

Package: rhino (via r-universe)

October 23, 2024

Title A Framework for Enterprise Shiny Applications

Version 1.10.1.9000

Description A framework that supports creating and extending enterprise Shiny applications using best practices.

URL <https://appsilon.github.io/rhino/>,
<https://github.com/Appsilon/rhino>

BugReports <https://github.com/Appsilon/rhino/issues>

License LGPL-3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Depends R (>= 2.10)

Imports box (>= 1.1.3), box.linters (>= 0.10.5), box.lsp, cli, config, fs, glue, lintr (>= 3.0.0), logger, purrr, renv, rstudioapi, sass, shiny, styler, testthat (>= 3.0.0), utils, withr, yaml

Suggests covr, knitr, mockery, rcmdcheck, rex, rlang, rmarkdown, shiny.react, spelling

LazyData true

Config/testthat/edition 3

Config/testthat/parallel true

Language en-US

Repository <https://appsilon.r-universe.dev>

RemoteUrl <https://github.com/appsilon/rhino>

RemoteRef HEAD

RemoteSha 7c6106f8857aa5ab647f22075b006f6191a34bb0

Contents

app	2
build_js	3
build_sass	4
dependencies	5
diagnostics	6
format_js	7
format_r	7
format_sass	8
init	9
lint_js	10
lint_r	11
lint_sass	11
log	12
react_component	13
rhinos	14
test_e2e	14
test_r	15
Index	16

app

Rhino application

Description

The entrypoint for a Rhino application. Your `app.R` should contain nothing but a call to `rhino::app()`.

Usage

```
app()
```

Details

This function is a wrapper around `shiny::shinyApp()`. It reads `rhino.yml` and performs some configuration steps (logger, static files, box modules). You can run a Rhino application in typical fashion using `shiny::runApp()`.

Rhino will load the `app/main.R` file as a box module (`box::use(app/main)`). It should export two functions which take a single `id` argument - the `ui` and `server` of your top-level Shiny module.

Value

An object representing the app (can be passed to `shiny::runApp()`).

Legacy entrypoint

It is possible to specify a different way to load your application using the `legacy_entrypoint` option in `rhino.yml`:

1. `app_dir`: Rhino will run the app using `shiny::shinyAppDir("app")`.
2. `source`: Rhino will `source("app/main.R")`. This file should define the top-level `ui` and `server` objects to be passed to `shinyApp()`.
3. `box_top_level`: Rhino will load `app/main.R` as a box module (as it does by default), but the exported `ui` and `server` objects will be considered as top-level.

The `legacy_entrypoint` setting is useful when migrating an existing Shiny application to Rhino. It is recommended to transform your application step by step:

1. With `app_dir` you should be able to run your application right away (just put the files in the app directory).
2. With `source` setting your application structure must be brought closer to Rhino, but you can still use `library()` and `source()` functions.
3. With `box_top_level` you can be confident that the whole app is properly modularized, as box modules can only load other box modules (`library()` and `source()` won't work).
4. The last step is to remove the `legacy_entrypoint` setting completely. Compared to `box_top_level` you'll need to make your top-level `ui` and `server` into a **Shiny module** (functions taking a single `id` argument).

Examples

```
## Not run:
# Your `app.R` should contain nothing but this single call:
rhino::app()

## End(Not run)
```

build_js

Build JavaScript

Description

Builds the `app/js/index.js` file into `app/static/js/app.min.js`. The code is transformed and bundled using **Babel** and **webpack**, so the latest JavaScript features can be used (including ECMAScript 2015 aka ES6 and newer standards). Requires Node.js to be available on the system.

Usage

```
build_js(watch = FALSE)
```

Arguments

`watch` Keep the process running and rebuilding JS whenever source files change.

Details

Functions/objects defined in the global scope do not automatically become window properties, so the following JS code:

```
function sayHello() { alert('Hello!'); }
```

won't work as expected if used in R like this:

```
tags$button("Hello!", onclick = 'sayHello()');
```

Instead you should explicitly export functions:

```
export function sayHello() { alert('Hello!'); }
```

and access them via the global App object:

```
tags$button("Hello!", onclick = "App.sayHello()")
```

Value

None. This function is called for side effects.

Examples

```
if (interactive()) {  
  # Build the `app/js/index.js` file into `app/static/js/app.min.js`.  
  build_js()  
}
```

build_sass

Build Sass

Description

Builds the app/styles/main.scss file into app/static/css/app.min.css.

Usage

```
build_sass(watch = FALSE)
```

Arguments

watch	Keep the process running and rebuilding Sass whenever source files change. Only supported for sass: node configuration in rhino.yml.
-------	--

Details

The build method can be configured using the `sass` option in `rhino.yml`:

1. `node`: Use **Dart Sass** (requires Node.js to be available on the system).
2. `r`: Use the `{sass}` R package.

It is recommended to use Dart Sass which is the primary, actively developed implementation of Sass. On systems without Node.js you can use the `{sass}` R package as a fallback. It is not advised however, as it uses the deprecated **LibSass** implementation.

Value

None. This function is called for side effects.

Examples

```
if (interactive()) {  
  # Build the `app/styles/main.scss` file into `app/static/css/app.min.css`.  
  build_sass()  
}
```

dependencies

Manage dependencies

Description

Install, remove or update the R package dependencies of your Rhino project.

Usage

```
pkg_install(packages)
```

```
pkg_remove(packages)
```

Arguments

`packages` Character vector of package names.

Details

Use `pkg_install()` to install or update a package to the latest version. Use `pkg_remove()` to remove a package.

These functions will install or remove packages from the local `{renv}` library, and update the `dependencies.R` and `renv.lock` files accordingly, all in one step. The underlying `{renv}` functions can still be called directly for advanced use cases. See the [Explanation: Renv configuration](#) to learn about the details of the setup used by Rhino.

Value

None. This functions are called for side effects.

Examples

```
## Not run:
# Install dplyr
rhino::pkg_install("dplyr")

# Update shiny to the latest version
rhino::pkg_install("shiny")

# Install a specific version of shiny
rhino::pkg_install("shiny@1.6.0")

# Install shiny.i18n package from GitHub
rhino::pkg_install("Apsilon/shiny.i18n")

# Install Biobase package from Bioconductor
rhino::pkg_install("bioc::Biobase")

# Install shiny from local source
rhino::pkg_install("~/path/to/shiny")

# Remove dplyr
rhino::pkg_remove("dplyr")

## End(Not run)
```

diagnostics

Print diagnostics

Description

Prints information which can be useful for diagnosing issues with Rhino.

Usage

```
diagnostics()
```

Value

None. This function is called for side effects.

Examples

```
if (interactive()) {
  # Print diagnostic information.
  diagnostics()
}
```

format_js	<i>Format JavaScript</i>
-----------	--------------------------

Description

Runs [prettier](#) on JavaScript files in app/js directory. Requires Node.js installed.

Usage

```
format_js(fix = TRUE)
```

Arguments

fix	If TRUE, fixes formatting. If FALSE, reports formatting errors without fixing them.
-----	---

Details

You can prevent prettier from formatting a given chunk of your code by adding a special comment:

```
// prettier-ignore
```

Read more about [ignoring code](#).

Value

None. This function is called for side effects.

format_r	<i>Format R</i>
----------	-----------------

Description

Uses the `{styler}` and `{box.linters}` packages to automatically format R sources. As with `styler`, carefully examine the results after running this function.

Usage

```
format_r(paths, exclude_files = NULL)
```

Arguments

paths	Character vector of files and directories to format.
exclude_files	Character vector with regular expressions of files that should be excluded from styling.

Details

The code is formatted according to the `styler::tidyverse_style` guide with one adjustment: spacing around math operators is not modified to avoid conflicts with `box::use()` statements.

If available, `box::use()` calls are reformatted by styling functions provided by `{box.linters}`. These include:

- Separating `box::use()` calls for packages and local modules
- Alphabetically sorting packages, modules, and functions.
- Adding trailing commas

`box.linters::style_*` functions require the `tree-sitter` and `tree-sitter-r` packages. These, in turn, require `R >= 4.3.0`. `format_r()` will continue to operate without these but will not perform `box::use()` call styling.

For more information on `box::use()` call styling please refer to the `{box.linters}` styling functions [documentation](#).

Value

None. This function is called for side effects.

Examples

```
if (interactive()) {
  # Format a single file.
  format_r("app/main.R")

  # Format all files in a directory.
  format_r("app/view")
}
```

format_sass

Format Sass

Description

Runs [prettier](#) on Sass (.scss) files in `app/styles` directory. Requires Node.js installed.

Usage

```
format_sass(fix = TRUE)
```

Arguments

`fix` If TRUE, fixes formatting. If FALSE, reports formatting errors without fixing them.

Details

You can prevent prettier from formatting a given chunk of your code by adding a special comment:

```
// prettier-ignore
```

Read more about [ignoring code](#).

Value

None. This function is called for side effects.

init	<i>Create Rhino application</i>
------	---------------------------------

Description

Generates the file structure of a Rhino application. Can be used to start a fresh project or to migrate an existing Shiny application created without Rhino.

Usage

```
init(
  dir = ".",
  github_actions_ci = TRUE,
  rhino_version = "rhino",
  force = FALSE
)
```

Arguments

dir	Name of the directory to create application in.
github_actions_ci	Should the GitHub Actions CI be added?
rhino_version	When using an existing <code>renv.lock</code> file, Rhino will install itself using <code>renv::install(rhino_version)</code> . You can provide this argument to use a specific version / source, e.g. <code>"Appsilon/rhino@v0.4.0"</code> .
force	Boolean; force initialization? By default, Rhino will refuse to initialize a project in the home directory.

Details

The recommended steps for migrating an existing Shiny application to Rhino:

1. Put all app files in the app directory, so that it can be run with `shiny::shinyAppDir("app")` (assuming all dependencies are installed).
2. If you have a list of dependencies in form of `library()` calls, put them in the `dependencies.R` file. If this file does not exist, Rhino will generate it based on `renv::dependencies("app")`.
3. If your project uses `{renv}`, put `renv.lock` and `renv` directory in the project root. Rhino will try to only add the necessary dependencies to your lockfile.
4. Run `rhino::init()` in the project root.

Value

None. This function is called for side effects.

lint_js	<i>Lint JavaScript</i>
---------	------------------------

Description

Runs **ESLint** on the JavaScript sources in the `app/js` directory. Requires Node.js to be available on the system.

Usage

```
lint_js(fix = FALSE)
```

Arguments

`fix` Automatically fix problems.

Details

If your JS code uses global objects defined by other JS libraries or R packages, you'll need to let the linter know or it will complain about undefined objects. For example, the `{leaflet}` package defines a global object `L`. To access it without raising linter errors, add `/* global L */` comment in your JS code.

You don't need to define `Shiny` and `$` as these global variables are defined by default.

If you find a particular ESLint error inapplicable to your code, you can disable a specific rule for the next line of code with a comment like:

```
// eslint-disable-next-line no-restricted-syntax
```

See the [ESLint documentation](#) for full details.

Value

None. This function is called for side effects.

Examples

```
if (interactive()) {  
  # Lint the JavaScript sources in the `app/js` directory.  
  lint_js()  
}
```

lint_r	<i>Lint R</i>
--------	---------------

Description

Uses the {lintr} package to check all R sources in the app and tests/testthat directories for style errors.

Usage

```
lint_r(paths = NULL)
```

Arguments

paths Character vector of directories and files to lint. When NULL (the default), check app and tests/testthat directories.

Details

The linter rules can be **adjusted** in the .lintr file.

You can set the maximum number of accepted style errors with the legacy_max_lint_r_errors option in rhino.yml. This can be useful when inheriting legacy code with multiple styling issues.

The `box.linters::namespaced_function_calls()` linter requires the {treesitter} and {treesitter.r} packages. These require R >= 4.3.0. lint_r() will continue to run and skip namespaced_function_calls() if its dependencies are not available.

Value

None. This function is called for side effects.

lint_sass	<i>Lint Sass</i>
-----------	------------------

Description

Runs **Stylelint** on the Sass sources in the app/styles directory. Requires Node.js to be available on the system.

Usage

```
lint_sass(fix = FALSE)
```

Arguments

fix Automatically fix problems.

Value

None. This function is called for side effects.

Examples

```
if (interactive()) {  
  # Lint the Sass sources in the `app/styles` directory.  
  lint_sass()  
}
```

log

Logging functions

Description

Convenient way to log messages at a desired severity level.

Usage

log

Format

An object of class `list` of length 7.

Details

The `log` object is a list of logging functions, in order of decreasing severity:

1. `fatal`
2. `error`
3. `warn`
4. `success`
5. `info`
6. `debug`
7. `trace`

Rhino configures logging based on settings read from the `config.yml` file in the root of your project:

1. `rhino_log_level`: The minimum severity of messages to be logged.
2. `rhino_log_file`: The file to save logs to. If NA, standard error stream will be used.

The default `config.yml` file uses `!expr Sys.getenv()` so that log level and file can also be configured by setting the `RHINO_LOG_LEVEL` and `RHINO_LOG_FILE` environment variables.

The functions re-exported by the `log` object are aliases for `{logger}` functions. You can also import the package and use it directly to utilize its full capabilities.

Examples

```
## Not run:
box::use(rhino[log])

# Messages can be formatted using glue syntax.
name <- "Rhino"
log$warn("Hello {name}!")
log$info("{1:3} + {1:3} = {2 * (1:3)}")

## End(Not run)
```

react_component	<i>React components</i>
-----------------	-------------------------

Description

Declare the React components defined in your app.

Usage

```
react_component(name)
```

Arguments

name	The name of the component.
------	----------------------------

Details

There are three steps to add a React component to your Rhino application:

1. Define the component using JSX and register it with `Rhino.registerReactComponents()`.
2. Declare the component in R with `rhino::react_component()`.
3. Use the component in your application.

Please refer to the [Tutorial: Use React in Rhino](#) to learn about the details.

Value

A function representing the component.

Examples

```
# Declare the component.
TextBox <- react_component("TextBox")

# Use the component.
ui <- TextBox("Hello!", font_size = 20)
```

rhinos

Population of rhinos

Description

A dataset containing population of 5 species of rhinos.

Usage

```
rhinos
```

Format

A data frame with 58 rows and 3 variables:

Year year

Population rhinos population

Species rhinos species

Source

<https://ourworldindata.org/>

test_e2e

Run Cypress end-to-end tests

Description

Uses **Cypress** to run end-to-end tests defined in the `tests/cypress` directory. Requires Node.js to be available on the system.

Usage

```
test_e2e(interactive = FALSE)
```

Arguments

`interactive` Should Cypress be run in the interactive mode?

Details

Check out: [Tutorial: Write end-to-end tests with Cypress](#) to learn how to write end-to-end tests for your Rhino app.

If you want to write end-to-end tests with `{shinytest2}`, see our [How-to: Use shinytest2](#) guide.

Value

None. This function is called for side effects.

Examples

```
if (interactive()) {  
  # Run the end-to-end tests in the `tests/cypress` directory.  
  test_e2e()  
}
```

test_r

Run R unit tests

Description

Uses the {testthat} package to run all unit tests in tests/testthat directory.

Usage

```
test_r()
```

Value

None. This function is called for side effects.

Examples

```
if (interactive()) {  
  # Run all unit tests in the `tests/testthat` directory.  
  test_r()  
}
```

Index

* datasets

log, [12](#)

rhinos, [14](#)

app, [2](#)

box.linters::namespaced_function_calls(),
[11](#)

build_js, [3](#)

build_sass, [4](#)

dependencies, [5](#)

diagnostics, [6](#)

format_js, [7](#)

format_r, [7](#)

format_sass, [8](#)

init, [9](#)

lint_js, [10](#)

lint_r, [11](#)

lint_sass, [11](#)

log, [12](#)

pkg_install(dependencies), [5](#)

pkg_remove(dependencies), [5](#)

react_component, [13](#)

rhinos, [14](#)

test_e2e, [14](#)

test_r, [15](#)