Workbench Developer's Guide

Zhian N. Kamvar

2023-12-31

Table of contents

Preface

The Carpentries Workbench is a open-source and portable lesson infrastructure built with the R programming language. Despite it being built in R, contributors do not need to know any R in order to use it to build reliable, stylish, and accessible lessons.

This book serves as development documentation for The Carpentries Workbench. It was written between June and December 2023 primarily to orient new developers and contributors to The Workbench ecosystem.

> **Under Construction**
>
> The content for this site is still being written! Please check back if what you are looking for does not exist or, open an issue at https://github.com/carpentries/workbench-dev/issues/ to request it!

> **Prerequisite**
>
> This book assumes familiarity with R Package Development. If you are unfamiliar, please read R Packages (2e) (Wickham and Bryan 2023).

1 Introduction

The core of The Carpentries Workbench consists of three packages:

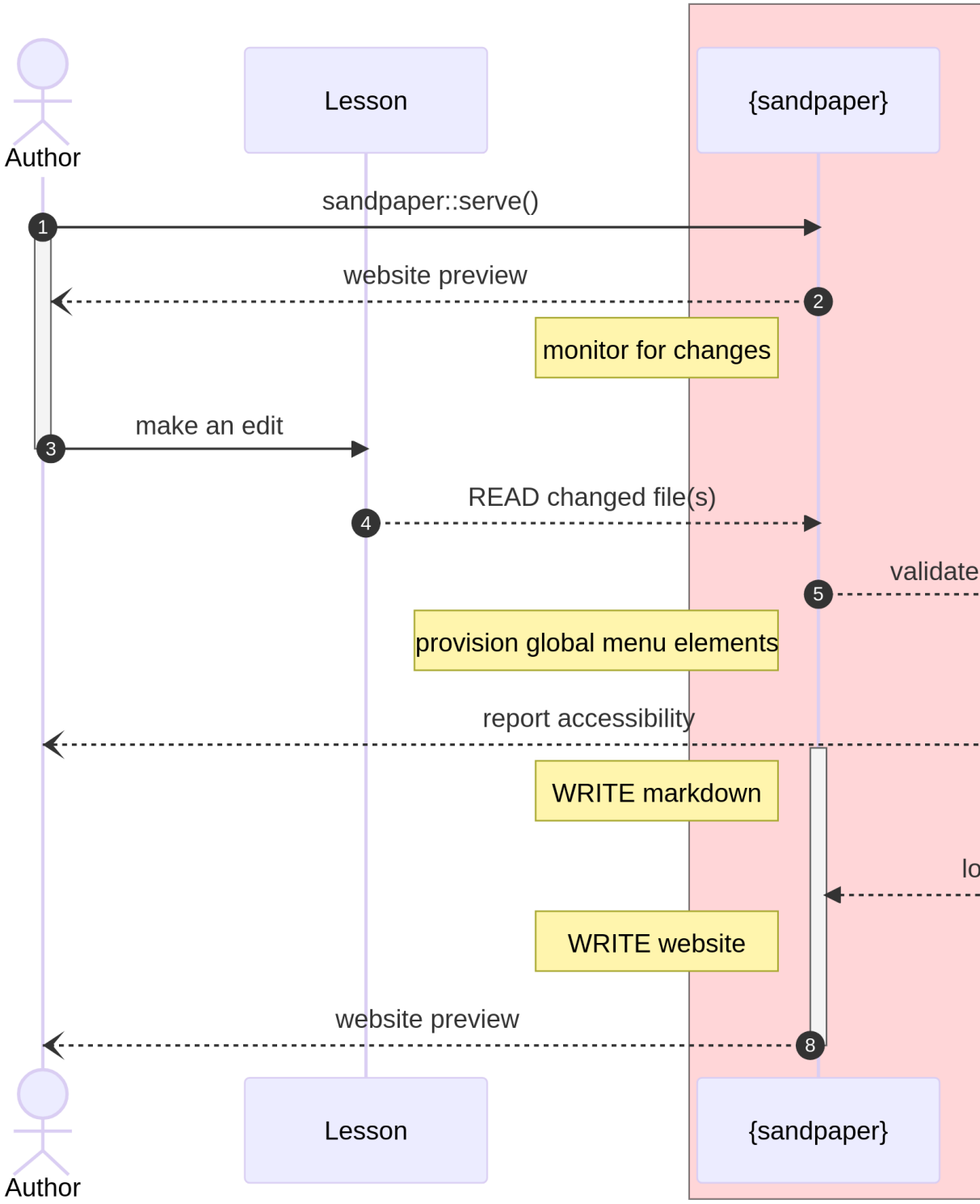
- {sandpaper}: user interface and workflow engine
- {pegboard}: parsing and validation engine
- {varnish}: HTML templates, CSS, and JS elements

These packages are all available and released to the Carpentries R-Universe, which checks for updates to the source packages hourly.

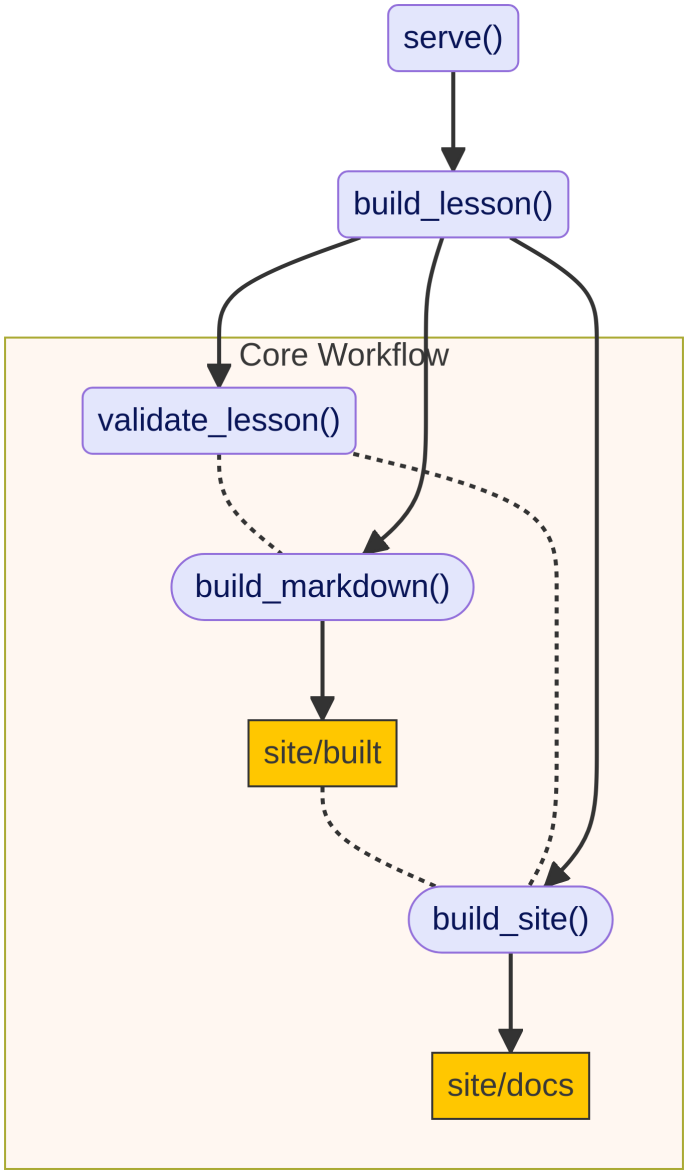
1.1 Building Lessons

In a broad sense, this is what happens when you run `sandpaper::serve()` or `sandpaper::build_lesson()`. The interaction between the three Workbench packages, the lesson content, and the author can be summarised like this where the author makes an edit:

> **Summary Content**
>
> This content is a general picture of what happens between the packages. For a more in-depth discussion and more detailed diagrams, please visit the Flow Diagrams page.


Author
Lesson
{sandpaper}
sandpaper::serve()
1
website preview
2
monitor for changes
make an edit
3
READ changed file(s)
4
validate
5
provision global menu elements
report accessibility
WRITE markdown
lo
WRITE website
website preview
8
Author
Lesson
{sandpaper}

In terms of folder structure, the workflow runs the two-step workflow to first render markdown files into `site/built` and then uses those files to render the HTML, CSS, and JavaScript into `site/built`. These workflows are detailed in The Workflows Chapter.



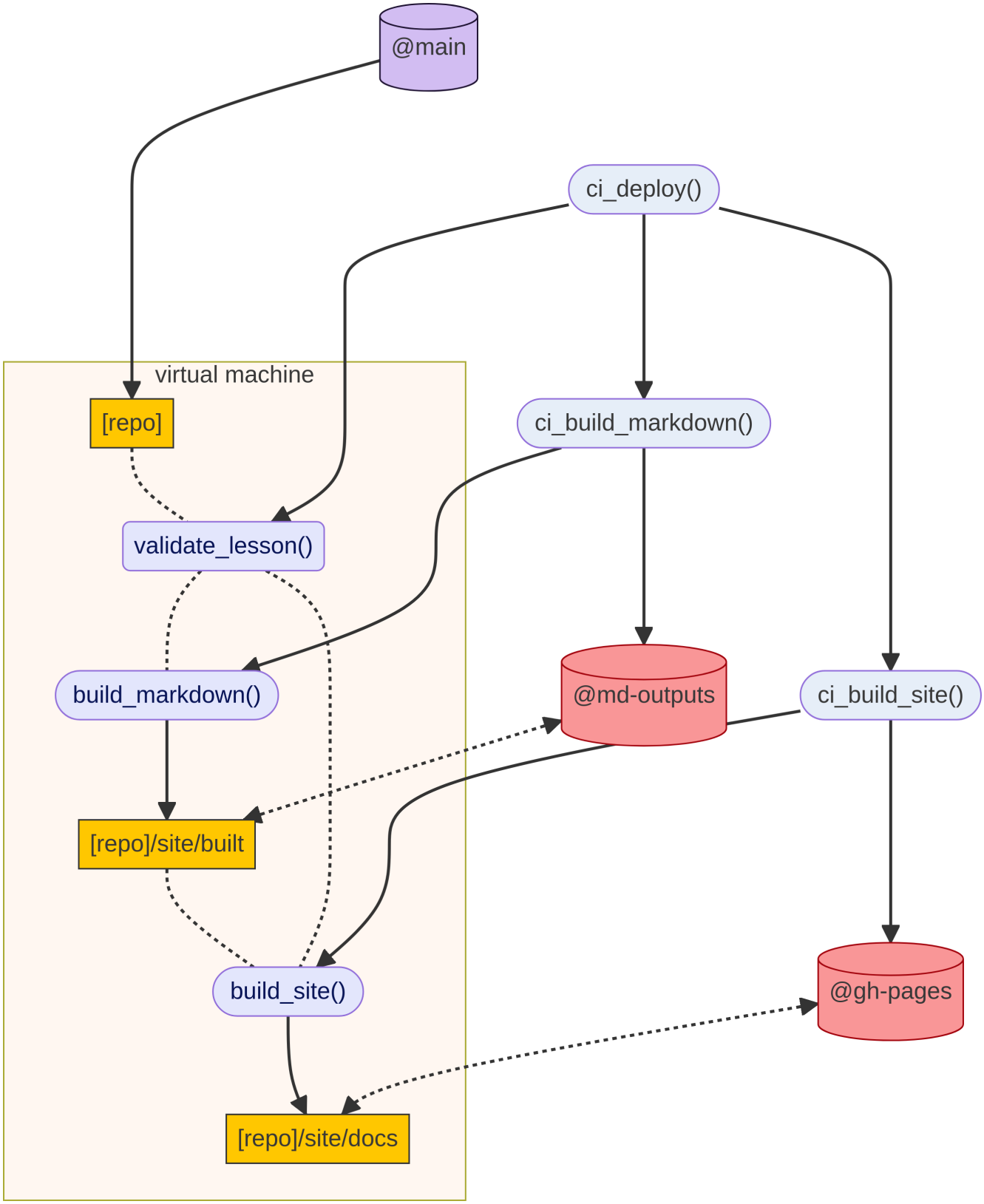
> **i** Resource folder names
>
> The names of the folders inside `site/` are considered internal resources and they can change at any time. The reason why the folder for the final website output is called `site/docs/` is because we use the {pkgdown} package to provision the website without needing to bundle the templates inside of {sandpaper}, but we never got around to explicitly changing the name of that folder.

The `site/docs` folder contains the full website that can be safely used offline. This is the core of the workflow and is used both locally and in a remote setting. The only difference with the remote setting is that we use a few Git tricks to provision the markdown cache without needing to store it in the default branch.

1.2 Building Lessons Remotely (e.g. on GitHub)

In the remote workflow, **we still use the same workflow as above**, except now we use `ci_deploy()` to link the branches and folders using worktrees, which you can think of as Git branches assigned to separate folders.

> **i** Platform Independence
>
> When we developed The Workbench, GitHub was the most widely used platform for social coding that represented the easiest way for newcomers to contribute to our lessons. We used this knowledge to build the workflows for the lessons on GitHub, but we were also aware of the valid criticisms of GitHub and the dangers of vendor lock-in.
> Thus, while the lessons are deployed using GitHub workflows and we have features that handle pull requests and updates, **the core deployment features remain platform-independent**. The workflows are merely instructions that we provide for GitHub to set up the workbench and to run the individual functions. In theory, **any platform can be configured to deploy lessons via The Workbench.**
> In fact, in a pinch when GitHub workflows are not working properly, a lesson maintainer could run `sandpaper:::ci_deploy()` to render and deploy a local copy of the lesson.


@main
ci_deploy()
virtual machine
[repo]
ci_build_markdown()
validate_lesson()
build_markdown()
@md-outputs
ci_build_site()
[repo]/site/built
build_site()
@gh-pages
[repo]/site/docs

1.3 Development

Development of The Workbench is overseen by Zhian N. Kamvar. New features are added incrementally as pull requests. Pushes to the main branch are *rare* and discouraged. New features must have tests associated (with the exception of {varnish}).

If you are interested, we have documentation for the release process available.

1.4 Documentation

Reference documentation for individual functions for each package is written alongside the function using {roxygen2}.

This documentation is generated by `devtools::document()`

1.5 Testing

Tests for each package live in `tests/testthat/` and follow a `test-[file-name].R` naming convention. These are controlled by the {testthat} package and run by `devtools::test()`.

You can find more information about testing the core packages in Testing The Workbench

1.6 Continous Integration

The continous integration for each package tests on Ubuntu, MacOS, and Windows systems with the last five versions of R (same as the RStudio convention).

More information about the Continous Integration can be found in the Continuous Integration section of the testing section.

Coming up:

- Testing Pull Requests (Locally and on your fork)
- Resources for R package development
- Adding functionality to {sandpaper}
- Adding functionality to {pegboard}
- Adding styling elements to {varnish}
- Adding functionality to carpentries/actions

2 System Setup

2.1 Software Tools

Development of Workbench components requires the same toolchain for working on lessons:

- R
- pandoc
- Git

It is recommended to have the latest versions of R and pandoc available. You need at least git 2.28 for security purposes.

```
R version
---


R version 4.3.1 (2023-06-16) -- "Beagle Scouts"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under the terms of the
GNU General Public License versions 2 or 3.
For more information about these matters see
https://www.gnu.org/licenses/.



pandoc version
---


pandoc 2.19.2
Compiled with pandoc-types 1.22.2.1, texmath 0.12.5.2, skylighting 0.13,
citeproc 0.8.0.1, ipynb 0.2, hslua 2.2.1
Scripting engine: Lua 5.4
User data directory: /home/runner/.local/share/pandoc
```

```
Copyright (C) 2006-2022 John MacFarlane. Web:  https://pandoc.org
This is free software; see the source for copying conditions. There is no
warranty, not even for merchantability or fitness for a particular purpose.


git version
---


git version 2.41.0
```

2.2 R Packages

Once you have these installed, make sure to install ALL of the dependencies for the workbench:

```
install.packages(c("sandpaper", "pegboard", "varnish", "tinkr"),
  dependencies = TRUE,
  repos = c(getOption("repos"), "https://carpentries.r-universe.dev"))
```

Working on Linux?

If you are on Linux, you will run into a couple of fun aspects that you may already be familiar with, especially if you have ever tried to install bioinformatic software:

1. having to also install some extra C libraries (which are akin to R packages, but for C), such as the `xslt` library.
2. having to build all packages from source

You can find detailed instructions in The Sandpaper Setup Guide, but the relevant commands are below.

System Dependencies

Here is the gist **for Ubuntu Users to get system dependencies set up.** Use The Carpentries R-Universe API to get all of the system dependencies. Here's how to do that via CURL:

```
curl https://carpentries.r-universe.dev/stats/sysdeps 2> /dev/null | jq -r '.headers[0] | s
```

This list can be sent to `apt-get install` to install everything:

```
sudo apt-get install -y \
  $(curl https://carpentries.r-universe.dev/stats/sysdeps 2> /dev/null | jq -r '.headers[0]
  || echo "Not on Ubuntu"
```

Binary Packages

To get binary packages for your system, I will admit that it's *slightly confusing* because they bury the instructions for registering your system to use binaries in the *admin* pages and even then, it's kinda long. The gist is that you need to do two things:

1. set your HTTPUserAgent header to state your R version and platform
2. add the packagemanager CRAN-like repository to R's options:

Here's a script that you can copy and paste into `~/.Rprofile` which will be run every time you start R

```
local({
  # Set the default HTTP user agent to get pre-built binary packages
  RV <- getRversion()
  OS <- paste(RV, R.version["platform"], R.version["arch"], R.version["os"])
  codename <- sub("Codename.\t", "", system2("lsb_release", "-c", stdout = TRUE))
  options(HTTPUserAgent = sprintf("R/%s R (%s)", RV, OS))

  # register the repositories for The Carpentries and CRAN
  options(repos = c(
    carpentries = "https://carpentries.r-universe.dev/",
    CRAN = paste0("https://packagemanager.posit.co/all/__linux__/", codename, "/latest")
  ))
})
```

When you have this set up, you can then install the workbench packages:

```
# Install The Workbench and dependencies
install.packages(c("sandpaper", "varnish", "pegboard", "tinkr"), dep = TRUE)
```

The {sandpaper} package comes with the {usethis} package embedded (though this may change in the future). In addition, you will need the {devtools} for development.

I would also *highly* recommend the {pandoc} package for managing pandoc versions (NOTE: this requires you to have a personal access token set up).

```
install.packages("devtools")
install.packages("pandoc")
```

Once you have devtools, be sure to run `devtools::dev_sitrep()` and `usethis::git_sitrep()` to make sure you have the tools to build The Workbench:

```
devtools::dev_sitrep()
#>   R
#> • version: 4.3.0
#> • path: '/usr/lib/R/'
#>   devtools
#> • version: 2.4.5
#>   dev package
#> • package: <unset>
#> • path: <unset>
#>  All checks passed

usethis::git_sitrep()
#> Git config (global)
#> • Name: 'Zhian N. Kamvar'
#> • Email: 'zkamvar@gmail.com'
#> • Global (user-level) gitignore file: <unset>
#> • Vaccinated: FALSE
#>  See `?git_vaccinate` to learn more
#>  Defaulting to 'https' Git protocol
#> • Default Git protocol: 'https'
#> • Default initial branch name: 'main'
#> GitHub
#> • Default GitHub host: 'https://github.com'
#> • Personal access token for 'https://github.com': '<discovered>'
#> • GitHub user: 'zkamvar'
#> • Token scopes: 'gist, repo, user, workflow'
#> • Email(s): 'zkamvar@gmail.com (primary)', ...
#> Git repo for current project
#>  No active usethis project
```

Created on 2023-05-30 with reprex v2.0.2

2.3 Development Workflow

This development workflow is known as Test Driven Development in which a test is written *before* things work. This way, we can confirm that a bug is fixed once it passes the tests and we have confidence that it will not fail again.

1. open RStudio and switch to the project for the package you are working on
2. checkout a new branch for your feature/bug
3. **load package** via `devtools::load_all()` or ctrl+shift+L (use cmd on macOS) to load the package NAMESPACE
4. **run tests** (either via `devtools::test()` or ctrl+shift+T to run the entire test suite OR to test a single file, use the "run tests" button in a test file or run `testthat::test_local(filter = '[FILE SLUG]')`
5. **modify tests** for new functionality/bug fix
6. **add functionality/bug fix** and move to 3 unless you are ready to push
7. run check with `devtools::check()` or ctrl+shift+E

3 Testing The Workbench

> ⚠ This section is still under construction!
>
> We are still assembling the documentation for this part of the site. If you would like to contribute, please feel free to open an issue.

3.1 Introduction

> The first stage of your testing journey is to become convinced that testing has enough benefits to justify the work. For some of us, this is easy to accept. Others must learn the hard way.
>
> — Wickham and Bryan, Testing Basics, **R Packages** second edition

> If you use software that lacks automated tests, you are the tests.
>
> — Jenny Bryan source tweet (2018-09-22 01:13 UTC)

Every single package that runs code in the lesson infrastructure is tested before it ever reaches any lesson. This is important because we want to give the lesson authors and maintainers as much freedom as they need to write a lesson while maintaining predictability and integrity. We also want to give our community confidence that this system works.

Whenever a new feature or bug fix is added to The Workbench, it is imperative that a test is associated and verified before it gets sent into production.

Tests can be run locally and via continuous integration. This page introduces some of the testing strategies used in The Workbench and the caveats that come with these strategies.

3.2 Unit Testing

The tests under `test/testthat/` are run in alphabetical order using the {testthat} package (see https://r-pkgs.org/testing-basics.html) via `devtools::test()` or `devtools::check()`.

3.2.1 Conditionally Skipped Tests

The tests often need special conditions in order to run and sometimes those conditions are not possible. One of the most common conditions to skip is if the testing happens on CRAN. They are very hawkish about how long test suites can run and it's often difficult to detect the state of a CRAN machine, so it's better to skip long-running tests or those with complex environmental dependencies on CRAN (which does not yet apply to {sandpaper}).

3.3 Continous Integration

All the unit tests are run in continuous integration for every push and pull request that occurs. They also run every week. This provisions the current releases of the R package dependencies along with development versions of critical dependencies such as {renv}.

In continous integration, we run on with the following conditions to make sure it works not only on GitHub, but also on local user machines:

- test coverage (no package structure) with released versions on Ubuntu Linux (though reporting is stalled)
- For each platform (Ubuntu Linux, macOS, and Windows)
  - `R CMD check`, which checks the structure of the package and documentation
  - all run on these versions of R: current, devel, and two previous R versions

Because of occasional provisioning failures on macOS and Windows, we require only that Ubuntu Linux latest version passes check for merging pull requests.

3.4 Lesson Integration Testing

Unit tests are great for testing all functionality using known and stable inputs, it is important to test using known inputs that are in a constant state of flux: live lessons. This is where The Workbench Integration Test comes in. **It will run a weekly test on a defined set of lessons using the current development versions of varnish and sandpaper.** These tests are useful for three purposes:

1. ongoing integrity for the workbench lessons lessons that use different features of The Workbench such as R code execution
2. real-world effects of new sandpaper and varnish versions (including those in pull requests)
3. inspecting changes in HTML and markdown output

These lessons that we use are

Instructor Training Lesson with the most content, contributors, activity, and used features. This particular lesson is a bit of a stress test for the infrastructure.

R for SocialScientists This is one of the first R-based lessons to be transitioned and it uses the tidyverse as a dependency.

Workbench Documentation The workbench documentation. If this doesn't work, nothing will.

Raster and Vector Geospatial Data with R This lesson uses R packages that rely on a geospatial software stack, which can be complex. Failures here likely mean that there are problems with provisioning external C libraries.

BioConductor RNAseq Lesson using BioConductor R packages by people at BioConductor. If this does not work, then there likely is a provisioning problem between BioConductor and {renv}.

3.4.1 Testing sandpaper and varnish pull requests

To test a pull request version, you can head over the the main workflow and use the button that says "Run Workflow". When you want to test a varnish or sandpaper pull request, you can use the `[REPO]#[PR]` syntax (e.g. `carpentries/sandpaper#429` to run sandpaper pull request 429) in the entry fields for varnish and sandpaper version. If you don't have a pull request to work from, you can use the `[REPO]@[REF]` syntax (e.g. `carpentries/sandpaper@test-this-thing` to run sandpaper test-this-thing branch).

3.4.2 Inspecting changes

The output for all the tests are stored in branches that are named respective for their test repositories. For example, `datacarpentry/r-socialsci/markdown` and `datacarpentry/r-socialsci/site` contain the markdown and HTML outputs for the R for Social Scientists lesson. By inspecting the diffs from the commits, you can see how the output has changed from run to run, which is useful if you are confirming that a feature will be automatically deployed.

3.5 Ad-hoc Testing

There are times when you cannot automate your way through testing and you just have to suck it up and get your (virtual) hands dirty. You find yourself in this kind of situation if you are testing out GitHub workflows, GitHub actions, or if you are implementing a new feature and you need to see that it works reliably and safely. It's this point in time when you use ad-hoc testing on a brand new lesson repository that you give yourself permission to mess around in and will delete when finished.

3.5.1 GitHub Workflows

A GitHub Workflow is a YAML document that lives inside the `.github/workflows` folder of a repository. This sets up the environment needed to build a lesson. When you debug these, ask yourself if you really want to update the GitHub Actions instead. Because these are copied to each lesson, they need to be updated in each lesson (which is accomplished through the automated pull request workflow). If you've determined that the workflow needs to be modified, you can modify them inside your test lesson until you get the desired results. Once that is done, copy them over to `sandpaper/inst/workflows` and add a NEWS item that's under the heading `## CONTINUOUS INTEGRATION` describing your change.

3.5.2 GitHub Actions

A GitHub Action is a single step in a GitHub Workflow and can be written in nearly any language.

The GitHub actions that are in carpentries/actions are written in BASH, node JavaScript, and R and cobbled together with YAML. When developing a new feature, work on a branch and then, in your shiny new test lesson, replace the `@main` in the GitHub Workflows to your branch name. This way, you can know immediately if the fix or feature worked without having to interrupt someone's flow.

3.5.3 New config items or settings

If you implement a new config item (e.g. a `lang:` tag), a temporary lesson is a great way to test it. To do so, you can use the `sandpaper:` or `varnish:` keys in your lesson config to specify the version of sandpaper or varnish you want to test.

Part I

Building Lessons

4 Flow Diagrams

4.1 Introduction

This section builds on The broad workflow and details the internal process that are invoked with the `sandpaper::build_lesson()` function. If you look at the source for this function, it contains a total of sevens significant lines of code (many more due to documentation and comments).

The pre-flight steps all happen before a single source file is built. These check for pandoc, validate the lesson, and configure global elements. The last two lines are responsible for building the site and combining them with the global variables and templates.

Users will invoke this function in the following ways:

venue	function	purpose
local	`sandpaper::build_lesson()`	render content for offline use
local	`sandpaper::serve()`	dynamically render and preview content
remote	`sandpaper:::ci_deploy()`	render content and deploy to branches

All of these methods will call `sandpaper::validate_lesson()` (which also sets up global metadata and menu variables) and the two-step internal functions `sandpaper:::build_markdown()` and `sandpaper:::build_site()`. Below, I break down and detail the process for each.

4.2 Preflight Checks

Before a lesson can be built, we need to confirm the following:

1. We have access to the tools needed to build a lesson (e.g. pandoc). This is achieved via the `sandpaper::check_pandoc()`
2. We are inside a lesson that can be built with The Carpentries Workbench

4.3 `validate_lesson()`

The lesson validator is a bit of a misnomer. Yes, it does peform lesson validation, which it does so through the methods in the `pegboard::Lesson` R6 class.

In order to use thse methods, it first loads the lesson, via the `sandpaper::this_lesson()` function, which loads *and* caches the `pegboard::Lesson` object. It also caches elements that are mostly duplicated across episodes with small tweaks for each episode:

- metadata in JSON-LD format
- sidebar
- extras menu for learner and instructor views

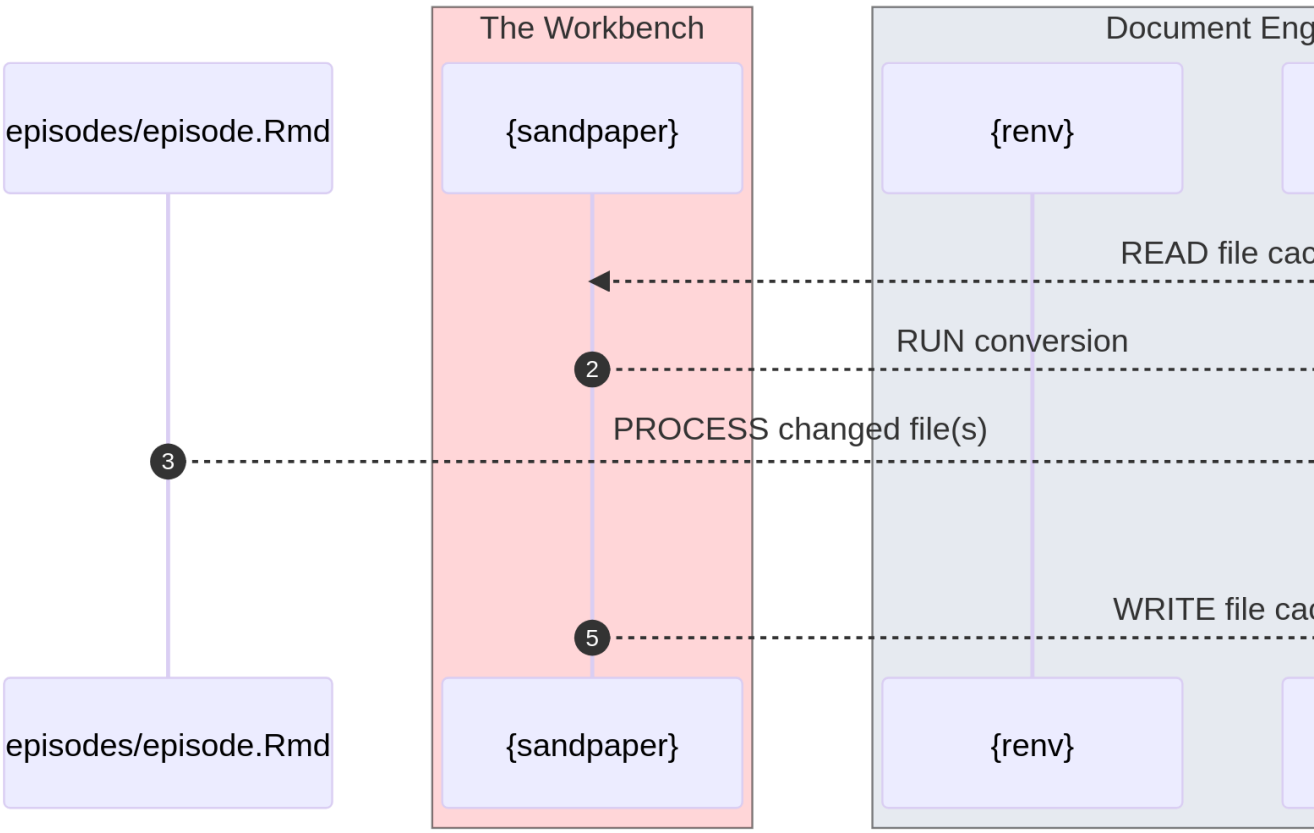
4.4 `build_markdown()`

4.4.1 Generating Markdown

Markdown generation for the lesson is controlled by the internal function `sandpaper:::build_markdown()`.

When a lesson contains R Markdown files, these need to have content rendered to markdownsot hat we can further process them. This content is processed with the {knitr} R package *in a separate R process*. Markdown source content on the other hand is copied to the `site/built` folder.

Because R Markdown files can take some time to render, we use MD5 sums of the episode contents (stored in the `site/built/md5sum.txt` file) to skip any files that have not changed.



Package Cache and Reproducibility

One package that is missing from the above diagram is {renv} and that's partially because it has an indirect effect on the lesson: it provisions the packages needed to build the lesson. When episodes are rendered from R Markdown to Markdown, we attempt to reproduce the build environment as closely as possible by using the {renv} package. If the global package cache from {renv} is available, then the lesson profile is activated before the episode is sent to {knitr} and R will use the packages provided in that profile. This has two distinct advantages:

1. The user does not have to worry about overwriting packages in their own library (i.e. a graduate researcher working on their dissertation does not want to have to rewrite their analyses because of a new version of {sf})
2. The package versions will be the same as the versions on the GitHub version of the site, which means that there will be no false positives of new errors popping up

For details on the package cache, see the Building Lessons With A Package Cache article.

At this step, the markdown has been written and the state of the cache is updated so if we re-run this function, then it will show that no changes have occured. After this step, the internal function `sandpaper:::build_site()` is run where the markdown file that we just created is converted to HTML with pandoc and stored in an R object. This R object is then manipulated and then written to an HTML file with the {varnish} website templates applied.

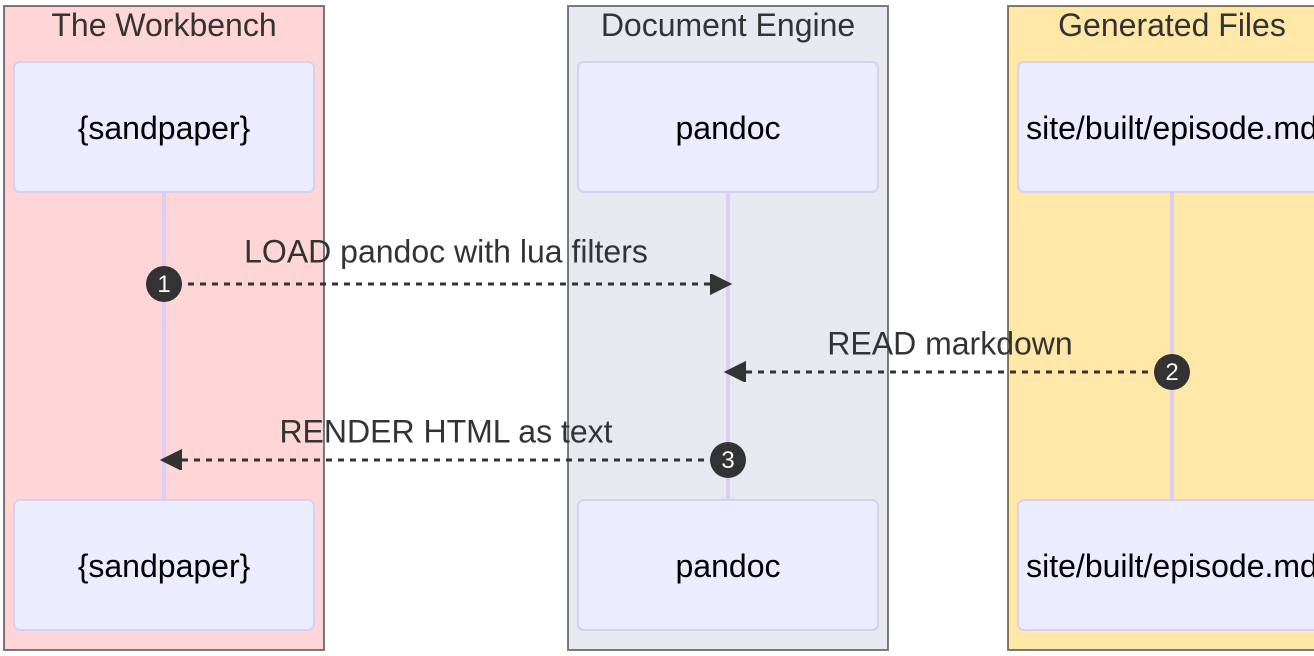
We use this function in the pull request workflows to demonstrate the changes in markdown source files, which is useful when package versions change, causing the output to potentially change.

4.5 `build_site()`

The following sections will discuss the HTML generation (the following section), manipulation (the section after that), and applying the template (the final section) separately because, while these processes are each run via the internal `sandpaper:::build_site()` function, they are functionally separate.

4.5.1 Generating HTML

Each markdown file is processed into HTML via pandoc and returned to R as text. This is done via the internal function `sandpaper:::render_html()`.

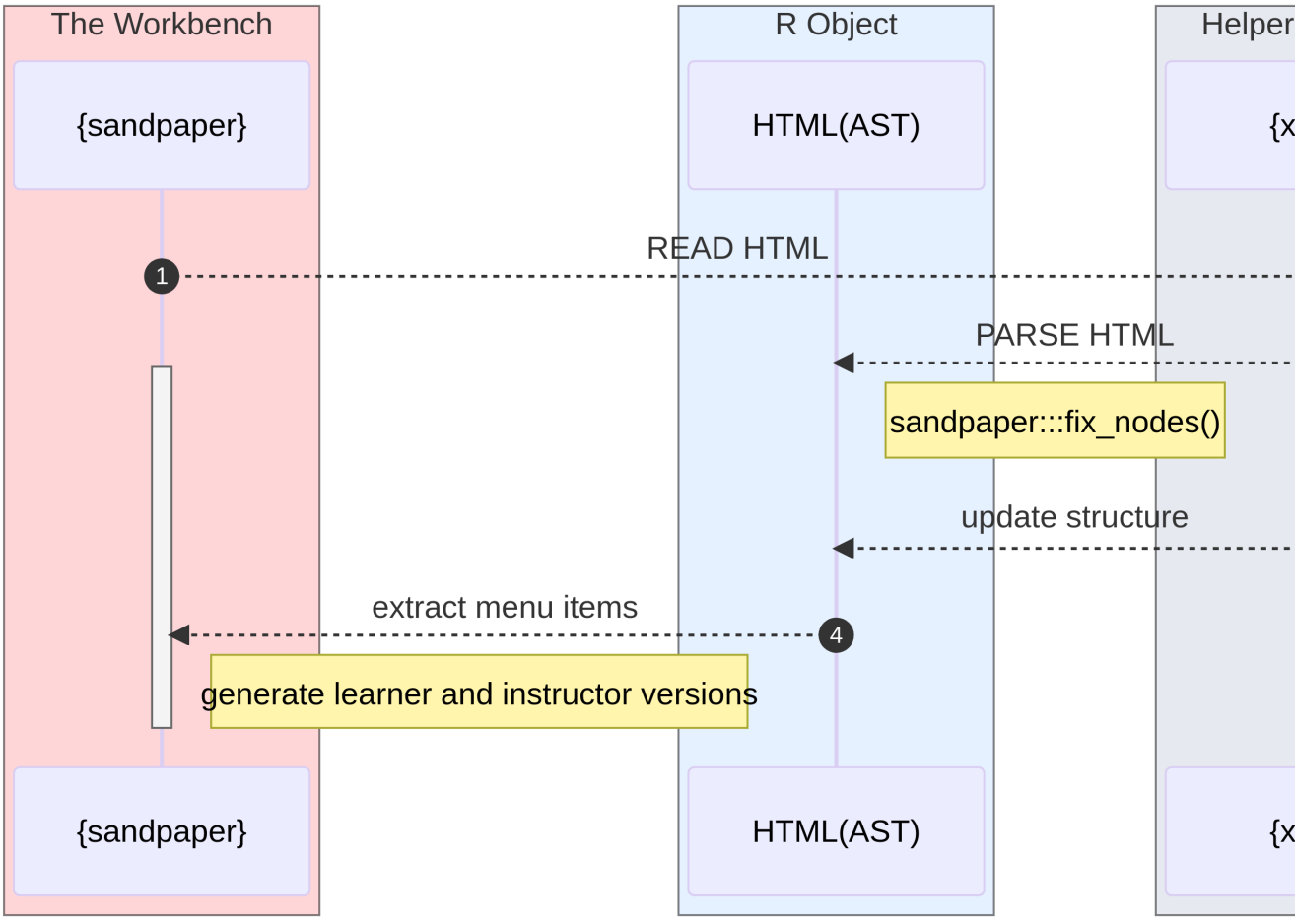


From here, the HTML exists as the internal body content of a website without a header, footer, or any styling. It is nearly ready for insertion into a website template. The next section details the flow we use to tweak the HTML content.

4.5.2 Processing HTML

The HTML needs to be tweaked because the output from pandoc, even with our lua filters, still needs some modification. We tweak the content by first converting the HTML into an Abstract Syntax Tree (AST). This allows us to programmatically manipulate tags in the HTML without resorting to using regular expressions.

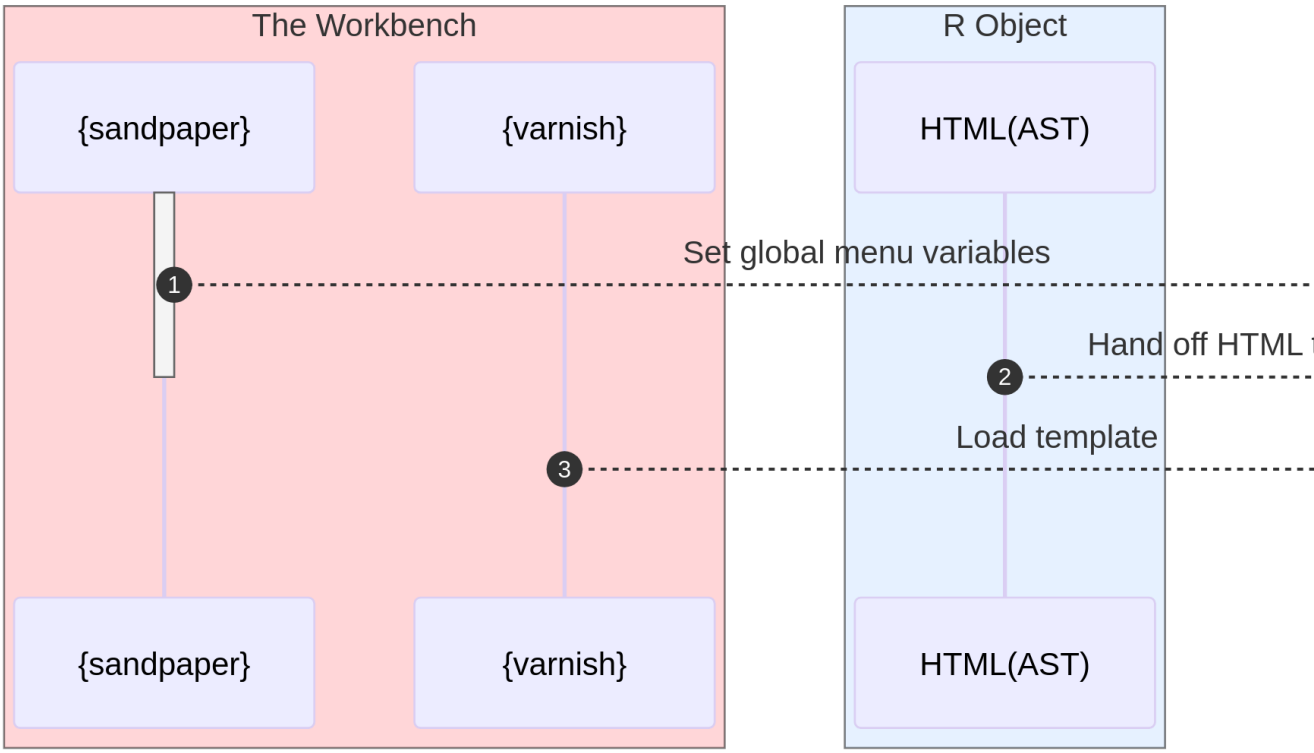
In this part, we update links, images, headings, structure that we could not fix using lua filters. We then use the information from the episode to complete the global menu variable with links to the second level headings in the episode.



> **Working With XML**
>
> Working with XML data is perhaps one of the strangest experiences for an R user because in R, functions will normally return a copy of the data, but when working with an XML document parsed by {xml2}, the data is modified *in place*.
> It allows us to do neat things, but there is a learning curve associated.

4.5.3 Applying Website Template

Now that we have an HTML AST that has been corrected and associated metadata, we are ready to write this to HTML. This process is achieved by passing the AST and metadata to {pkgdown} where it performs a little more manipulation, applies the {varnish} template, and writes it to disk.



Part II

{sandpaper} User Interface

5 The {sandpaper} package

5.1 Introduction

> sandpaper (n.)
>
> Heavy paper coated on one side with sand or other abrasive material and used for smoothing surfaces.

The {sandpaper} package is the user interface for The Carpentries workbench and orchestrates the building and deployment of lessons. It helps lesson developers, authors, and maintainers to smooth out their contributions. Because of it's user-facing and modular nature, it is the most complex package in The Workbench.

People who want to use {sandpaper} will generally use it for one of these five things[1]:

1. Creating lessons
2. Contributing to lessons
3. Maintaining lessons
4. Rendering a portable lesson site
5. Rendering a lesson site with continous integration (GitHub Actions)

Importantly, **all of these points must be achievable by someone with little to no experience with R or any other programming language.**

5.2 Not a static site generator

One important distinction I would like to make is that *{sandpaper} is not a static site generator*. Yes, it may act like a static site generator because it creates static sites that are portable from markdown sources. However, it differs in that it is not intended to be as flexible as many other static site generators. This may seem like a negative point, but in the context of Carpentries Lessons, it is an asset.

Many static site generators are extremely flexible at the cost of requiring the user to think deeply about the model of the website structure, deployment, and maintenance. For a single

[1] As has been the case ever since the README was first written in August 2020 before any code was created. Fun fact, much of the README remains in tact and accurate: README 2020-08-04

website, this is fine, but for distributed lessons like those in The Carpentries, it is much more difficult to use a static site generator because lesson maintainers should only have to focus on the content, not the mechanics or style. The {sandpaper} package builds *lesson websites* and nothing more. It is possible to use it for a narrative analysis, but at the end of the day, it will still be a lesson website.

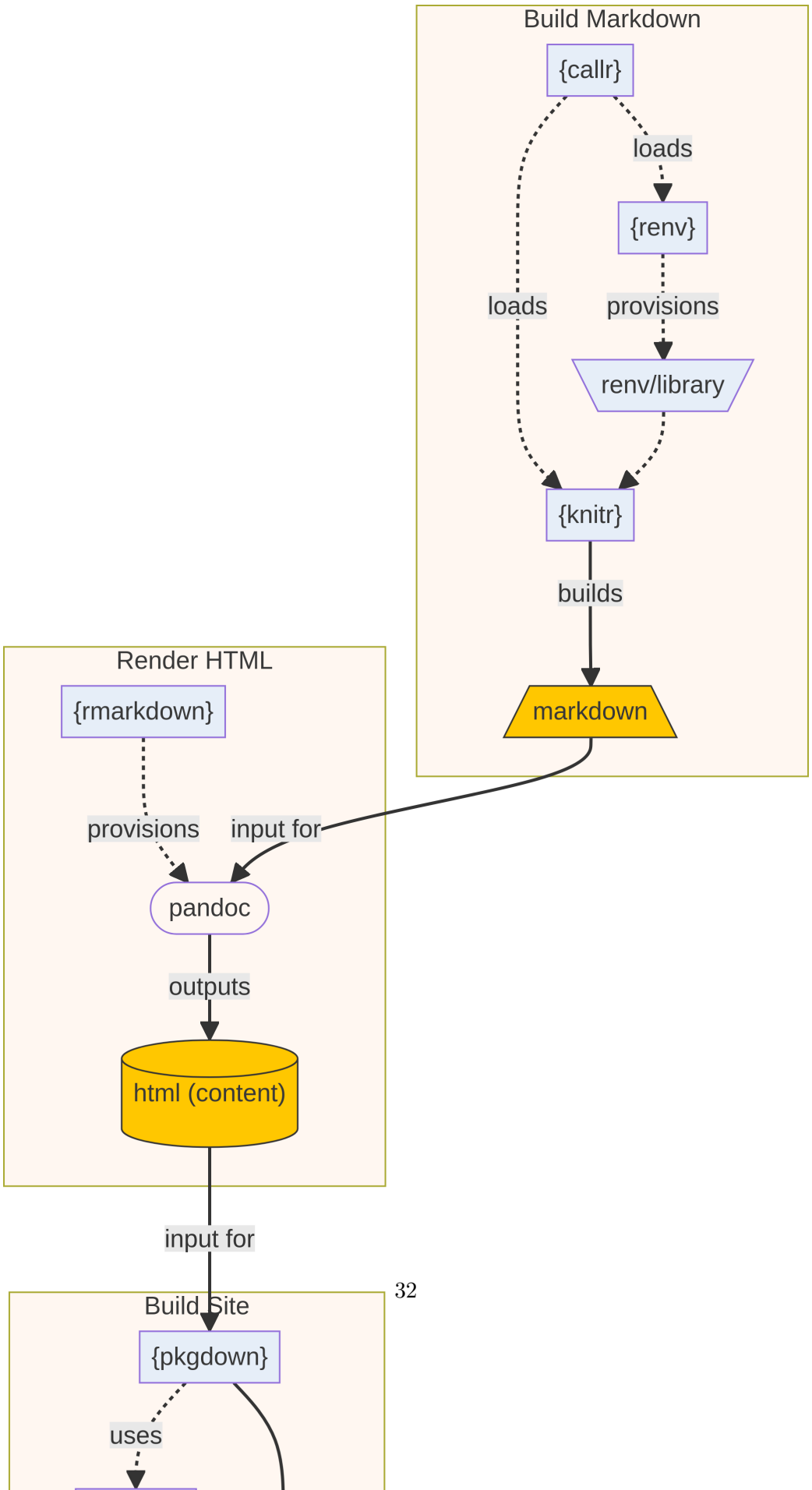
5.3 You had ~~one~~ several jobs

In terms of it's relation to the other Workbench packages, {sandpaper} depends on {pegboard} to validate lessons, extract questions and timings for the schedule, and to extract the code handout. It *relies* on {varnish} to provide the HTML, CSS, and JS templates that create the websites via {pkgdown}. Its other dependencies are varied in purpose. It relies on other R packages and pandoc to do much of the work.

5.3.1 Building Websites

At it's core, {sandpaper} provides a workflow to process R Markdown to Markdown (if the lesson is purely markdown-based, then it Markdown is simply copied), generate HTML, and package that HTML content into a website framework with the following ideals:

1. When processing R Markdown, the user's environment should not be modified
2. Generated content should be auditable, but not part of the main git history
3. Processes for generating markdown, HTML, and the website should be modular such that if a better tool comes along, it can serve as a drop-in replacement


Build Markdown
{callr}
loads
{renv}
loads
provisions
renv/library
{knitr}
builds
markdown
Render HTML
{rmarkdown}
provisions
input for
pandoc
outputs
html (content)
input for
Build Site
{pkgdown}
uses

6 Testing {sandpaper}

6.1 Introduction

{sandpaper} is the largest package and the main user and deployment interface for The Workbench. The tests are all designed to work within a lesson context, which means a couple of things to be aware of.

1. **Some tests will take a long time to run.** IO procedures are often one of the most time-consuming steps in computing and {sandpaper} does a lot of it.
2. **Most tests will be dependent on the previous tests in a file.** Because the tests work on a folder on disk, we need to provision different scenarios of lesson state. It is far simpler to do this by side-effect rather than copying, setting up, and tearing down the state of the lesson for each and every test.

6.2 Test Setup

There is nothing that you as the contributor/developer need to do to set up for running these tests beyond what you have already done in your development setup. This section describes conceptually how {testthat} and {sandpaper} setup the testing environment after you run `devtools::test()`.

In short, this is the process:

1. {sandpaper} is loaded
2. Helper test files are loaded
3. The setup script is loaded, which provisions the test lesson
4. Each test file matching`tests/testthat/test-*.R` is run, resetting the test lesson at the top of each file.

6.2.1 Test Helpers

Sandpaper has three test helpers that handle some of the more tedious side-effects of testing.

`helper-hash.R` Expectation `expect_hashed()` that an episode file MD5 sum (expected) matches the MD5 sum (actual) we recorded in the `site/built/md5sum.txt` database.

`helper-processing.R` Provides an output string that demonstrates a screen output of a file being processed by {knitr}. This is used with `expect_output()`.

`helper-snap-transform.R` A function that is passed to the transform parameter of `expect_snapshot()` and will mask the temporary directory path so that the snapshot does not continuously invalidate on every run.

6.2.2 Setup Script

The first script to run is `tests/testthat/setup.R`, where a test lesson and a local git remote is created and stored in a temporary location for the duration of the test suite and a reset function is exposed for the tests.

6.3 Conditionally Skipped Tests

Each link below will open a code search for the different types of skipped tests in {sandpaper}

`skip_on_os` These are often tests that are skipped on Windows, usually for the reason that {renv} and filepaths behave slightly differently on Windows in a continuous integration setting.

`skip_if` Tests that are skipped under various conditions. For example, if Git is not installed on a system, we should not test any of the CI functions because they rely on Git.

`skip("<for reasons>")` This pattern is more rare. It's really useful in a situation where you are refactoring and know that a lot of tests will fail. If you sprinkle in these skips, you can focus on testing the core functionality of the refactor and then address the side-effects. Regarding the skips that remain: during testing, sometimes we encounter a ghost in the machine and we cannot set up the right conditions to run the test properly or the test was created with an earlier model of the package that we haven't been able to shake. In these cases, instead of deleting the test or commenting out code, we add the `skip()` function and write a message of *why* we skipped it so if we need to come back to it later, we can.

6.4 Continuous Integration

6.4.1 Package Cache

Running tests on Continuous Integration is tricky in part because we need to set up a {renv} package cache to work on Mac, Windows, and Linux systems. In practise, we have to set up a specific `RENV_PATHS_ROOT` folder for each system.

6.4.1.1 Windows

For Windows, the setup is even more complex because there are weird caveats in how pandoc and {renv} work on the CI version of Windows.

6.4.2 Dependencies

At the moment, we test the current versions of dependencies when we are running tests in the `test-coverage.yaml` file. For the `R-CMD-check.yaml` file, however, we test the development version of {renv}. The reason *why* we do this is because in March 2023, {renv} 0.17.0 was released and subsequently broke bioconductor-based R Markdown lessons and new R Markdown lessons that needed to be bootstrapped (see sandpaper#406).

This can lead to a situation where the tests will pass on the test coverage check, but fail for `R CMD check`, which is diagnostic because it tells us that there is an upstream issue in {renv} that we can address before it becomes a problem after a CRAN release.

7 The R package cache

7.1 Introduction

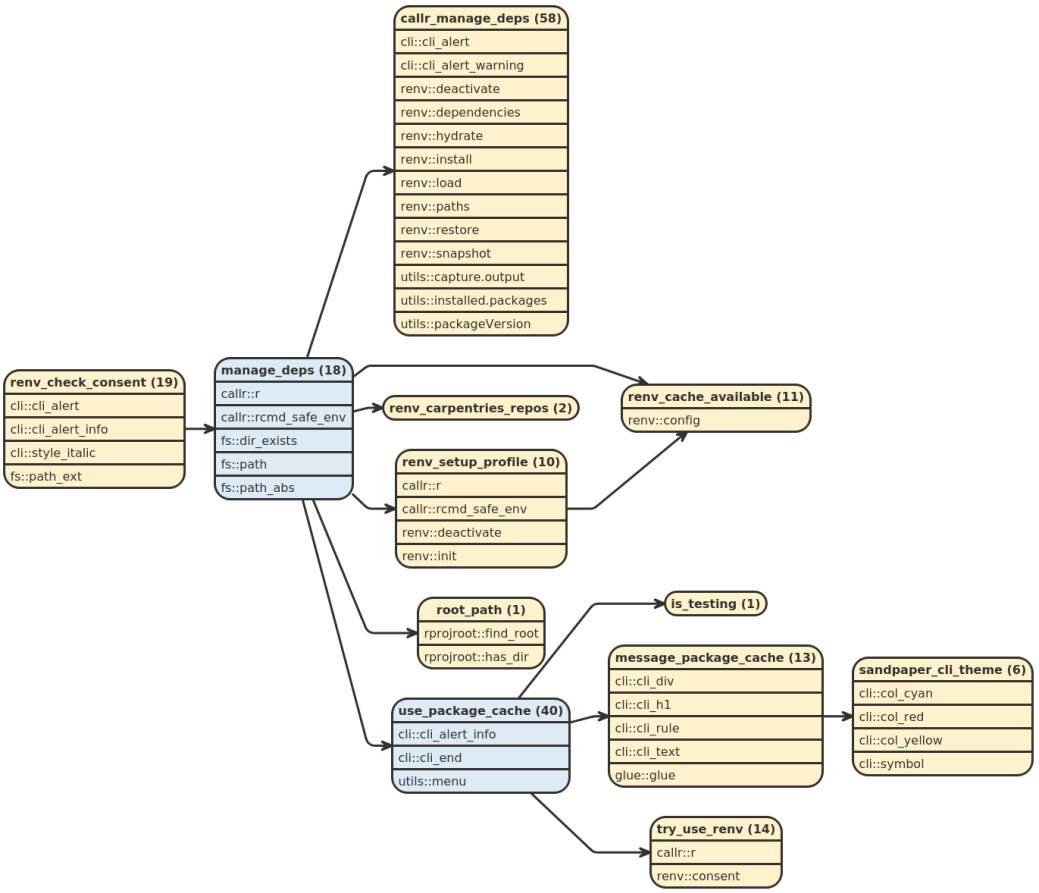
All lessons that use The Workbench can build using either Markdown or R Markdown file formats. The R package cache allows for R Markdown file formats to be built reproducibly and consistently. The cache is expected to be mindful of these formats in the following ways (as taken from Building Lessons With A Package Cache):

- *reliable setup*: the version of the lesson built on the carpentries website will be the same as what you build on your computer because the packages will be identical
- *environmentally friendly*: The lesson dependencies are NOT stored in your default R library and they will not alter your R environment.
- *transparent*: any additions or deletions to the cache will be recorded in the lockfile, which is tracked by git.

The package cache is *only* used when building lessons with R Markdown elements and performs the folowing tasks in a separate R session to avoid polluting the user's environment:

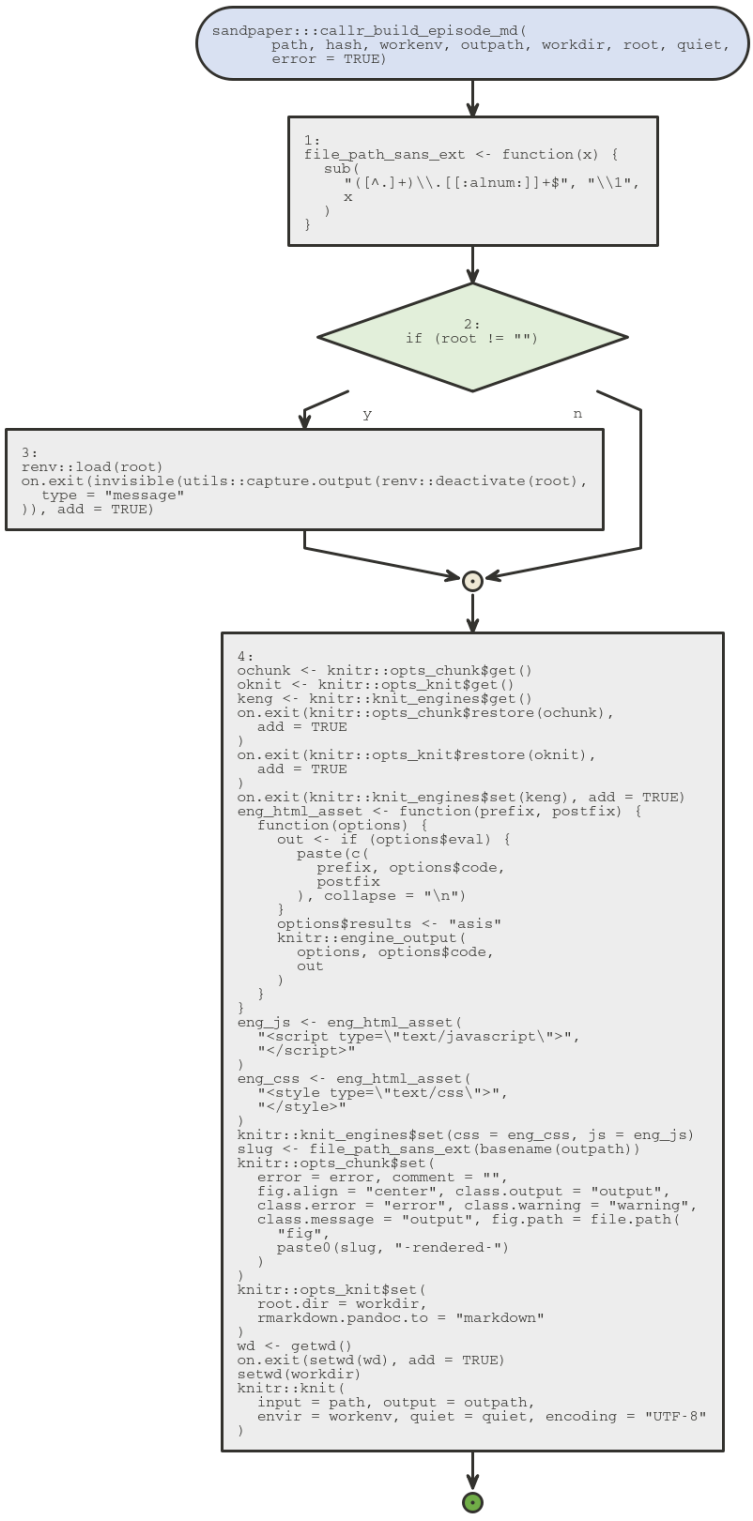
1. *Before markdown is built*: We **check for consent** to use the package cache with `sandpaper:::renv_check_consent()` and then provision any packages needed for the lesson with `sandpaper::manage_deps()`

   > **i** Dependency Tree for `renv_check_consent()`
   >
   > The `renv_check_consent()` checks if the user has run `sandpaper::use_package_cache()`, which allows {renv} to create and maintain a global package cache on their system.



2. *During each markdown rendering*: If we have consent, we **load the {renv} profile** to reproducibly render the markdown document

> **i** Flow Diagram for `build_episode_md()`
>
> This flow represents the build process for *each R Markdown file*. This function is called from within a separate R process. The only conditional here determines if the {renv} environment needs to be loaded.


sandpaper:::callr_build_episode_md(
path, hash, workenv, outpath, workdir, root, quiet,
error = TRUE)
1:
file_path_sans_ext <- function(x) {
sub(
"([^.]+)\\.[[:alnum:]]+$", "\\1",
x
)
}
2:
if (root != "")
y
n
3:
renv::load(root)
on.exit(invisible(utils::capture.output(renv::deactivate(root),
type = "message"
)), add = TRUE)
4:
ochunk <- knitr::opts_chunk$get()
oknit <- knitr::opts_knit$get()
keng <- knitr::knit_engines$get()
on.exit(knitr::opts_chunk$restore(ochunk),
add = TRUE
)
on.exit(knitr::opts_knit$restore(oknit),
add = TRUE
)
on.exit(knitr::knit_engines$set(keng), add = TRUE)
eng_html_asset <- function(prefix, postfix) {
function(options) {
out <- if (options$eval) {
paste(c(
prefix, options$code,
postfix
), collapse = "\n")
}
options$results <- "asis"
knitr::engine_output(
options, options$code,
out
)
}
}
eng_js <- eng_html_asset(
"<script type=\"text/javascript\">",
"</script>"
)
eng_css <- eng_html_asset(
"<style type=\"text/css\">",
"</style>"
)
knitr::knit_engines$set(css = eng_css, js = eng_js)
slug <- file_path_sans_ext(basename(outpath))
knitr::opts_chunk$set(
error = error, comment = "",
fig.align = "center", class.output = "output",
class.error = "error", class.warning = "warning",
class.message = "output", fig.path = file.path(
"fig",
paste0(slug, "-rendered-")
)
)
knitr::opts_knit$set(
root.dir = workdir,
rmarkdown.pandoc.to = "markdown"
)
wd <- getwd()
on.exit(setwd(wd), add = TRUE)
setwd(workdir)
knitr::knit(
input = path, output = outpath,
envir = workenv, quiet = quiet, encoding = "UTF-8"
)

This chapter will go into the history of using a package cache in Carpentries lessons, dig into the design principles, and understand challenges for moving forward.

7.2 A Bit of History

As of this writing, The Workbench is able to reproducibly build and deploy lessons across machines and infrastructures thanks to the package cache provided by {renv}, but it is important to understand how we got here and what the motivations were, because the tools we have now simply did not exist in 2016. This section dives a bit into the history of writing R markdown content in Carpentries lessons and what lessons… were learned.

Software Carpentry Lessons have been able to handle content written in R Markdown since July 2014. This process was disrupted in June 2016 with the release of the new styles template, but luckily, François Michonneau swooped in to the rescue by providing a templating setup that would not only render the Markdown, but keep the output separate. The very next month, François submitted carpentries/styles#83, which added the capability to automatically detect and install packages needed to build an R-based lesson.

Of course, back in 2016, in order to deploy an R-based lesson, you still had to build it locally, which sounds simple until you consider the aspect of reproducibility (Marwick 2016; Wilson et al. 2017). If you build the same document on two different machines, there is no guarantee that you will get the same results. Thus, in May 2018, Raniere Silva added the ability to build R-based lessones on Continuous Integration. Finally, in 2020, Maxim Belkin added GitHub Workflows to the styles repository so that we no longer had to rely on TravisCI.

These changes allowed a single, definitive source for lessons to be built, but alas, they still were not *reproducible* because the packages used to build the lesson were always being run with the most recent versions. This lead to problems with outputs changing or worse, the entire build failing (see swcarpentry/r-novice-gapminder#746. Moreover, lesson maintainers of thes R lessons encountered the following problems:

1. Every time they built their lessons locally, their default R package library would update. This was especially a problem for maintainers who were working on their dissertations and really could not afford to lose work due to their packages changing.
2. Maintainers were unsure of what would happen to the lesson with any given pull request and would have to manually run the results or trust the contributor.
3. All the normal struggles with Jekyll.

Reproducibility is *hard*. Software ecosystems are always shifting like the sand dunes in the Sahara (Vaniea and Rashidi 2016). By late 2020, we had come a long way in terms of automating the build process for R-based lessons, but there were still many hills to climb. It was in this context that we developed the use of {renv} and the R package cache to automate

package provisioning, caching, and auditable updating so that R-based lessons could reliably be deployed with no surprises.

7.3 Design Principles

It is worth reading through carpentries/sandpaper#21 to see the discussion and thoughts around the origin of the design for using this feature. It was implemented during a three week period between 2021-08-24 and 2021-09-16, as detailed in the pull request carpentries/sandpaper#158.

It's easy to think about a package cache as a way to declare dependencies for a lesson in a reproducible way, but it's more than that. The philosophy of R packages on CRAN is that they all need to work with the latest versions of each other, so a lesson that teaches R should be reasonably up-to-date. Thus, any given package must be able to do the following four things in the package cache:

1. enter the package cache and record the version number
2. exit the package cache and be removed from the record
3. update to the latest version
4. pin to a specific version

Every good tool for handling a package cache does these four things, and the workflow to add a new package with {renv} (before version 1.0.0) flows like this:

1. Contributor A would like to add {packageA} to the project
2. Contributor A opens the project and installs {packageA}
3. Contributor A adds {packageA} to the project content
4. Contributor A takes a snapshot of the project to the lockfile

In this scenario, Contributor A must explicitly install the package to the project before they can use it regardless of whether or not they have that package on their system. Steps 2 and 4 of this workflow involve explicitly working with the package cache and, in my experience, R users will often jump directly to step 3, which leads to failed builds and frustration.

When designing a user interface, you also need to think about what distractions and real-life stressors the lesson maintainer/developer/contributor is potentially dealing with. You want to minimize the amount of fuss that the contributor needs to do to get something working. **A contributor should only have to add a package once to the lesson content to make it available.** This means that the following steps are taken to add a new package:

1. Contributor A has {packageA} in their library and wants to demonstrate the use of {packageA} in the lesson.

2. Contributor A inserts a code block in the lesson that declares `library("packageA")` or `packageA::fun()` somewhere in the code block:

````
```{r}
# load the packageA package
library("packageA")
```
````

After that, when the lesson is built, {packageA} is automatically added to the lockfile. If Contributor A decideds to no longer use {packageA}, they can remove it from the lesson content and the next build will remove {packageA} from the lockfile. In this way, the lesson content remains the source of truth for the packages used in the lesson and the lockfile represents the metadata associated with the lesson.

That means the following WRT to packages in a lesson:

1. Users should not have to know that the lockfile exists
2. The packages used in the lesson should be locked to specific versions and be reproducible across machines
3. The packages used in the lesson should *not* overwrite the packages in the user's default R library
4. Any package that is missing from the user's machine should automatically be provisioned
5. All packages used should be the correct version
6. The lockfile should be auditable
7. The lockfile defining the package versions should update/remove packages according to the contents of the lesson
8. Users should be able to specify versions in the lockfile easily
9. Users should be able to automatically update the lockfile

Part III

{pegboard} Validation and Parsing

8 The {pegboard} package

8.1 Introduction

The {pegboard} package was the very first package written for The Carpentries Workbench. It's initial purpose was to parse the lessons based on the styles lesson infrastructure to figure out how lesson authors and maintainers were using the challenges and solutions. You can see this analysis in a vignette I started in May 2020 [1].

It's purpose now is two-fold:

1. parse and validate the lessons for structural markdown elements
2. translate Carpentries-style materials from the styles lesson infrastructure (Jekyll-based) to The Workbench (Pandoc-based)

8.1.1 Dependencies

The dependency footprint of {pegboard} is intended to be small. The package is intended for validation and translation. It is meant to be stable.

Pegboard was built on top of the {tinkr} package, intially developed by Maëlle Salmon (rOpenSci) and now maintained by Zhian Kamvar. This package uses the CommonMark C library to parse Markdown into XML and then uses a custom XSLT stylesheet to translate Markdown back to XML. One of the key advantages that CommonMark's XML gives us is the ability to extract line numbers and positions for markdown elements, which allows us to accurately report any Markdown issues to the user.

The objects used in pegboard are created with the {R6} package, which implements *encapsulated object orientated programming* for R. This style of programming is *very* similar to that of Python or Java. One of the reasons why we use {R6} is because objects created in this system are *modified in place* by their methods. This is important because the package to manipulate XML data, {xml2}, is built directly on top of the libxml2 C library, which also modifies objects in place, but it's not inherently obvious when you work with them, so having a formal system like {R6} to encapsulate them makes more sense than a functional programming framework.

[1] This vignette no longer exists. It stopped working in November 2021, because of updates to the dependencies and the lessons. The vingette took a few minutes to build because it needed to download the lessons, and it was no longer appropriate as {pegboard} was shaping up to be the lesson validator and translator.

Part IV

{varnish} Web Styling

9 The {varnish} package

9.1 Introduction

The {varnish} package is a weird little package in that it does not contain any actual R code. It's purpose is to host HTML templates along with the CSS and JavaScript needed to display the lesson.

We take advantage of the fact that the only thing actually *required* to install an R package is the presence of a `DESCRIPTION` file[1]. These all live inside the `inst/` folder, which is a place that allows files to be installed along with the package.

The {pkgdown} package uses this as a mechanism for package developers to override the default styling, creating customized documentation websites such as the rOpenSci documentation sites: https://docs.ropensci.org/rotemplate/.

This allows people to update the lesson styling however they wish[2] and while we *could* include it in {sandpaper}, it's best kept separate so that people can update {varnish} without needing to update the entire tool suite.

9.2 Design and Implementation Background

The design for the frontend was created by Emily de la Mettrie in 2021 after consultation with Zhian N. Kamvar and François Michonneau using examples from The Unix Shell and parts of Exploring Data Frames for content cues.

The final figma design project[3] was then handed off to a team at Bytes.co, who translated the designs to CSS and JavaScript, subcontracted an a11y testing company to interactively test the prototype for a11y issues.

The prototype we recieved from Bytes.co was a Jekyll template serving HTML files. Zhian created a staging repository called shellac to transform the site from one that was served via

[1] for example, a DESCRIPTION file is a low-rent way of specifying depedencies for a manuscript).

[2] There is a caveat, though. There are some components, such as the sidebar and overview callouts that are built on the {sandpaper} side and then fed into the template.

[3] The final figma design project for The Workbench can be found in this link. To preview what it looks like, click the "Present" button (iconized as a triangle play button) in the top right.

static site generator to one that was standalone. The preview is preserved at https://zkamvar.github.io/shellac/student_carpentries.html.

This site was then stripped of the added into {varnish} in carpentries/varnish#14 between 2022-01-10 and 2022-01-24, when the 1.0.0 release of {varnish} was created and the sandpaper docs website was updated to use the new version of the HTML, CSS, and JavaScript.

Part V

Package Distribution

10 Release Process for Workbench Packages

10.1 Background

The workbench contains three main packages:

- {sandpaper}: user interface and workflow engine
- {pegboard}: parsing and validation engine
- {varnish}: HTML templates, CSS, and JS elements

Each of these packages are available on the Carpentries R-Universe and new versions are checked for hourly. This allows folks to get up-to-date versions of The Workbench packages built for their system without running out of GitHub API query attempts.

In order to maintain quality, packages are only sent to the R-Universe if they have been formally released on GitHub (as specified in the packages.json configuration file). This allows us to incrementally add new experimental features without changing the stable deployments.

10.2 Release Process

When a package is ready for release we use the following checklist:

- ☐ Update version number in DESCRIPTION
- ☐ Add NEWS for the changes in this version
- ☐ Ensure all changes are committed and pushed
- ☐ add new signed tag with the name " X.Y.Z"

```
# example: create a signed (-s) tag for sandpaper 3.3.3
git tag -s 3.3.3 -m 'sandpaper 3.3.3'
```

- ☐ create a release on github from the new tag

> **Note**
>
> Zhian likes to create tags via the command line because he has set up his git configuration to use a gpg signature so the tags and the releases are both verified.

The last two items can be achieved in a single step with the github cli with the command `gh release create X.Y.Z` for the version number

```
gh release create 3.3.3
# ? Title (optional) sandpaper 3.3.3
# ? Release notes  [Use arrows to move, type to filter]
#   Write my own
# > Write using generated notes as template
#   Leave blank
```

Selecting "Write using generated notes as a template" opens an editor and populates it with the pull requests that have been accepted since the last release.

Once the relase is created on GitHub, then the package will be available on the R-Universe in about an hour or less.

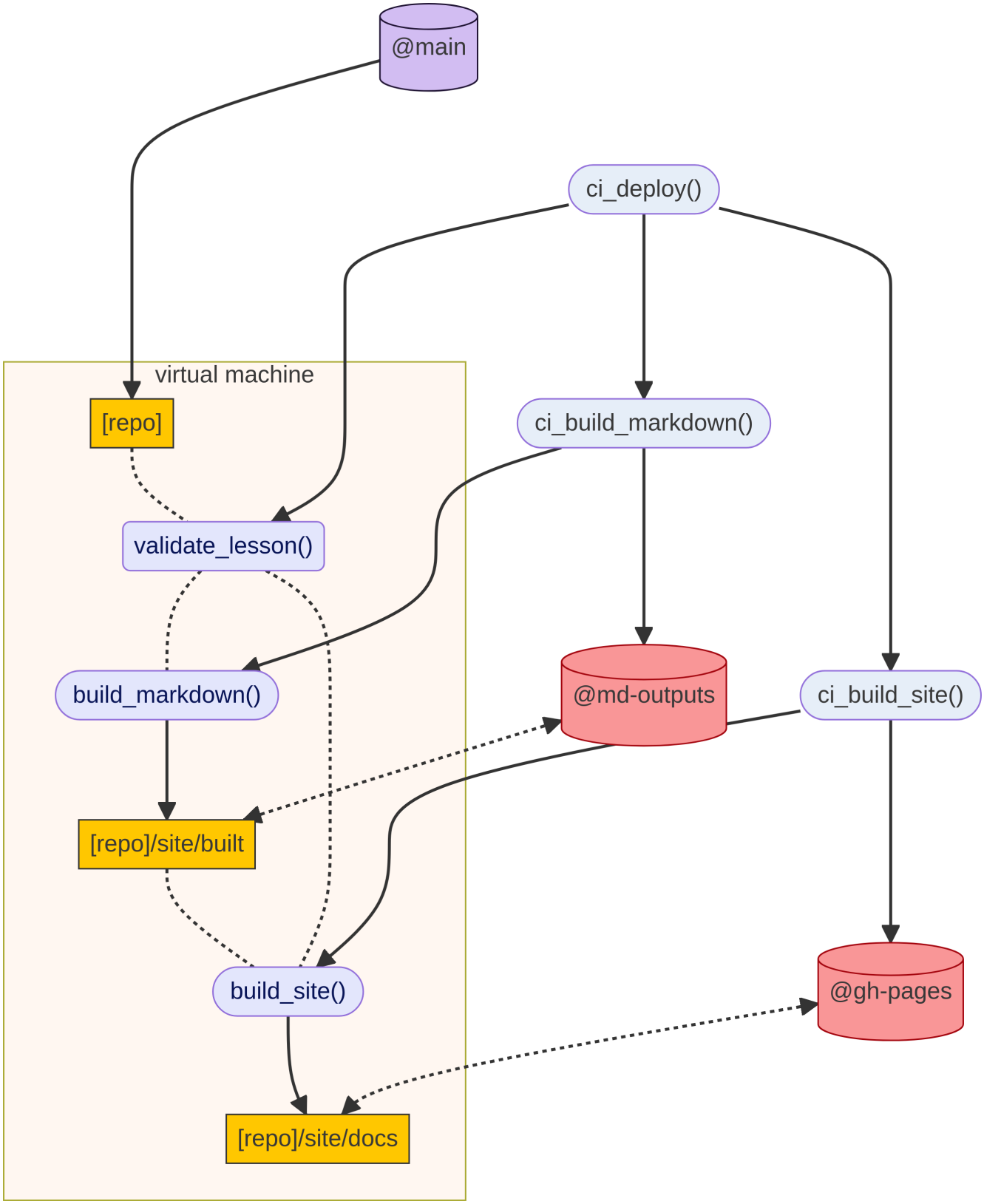
Part VI

Remote Deployment and Management

11 Introduction

11.1 Philosophy

The Workbench was built to be platform-independent, so that you could continue to deploy its features without relying on specific features of GitHub. We use the exact same two step process of building the markdown and then passing that to the HTML renderer, with a twist. The core deployment function in the workbench is `sandpaper:::ci_deploy()`, which will deploy the rendered markdown and the HTML to separate *orphan branches* in the Git repository that are mapped as *git worktrees* to the `site/built` and `site/docs` folders during the build process. In each step, when the build is successful, the results are pushed to the respective worktree before moving to the next step. When the process is done (regardless of outcome), the worktrees are torn down gracefully.


@main
ci_deploy()
virtual machine
[repo]
ci_build_markdown()
validate_lesson()
build_markdown()
@md-outputs
ci_build_site()
[repo]/site/built
build_site()
@gh-pages
[repo]/site/docs

This allows us to retain the commit history from building not just the HTML, but also the markdown outputs without interfering with the commit history for the lesson source. It *also* gives us the ability to use these branches as a cache so that the lesson doesn't have to re-build from scratch every time, but the biggest advantage is in the things that go beyond just deploying lessons.

11.1.1 Proof

One way to prove this works on any system that uses Git, you can create a lesson, push it to GitHub, and then render the contents before GitHub is done setting up its runners.

The file `test-deploy.R` will do just that[1]:

```
# Load The packages ------------------------------------
library(sandpaper)
library(usethis)
library(withr)
library(ids)
library(fs)
# Generate the Lesson ----------------------------------
tmp <- tempfile()
id <- paste0("TEST-", adjective_animal(style = "kebab"))
dir_create(tmp)
lsn <- path(tmp, id)
create_lesson(lsn, open = FALSE)
# Push the Lesson To Github ----------------------------
with_dir(lsn, {
  use_github()
})
# Render and Deploy the Lesson -------------------------
with_dir(lsn, {
  sandpaper:::ci_deploy()
})
# Set GitHub Pages -------------------------------------
with_dir(lsn, {
  use_github_pages()
})
```

You must run it in a non-interactive fashion:

[1]Please note: this will only work if you have a GitHub PAT set up so that {usethis} can interact with the GitHub API.

```
Rscript remote/test-deploy.R
```

Now you can visit the GitHub repository and if you wait ~30 seconds, GitHub will have created a website for you and will still be setting up the lesson engine. This shows that it is possible to deploy as long as you have the following:

1. A system with {sandpaper} and Git set up properly
2. push access to a remote Git repository

In fact, if you look at the example for `ci_deploy()`, you will see that it creates a lesson and remote repository and walks you through the process that happens.

The challenge when deploying a Workbench lesson then lies in the step of provisioning the virtual machine or docker container to build a lesson when it updates.

11.2 Beyond Deployment

Having a single workflow for deployment is fine, but in the context of a lesson that will generate its content, other tools are needed to avoid the element of surprise from taking over when a change is made to the lesson. On the converse side, tools are needed to bring in updates that can affect the security and accuracy of the lesson.

11.2.1 Pull Request Management

The norm for working on GitHub is a trunk-based workflow—small branches containing different features or bug fixes are created and then merged into the default branch after review. If new content is added or packages update, it is important to have mechanisms to verify that the contents of a lesson and to intervene if something is incorrect before the changes happen.

11.2.2 Updating Compontents

The update workflows are there because we understand that a data science lesson does not live in isolation and it cannot be built in isolation—contents and tools need to be updated as the software ecosystem changes. Thus, just like we provide the {sandpaper} functions sandpaper::update_cache() and sandpaper::update_github_workflows(), these are also available as GitHub workflows that will create a pull request (if it has permissions).

11.3 In Practice

We use GitHub Workflows to build and deploy our lessons[2] and the rest of the chapters in this section will discuss *how* we set these up, but *within the context of GitHub*. Remember that our philosophy is that the workbench should be deployable anywhere. These workflows are responsible for provisioning GitHub's Ubuntu 22.04 Runner Image with the packages and software needed to build a lesson with The Workbench.

11.3.1 Workflows

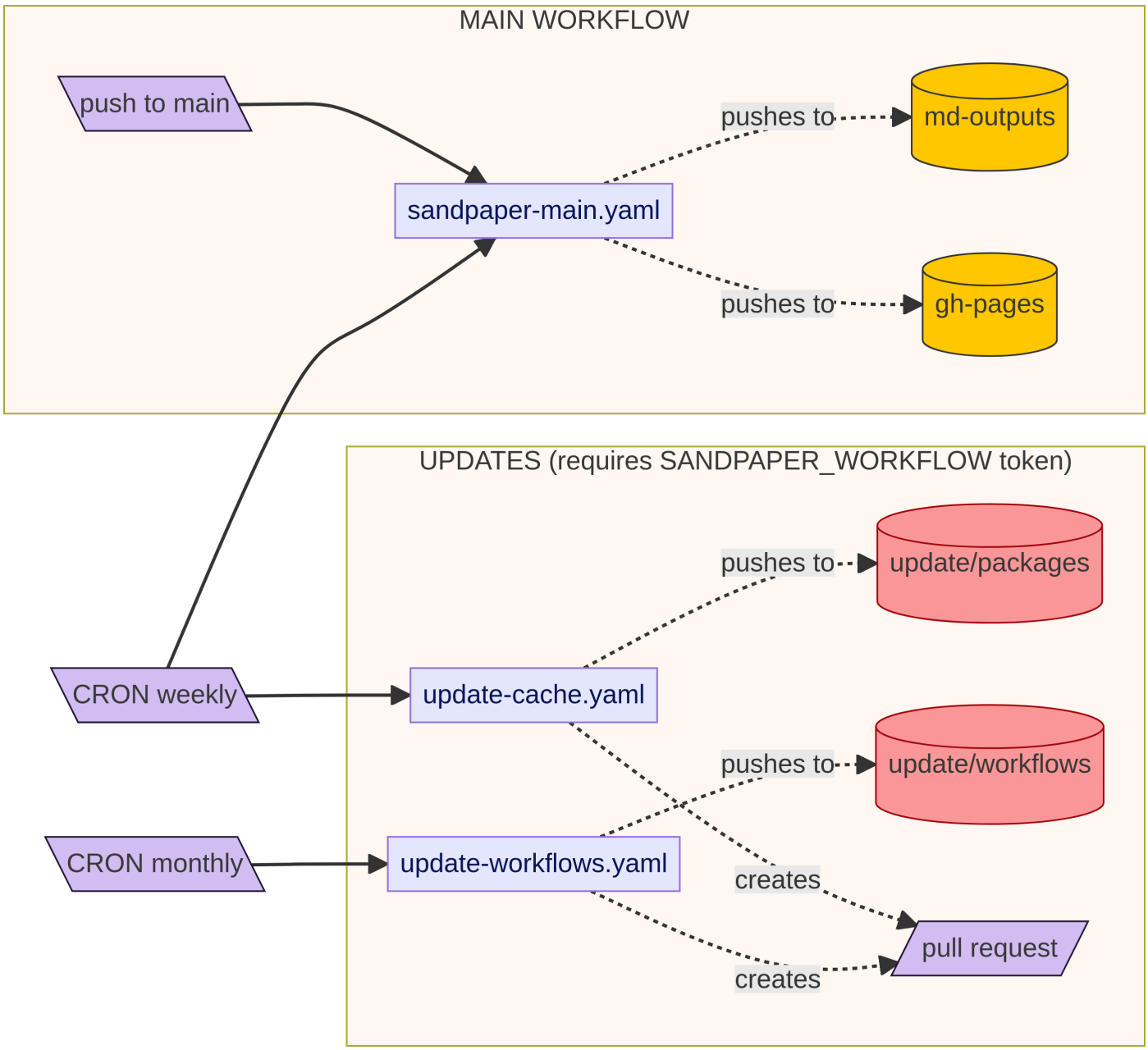
There are broadly four categories of workflows, where an asterisk (*) denotes workflows that can be manually triggered by maintainers and a carrot (^) denotes workflows that require a personal access token to create a pull request

1. Deployment* (`sandpaper-main.yaml`)
2. Pull Request Responders (`pr-preflight.yaml`, `pr-receive.yaml`)
3. Updates*^ (`update-cache.yaml`, `update-workflows.yaml`)
4. Pull Request Preview Managers (`pr-comment.yaml`, `pr-close-signal.yaml`, `pr-post-remove-branch.yaml`)

These workflows are individually documented in the sandpaper repository

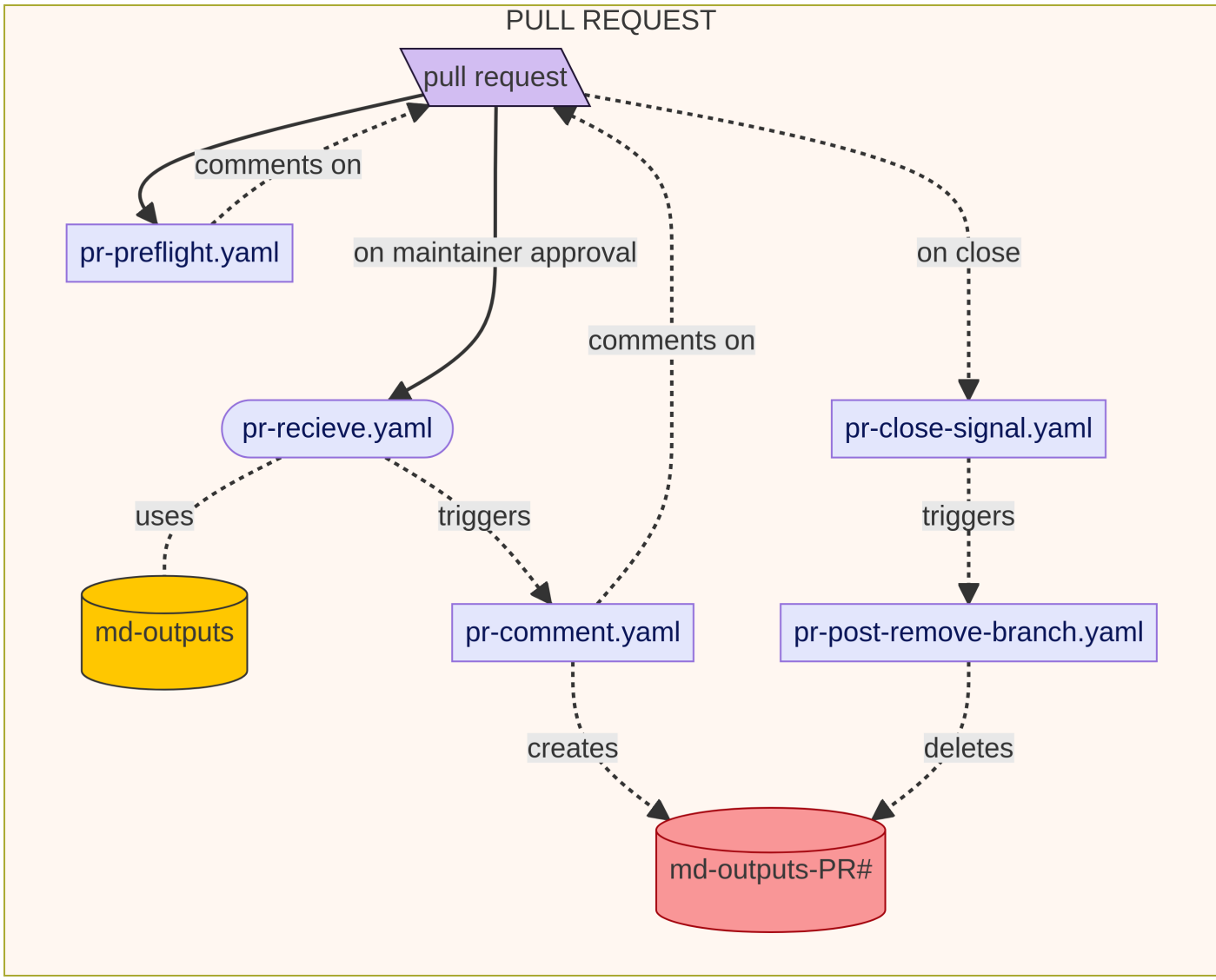
These workflows are interrelated and have different triggers. Below are a set of diagrams that disambiguates these relationships. First up are the workflows that are run on a schedule **and on demand**. Note that the update workflows will only push to a branch if any updates exist, otherwise, they will exit silently.

[2]GitHub can be a bit confusing with it's terminology and fluid concepts. Their resource for Understanding GitHub Actions may help, but here's how I think about it In this publication, whenever I refer to a *GitHub Workflow*, this is a YAML file that lives inside of a repository that tells GitHub how to set up its machine to build the lesson. It's like a recipe plan and shopping list for a dinner. On the other hand, when I refer to a *GitHub Action*, this is a self-contained piece of software that will perform a specific task *within a workflow*. This is akin to a specific kitchen utensil, ingredient or spice within a recipe.



Notice how **none of the workflows push to main**. The update workflows will push to the `update/*` branches and then create a pull request. It's common to find workflows that will perform updates and then immediately push to the default branch (which is the case for the lesson-transition workflow), but it's important to remember that a workflow that does *automatic* updates prevents the maintainers from critically inspecting the changes to the com-

ponents. This is especially true of the `update-cache.yaml` workflow, which will update the {renv} lockfile. By passing it through the pull request process first, we can give the maintainers a way to audit the changes coming through.



11.3.2 Actions

These workflows use a series of Custom GitHub Actions (aside from the official GitHub actions of `checkout` and `cache`) which can be found in the following repositories:

- https://github.com/carpentries/actions a combination of both composite and JavaScript Actions that perform the duties for provisioning the workbench, provisioning packages for R-based lessons, validating pull requests, downloading data from previous runs, commenting on pull requests, and updating components.
- https://github.com/r-lib/actions similar to carpentries/actions, but these are used in our workflows to provision R (that is, set up the correct environment variables) and to provision pandoc. Many of these actions are designed for packages and we use them heavily in the workbench development.
- https://github.com/carpentries/create-pull-request a fork of a popular action that will create a pull request *from a Github Workflow*. This is a fork so that we can make sure that we will keep it secure.

Each repository has the actions documented to a degree, but we will discuss the implications and design of the actions in a following chapter.

12 Deployment

As we saw in the previous chapter, deployment of lesson content requires two things:

1. a system provisioned with R, pandoc, and Git that is capable of installing any R packages (including those that require compillation or external C libraries).
2. access to a remote Git repository with the ability to create and push branches.

Point 2 is easily taken care of by any remote Git hosting service such as GitHub, GitLab, or GitTea. Point 1 is also fairly straightforward because many hosting services will be able to run Ubuntu or some other flavour of Linux[1]. The challenge is: how do you do this in a way that is reliable, up-to-date, and *fast*? When you consider the fact that some lessons will be using R Markdown to render content with an arbitrary set of packages and that those packages are not known until build time, this challenge becomes even more difficult.

Whatever workflow is building the workbench needs to do these things, which are covered in the sections below:

1. provision R
2. provision pandoc
3. install and cache a Workbench installation
4. install and cache the required packages for building markdown lessons

12.1 Provisioning R

We use the r-lib/actions/setup-r@v2 action to set up R's environmental variables. The way we use it, it does not actually install R because R comes installed by default on GitHub's Ubuntu runners [2], saving about a minute's worth of installation time.

Another alternative is to use containers from the Rocker project or the base image of the R-universe if you want to work with Docker containers.

[1] Perhaps not RedHat or CentOS, which are notoriously strict about updating their C libraries.
[2] see https://github.com/carpentries/sandpaper/pull/279 for information about how we found that out.

12.2 Provisioning pandoc

Again, we use r-lib for this task, with the r-lib/actions/setup-pandoc@v2 action. Note: we use the default installation of pandoc, which is 2.19.2 as of this writing, and is expected to match the version of pandoc used in RStudio.

Alternative ways to provision pandoc can be found in pandoc's installation guide.

12.3 Caching

In order to ensure that the process moves as rapidly as possible, we need to be able to cache the packages used in our workflow. This is a bit of a complex topic, but GitHub has written up a guide for caching dependencies. Effectively, our strategy for the cache is to ensure we can restore the packages that we have previously downloaded and (in the case of provisioning the workbench) installing only the updates we need. To do this, we need to define the folder to restore from and two keys:

0. Folder: path to the R package library used, which for a normal R installation is the `${R_LIBS_USER}` environment variable. For a {renv} cache, it will be in the `${RENV_PATHS_ROOT}` environment variable.
1. `key` a very specific key to restore a valid cache that can be used immediately
2. `restore-keys` a less-specific key to restore an invalid cache that might be able to be updated

12.3.1 Restoring an outdated cache with `restore-keys`

In the case of R packages, you need to be concerned about two things to restore a cache, valid or not:

1. The OS version
2. The R version

Remember that R packages in the libary are built for the specific operating system and thus, if the operating system changes, the cache cannot be recovered. Similarly, if the R version has changed, a package *may* work, but there is no guarantee that it *will* work and thus the cache cannot be recovered.

Thus, when we define the prefix for the `restore-keys` and the `key`, we define it like this:

```
restore-keys: ${{ os-version }}-${{ r-version }}-
```

The important part of having the `restore-keys` is that, when it is hit, the cache will be updated when the workflow completes, thus recording the updates that were found.

> **i** `cache-version` key
>
> You might see reference to a `$CACHE_VERSION` key in some of the workflows. This is an outdated feature that will be removed in the future. There are times when a cache becomes borked and it needs to be reset quickly. Sometime in 2022, GitHub added the ability for maintainers to delete a cache with a button click in the "actions" tab, so this became trivial.
> Before then, we had to rely on a trick that had maintainers register a `CACHE_VERSION` secret, which was recommended to set it to the date. This was tacked on to the end of the restore key so that if a cache needed to be reset, a maintainer could update the `CACHE_VERSION` secret and the cache would be invalidated completely.

12.3.2 Restoring a valid cache with `key`

A valid cache can be restored and used immediately without needing to reinstall any resources. To determine the validity of a particular cache, it's important to understand where your updates are coming from.

12.3.2.1 `key` For The Workbench

In terms of The Workbench, you want to make sure you pull in any updates from the R-universe and CRAN, because they will have any bugfixes that we did not consider. In practise, the way we do this is by saving a file from the output of `remotes::package_deps()`, which will get the recursive dependencies for The Workbench, check their package versions against the versions of the installed package and report which packages have updates available.

However, you might be noticing something that is amiss: if you are checking for outdated packages before any packages are installed, then how does this help with invalidating the cache if you don't know the package versions you have installed before you restore from the cache? The trick is that whenever you query the packages in this way, you are always comparing against a (near) empty R library, so the result will be the same across runs *if and only if* none of the packages have updated in the upstream repositories.

> **i** Example: cowsay
>
> For example, this is the output of checking for dependencies of the {cowsay} package:

```
> deps <- remotes::package_deps("cowsay")
> deps
Needs update ----------------------------
 package  installed available is_cran remote
 rmsfact  NA        0.0.3     TRUE    CRAN
 fortunes NA        1.5-4     TRUE    CRAN
 crayon   NA        1.5.2     TRUE    CRAN
 cowsay   NA        0.8.2     TRUE    CRAN
```

When the data frame is saved in the workspace and the file is hashed, it will produce the same hash across builds because the `installed` the only column that will vary will be the `available` column (unless "cowsay" changes dependencies).

Idea for the future

It might be possible for us to centralize this caching so that we run `remotes::package_deps("sandpaper")` *outside* of the workflow in https://files.carpentries.org so that we can save a few seconds of time installing remotes and querying dependencies if the restore is successful.

12.3.2.2 `key` for the {renv} package cache

The goal of the {renv} package cache is only to reliably restore it for reproducibility (say that five times fast), thus, the only key that we need to validate the cache is the `renv/profiles/lesson-requirements/renv.lock` file itself. If the hash for that is identical across runs, then the hash is valid and we can restore the cache as usual.

12.4 Provisioning The Workbench

The Workbench Packages are provisioned with the carpentries/actions/setup-sandpaper@main composite action.

This workflow does the following:

1. Sets up environment variables and options to allow us to fetch from the R-universe and RSPM
2. Fetches the system dependencies json record from the R-universe and installs them [3].
3. Restores the Cache and Installs any missing/outdated dependencies
4. Installs any custom versions of sandpaper/varnish

[3] See the ssl error of July 2023 for one consequence of this

12.5 Provisioning The Package Cache

The {renv} package cache is provisioned with the carpentries/actions/setup-lesson-deps@main composite action. This action relies on the {vise} package to determine and install the system dependencies for the packages in the cache (e.g. it's the reason why the spatial packages can be installed for the spatial lessons).

This has the following steps and is run only if the `renv/` folder exists.

1. Sets up environment variables and options to allow us to fetch from the R-universe and RSPM
2. Determine and install system dependencies from the lockfile[4]
3. provision the packages with `sandpaper::manage_deps()`

12.6 A bit of History

In theory, these can all be taken care of with a Docker container and, indeed, we have written a `Dockerfile` to do just this, piggybacking off of the R-universe base image. You might be wondering: why don't we use a Docker image to build the lessons? Why do we use the runners for GitHub Actions? When we initially built The Workbench, building R packages on Ubuntu *always* required compilation, so we used macOS runners so that we could get compiled packages most of the time. The key point here is *most* of the time.

The release cycle of R packages on CRAN will release the source of the package first and build the binaries for macOS and Windows in the few days following. Importantly, these binaries would have the C libraries bundled with the packages that required them, so the installation would *just work*. During these few days, users will be prompted with a message asking them if they would like to install the binary version or compile the latest source. However, on GitHub runners, the machine always defaulted to the latest version, so sometimes, just after a package updated, we would get issues where a package (e.g. {stringi}) would fail to compile because the proper C library was not installed. This was especially problematic for a situation where we needed to provision an arbitrary set of packages for R-based lessons.

In November 2021, we officially switched our runners over to use Ubuntu with carpentries/actions#31 and carpentries/sandpaper#211. These allowed us to use the binary packages from the Posit Package Manger (previously RStudio Package Manager) to provision our builds *and* parse the necessary system dependencies.

This code initially lived inside of the github workflow, but the code got complex enough that it was worthwhile to port it to a specific package, which eventually became {vise}. This package

[4]At the moment, this is sort of complex because {remotes} package has not been updated on CRAN since 2021 and does not know about ubuntu 22.04, which is the version of runners that GitHub uses. We do this because installing from CRAN is faster.

was intended as a way to split off the {renv} system from {sandpaper} into its own package (akin to something like {capsule}, but I never had the bandwitdh to properly separate the {renv} components from the {sandpaper} components (though that may be easier now that the {flow} package exists for analysis of code pathways).

13 Summary

In summary, this book has no content whatsoever.

References

Marwick, Ben. 2016. "Computational Reproducibility in Archaeological Research: Basic Principles and a Case Study of Their Implementation." *Journal of Archaeological Method and Theory* 24 (2): 424–50. https://doi.org/10.1007/s10816-015-9272-9.

Vaniea, Kami, and Yasmeen Rashidi. 2016. "Tales of Software Updates." *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, May. https://doi.org/10.1145/2858036.2858303.

Wickham, Hadley, and Jennifer Bryan. 2023. "R Packages (2e)." https://r-pkgs.org/.

Wilson, Greg, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. 2017. "Good Enough Practices in Scientific Computing." Edited by Francis Ouellette. *PLOS Computational Biology* 13 (6): e1005510. https://doi.org/10.1371/journal.pcbi.1005510.

A Standard Operating Procedures

For any issue or feature addition there are a few general strategies to follow while working with any of the packages in The Workbench. These strategies will make it easier to ensure the quality of work with the bonus of being able to more easily collaborate with other maintainers and contributors to this project.

A.1 Get a Reproducible Example

The most powerful tool in the developer's toolbox for demonstrating an an issue or a new feature is the reproducible example. It is no surprise that Jenny Bryan (of "Everything I Know Is From Jenny Bryan" fame) is the maintainer of the {reprex} package and has an incredibly useful webinar on how to create reproducible examples with {reprex}.

> ⚠ System-Dependent Issues
>
> There are times when you cannot create a reproducible example that will produce the results you want to see. This is often the case when there is a system-dependent issue. In these cases, you should share the reprex with someone who has the system in question and have them run it.

A.2 Git/GitHub Etiquitte

In The Workbench practice, commits are never pushed to the main branch of the packages. **All issues and features are fixed/made by pull requests** because they give a better accounting of what was changed and why. Using this, it is good to strive for these practises[1]:

0. If an issue for the feature/bug does not exist, create it with documentation about where the error exists (or where the feature should be). Use the "copy permalink" when you click on a line number in the source view of GitHubto create a preview of the code as it existed before the feature/fix.

[1] I say *strive* for these practises because I do not always follow them myself, but they are there for very good reasons. For example: testing before you write code helps you design your code to do the thing you want it to do AND it helps avoid confirmation bias in the tests.

1. **create a new branch** from `main` that *describes the purpose* and *includes the issue number*. For example: `avoid-gert-config-449` removed a bit of code that was using the {gert} package to check for a git user, which was unnecessary.
2. push the branch to GitHub[2] and **create a new *draft* pull request**
3. **Add a test** that will reproduce the bug (if possible) before any code
4. **COMMIT OFTEN**. There are MANY times when I have been working on a feature deeply and have forgotten to commit, only to lose track of what changes I made and why[3]. Remember that git is your lab notebook.
5. **Iterate** writing code and committing until the tests for that particular function passes.
6. Write a new item in NEWS and bump the version in DESCRIPTION.
7. Push to GitHub and mark as ready for review; tag `@carpentries/maintainers-workbench`

A.2.1 Pull Request Reviews

While The Workbench was being rapidly developed by Zhian Kamvar, all pull requests were merged by Zhian, even if he created them. After the launch of The Workbench across all official lessons, The Workbench Maintainer Group will now review all pull requests. In order to ensure pull request reviews are equitably given, please read Alex Hill's blog post entitled "The Art of Giving and Receiving Code Reviews".

A.3 Addressing Issues

One big challenge with issues is that you cannot know *where* the issue is coming from. If you do nothing else, please **watch this Keynote talk from Jenny Bryan**: Object of type 'closure' is not subsettable. This will give you the right mindset for broadly approaching issues in R package development, which can be summarised as:

1. Turn it off and on again (Restart R in a clean session)
2. Make a reprex
3. Dig into the error (via `traceback()` and/or a debugger)
4. Future-proof your fixes and make things within reach

With these tools, if you cannot reproduce the error, you can guide the user to give you the information you need to reproduce the error and then you can follow the git guidelines to iterate on the fix.

[2] In RStudio, when you use the interface to create a new branch, it will generally also push the branch to GitHub for you. If it does not do this, you will need to push the branch with `git push -u`. I do this often enough that I've created a git alias (which can go in your `~/.gitconfig` file):
```
pushb = !"git push --set-upstream $(git remote | head -n 1) $(git symbolic-ref HEAD
--short)"
```

[3] You do not have to push after each commit. Sometimes it is strategic to collect a little stack of local commits in case you were experimenting and need to revert back to one that gave you a working state.

> **Think of the big picture**
>
> As you are fixing an issue, consider the larger picture of the issue. Sometimes you will run into an issue that will technically fix it, but still leave the possibility for future issues to open up. **The most important thing is to document what future work needs to be done to ensure a robust solution while it is still fresh in your mind.** This can be in an issue or in the comments of the code itself.
> Sometimes an issue is urgent enough that a quick fix is what is needed, and in those cases, it's best to open up a followup issue that addresses what would be needed for a more robust fix. Sometimes the solution is splitting a large function into smaller functions, but other times it may be a refactor of the internal data structure.

A.4 Creating New Features

If you want to create a new feature, the most important thing to do is to clearly define the audience, scope, dependencies, inputs and the outputs of the feature that you want to create. It's best to strive for new features that bolster existing features (e.g. `sandpaper::serve()` provides an continously updating version of `sandpaper::build_lesson()`) or those that are modular and optional (e.g. the `fail_on_error` config option for lessons using R Markdown force an error in code blocks to stop the lesson from rendering unless those code blocks have the `error = TRUE` option).

One of the most important things to consider when adding new features is the maintenance and deployment workflow. Maintainers and contributors should *not* need to worry about function arguments, GitHub Actions, or new Markdown syntax in order to implement a new feature that will be deployed automatically to all of the lessons. **A new feature should not break their workflow or the deployment workflow.**

> **Simple Example**
>
> As an example, when we were discussion folder organistaion, The original config template had a "schedule" section instead of "episodes". When I replaced "schedule" with "episodes", I added functionality to allow for old-style lessons to move forward so that any existing test lessons did not break. To this day, you can still create a lesson using schedule instead of episodes as the keyword.

B Examples and Flight Rules

aka: what to do when something's not working

This part of the developers manual is a living document and serves as a window into the mind of a Workbench Maintainer as they address issues and new feature requests.

The following sections of this appendix will not distinguish between bugs and features, because often the process for fixing bugs is not sufficiently different than that of adding a new feature (though there are exceptions, which I will highlight below). Rather, a more logical division is to distinguish between if the issue is within The Workbench Packages, In an Upstream dependency, with GitHub Actions, and finally, issues that are more structural in nature and require more extensive user experience (UX) testing.

B.1 Within The Workbench R Packages

Bugs or features in this category are entirely within our control and are theoretically the easiest/most quick implementations. Items in this category can be split into either single package items which can be fixed with a single pull request or cross-package items, which require coordination of pull requests to achieve.

B.1.1 Single Package

In this section, we outline issues that are addressed within a single package. Note that this does not indicate that these issues are straightforward to address.

B.1.1.1 Single-Function Issues

If you are here, you have determined that the bug or feature that you are working on will affect a single function or data pathway. These issues are often the most straightforward to address. Below, I've documented narratives for these issues.

B.1.1.1.1 * Markdown file for 404 page created with read-only permissions

B.1.1.1.1.1 * s479 Situation

- Issue: carpentries/sandpaper#479
- Resolver: zkamvar
- Type: Local
- OS: Rocky Linux 8.8
- Package: sandpaper

The user is attempting to build a lesson, but they are unable to because an error appears during the "Create 404 page" step:

```
  Creating 404 page
Error in file(file, ifelse(append, "a", "w")) :
  cannot open the connection
In addition: Warning messages:
[snip]
7: In file(file, ifelse(append, "a", "w")) :
  cannot open file '/tmp/RtmpmaiZ5B/file73cdc48b90540.md': Permission denied
```

B.1.1.1.1.2 * s479 Diagnosis

This permissions issue was a problem with copying a read-only file without adjusting the subsequent permissions. It stemmed from `build_404()` calling `render_html()` with the presence of a `links.md` document at the top of the lesson. Packages in the user's R library are installed by the systems administrator and the user does not have permissions to write or append any file in that folder. When `build_404()` runs, it passes a template markdown file to `render_html()`. If there is an option called `sandpaper.links` set to a filepath, then this function will copy the input file to a temporary file and append the links to that temporary file before rendering it to HTML. Because the input file was read-only, copying the file to the temporary directory retained its permissions and we were unable to append the links.

B.1.1.1.1.3 * s479 Solution

Pull Request: carpentries/sandpaper#482

The solution was to add a single line adding user write permissions before the links file was appended: `fs::file_chmod(tmpin, "u+w")`.

B.1.1.1.1.4 * s479 Alternative Solution

This additionally could have been avoided by temporarily unsetting the `sandpaper.links` option for the `build_404()` function before it calls `render_html()`. This would prevent it entering the loop where it appends the links, making the process *slightly* faster. The only downside is that we would need to do that for all of the other templated pages (though I believe that might be the only one).

B.1.1.1.1.5 * s479 Narrative

- Issue: carpentries/sandpaper#479
- First impression: this is running on a version of Linux that Zhian does not know anything about. My thoughts are that it will be a difficult fix.
- I suspect the bug might be due to the {fs} package and ask for the user if they could test out the following code snippet:

```
tmp <- fs::file_temp(ext = ".md")
cat("test\n", file = tmp, append = TRUE)
readLines(tmp)
```

- I looked again did not recognise the code snippet where the issue seemed to be (`file(file, ifelse(append, 'a', 'w'))`) - so this seems to be a problem with a non-Workbench package.
- initially went looking for this code in the `fs` package, but GitHub search showed that this code doesn't appear there either.
- next guess is the `cat` function, which is used to add link references to the end of files. I searched the R code base and found the snippet in the `cat()`function
- that seemed to be where that code snippet was coming from, but the problem really originated a few lines above the call to cat: when the template for the 404 page (which is saved where the package was installed) is copied—`file_copy()` copies a file and *all of its permissions*—so the copy is read-only for non-admin users.
- I opened a PR to test for the error, then applied a fix to prevent it from being thrown again.
- Asked the reporter to install the patch on their system and report back on whether it worked.
- They reported back that it did—and pointed out a typo!
- After merging PR, create a new release (see process in The Release Workflow)

B.1.1.2 Multi-Function Issues

B.1.1.3 Test Failures With No User Impact

B.1.2 Aross Packages

B.2 Upstream R Packages

B.2.1 renv

B.2.1.1 Background

The {renv} package is a key player for allowing {sandpaper} to provision and maintain the R packages required to build R-based lessons. The motivation and strategy for how it works can be found in the Building Lessons With A Package Cache article in the {sandpaper} documentation.

It's worth diving into carpentries/sandpaper#21 to see the discussion and thoughts around the origin of the design for using this feature. It was implemented during a three week period between 2021-08-24 and 2021-09-16, as detailed in the pull request carpentries/sandpaper#158.

B.2.1.2 In practices

We have to consider {renv} in practice from the standpoint of both local computers and on GitHub, which can behave very differently and require different tools to address their tasks.

There are three tools and packages that use {renv}:

1. {sandpaper} is designed to provide a way to manage dependencies in a lesson
2. {vise} was originally intended as a project to split out {sandpaper} code that used {renv} to simplify the testing. At the moment, it provides utilities for automatically provisioning C libraries on Ubunutu Linux and running the equivalent of `sandpaper::update_cache()` in a GitHub Actions context.
3. carpentries/actions these contain R code within YAML files that will call {renv} and {vise}.

B.2.1.3 Debugging Tips

B.2.1.3.1 Setting up a reproducible environment

Browse the {renv} issues opened by @zkamvar. In nearly all of these issues, I provide a reproducible example. They generally follow the pattern of:

1. create a temporary file
2. make it a directory and move there
3. add any files that are needed *before* the renv project is set up (if specific to problem)
4. set up a {renv} project with `renv::init()`
5. demonstrate problem

```
tmp <- tempfile()
dir.create(tmp)
setwd(tmp)
writeLines("library(R6)", "test.R")
renv::init(profile = "test")
# demonstrate problem here
```

B.2.1.3.2 Choosing a minimal example

Presenting a failing CI run with a Workbench lesson *is* reproducible, but it's not minimal. Presenting the same with a smaller lesson is still not minimal. Often times, the issue involves detecting and installing a new package. In this case, you want to choose a package that has few to no dependencies and *is not listed as a dependency for knitr*. One example that I use often is the {cowsay} package. It has a total of three dependencies and is not depended on by anything. If I need a quick package with zero dependencies, I will reach for {R6} or {aweek}. Both of these packages are under 100 Kilobytes, are pure R code, do not have any dependencies, and do not require compillation.

B.2.1.3.3 When you just can't reproduce it locally

There have been times when {renv} seems to fail *only* on GitHub Actions. This was the case for {renv} version 0.17.0, as I reported in rstudio/renv#1161.

The important thing in these situations is to stay calm and try to narrow down as much as possible the exact conditions that will create the problem. Once you have those, you can open an issue on the {renv} issue tracker. If you are at this point, it's important to not expect this to be resolved quickly, because it is likely out of your control. The best you can do is to try the debugging techniques that Kevin provides and report back on the issue thread.

Once the issue is resolved: **Thank Kevin for his help.** This is a very important point. Maintainers often only hear from their users if somehting is going wrong, so it's important to let them know that they are appreciated.

B.3 GitHub Actions

There is a category of issues that fail explicitly with GitHub actions only. As with the other issues, it is important not to panic when you encounter these, but instead, take stock of what happens when these happen. In general, here are the steps you should take to diagnosing problems:

0. Prepare your inbox to filter messages from GitHub. A good way is to check that the subject line matches "Run failed 01 Build and Deploy" or "Run failed 03 Update Cache".
1. Find and read the error message from the log. You can find this by looking for the red X and expanding that section. It will normally scroll down to the error.
2. Read the *context* of the error message. Most errors happen because of problems upstream, so it's important to see what was happening when that error happened
3. Restart the action. GitHub has a button near the top right of the action. If it's ephemeral (e.g. networking error) then it will run fine on the next run.
4. Test the build locally
5. Test it on the `carpentries/sandpaper-docs` repository
6. Test it on a brand new repository created via the templates (https://bit.ly/new-lesson-md or https://bit.ly/new-lesson-rmd)

Below I will attempt to outline common problem areas and their potential solutions. Note: this is not an exhaustive list, but you have to remember that in these situations, you are effectively debugging someone else's computer, so patience is absolutely required because this is more difficult than plucking a single grain of rice with metal chopsticks.

B.3.1 Networking Failures

These are often the easiest problems to solve. You will encounter an error that says somthing like "network timed out" and it fails when either checking out the repository or installing pandoc. In these situations, you should check https://status.github.com to see if this is a problem with GitHub, and then you should restart the build. Most of the time, the build will work on the second try.

B.3.2 Upstream System Dependency Issues

SSL error

B.3.2.0.1 * bioc-intro109 Situation

- Issue: https://github.com/carpentries-incubator/bioc-intro/issues/109
- Resolver: NA
- Type: Remote
- OS: Ubuntu 22.04
- Package: NA

On 2023-07-20, @lgatto, who maintains the carpentries-incubator/bioc-intro repository reported an error in their builds during the "Setup Package Cache" of the `sandpaper-main.yaml` workflow:

```
Register Repositories
Using github PAT from envvar GITHUB_PAT
Error: Error: Failed to install 'vise' from GitHub:
  SSL peer certificate or SSH remote key was not OK: [api.github.com] SSL: no alternative ce
Execution halted
Error: Process completed with exit code 1.
```

This error was causing the carpentries/vise package to not be installed and the {renv} package cache could not be provisioned for lessons. Because it involved the {renv} package cache, this was a problem that was limited to R-based lessons.

B.3.2.0.2 * bioc-intro109 Diagnosis

The cause of the problem was identified by Gabor Csardi as a bug specific to the development version of `curl` on Ubuntu: https://bugs.launchpad.net/ubuntu/+source/curl/+bug/2028170.

The key error message here was `SSL: no alternative certificate subject name matches target host name 'api.github.com'`. The `curl` program was checking the validity of the SSL certificate for `github.com`, but it was ignoring the rules for `*.github.com`, so when it found `api.github.com`, it saw that as an invalid site that should not be trusted.

Even though the default version of curl on GitHub's runners is *not* the dev version, when we provision the workbench in the "Setup Lesson Engine" step, we query the system dependencies from the R-universe:

```
$ curl https://carpentries.r-universe.dev/stats/sysdeps 2> /dev/null \
  | jq -r '.headers[0] | select(. != null)'
```

```
libcurl4-openssl-dev
libfontconfig-dev
libfreetype-dev
libfribidi-dev
libharfbuzz-dev
libicu-dev
libgit2-dev
libjpeg-turbo8-dev
libpng-dev
libtiff-dev
libxml2-dev
libxslt1-dev
libssl-dev
```

This installed the development version of curl into our system and introduced the bug during the "Setup Package Cache" step.

B.3.2.0.3 * bioc-intro109 Solution

The solution was to wait for a fix.

B.3.2.0.4 * bioc-intro109 Alternative Solution

An alternative solution that we tested was to switch the download method for R packages to "wget", but this was a non-starter because this caused the installation times for R packages *and* the workbench to rise because they would need to be compiled (posit PPM did not serve the binary packages over wget).

B.3.2.0.5 * bioc-intro109 Narrative

- I get the notification of the issue and @lgatto suspects it's because of {vise}.
- I install {vise} using `remotes::install_github("carpentries/vise")` to check that it's installable and I am successful.
- I rerun the action and get the same error. I run the action on the carpentries/sandpaper-docs repository, but it also fails similarly. This confirms that it's a broader issue.
- I thought maybe also it was a problem in the {remotes} package since it hasn't been updated on CRAN since 2021.

- I found https://github.com/r-lib/remotes/issues/762 which points to an issue in `utils::download.file()` as the source of the problems with curl.
- I notice that {vise} is still recorded as "zkamvar/vise" in the github actions files. I change it in carpentries/actions@2de7e3fef
- This does not fix the issue and I post to mastodon: https://fosstodon.org/@zkamvar/110741428815173845
- I update the installation for vise to use "wget" as a fallback (see this comparison of commits
- I create a new branch in carpentries/actions to set `options(download.file.method = "wget")` in setup-sandpaper (view the diff).
- I create https://github.com/zkamvar/2023-07-19-test-actions, which fails initially and replace all `@main` declarations in `.github/workflows/` with `@2023-07-ssl-errors` to test the fix that I created. The next run also fails and it takes > 17 minutes to set up the workbench (which normally takes ~1 minute or less). I understand that the "wget" method will not work.
- Gabor Csardi identifies the issue as coming from the dev version of curl
- I check the version of curl in the ubuntu runner image, which I found by expanding "Setup Job" and then "Runner Image" in the GitHub action run and finding the URL above, but it is not devel.
- I create https://github.com/zkamvar/test-github-actions-ssl to as a MWE of the error.
- I create a workflow that runs the command from the ubuntu bug report and find that it works.
- I update the workflow so that it demonstrates the working example and then demonstrates the failing example. It successfully fails
- I wait and check the bug has been fixed for ubuntu and I rerun the workflow. It succeeds and I re-run the bioc-intro and sandpaper-docs builds to find that they work.

B.3.3 Workflow Mis-Configuration

Sometimes the workflow files themselves are mis-configured. In these cases, the fix will involve updating the files in sandpaper's `inst/workflows/` directory. If the `pr-comment.yaml` or `update-workflows.yaml` files are not affected, then a pull request will be created automatically to all lessons within a week after you submit the patch, but if these files are affected, then you will have to manually submit the patch.

files with spaces in names cause `pr-comment.yaml` workflow to fail

B.3.3.0.1 * s399 Situation

- Issue: carpentries/sandpaper#399
- Resolver: zkamvar
- Type: Remote

- OS: Ubuntu 22.04
- Package: sandpaper

B.3.3.0.2 * s399 Diagnosis

B.3.3.0.3 * s399 Solution

B.3.3.0.4 * s399 Alternative Solution

B.3.3.0.5 * s399 Narrative

B.3.4 Permissions Changes

B.3.4.1 carpentries-bot

B.3.5 Actions

B.4 Structural Features