the package and install it:

```
devtools::document()  
devtools::check()  
devtools::install()
```

`devtools::document()` writes the help pages to `man/`. `devtools::check()` runs the same check as the workflow on GitHub. A warning about the license is expected at this point and

goes away in the next section. `devtools::install()` installs the package on your computer, so that the README and the website can load the data.

Commit and push.

3.2 Complete the DESCRIPTION file

DESCRIPTION describes the package as a whole, with its title, what it contains, who made it and under which license. The citation files, the README, the website and the metadata all read from it, so a value that is wrong here is wrong everywhere. Fix it here and never in a generated file.

3.2.1 Authors

Add yourself as author and maintainer:

```
usethis::use_author(  
  given = "Your First Name",  
  family = "Your Last Name",  
  role = c("aut", "cre"),  
  email = "your.email@example.com",  
  comment = c(ORCID = "0000-0000-0000-0000")  
)
```

`aut` marks an author and `cre` the maintainer, the person who receives questions about the package. The ORCID iD reaches the citation files, the Zenodo record and the metadata, so give it when you have one.

Add every other person who contributed with the same call, with `role = "aut"` for an author and `role = "ctb"` for a contributor. To collect the names, email addresses, roles and ORCID iDs, open an issue in the repository titled “Author information for DESCRIPTION” and ask everyone to add their line.

3.2.2 The fields washr writes

```
update_description()
```

The call sets the fields the review standard expects and keeps every value that is already there. It writes the language, the date, the repository and bug report URLs, which it takes from the repository named in `DESCRIPTION`, and the CC BY 4.0 license when the package has none yet. It also writes a few `Config/washr/` fields, such as the funding sentence of the website and the analytics domain, with the openwashdata values for a package under openwashdata. The later steps read them from `DESCRIPTION`. The vignette section [Publishing outside openwashdata](#) explains how another organisation sets its own values.

Now open `DESCRIPTION` and write the title and the description. The title is one line in title case without a final period. The description is one paragraph that says what the data contains, where it comes from and when it was collected.

3.2.3 Keywords, coverage and source

Three facts have no other home and go into `DESCRIPTION` by hand. Add three lines with your own values:

```
X-schema.org-keywords: sanitation, faecal sludge, Kampala
X-schema.org-spatialCoverage: Kampala, Uganda
X-schema.org-temporalCoverage: 2022-03-01/2022-09-30
```

The keywords are a comma separated list. For an openwashdata package, start with `open data` and `washdata`, then add the topic and the country or region. The spatial coverage is a place name. The temporal coverage is the first and the last date of the data, separated by a slash.

When the data comes from a published article, add its DOI as well:

```
X-schema.org-isBasedOn: https://doi.org/10.2166/wh.2026.173
```

The citation files then cite the article next to the data package, and the metadata records it as the source. Several DOIs are separated by commas.

Run `devtools::document()`, `devtools::check()` and `devtools::install()` again, then commit and push.

3.3 Generate the metadata

One function derives the machine readable metadata of your package from the files you have already written, which are `DESCRIPTION`, the dictionary, the citation file and the exports in `inst/extdata/`.

```
update_metadata()
```

It writes a schema.org description of the dataset as JSON-LD into `pkgdown/templates/in-header.html`. The website build in the next chapter embeds it in the head of every page, where dataset search engines such as Google Dataset Search read it. The function ends with the fields it could not fill and where to fill them. Before the release, the DOI is the one field that stays blank.

Nothing in this file is edited by hand. When a value is wrong, change its source and run the function again. Run it again after every change to `DESCRIPTION`, the dictionary or the exports, and once more after the DOI exists (Chapter 5). A second run with nothing changed writes nothing.

Commit and push.

4 Publishing the data

The package now has data, a dictionary, help pages and metadata. This chapter writes the README that GitHub shows on the repository page and builds the website that presents the dataset.

4.1 Write the README

4.1.1 Create it from the template

```
setup_readme(has_example = TRUE)
```

washr writes README.Rmd from its template. The file holds the title and badges, an introduction placeholder, the installation instructions, a download table with one row per dataset and links to the CSV and XLSX exports, a section for the first dataset with its dimensions, a preview and the variable table from the dictionary, an Example section with a scaffold for a first plot, the license, and a citation section that names the version and the DOI in plain text once they exist. Without `has_example = TRUE` the Example section is left out.

4.1.2 Write the text

Open README.Rmd and fill in the parts that are yours.

- The introduction. One paragraph of three to five sentences in place of “The goal of ... is to”. Say who collected the data, how, when and where, and under which license or permission the source data was shared.
- The data sentence. Complete “The dataset ... contains data about”.
- A block per further dataset. The template documents the first dataset in `data/`. For each further one, copy the block from its heading to the end of its variable table and replace the dataset name in the heading, in the sentence, in the preview chunk and in the filter of the variable table.

- The example. The plot code is commented out. Fill in variables of your dataset, write readable axis labels and a title, and uncomment the block together with the `library(ggplot2)` line. One plot that gives a useful first look at the data is enough. Say in the text what the plot shows.

The installation instructions, the download table, the dimensions, the previews, the variable tables, the license and the citation are generated. Leave them as they are.

4.1.3 Build it

README.md is built from README.Rmd and never edited by hand:

```
devtools::build_readme()
```

The build loads the packages named in the installation chunk (dplyr, knitr, readr, stringr, gt and kableExtra), so install the ones you are missing when the build stops. Commit both README files and push. GitHub shows README.md on the repository page.

4.2 Build the website

The website is built by [pkgdown](#) from the README, the help pages and the metadata. The review standard has it deployed by a GitHub Actions workflow to the `gh-pages` branch of the repository, so that the built site is never committed and never goes stale. Set up the workflow first, then let washr write the configuration and build.

4.2.1 Add the deployment workflow

```
usethis::use_github_action("pkgdown")
```

The call writes the workflow `.github/workflows/pkgdown.yaml`. It runs on every push to the main branch, builds the site and writes it to the `gh-pages` branch. Nothing else changes yet.

4.2.2 Write the configuration and build the site

```
setup_website()
```

The first run writes `_pkgdown.yml` from the `washr` template, with the address of the site, the analytics header, the funding sentence in the sidebar and a reference index with one entry per dataset. Every value comes from `DESCRIPTION`. Then it builds the site into `docs/`, which takes a minute. Because the deployment workflow exists, `docs/` stays in `.gitignore`, and the local build is a preview. Open `docs/index.html` in a browser to see it. Every later run keeps `_pkgdown.yml` as it is and rebuilds the site.

`pkgdown` expects the address of the site in `DESCRIPTION` as well, next to the repository URL. Add it, with the value of the `url` line of `_pkgdown.yml`:

```
desc::desc_add_urls("https://openwashdata.github.io/yourpackagename")
```

Note

A repository that cannot run GitHub Actions can serve the site from the main branch instead. Skip the workflow and run `setup_website()`. Without a workflow it removes `docs` from `.gitignore`, so that the built site is committed. In the Pages settings of the repository, choose the main branch and the `/docs` folder as the source. The review standard prefers the workflow, because a committed site is only as current as its last build.

4.2.3 Apply the brand

```
use_brand()
```

This copies the brand file and the logos of the `openwashdata` brand from the latest release of the brand repository into your package, adds them to `.Rbuildignore`, wires `_pkgdown.yml` to them and records the release in `DESCRIPTION`. The site now uses the `openwashdata` fonts and colours. Run it again after a new brand release. A run with no new release changes nothing. For a package under another organisation, the `Config/washr/brand-source` field in `DESCRIPTION` names the brand repository to copy from, or `none`.

4.2.4 Check, commit and publish

Run `devtools::document()`, `devtools::check()` and `devtools::install()` once more, then commit everything and push. The push starts the pkgdown workflow on GitHub, which builds the site and writes it to the `gh-pages` branch. The branch appears with the first run, after a few minutes.

Then open the repository on GitHub, go to “Settings” and “Pages”, and under “Build and deployment” choose “Deploy from a branch”, with the `gh-pages` branch and the root folder as the source. The site is live at the address in `_pkgdown.yml` a minute later, and every later push rebuilds it.

Reports about the package in the openwashtdata look, as PDF or Word documents, use the [quarto-owd](#) Quarto extension. It reads the brand file that `use_brand()` installed.

5 Creating a release and a DOI

A Digital Object Identifier (DOI) makes the dataset citable. It points at a fixed version of the data in an archive that outlives the repository. [Zenodo](#) assigns one for every release of a GitHub repository it watches. This chapter writes the citation files, checks that the package is ready, creates the release and brings the DOI back into the package.

5.1 Write the citation files

```
update_citation()
```

The call writes three files from DESCRIPTION. CITATION.cff is what GitHub reads for its “Cite this repository” button. inst/CITATION is what citation("yourpackagename") prints in R. .zenodo.json is what Zenodo reads when it archives a release. From that file Zenodo takes the resource type (Dataset), the openwashedata community, the creators with their ORCID iDs, the license and the keywords, so that the record needs no edits by hand. When DESCRIPTION names a source article in X-schema.org-isBasedOn, the citation files cite it next to the data package.

Commit the three files and push. Zenodo reads .zenodo.json from the release, so it has to be in the repository before the release is created.

5.2 Check that the package is ready

```
check_publication_readiness()
```

The function reads the package and reports, item by item, what is in place and what is missing. It covers the metadata, the dictionary, the documentation, the citation files and the check workflow, and for every gap it names the step that closes it. It writes nothing, so run it as often as you like, and go back to the chapter it points to until no item fails. Before the first release, the DOI is the one expected gap.

5.3 Connect the repository to Zenodo

Log in to [Zenodo](#) with your GitHub account. Open the menu next to your email address in the top right corner, choose “GitHub”, find your repository in the list and flip its switch to “ON”. From now on Zenodo archives every release of the repository. Each release gets its own version DOI, and the record carries one concept DOI that always resolves to the latest version.

5.4 Create the release

Set `Version` in `DESCRIPTION` to the version you release, for example 1.0.0 for the first publication. Run `update_citation()` once more, so that the citation files carry the version, and `devtools::build_readme()`, so that the README names it. Commit and push.

On GitHub, open the repository, click “Releases” and “Draft a new release”. Create a tag with the version and a v in front, such as v1.0.0, give the release the same name and a short description, and publish it. Zenodo picks the release up and creates the archive. This can take up to an hour. The DOIs then appear on the record and on your Zenodo GitHub page.

Open the record on Zenodo and check it. The resource type is Dataset, the community is openwashdata, the creators and the license are right. All of this came from `.zenodo.json`. If something is wrong, fix it in `DESCRIPTION`, run `update_citation()` again for the next release, and correct the record once in the Zenodo editor.

5.5 Bring the DOI into the package

Use the concept DOI, the one Zenodo shows under “Cite all versions”. It stays the same across releases, so a citation always resolves to the latest version, while the version in the citation file says which one was used. Back in RStudio, write it into the citation files, the README and the metadata with two calls:

```
update_citation(doi = "10.5281/zenodo.11185699")
update_metadata()
```

`update_citation()` regenerates the citation files with the DOI, adds the DOI badge to `README.Rmd` and rebuilds `README.md`. `update_metadata()` carries the DOI into the JSON-LD of the site. Run `check_publication_readiness()` once more, then commit the changed files and push. The pkgdown workflow rebuilds the site, and the citation in the README and on the website now shows how to cite the dataset.

A package with a Zenodo DOI appears in the [openwashdata data catalogue](#) within a week.

5.6 Later versions

When the data or the documentation changes after publication, raise the version in **DESCRIPTION**, run the steps of this chapter again and create a new release. Zenodo archives the new version under a new version DOI, and the concept DOI keeps resolving to the latest one. The download links in the README point at the release tag of the version in **DESCRIPTION**, so that a citation of a version leads to the same files later.

5.7 ETH Research Collection submission

Note

This section applies only to researchers at Global Health Engineering, ETH Zurich.

For a submission to the [ETH Research Collection](#), go to the research data section and choose “dataset” as the submission type. Under organisational unit, select “tilley”, and set the license to “Creative Commons Attribution 4.0 International”. These settings match the data package and keep the data open while protecting your rights as the creator.

6 Troubleshooting

This guide cannot cover every situation. The problems below have come up more than once, with what fixed them. If you run into one that is missing, please open an issue in the [source repository](#).

6.1 Start with the readiness report

`check_publication_readiness()` reads the package and names, for every item that fails, the step that closes it. Run it first whenever a later step stops with a message about a missing file.

6.2 `setup_dictionary()` stops because the dictionary exists

Run `update_dictionary()` instead. It brings the dictionary in line with the data and keeps every description you wrote (Chapter 2).

6.3 `setup_website()` stops because README.md is missing

The website is built from the README. Run `setup_readme()` and `devtools::build_readme()` first (Chapter 4).

6.4 `pkgdown` reports “URLs not ok”

The address of the site is missing from DESCRIPTION. Add it with `desc::desc_add_urls()` and the value of the `url` line of `_pkgdown.yml`, then build again (Chapter 4).

6.5 Git fails when pushing large files

When you push large files such as raw datasets, Git may fail. Raise the buffer size with the following line in the Terminal, from the folder of your project:

```
git config http.postBuffer 524288000
```

6.6 R CMD check

A few messages from `devtools::check()` come up again and again.

“unable to verify current time” relates to the time synchronisation of your computer. This [Stack Overflow answer](#) has the fix.

“Non-standard files/directories found at top level” means that the package root holds files R does not expect there. The fix is to list them in `.Rbuildignore`, as this [Stack Overflow thread](#) explains. The `washr` functions add their own files to `.Rbuildignore`, so this message points at a file you added.

Platforms change their interfaces from time to time, so a button or a menu named in this guide may have moved. The documentation of the platform is the place to look then.