Package 'rhino'

September 7, 2022

Title A Framework for Enterprise Shiny Applications

Version 1.1.1

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

RoxygenNote 7.2.1

Depends R (>= 2.10)

Imports box, cli, config, fs, glue, lintr (>= 2.0.0), logger, purrr, renv, rstudioapi, sass, shiny, styler, testthat (>= 3.0.0), utils, withr, yaml

Suggests knitr, mockery, rcmdcheck, rmarkdown

LazyData true

Config/testthat/edition 3

Config/testthat/parallel true

NeedsCompilation no

Author Kamil Zyla [aut, cre],
Jakub Nowicki [aut],
Tymoteusz Makowski [aut],
Marek Rogala [aut],
Appsilon Sp. z o.o. [cph]

Maintainer Kamil Zyla <kamil@appsilon.com>

Repository CRAN

Date/Publication 2022-09-07 13:40:02 UTC

R **topics documented:**

---

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 and the `yarn` command 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 and the yarn command 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 yarn 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()
}
```

---

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_r   *Format R*

Description

Uses the {styler} package to automatically format R sources.

Usage

```
format_r(paths)
```

Arguments

paths   Character vector of files and directories to format.

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.

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")
}
```

---

init   *Create Rhino application*

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")
```

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 renv.lock file, Rhino will install itself using renv::install(rhino_version).
     You can provide this argument to use a specific version / source, e.g."Appsilon/rhino@v0.4.0".

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   *Lint JavaScript*

Description

Runs [ESLint](#) on the JavaScript sources in the app/js directory. Requires Node.js and the yarn
command 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   *Lint R*

Description

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

Usage

```
lint_r()
```

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.

Value

None. This function is called for side effects.

---

lint_sass   *Lint Sass*

Description

Runs Stylelint on the Sass sources in the `app/styles` directory. Requires Node.js and the `yarn` command 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)
```

---

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 and the `yarn` command to be available on the system.

Usage

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

Arguments

interactive   Should Cypress be run in the interactive mode?

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 {testhat} 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