Package 'dash'

June 4, 2020

Title An Interface to the 'dash' Ecosystem for Authoring Reactive Web Applications

Version 0.5.0

Description A framework for building analytical web applications, 'dash' offers a pleasant and productive development experience. No JavaScript required.

Depends R (>= 3.0.2)

Imports dashHtmlComponents (== 1.0.3), dashCoreComponents (== 1.10.0), dashTable (== 4.7.0), R6, fiery (> 1.0.0), routr (> 0.2.0), plotly, reqres (>= 0.2.3), jsonlite, htmltools, assertthat, digest, base64enc, mime, crayon, brotli

Suggests testthat

Collate 'utils.R' 'dependencies.R' 'dash-package.R' 'dash.R' 'imports.R' 'print.R' 'internal.R'

License MIT + file LICENSE

Encoding UTF-8

LazyData true

KeepSource true

RoxygenNote 7.1.0

URL https://github.com/plotly/dashR

BugReports https://github.com/plotly/dashR/issues

NeedsCompilation no

Author Chris Parmer [aut],
Ryan Patrick Kyle [aut, cre] (<https://orcid.org/0000-0001-5829-9867>),
Carson Sievert [aut] (<https://orcid.org/0000-0002-4958-2844>),
Hammad Khan [aut],
Plotly Technologies [cph]

Maintainer Ryan Patrick Kyle <ryan@plotly.com>

Repository CRAN

Date/Publication 2020-06-04 09:40:06 UTC

R topics documented:

`dash-package` *An Interface to the Dash Ecosystem for Authoring Reactive Web Applications*

Description

Dash is a productive framework for building web applications in R, Python, and Julia.

Written on top of Fiery, Plotly.js, and React.js, Dash for R is ideal for building data visualization apps with highly custom user interfaces in pure R. It's particularly suited for anyone who works with data in R.

Through a couple of simple patterns, Dash abstracts away all of the technologies and protocols that are required to build an interactive web-based application. Dash is simple enough that you can bind a user interface around your R code in an afternoon.

Dash apps are rendered in the web browser. You can deploy your apps to servers and then share them through URLs. Since Dash apps are viewed in the web browser, Dash is inherently cross-platform and mobile ready.

There is a lot behind the framework. To learn more about how it is built and what motivated Dash, watch our talk from Plotcon or read our announcement letter.

Dash is an open source package, released under the permissive MIT license. Plotly develops Dash and offers a platform for easily deploying Dash apps in an enterprise environment. If you're interested, please get in touch.

Author(s)

Maintainer: Ryan Patrick Kyle <ryan@plotly.com>

Authors:

- Chris Parmer <chris@plotly.com>
- Ryan Patrick Kyle <ryan@plotly.com>
- Carson Sievert
- Hammad Khan <hammadkhan@plotly.com>

Other contributors:

- Plotly Technologies [copyright holder]

See Also

Useful links:

- http://dashr.plotly.com
- https://github.com/plotly/dashR
- Report bugs at https://github.com/plotly/dashR/issues

`clientsideFunction` *Define a clientside callback*

Description

Create a callback that updates the output by calling a clientside (JavaScript) function instead of an R function. Note that it is also possible to specify JavaScript as a character string instead of passing `clientsideFunction`. In this case Dash will inline your JavaScript automatically, without needing to save a script inside `assets`.

Usage

```
clientsideFunction(namespace, function_name)
```

Arguments

`namespace`	Character. Describes where the JavaScript function resides (Dash will look for the function at `window[namespace][function_name]`.)
`function_name`	Character. Provides the name of the JavaScript function to call.

Details

With this signature, Dash's front-end will call `window.my_clientside_library.my_function` with the current values of the `value` properties of the components `my-input` and `another-input` whenever those values change. Include a JavaScript file by including it your assets/ folder. The file can be named anything but you'll need to assign the function's namespace to the `window`. For example, this file might look like:

```
window.my_clientside_library = {
my_function: function(input_value_1, input_value_2) {
   return (
     parseFloat(input_value_1, 10) +
       parseFloat(input_value_2, 10)
   );
}
}
```

Examples

```
## Not run:
app$callback(
  output('output-clientside', 'children'),
  params=list(input('input', 'value')),
  clientsideFunction(
  namespace = 'my_clientside_library',
  function_name = 'my_function'
  )
)

# Passing JavaScript as a character string
app$callback(
 output('output-clientside', 'children'),
 params=list(input('input', 'value')),
 "function (value) {
       return 'Client says \"' + value + '\"';
 }"
)
## End(Not run)
```

`Dash` *R6 class representing a Dash application*

Description

A framework for building analytical web applications, Dash offers a pleasant and productive development experience. No JavaScript required.

Format

An R6::R6Class generator object

Public fields

`server` A cloned (and modified) version of the fiery::Fire object provided to the `server` argument (various routes will be added which enable Dash functionality).

`config` A list of configuration options passed along to dash-renderer. Users shouldn't need to alter any of these options unless they are constructing their own authorization front-end or otherwise need to know where the application is making API calls.

Methods

Public methods:

- `Dash$new()`
- `Dash$layout_get()`
- `Dash$layout()`

- Dash$react_version_set()
- Dash$callback()
- Dash$callback_context()
- Dash$get_asset_url()
- Dash$get_relative_path()
- Dash$strip_relative_path()
- Dash$index_string()
- Dash$interpolate_index()
- Dash$title()
- Dash$run_server()
- Dash$clone()

Method `new()`**:** Create and configure a Dash application.

Usage:

```
Dash$new(
  server = fiery::Fire$new(),
  assets_folder = "assets",
  assets_url_path = "/assets",
  eager_loading = FALSE,
  assets_ignore = "",
  serve_locally = TRUE,
  meta_tags = NULL,
  url_base_pathname = "/",
  routes_pathname_prefix = NULL,
  requests_pathname_prefix = NULL,
  external_scripts = NULL,
  external_stylesheets = NULL,
  compress = TRUE,
  suppress_callback_exceptions = FALSE,
  show_undo_redo = FALSE
)
```

Arguments:

`server` fiery::Fire object. The web server used to power the application.

`assets_folder` Character. A path, relative to the current working directory, for extra files to be used in the browser. All .js and .css files will be loaded immediately unless excluded by `assets_ignore`, and other files such as images will be served if requested. Default is `assets`.

`assets_url_path` Character. Specify the URL path for asset serving. Default is `assets`.

`eager_loading` Logical. Controls whether asynchronous resources are prefetched (if `TRUE`) or loaded on-demand (if `FALSE`).

`assets_ignore` Character. A regular expression, to match assets to omit from immediate loading. Ignored files will still be served if specifically requested. You cannot use this to prevent access to sensitive files.

`serve_locally` Logical. Whether to serve HTML dependencies locally or remotely (via URL).

`meta_tags` List of lists. HTML <meta> tags to be added to the index page. Each list element should have the attributes and values for one tag, eg: `list(name = 'description',content = 'My App')`.

`url_base_pathname` Character. A local URL prefix to use app-wide. Default is `/`. Both `requests_pathname_prefix` and `routes_pathname_prefix` default to `url_base_pathname`. Environment variable is `DASH_URL_BASE_PATHNAME`.

`routes_pathname_prefix` Character. A prefix applied to the backend routes. Environment variable is `DASH_ROUTES_PATHNAME_PREFIX`.

`requests_pathname_prefix` Character. A prefix applied to request endpoints made by Dash's front-end. Environment variable is `DASH_REQUESTS_PATHNAME_PREFIX`.

`external_scripts` List. An optional list of valid URLs from which to serve JavaScript source for rendered pages.

`external_stylesheets` List. An optional list of valid URLs from which to serve CSS for rendered pages.

`compress` Logical. Whether to try to compress files and data served by Fiery. By default, `brotli` is attempted first, then `gzip`, then the `deflate` algorithm, before falling back to `identity`.

`suppress_callback_exceptions` Logical. Whether to relay warnings about possible layout mis-specifications when registering a callback.

`show_undo_redo` Logical. Set to `TRUE` to enable undo and redo buttons for stepping through the history of the app state.

Method `layout_get()`**:** Retrieves the Dash application layout.

Usage:

```
Dash$layout_get(render = TRUE)
```

Arguments:

`render` Logical. If the layout is a function, should the function be executed to return the layout? If `FALSE`, the function is returned as-is.

Details: If render is `TRUE`, and the layout is a function, the result of the function (rather than the function itself) is returned.

Returns: List or function, depending on the value of `render` (see above). When returning an object of class `dash_component`, the default `print` method for this class will display the corresponding pretty-printed JSON representation of the object to the console.

Method `layout()`**:** Set the Dash application layout (i.e., specify its user interface).

Usage:

```
Dash$layout(value)
```

Arguments:

`value` An object of the `dash_component` class, which provides a component or collection of components, specified either as a Dash component or a function that returns a Dash component.

Details: `value` should be either a collection of Dash components (e.g., dccSlider, htmlDiv, etc) or a function which returns a collection of components. The collection of components must be nested, such that any additional components contained within `value` are passed solely as `children` of the top-level component. In all cases, `value` must be a member of the `dash_component` class.

Method `react_version_set()`**:** Update the version of React in the list of dependencies served by dash-renderer to the client.

Usage:

```
Dash$react_version_set(version)
```

Arguments:

`version` Character. The version number of React to use.

Method `callback()`**:** Define a Dash callback.

Usage:

```
Dash$callback(output, params, func)
```

Arguments:

`output` Named list. The `output` argument provides the component `id` and `property` which will be updated by the callback; a callback can target one or more outputs (i.e. multiple outputs).

`params` Unnamed list; provides input and state statements, each with its own defined `id` and `property`.

`func` Function; must return output provided input or state arguments. `func` may be any valid R function, or a character string containing valid JavaScript, or a call to clientsideFunction, including `namespace` and `function_name` arguments for a locally served JavaScript function.

Details: Describes a server or clientside callback relating the values of one or more `output` items to one or more `input` items which will trigger the callback when they change, and optionally `state` items which provide additional information but do not trigger the callback directly.
The `output` argument defines which layout component property should receive the results (via the output object). The events that trigger the callback are then described by the input (and/or state) object(s) (which should reference layout components), which become argument values for R callback handlers defined in `func`.
Here `func` may either be an anonymous R function, a JavaScript function provided as a character string, or a call to `clientsideFunction()`, which describes a locally served JavaScript function instead. The latter two methods define a "clientside callback", which updates components without passing data to and from the Dash backend. The latter may offer improved performance relative to callbacks written purely in R.

Method `callback_context()`**:** Request and return the calling context of a Dash callback.

Usage:

```
Dash$callback_context()
```

Details: The `callback_context` method permits retrieving the inputs which triggered the firing of a given callback, and allows introspection of the input/state values given their names. It is only available from within a callback; attempting to use this method outside of a callback will result in a warning.
The `callback_context` method returns a list containing three elements: `states`, `triggered`, `inputs`. The first and last of these correspond to the values of `states` and `inputs` for the current invocation of the callback, and `triggered` provides a list of changed properties.

Returns: List comprising elements `states`, `triggered`, `inputs`.

Method `get_asset_url()`**:** Return a URL for a Dash asset.

Usage:

```
Dash$get_asset_url(asset_path, prefix = self$config$requests_pathname_prefix)
```

Arguments:

`asset_path` Character. Specifies asset filename whose URL should be returned.

`prefix` Character. Specifies pathname prefix; default is to use `requests_pathname_prefix`.

Details: The `get_asset_url` method permits retrieval of an asset's URL given its filename. For example, `app$get_asset_url('style.css')` should return /assets/style.css when `assets_folder = 'assets'`. By default, the prefix is the value of `requests_pathname_prefix`, but this is configurable via the `prefix` parameter. Note: this method will present a warning and return `NULL` if the Dash app was not loaded via `source()` if the `DASH_APP_PATH` environment variable is undefined.

Returns: Character. A string representing the URL to the asset.

Method `get_relative_path()`**:** Return relative asset paths for Dash assets.

Usage:

```
Dash$get_relative_path(
  path,
  requests_pathname_prefix = self$config$requests_pathname_prefix
)
```

Arguments:

`path` Character. A path string prefixed with a leading `/` which directs at a path or asset directory.

`requests_pathname_prefix` Character. The pathname prefix for the application when deployed. Defaults to the environment variable set by the server, or `""` if run locally.

Details: The `get_relative_path` method simplifies the handling of URLs and pathnames for apps running locally and on a deployment server such as Dash Enterprise. It handles the prefix for requesting assets similar to the `get_asset_url` method, but can also be used for URL handling in components such as `dccLink` or `dccLocation`. For example, `app$get_relative_url("/page/")` would return /app/page/ for an app running on a deployment server. The path must be prefixed with a `/`.

Returns: Character. A string describing a relative path to a Dash app's asset given a `path` and `requests_pathname_prefix`.

Method `strip_relative_path()`**:** Return a Dash asset path without its prefix.

Usage:

```
Dash$strip_relative_path(
  path,
  requests_pathname_prefix = self$config$requests_pathname_prefix
)
```

Arguments:

`path` Character. A path string prefixed with a leading `/` which directs at a path or asset directory.

`requests_pathname_prefix` Character. The pathname prefix for the app on a deployed application. Defaults to the environment variable set by the server, or `""` if run locally.

Details: The `strip_relative_path` method simplifies the handling of URLs and pathnames for apps running locally and on a deployment server such as Dash Enterprise. It acts almost opposite to the `get_relative_path` method, by taking a relative path as an input, and returning the `path` stripped of the `requests_pathname_prefix`, and any leading or trailing `/`. For example, a path string /app/homepage/, would be returned as `homepage`. This is particularly useful for `dccLocation` URL routing.

Method `index_string()`**:** Specify a custom index string for a Dash application.

Usage:

```
Dash$index_string(string)
```

Arguments:

`string` Character; the index string template, with interpolation keys included.

Details: The `index_string` method allows the specification of a custom index by changing the default `HTML` template that is generated by the Dash UI. #' Meta tags, CSS, and JavaScript are some examples of features that can be modified. This method will present a warning if your HTML template is missing any necessary elements and return an error if a valid index is not defined. The following interpolation keys are currently supported:

{%metas%} Optional - The registered meta tags.

{%favicon%} Optional - A favicon link tag if found in assets.

{%css%} Optional - Link tags to CSS resources.

{%config%} Required - Configuration details generated by Dash for the renderer.

{%app_entry%} Required - The container where Dash React components are rendered.

{%scripts%} Required - Script tags for collected dependencies.

Example of a basic HTML index string: `"<!DOCTYPE html>`

```
<html>
 <head>
  {%meta_tags%}
     <title>{{
     {%favicon%}
     {%css_tags%}
 </head>
  <body>
    {%app_entry%}
    <footer>
     {%config%}
     {%scripts%}
    </footer>
  </body>
</html>"
```

Method `interpolate_index()`**:** Modify index template variables for a Dash application.

Usage:

```
Dash$interpolate_index(template_index = private$template_index[[1]], ...)
```

Arguments:

`template_index` Character. A formatted string with the HTML index string. Defaults to the initial template.

`...` Named list. The unnamed arguments can be passed as individual named lists corresponding to the components of the Dash HTML index. These include the same argument as those found in the `index_string()` template.

Details: With the `interpolate_index` method, one can pass a custom index with template string variables that are already evaluated. Directly passing arguments to the `template_index` has the effect of assigning them to variables present in the template. This is similar to the `index_string` method but offers the ability to change the default components of the Dash index as seen in the example below.

Examples:

```
library(dash)
app <- Dash$new()

sample_template <- "<!DOCTYPE html>
<html>
<head>
{%meta_tags%}
<title>Index Template Test</title>
{%favicon%}
{%css_tags%}
</head>
<body>
{%app_entry%}
<footer>
{%config%}
{%scripts%}
</footer>
</body>
</html>"

# this is the default configuration, but custom configurations
# are possible -- the structure of the "config" argument is
# a list, in which each element is a JSON key/value pair, when
# reformatted as JSON from the list:
#  e.g. {"routes_pathname_prefix":"/", "ui":false}
config <- sprintf("<script id='_dash-config' type='application/json'> %s </script>",
                  jsonlite::toJSON(app$config, auto_unbox=TRUE))

app$interpolate_index(
  sample_template,
  metas = "<meta_charset='UTF-8'/>",
 app_entry = "<div id='react-entry-point'><div class='_dash-loading'>Loading...</div></div>",
  config = config,
  scripts = "")
```

Method `title()`**:** Set the title of the Dash app

Usage:

```
Dash$title(string = "Dash")
```

Arguments:

`string` Character. A string representation of the name of the Dash application.

Details: If no title is supplied, Dash for R will use 'Dash'.

Method `run_server()`**:** Start the Fiery HTTP server and run a Dash application.

Usage:

```
Dash$run_server(
  host = Sys.getenv("HOST", "127.0.0.1"),
  port = Sys.getenv("PORT", 8050),
  block = TRUE,
  showcase = FALSE,
  use_viewer = FALSE,
  dev_tools_prune_errors = TRUE,
  debug = Sys.getenv("DASH_DEBUG"),
  dev_tools_ui = Sys.getenv("DASH_UI"),
  dev_tools_props_check = Sys.getenv("DASH_PROPS_CHECK"),
  dev_tools_hot_reload = Sys.getenv("DASH_HOT_RELOAD"),
  dev_tools_hot_reload_interval = Sys.getenv("DASH_HOT_RELOAD_INTERVAL"),
 dev_tools_hot_reload_watch_interval = Sys.getenv("DASH_HOT_RELOAD_WATCH_INTERVAL)"),
  dev_tools_hot_reload_max_retry = Sys.getenv("DASH_HOT_RELOAD_MAX_RETRY"),
  dev_tools_silence_routes_logging = NULL,
  ...
)
```

Arguments:

`host` Character. A string specifying a valid IPv4 address for the Fiery server, or 0.0.0.0 to listen on all addresses. Default is 127.0.0.1 Environment variable: `HOST`.

`port` Integer. Specifies the port number on which the server should listen (default is `8050`). Environment variable: `PORT`.

`block` Logical. Start the server while blocking console input? Default is `TRUE`.

`showcase` Logical. Load the Dash application into the default web browser when server starts? Default is `FALSE`.

`use_viewer` Logical. Load the Dash application into RStudio's viewer pane? Requires that `host` is either 127.0.0.1 or `localhost`, and that Dash application is started within RStudio; if `use_viewer = TRUE` and these conditions are not satisfied, the user is warned and the app opens in the default browser instead. Default is `FALSE`.

`dev_tools_prune_errors` Logical. Reduce tracebacks such that only lines relevant to user code remain, stripping out Fiery and Dash references? Only available with debugging. `TRUE` by default, set to `FALSE` to see the complete traceback. Environment variable: `DASH_PRUNE_ERRORS`.

`debug` Logical. Enable/disable all the Dash developer tools (and the within-browser user interface for the callback graph visualizer and stack traces) unless overridden by the arguments or environment variables. Default is `FALSE` when called via `run_server`. For more information, please visit https://dashr.plotly.com/devtools. Environment variable: `DASH_DEBUG`.

`dev_tools_ui` Logical. Show Dash's developer tools UI? Default is `TRUE` if `debug == TRUE`, `FALSE` otherwise. Environment variable: `DASH_UI`.

`dev_tools_props_check` Logical. Validate the types and values of Dash component properties? Default is `TRUE` if `debug == TRUE`, `FALSE` otherwise. Environment variable: `DASH_PROPS_CHECK`.

`dev_tools_hot_reload` Logical. Activate hot reloading when app, assets, and component files change? Default is `TRUE` if `debug == TRUE`, `FALSE` otherwise. Requires that the Dash application is loaded using `source()`, so that `srcref` attributes are available for executed code. Environment variable: `DASH_HOT_RELOAD`.

`dev_tools_hot_reload_interval` Numeric. Interval in seconds for the client to request the reload hash. Default is `3`. Environment variable: `DASH_HOT_RELOAD_INTERVAL`.

`dev_tools_hot_reload_watch_interval` Numeric. Interval in seconds for the server to check asset and component folders for changes. Default `0.5`. Environment variable: `DASH_HOT_RELOAD_WATCH_INTERVAL`.

`dev_tools_hot_reload_max_retry` Integer. Maximum number of failed reload hash requests before failing and displaying a pop up. Default `0.5`. Environment variable: `DASH_HOT_RELOAD_MAX_RETRY`.

`dev_tools_silence_routes_logging` Logical. Replace Fiery's default logger with `dashLogger` instead (will remove all routes logging)? Enabled with debugging by default because hot reload hash checks generate a lot of requests.

`...` Additional arguments to pass to the `start` handler; see the fiery documentation for relevant examples.

Details: Starts the Fiery server in local mode and launches the Dash application. If a parameter can be set by an environment variable, that is listed too. Values provided here take precedence over environment variables. . If provided, `host`/`port` set the `host`/`port` fields of the underlying fiery::Fire web server. The `block`/`showcase`/`...` arguments are passed along to the `ignite()` method of the fiery::Fire server.

Examples:

```
if (interactive() && require(dash)) {
  library(dashCoreComponents)
  library(dashHtmlComponents)
  library(dash)

  app <- Dash$new()
  app$layout(htmlDiv(
    list(
      dccInput(id = "inputID", value = "initial value", type = "text"),
      htmlDiv(id = "outputID")
    )
   )
  )

  app$callback(output = list(id="outputID", property="children"),
               params = list(input(id="inputID", property="value"),
                             state(id="inputID", property="type")),
    function(x, y)
      sprintf("You've entered: '%s' into a '%s' input control", x, y)
  )

  app$run_server(showcase = TRUE)
}
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Dash$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
## ------------------------------------------------
## Method `Dash$interpolate_index`
## ------------------------------------------------

library(dash)
app <- Dash$new()

sample_template <- "<!DOCTYPE html>
<html>
<head>
{%meta_tags%}
<title>Index Template Test</title>
{%favicon%}
{%css_tags%}
</head>
<body>
{%app_entry%}
<footer>
{%config%}
{%scripts%}
</footer>
</body>
</html>"

# this is the default configuration, but custom configurations
# are possible -- the structure of the "config" argument is
# a list, in which each element is a JSON key/value pair, when
# reformatted as JSON from the list:
#  e.g. {"routes_pathname_prefix":"/", "ui":false}
config <- sprintf("<script id='_dash-config' type='application/json'> %s </script>",
                  jsonlite::toJSON(app$config, auto_unbox=TRUE))

app$interpolate_index(
  sample_template,
  metas = "<meta_charset='UTF-8'/>",
 app_entry = "<div id='react-entry-point'><div class='_dash-loading'>Loading...</div></div>",
  config = config,
  scripts = "")

## ------------------------------------------------
## Method `Dash$run_server`
## ------------------------------------------------

if (interactive() && require(dash)) {
```

```
  library(dashCoreComponents)
  library(dashHtmlComponents)
  library(dash)

  app <- Dash$new()
  app$layout(htmlDiv(
    list(
      dccInput(id = "inputID", value = "initial value", type = "text"),
      htmlDiv(id = "outputID")
    )
   )
  )

  app$callback(output = list(id="outputID", property="children"),
               params = list(input(id="inputID", property="value"),
                             state(id="inputID", property="type")),
    function(x, y)
      sprintf("You've entered: '%s' into a '%s' input control", x, y)
  )

  app$run_server(showcase = TRUE)
}
```

---

dependencies *Input/Output/State definitions*

Description

Use in conjunction with the `callback()` method from the dash::Dash class to define the update logic in your application.

Usage

```
output(id, property)

input(id, property)

state(id, property)

dashNoUpdate()
```

Arguments

`id`	a component id
`property`	the component property to use

Details

The `dashNoUpdate()` function permits application developers to prevent a single output from updating the layout. It has no formal arguments.

print.dash_component *Output a dash component object as JSON*

Description

Objects of the dash_component class support a print method, which first processes the nested list object, and then returns its JSON representation.

Usage

```
## S3 method for class 'dash_component'
print(x, ...)
```

Arguments

x	an object of class dash_component
...	not currently used

Index