Package: blueprintr (via r-universe)

October 13, 2024

Title Automagically Document and Test Datasets Using Targets Or Drake

Version 0.2.7

Description Documents and tests datasets in a reproducible manner so that data lineage is easier to comprehend for small to medium tabular data. Originally designed to aid data cleaning tasks for humanitarian research groups, specifically large-scale longitudinal studies.

License MIT + file LICENSE

Suggests testthat (>= 2.1.0), covr, codetools, knitr, rmarkdown, kableExtra, rcoder, labelled, drake (>= 7.11.0), panelcleaner, kfa, callr, igraph, uuid, visNetwork

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

URL https://github.com/nyuglobalties/blueprintr

BugReports https://github.com/nyuglobalties/blueprintr/issues

Depends R (>= 3.5.0)

Imports targets, rlang, here, glue, magrittr, readr, lifecycle, tidytable, tidyselect (>= 1.2.0), snakecase, digest, data.table

VignetteBuilder knitr

Remotes nyuglobalties/panelcleaner

Repository https://nyuglobalties.r-universe.dev

RemoteUrl https://github.com/nyuglobalties/blueprintr

RemoteRef HEAD

RemoteSha fc12a4572f995e878d5e974cb244f6eb662c8e94

Contents

`annotations` *Access the blueprintr metadata at runtime*

Description

Access the blueprintr metadata at runtime

Usage

```
annotations(x)

annotation_names(x)

annotation(x, field)
```

```
super_annotation(x, field)

has_annotation(x, field)

has_super_annotation(x, field)

add_annotation(x, field, value, overwrite = FALSE)

set_annotation(x, field, value)

add_super_annotation(x, field, value)

remove_super_annotation(x, field)
```

Arguments

x	An object, most likely a variable in a data.frame
field	The name of a metadata field
value	A value to assign to an annotation field
overwrite	If TRUE, allows overwriting of existing annotation values

Functions

- annotations(): Gets a list of all annotations assigned to an object
- annotation_names(): Get the names of all of the annotations assigned to an object
- annotation(): Gets an annotation for an object
- super_annotation(): Gets an annotation that overrides existing annotations
- has_annotation(): Checks to see if an annotation exists for an object
- has_super_annotation(): Checks to see if an overriding annotation exists for an object
- add_annotation(): Adds an annotation to an object, with the option of overwriting an existing value
- set_annotation(): Alias to add_annotation(overwrite = TRUE)
- add_super_annotation(): Adds an overriding annotation to an object. Note that overriding annotations will overwrite previous assignments!
- remove_super_annotation(): Removes overriding annotation

attach_blueprints *Attach blueprints to a drake plan*

Description

Blueprints outline a sequence of checks and cleanup steps that come after a dataset is created. In order for these steps to be executed, the blueprint must be attached to a drake plan so that drake can run these steps properly.

Usage

```
attach_blueprints(plan, ...)

attach_blueprint(plan, blueprint)
```

Arguments

plan	A drake plan
...	Multiple blueprints
blueprint	A blueprint object

blueprint *Create a blueprint*

Description

Create a blueprint

Usage

```
blueprint(
  name,
  command,
  description = NULL,
  metadata = NULL,
  annotate = FALSE,
  metadata_file_type = c("csv"),
  metadata_file_name = NULL,
  metadata_directory = NULL,
  metadata_file_path = NULL,
  extra_steps = NULL,
  ...,
  class = character()
)
```

Arguments

name	The name of the blueprint
command	The code to build the target dataset
description	An optional description of the dataset to be used for codebook generation
metadata	The associated variable metadata for this dataset
annotate	If TRUE, during cleanup the metadata will "annotate" the dataset by adding variable attributes for each metadata field to make metadata provenance easier and responsive to code changes.
metadata_file_type	The kind of metadata file. Currently only CSV.
metadata_file_name	The file name for the metadata file. If the option blueprintr.use_local_metadata_path is set to TRUE, then the default file name will be the name of the blueprint script, minus the .R extension. Otherwise, this will default to the name of the blueprint.
metadata_directory	Where the metadata file will be stored. If the option blueprintr.use_local_metadata_path is set to TRUE, then the default location will be the folder where the blueprint script is located. Otherwise, this will default to here::here("blueprints")
metadata_file_path	Overrides the metadata file path generated by metadata_directory, name, and metadata_file_type if not NULL.
extra_steps	A list() of extra 'bpstep' objects, which add extra targets to the workflow after the desired dataset has completed its cleanup phase. Uses of this could include generating codebooks or other reports based on the built data. See bp_add_bpstep() for more details.
...	Any other parameters and settings for the blueprint
class	A subclass of blueprint capability, for future work

Value

A blueprint object

Cleanup Tasks

blueprintr offers some post-check tasks that attempt to match datasets to the metadata as much as possible. There are two default tasks that run:

1. Reorders variables to match metadata order.
2. Drops variables marked with dropped == TRUE if the dropped variable exists in the metadata.

The remaining tasks have to be enabled by the user:

- If labelled = TRUE in the blueprint() command, all columns will be converted to labelled() columns, provided that at least the description field is filled in. If the coding column is present in the metadata, then categorical levels as specified by a coding() will be added to the column as well. In case the description field is used for detailed column descriptions, the title field can be added to the metadata to act as short titles for the columns.

`blueprint_macros` *Macros for blueprint authoring*

Description

`blueprintr` uses code inspection to identify and trace dataset dependencies. These macro functions signal a dependency to `blueprintr` and evaluate to symbols to be analyzed in the `drake` plan.

Usage

```
.TARGET(bp_name, .env = parent.frame())

.BLUEPRINT(bp_name, .env = parent.frame())

.META(bp_name, .env = parent.frame())

.SOURCE(dat_name)

mark_source(dat)
```

Arguments

`bp_name`	Character string of blueprint's name
`.env`	The environment in which to evaluate the macro. For internal use only!
`dat_name`	Character string of an object's name, used exclusively for marking "sources"
`dat`	A data.frame-like object

Functions

- `.TARGET()`: Gets symbol of built and checked data
- `.BLUEPRINT()`: Gets symbol of blueprint reference in plan
- `.META()`: Gets symbol of metadata reference in plan
- `.SOURCE()`: Gets a symbol for an object intended to be a "data source"
- `mark_source()`: Mark an data.frame-like object as a source table

When to use

Generally speaking, the `.BLUEPRINT` and `.META` macros should be used for check functions, which frequently require context, e.g. in the form of configuration from the blueprint or coding expectations from the metadata. `.TARGET` is primarily used in blueprint commands, but there could be situations where a check depends on the content of another dataset.

It is important to note that the symbols generated by these macros are only understood in the context of a `drake` plan. The targets associated with the symbols are generated when blueprints are attached to a plan.

Sources

Sources are an ability to add variable UUIDs to objects that are not constructed using blueprints. This is often the case if the sourced table derives from some intermittent HTTP query or a file from disk. Blueprints have limited capability of configuring the underlying target behavior during the `_initial` phase, so often it is easier to do that sort of fetching and pre-processing before using blueprints. However, you lose the benefit of variable lineage when you don't use blueprints. "Sources" are simply data.frame-like objects that have the ".uuid" attribute for each variable so that variable lineage can cover the full data lifetime. Use `blueprintr::mark_source()` to add the UUID attributes, and then use `.SOURCE()` in the blueprints so lineage can be captured

Examples

```
.TARGET("example_dataset")
.BLUEPRINT("example_dataset")
.META("example_dataset")

blueprint(
  "test_bp",
  description = "Blueprint with dependencies",
  command =
    .TARGET("parent1") %>%
      left_join(.TARGET("parent2"), by = "id") %>%
      filter(!is.na(id))
)
```

---

bpstep *Define a step of blueprint assembly*

Description

Each step in the blueprint assembly process is contained in a wrapper 'bpstep' object.

Usage

```
bpstep(step, bp, payload, ...)
```

Arguments

`step`	The name of the step
`bp`	A 'blueprint' object to create the assembled step
`payload`	A 'bpstep_payload' object that outlines the code to be assembled depending on the workflow executor
`...`	Extensions to the bpstep, like "allow_duplicates"

Value

A 'bpstep' object

bpstep_payload *Create a step payload*

Description

The bpstep payload is the object that contains the target name and command, along with any other metadata to be passed to the execution engine.

Usage

```
bpstep_payload(target_name, target_command, ...)
```

Arguments

target_name	The target's name
target_command	The target's command
...	Arguments to be passed to the executing engine (e.g. arguments sent to targets::tar_target())

Value

A bpstep payload object

Examples

```
if (FALSE) {
  bpstep(
    step = "some_step",
    bp = some_bp_object,
    payload = bpstep_payload(
      "payload_name",
      payload_command()
    )
  )
}
```

bp_add_bpstep *Add custom bpstep to blueprint schema*

Description

`blueprint()` objects store custom bpstep objects in the "extra_steps" element. This function adds a new step to that element.

Usage

```
bp_add_bpstep(bp, step)
```

Arguments

bp	A blueprint
step	A bpstep object

Examples

```
if (FALSE) {
  # Based on the codebook export step
  step <- bpstep(
    step = "export_codebook",
    bp = bp,
    payload = bpstep_payload(
      target_name = blueprint_codebook_name(bp),
      target_command = codebook_export_call(bp),
      format = "file",
      ...
    )
  )

  bp_add_bpstep(
    bp,
    step
  )
}
```

---

bp_export_codebook *Instruct blueprint to export codebooks*

Description

Instruct blueprint to export codebooks

Usage

```
bp_export_codebook(
  blueprint,
  summaries = FALSE,
  file = NULL,
  template = NULL,
  title = NULL
)
```

Arguments

blueprint	A blueprint
summaries	Whether or not variable summaries should be included in codebook
file	Path to where the codebook should be saved
template	A path to an RMarkdown template
title	Optional title of codebook

Value

An amended blueprint with the codebook export instructions

Examples

```
## Not run:
test_bp <- blueprint(
  "mtcars_dat",
  description = "The mtcars dataset",
  command = mtcars
)

new_bp <- test_bp %>% bp_export_codebook()

## End(Not run)
```

---

bp_export_kfa_report *Instruct blueprint to generate kfa report*

Description

Instruct blueprint to generate kfa report

Usage

```
bp_export_kfa_report(
  bp,
  scale,
  path = NULL,
  path_pattern = NULL,
  format = NULL,
  title = NULL,
  kfa_args = list(),
  ...
)
```

Arguments

bp	A blueprint
scale	Which scale(s) to analyze
path	Path(s) to where the report(s) should be saved
path_pattern	Override the default location to save files (always rooted to the project root with here::here())
format	The output format of the report(s)
title	Optional title of report
kfa_args	Arguments forwarded to `kfa::kfa()` for this batch of scales
...	Arguments forwarded to the executing engine e.g. targets::tar_target_raw() or drake::target()

Value

An amended blueprint with the kfa report export instructions

Examples

```
## Not run:
test_bp <- blueprint(
  "mtcars_dat",
  description = "The mtcars dataset",
  command = mtcars
)

new_bp <- test_bp %>% bp_export_codebook()

## End(Not run)
```

`bp_extend` *Add custom elements to a blueprint*

Description

`blueprint()` objects are essentially just `list()` objects that contain a bunch of metadata on the data asset construction. Use `bp_extend()` to set or add new elements.

Usage

```
bp_extend(bp, ...)
```

Arguments

bp	A blueprint
...	Keyword arguments forwarded to blueprint()

Examples

```
if (FALSE) {
  bp <- blueprint("some_blueprint", ...)
  adjusted_bp <- bp_extend(bp, new_option = TRUE)
  bp_with_annotation_set <- bp_extend(bp, annotate = TRUE)
}
```

bp_include_panelcleaner_meta *Include panelcleaner mapping on metadata creation*

Description

panelcleaner defines a mapping structure used for data import of panel, or more generally longitudinal, surveys / data which can be used as a source for some kinds of metadata (currently, only categorical coding information). If the blueprint constructs a `mapped_df` object, then this extension will signal to blueprintr to extract the mapping information and include it.

Usage

```
bp_include_panelcleaner_meta(blueprint)
```

Arguments

`blueprint`	A blueprint that may create a `mapped_df` data.frame

Value

An amended blueprint with `mapped_df` metadata extraction set for metadata creation

bp_label_variables *Convert variables to labelled variables in cleanup stage*

Description

The haven package has a handy tool called "labeled vectors", which are like factors that can be interpreted in other statistical software like STATA and SPSS. See haven::labelled() for more information on the type. Running this on a blueprint will instruct the blueprint to convert all variables with non-NA `title`, `description`, or `coding` fields to labeled vectors.

Usage

```
bp_label_variables(blueprint)
```

Arguments

`blueprint`	A blueprint

Value

An amended blueprint with variable labelling in the cleanup phase set

checks *Evaluate checks on the blueprint build output*

Description

After building a dataset, it's beneficial (if not a requirement) to run tests on that dataset to ensure that it behaves as expected. `blueprintr` gives authors a framework to run these tests automatically, both for individual variables and general dataset checks. `blueprintr` provides three functions as models for developing these kinds of functions: one to check that all expected variables are present, one to check the variable types, and a generic function that checks if variable values are contained within a known set.

Usage

```
all_variables_present(df, meta, blueprint)

all_types_match(df, meta)
```

Arguments

`df`	The built dataset
`meta`	The dataset's metadata
`blueprint`	The dataset's blueprint

check_list *Create a quoted list of check calls*

Description

Create a quoted list of check calls

Usage

```
check_list(...)
```

Arguments

`...`	A collection of calls to be used for checks

cleanup *Run clean-up tasks and return built dataset*

Description

After checks pass, this step runs in the blueprint sequence. If any cleanup features are enabled, they will run on the dataset prior to setting the final blueprint target.

Usage

```
cleanup(results, df, blueprint, meta)
```

Arguments

results	A reference to the checks results. Currently used to ensure that this step runs after the checks step.
df	The built dataset
blueprint	The blueprint associated with the built dataset
meta	The metadata associated with the built dataset

create_metadata_file *Create a metadata file from a dataset*

Description

One of the targets in the blueprint workflow target chain. If a metadata file does not exist, then this function will be added to the workflow.

Usage

```
create_metadata_file(df, blueprint, ...)
```

Arguments

df	A dataframe that the metadata table describes
blueprint	The original blueprint for the dataframe
...	A variable list of metadata tables on which this metadata table depends

eval_checks *Evaluate all checks on a blueprint*

Description

Runs all checks – dataset and variable – on a blueprint to determine if a built dataset passes all restrictions.

Usage

```
eval_checks(..., .env = parent.frame())
```

Arguments

...	All quoted check calls
.env	The environment in which the calls are evaluated

Check functions

Check functions are simple functions that take in either a data.frame or variable at the minimum, plus some extra arguments if need, and returns a logical value: TRUE or FALSE. In blueprintr, the entire check passes or fails unlike other testing frameworks like pointblank. If you'd like to embed extra context for your test result, modify the "check.errors" attribute of the returned logical value with a character vector which will be rendered into a bulleted list. Note: if you embed reasons for a TRUE, the check will produce a warning in the targets or drake pipeline.

in_set *Test if x is a subset of y*

Description

Test if x is a subset of y

Usage

```
in_set(x, y)
```

Arguments

x	A vector
y	A vector representing an entire set

load_blueprint *Load a blueprint from a script file*

Description

Load a blueprint from a script file

Usage

```
load_blueprint(plan, file)

load_blueprints(plan, directory = here::here("blueprints"), recurse = FALSE)
```

Arguments

plan	A drake plan
file	A path to a script file
directory	A path to a directory with script files that are blueprints. Defaults to the "blueprints" directory at the root of the current R project.
recurse	Recursively loads blueprints from a directory if TRUE

Value

A drake_plan with attached blueprints

Empty blueprint folder

By default, blueprintr ignore empty blueprint folders. However, it may be beneficial to warn users if folder is empty, particularly during project setup. This helps identify any potential misconfiguration of drake plan attachment. To enable these warnings, set `option(blueprintr.warn_empty_blueprints_dirs = TRUE)`.

load_table_lineage *Read blueprints from folder and get lineage*

Description

Read blueprints from folder and get lineage

Usage

```
load_table_lineage(
  directory = here::here("blueprints"),
  recurse = FALSE,
  script = here::here("_targets.R")
)
```

Arguments

directory	A folder containing blueprint scripts
recurse	Should this function recursively load blueprints?
script	Where the targets/drake project script file is located. Defaults to using targets.

Value

An igraph of the table lineage for the desired blueprints

---

metadata *Convert an input dataframe into a metadata object*

Description

Convert an input dataframe into a metadata object

Usage

```
metadata(df)
```

Arguments

df	A dataframe that will be converted into a metadata object, once content checks pass.

---

mutate_annotation *Modify dataset variable annotations*

Description

Usually, metadata should be a reflection of what the data *should* represent and act as a check on the generation code. However, in the course of data aggregation, it can be common to perform massive transformations that would be cumbersome to document manually. This exposes a metadata-manipulation framework prior to metadata file creation, in the style of `tidytable::mutate`.

Usage

```
mutate_annotation(.data, .field, ..., .overwrite = TRUE)

mutate_annotation_across(
  .data,
  .field,
  .fn,
  .cols = tidyselect::everything(),
  .with_names = FALSE,
  ...,
  .overwrite = TRUE
)
```

Arguments

.data	A data.frame
.field	The name of the annotation field that you wish to modify
...	For mutate_annotation, named parameters that contain the annotation values. Like tidytable::mutate, each parameter name is a variable (that must already exist!), and each parameter value is an R expression, evaluated with .data as a data mask. For mutate_annotation_across, extra arguments passed to .fn
.overwrite	If TRUE, overwrites existing annotation values. Annotations have an overwriting guard by default, but since these functions are intentionally modifying the annotations, this parameter defaults to TRUE.
.fn	A function that takes in a vector and arbitrary arguments ... If .with_names is TRUE, then .fn will be passed the vector *and* the name of the vector, since it's often useful to compute on the metadata.
.cols	A tidyselect-compatible selection of variables to be edited
.with_names	If TRUE, passes a column *and* its name as arguments to .fn

Value

A data.frame with annotated columns

Examples

```
# Adds a "mean" annotation to 'mpg'
mutate_annotation(mtcars, "mean", mpg = mean(mpg))

# Adds a "mean" annotation to all variables in `mtcars`
mutate_annotation_across(mtcars, "mean", .fn = mean)

# Adds a "title" annotation that copies the column name
mutate_annotation_across(
  mtcars,
  "title",
  .fn = function(x, nx) nx,
  .with_names = TRUE
)
```

---

plan_from_blueprint *Create a drake plan from a blueprint*

Description

Creates a new drake plan from a blueprint

Usage

```
plan_from_blueprint(blueprint)
```

Arguments

blueprint	A blueprint

Value

A drake plan with all of the necessary blueprint steps

---

render_codebook *Render codebooks for datasets*

Description

Render codebooks for datasets

Usage

```
render_codebook(
  blueprint,
  meta,
  file,
  title = glue::glue("{ui_value(blueprint$name)} Codebook"),
  dataset = NULL,
  template = bp_path("codebook_templates/default_codebook.Rmd"),
  ...
)
```

Arguments

blueprint	A dataset blueprint
meta	A `blueprint_metadata` object related to the blueprint
file	Path to where the codebook should be saved
title	Title of the codebook
dataset	If included, a `data.frame` to be used as a source for summaries
template	Path to the knitr template
...	Extra parameters passed to `rmarkdown::render()`

`render_kfa_report` *Render k-fold factor analysis on scale using kfa*

Description

Generates a k-fold factor analysis report using the 'scale' field in the blueprintr data dictionaries. While not recommended, this function does allow for multiple loaded variables, delimited by commas. For example, 'var1' could have 'scale' be "SCALE1,SCALE2".

Usage

```
render_kfa_report(
  dat,
  bp,
  meta,
  scale,
  path = NULL,
  path_pattern = "reports/kfa-{snakecase_scale}-{dat_name}.html",
  format = NULL,
  title = NULL,
  ...
)
```

Arguments

`dat`	Source data
`bp`	The dataset's blueprint
`meta`	blueprintr data dictionary
`scale`	Scale identifier to be located in the 'scale' field
`path`	Where to output the report; defaults to the "reports" subfolder of the current working *project* folder.
`path_pattern`	If path is `NULL`, this is where the report will be saved. Variables available for use are: • `scale`: The scale name defined in the metadata • `snakecase_scale`: `scale` but in snake_case • `dat_name`: Name of the dataset (equivalent to the blueprint name)
`format`	The output format; defaults to 'html_document'
`title`	Optional title of the report
`...`	Arugments forwarded kfa::kfa()

Value

Path to where the generated report is saved

super_annotations *"Super Annotations"*

Description

As of blueprintr 0.2.1, there is now the option for metadata files to **always** overwrite annotations at runtime. Previously, this would be a conflict with mutate_annotation and mutate_annotation_across since the annotation phase happens during the blueprint cleanup phase, whereas these annotation manipulation tools occur at the blueprint initial phase. To resolve this, 0.2.1 introduces "super annotations", which are just annotations prefixed with "super.". However, the super annotations will *overwrite* the normal annotations during cleanup. This gives the annotation manipulation tools a means of not losing their work if `annotate_overwrite` is effectively enabled. To enable this functionality, set `options(blueprintr.use_improved_annotations = TRUE)`. This also has the side effect of **always** treating `annotate = TRUE` and `annotate_overwrite = TRUE`.

Usage

```
improved_annotation_option()

using_improved_annotations()
```

Functions

- `improved_annotation_option()`: Returns the option string for improved annotations
- `using_improved_annotations()`: Checks if improved annotations are enabled

tar_blueprint *Add a blueprint to a "targets" pipeline*

Description

Unlike drake, which requires some extra metaprogramming to "attach" blueprint steps to a plan, targets pipelines allow for direct target construction. Blueprints can thus be added directly into a `tar_pipeline()` object using this function. The arguments for `tar_blueprint()` are exactly the same as `blueprint()`. `tar_blueprints()` behaves like `load_blueprints()` but is called, like `tar_blueprint()`, directly in a `tar_pipeline()` object.

Usage

```
tar_blueprint(...)

tar_blueprints(directory = here::here("blueprints"), recurse = FALSE)

tar_blueprint_raw(bp)
```

Arguments

`...`	Arguments passed to `blueprint()`
`directory`	A folder containing R scripts that evaluate to `blueprint()` objects
`recurse`	Recursively loads blueprints from a directory if `TRUE`
`bp`	A blueprint object

Value

A `list()` of `tar_target` objects

Empty blueprint folder

By default, blueprintr ignore empty blueprint folders. However, it may be beneficial to warn users if folder is empty, particularly during project setup. This helps identify any potential misconfiguration of targets generation. To enable these warnings, set `option(blueprintr.warn_empty_blueprints_dirs = TRUE)`.

`variable_lineage` *Variable lineage*

Description

This is an experimental feature that traces variable lineage through an injection of a ".uuid" attribute for each variable. Previous attempts at variable lineage were conducted using variable names and heuristics of known functions. This approach yields a more consistent lineage.

Usage

```
load_variable_lineage(
  directory = here::here("blueprints"),
  recurse = FALSE,
  script = here::here("_targets.R")
)

filter_variable_lineage(
  g,
  variables = NULL,
  tables = NULL,
  mode = "all",
  cutoff = -1
)

vis_variable_lineage(..., g = NULL, cluster_by_dataset = TRUE)
```

Arguments

directory	A folder containing blueprint scripts
recurse	Should this function recursively load blueprints?
script	Where the targets/drake project script file is located. Defaults to using targets.
g	An igraph object. This defaults to a graph loaded with load_variable_lineage. However, use this if you want to inspect subgraphs of the variable lineage.
variables	Character vector of patterns for variable names to match. Note that each pattern is assumed to be disjoint (e.g. "if variable pattern A *or* variable pattern B"), but if tables is not NULL, the search will be joint (e.g. "if (variable pattern A *or* variable pattern B) *and* (table pattern A *or* table pattern B)").
tables	Character vector of patterns for table names to match. Note that each pattern is assumed to be disjoint (e.g. "if table pattern A *or* table pattern B"), but if variables is not NULL, the search will be joint (e.g. "if (table pattern A *or* table pattern B) *and* (variable pattern A *or* variable pattern B)").
mode	Which sort of relationships to include. Defaults to "all" (includes both relations *to* the target node in the graph and *from* the target node in the graph). See igraph::all_simple_paths() for more details.
cutoff	The number of node steps to consider in the graph traversal for filtering. Defaults to -1 (no limit on steps). See igraph::all_simple_paths() for more details.
...	Arguments passed to load_variable_lineage
cluster_by_dataset	If TRUE, variable nodes will be clustered into their respective dataset

Details

To enable the variable feature, set `options(blueprintr.use_variable_uuids = TRUE)`.

Functions

- `load_variable_lineage()`: Reads blueprintrs from folder to get variable lineage. Returns an igraph of the variable lineage.
- `filter_variable_lineage()`: Filter for specific variables to include in the lineage graph
- `vis_variable_lineage()`: Visualizes variable lineage with visNetwork. Returns an interactive graph.

---

vis_table_lineage *Visualize table lineage with visNetwork*

Description

Visualize table lineage with visNetwork

Usage

```
vis_table_lineage(..., g = NULL)
```

Arguments

...	Arguments passed to load_table_lineage
g	An igraph object, defaulting to the one created with load_table_lineage

Value

Interactive graph run by visNetwork

Index