Package: crawlee (via r-universe)

July 4, 2026

Title Tidy Interface for Reproducible Web Crawling

Version 0.1.0

Description A tidy, pipe-friendly toolkit for reproducible web crawling and structured data collection, inspired by the architecture of the 'Crawlee' library. Provides a unified crawler with a deduplicating, resumable request queue, content-type aware handlers, structured storage backends and rich console logging via 'cli'. Supports crawling HTML pages, sitemaps, RSS and Atom feeds and PDF documents, with optional headless-browser rendering and helpers for retrieval-augmented generation.

License MIT + file LICENSE

URL https://github.com/StrategicProjects/crawlee, https://strategicprojects.github.io/crawlee/

BugReports https://github.com/StrategicProjects/crawlee/issues

Depends R (>= 4.1.0)

Imports cli, httr2, R6, rlang, rvest, tibble, vctrs, xml2

Suggests arrow, chromote, DBI, dplyr, duckdb, httptest2, jsonlite, knitr, later, nanoparquet, pdftools, polite, promises, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

Language en-US

Roxygen list(markdown = TRUE)

RoxygenNote 8.0.0

Config/pak/sysreqs libxml2-dev libssl-dev

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

Date/Publication 2026-06-29 12:10:28 UTC

RemoteUrl https://github.com/strategicprojects/crawlee

RemoteRef HEAD

RemoteSha fef3fd8e7c115fff2dccc4b4091209f16ca7e573

Contents

`cr_autoscale` *Enable autoscaled parallel fetching*

Description

Like `cr_parallel()`, but the batch concurrency adapts at run time, the rough equivalent of Crawlee's autoscaled pool. After each batch the engine adjusts concurrency with an additive-increase / multiplicative-decrease rule: it grows by one when a batch is clean, and halves on back-pressure (a transport failure or an HTTP 429/500/502/503/504), staying within `[min, max]`.

Usage

```
cr_autoscale(crawler, min = 1L, max = 16L, max_active = NULL)
```

Arguments

`crawler`	A Crawler.
`min`, `max`	Concurrency bounds. The crawl starts at `min`.
`max_active`	Maximum simultaneously-active connections (defaults to the current concurrency).

Value

The crawler, invisibly.

Examples

```
crawler("https://example.com") |> cr_autoscale(min = 2, max = 16)
```

---

cr_chunk *Chunk text for retrieval-augmented generation*

Description

Splits documents into overlapping chunks suitable for embedding and retrieval. Works on a character vector (one element per document) or on a data frame, in which case the chunked column is replaced by `text` and all other columns are carried along as per-chunk metadata.

Usage

```
cr_chunk(
  data,
  text = NULL,
  size = 1000L,
  overlap = 200L,
  by = c("char", "word")
)
```

Arguments

`data`	A character vector or a data frame (e.g. the result of `cr_collect()`).
`text`	When `data` is a data frame, the (unquoted) column holding the text to chunk.
`size`	Target chunk size, in characters (`by = "char"`) or words (`by = "word"`).
`overlap`	Overlap between consecutive chunks, in the same unit as `size`. Must be smaller than `size`.
`by`	`"char"` (default) or `"word"`.

Value

A tibble with columns `doc_id`, `chunk_id` (within document), `chunk` (global index), `text`, `n_chars`, plus any carried metadata.

Examples

```
cr_chunk(c("um texto longo ...", "outro documento ..."),
         size = 10, overlap = 2, by = "word")
```

`cr_close` *Release a crawler's resources*

Description

Closes the headless browser session (if any) and the DuckDB connection (if the dataset uses the duckdb backend). Collect results with `cr_collect()` before closing a duckdb-backed crawl.

Usage

```
cr_close(crawler)
```

Arguments

`crawler`	A Crawler.

Value

The crawler, invisibly.

`cr_collect` *Collect crawl results*

Description

Collect crawl results

Usage

```
cr_collect(crawler)
```

Arguments

`crawler`	A Crawler that has been run.

Value

A tibble of all records pushed by handlers.

cr_dataset *Configure the dataset backend*

Description

Configure the dataset backend

Usage

```
cr_dataset(crawler, backend = "memory", path = NULL, table = "dataset")
```

Arguments

crawler	A Crawler.
backend	One of "memory" (default), "jsonl", "duckdb".
path	File (jsonl) or database (duckdb) path; required for the persistent backends.
table	Table name for the "duckdb" backend.

Value

The crawler, invisibly.

cr_embed *Attach embeddings to chunks*

Description

Adds an embedding list-column by applying a user-supplied, provider- agnostic embedding function in batches. crawlee never calls an external service itself: you pass embed_fn, which receives a character vector and returns either a numeric matrix (one row per input) or a list of numeric vectors. This keeps you free to use any provider or a local model.

Usage

```
cr_embed(data, embed_fn, text_col = "text", batch_size = 32L)
```

Arguments

data	A data frame with a text column (e.g. from cr_chunk()).
embed_fn	A function mapping a character vector to a numeric matrix (rows = inputs) or a list of numeric vectors.
text_col	Name of the text column. Defaults to "text".
batch_size	Number of texts per call to embed_fn.

Value

`data` with an added `embedding` list-column.

Examples

```
chunks <- cr_chunk(c("a b c d", "e f g h"), size = 2, overlap = 0, by = "word")
fake_embed <- function(x) matrix(nchar(x), nrow = length(x), ncol = 1)
cr_embed(chunks, fake_embed)
```

---

cr_export *Export chunks (and embeddings) for retrieval*

Description

Writes a chunk table to a retrieval-friendly format. `parquet` and `jsonl` preserve the `embedding` list-column natively; `csv` and `duckdb` serialise it to a `[...]` string for portability.

Usage

```
cr_export(
  data,
  path,
  format = c("parquet", "jsonl", "csv", "duckdb"),
  table = "chunks"
)
```

Arguments

`data`	A data frame (typically from `cr_chunk()` / `cr_embed()`).
`path`	Output file (or database) path.
`format`	One of `"parquet"`, `"jsonl"`, `"csv"`, `"duckdb"`.
`table`	Table name for the `"duckdb"` format.

Value

`path`, invisibly.

`cr_from_rss` *Discover URLs from an RSS or Atom feed*

Description

Fetches a feed and enqueues each item's link. The item title and date are attached to the request's `user_data` (available to handlers as `ctx$request$user_data`), so feed metadata can be carried into the dataset.

Usage

```
cr_from_rss(
  crawler,
  url,
  label = NULL,
  include = NULL,
  exclude = NULL,
  max = Inf
)
```

Arguments

`crawler`	A Crawler.
`url`	URL of an RSS or Atom feed.
`label`	Optional handler label routing the enqueued URLs.
`include, exclude`	Optional glob patterns (see `cr_on_html()`).
`max`	Maximum number of items to enqueue.

Value

The crawler, invisibly.

Examples

```
## Not run:
crawler() |>
  cr_on_html(\(ctx) ctx$push_data(list(
    url = ctx$request$url, title = ctx$request$user_data$title
  ))) |>
  cr_from_rss("https://example.com/feed.xml")

## End(Not run)
```

`cr_from_sitemap` *Discover URLs from a sitemap*

Description

Fetches a sitemap (or sitemap index, recursively) and enqueues the page URLs it lists. Supports gzipped sitemaps, glob filtering and a `since` filter on `<lastmod>` for incremental re-crawls of large sites that publish dated sitemaps.

Usage

```
cr_from_sitemap(
  crawler,
  url,
  label = NULL,
  include = NULL,
  exclude = NULL,
  since = NULL,
  max = Inf,
  max_levels = 3L
)
```

Arguments

`crawler`	A Crawler.
`url`	URL of a `sitemap.xml` or sitemap index.
`label`	Optional handler label routing the enqueued URLs.
`include, exclude`	Optional glob patterns (see `cr_on_html()`).
`since`	Optional date (or `YYYY-MM-DD` string); only URLs whose `<lastmod>` is on or after this date are enqueued (URLs without a `lastmod` are kept).
`max`	Maximum number of URLs to enqueue.
`max_levels`	Maximum recursion depth into nested sitemap indexes.

Value

The crawler, invisibly.

Examples

```
## Not run:
crawler() |>
  cr_on_html(\(ctx) ctx$push_data(list(url = ctx$request$url))) |>
  cr_from_sitemap("https://example.com/sitemap.xml", since = "2026-01-01")

## End(Not run)
```

cr_normalize_url *Normalise a URL into a canonical form*

Description

Produces a canonical representation of a URL used as the deduplication key (`unique_key`) of a request. Normalisation lower-cases the scheme and host, removes a trailing slash from the path, drops default ports and sorts the query parameters so that semantically identical URLs collapse to the same key.

Usage

```
cr_normalize_url(url)
```

Arguments

`url`	A character vector of URLs.

Value

A character vector of normalised URLs.

Examples

```
cr_normalize_url("HTTPS://Example.com:443/a/?b=2&a=1")
```

cr_on_html *Register an HTML handler*

Description

Registers a function called for each successfully fetched page whose request carries the given `label` (or for all pages when `label = NULL`). The handler receives a context object exposing the parsed page and the actions `push_data()` and `enqueue_links()`.

Usage

```
cr_on_html(crawler, handler, label = NULL)
```

Arguments

`crawler`	A Crawler.
`handler`	A function of one argument (the context). See **Context**.
`label`	Optional handler label. Requests enqueued with the same label are routed here; `NULL` registers the default handler.

Value

The crawler, invisibly.

Context

The `ctx` passed to a handler contains:

`request` The request list (`url`, `label`, `depth`, ...).

`response` The `httr2` response object.

`page` The parsed page (an `xml_document`) or `NULL`.

`push_data(data)` Append a record to the dataset.

`enqueue_links(...)` Discover and enqueue links from the page.

`log` Logging functions (`info`, `success`, `warn`, `error`).

Examples

```
crawler("https://example.com") |>
  cr_on_html(function(ctx) {
    ctx$push_data(list(url = ctx$request$url))
    ctx$enqueue_links()
  })
```

---

cr_on_pdf *Register a PDF handler*

Description

Registers a handler invoked for responses classified as PDF — by `Content-Type` (`application/pdf`) or a `.pdf` URL. The handler context adds PDF-specific helpers on top of the usual ones.

Usage

```
cr_on_pdf(crawler, handler, label = NULL)
```

Arguments

`crawler`	A Crawler.
`handler`	A function of one argument (the context). See **Context**.
`label`	Optional handler label; `NULL` registers the default PDF handler.

Details

Requests carrying an explicit `label` are always routed to the handler registered for that label (regardless of content kind); `label = NULL` registers the default PDF handler.

Value

The crawler, invisibly.

Context

In addition to the elements documented in `cr_on_html()`, a PDF handler's context provides:

`kind` `"pdf"`.

`pdf_text()` Extract text per page (requires the **pdftools** package), returning a character vector.

`body_raw()` The raw PDF bytes.

`save_body(key, ext)` Persist the PDF to the KeyValueStore.

Examples

```
## Not run:
crawler("https://example.com/report.pdf") |>
  cr_on_pdf(function(ctx) {
    text <- ctx$pdf_text()
    ctx$push_data(list(url = ctx$request$url, n_pages = length(text)))
    ctx$save_body(ext = "pdf")
  }) |>
  cr_run()

## End(Not run)
```

`cr_options` *Set crawler options*

Description

Set crawler options

Usage

```
cr_options(crawler, ...)
```

Arguments

`crawler`	A Crawler.
`...`	Named options to override. Recognised options: `concurrency`, `max_requests`, `max_depth`, `delay` (seconds between requests), `timeout`, `max_retries`, `user_agent`, `respect_robots`, `same_domain`, `log_level` (`"debug"`, `"info"`, `"warn"`, `"error"`, `"off"`).

Value

The crawler, invisibly.

Examples

```
crawler("https://example.com") |> cr_options(delay = 0.5, max_depth = 2)
```

`cr_parallel` *Enable parallel (concurrent) fetching*

Description

Switches the HTTP engine to fetch requests in concurrent batches via `httr2::req_perform_parallel()`, the rough equivalent of Crawlee's autoscaled pool. Network I/O runs concurrently while handlers still run sequentially in R, so there is no shared-state hazard. `robots.txt`, retries, `max_requests`/`max_depth` and queue checkpointing all still apply.

Usage

```
cr_parallel(crawler, concurrency = 4L, max_active = NULL)
```

Arguments

`crawler`	A Crawler.
`concurrency`	Number of requests per batch.
`max_active`	Maximum simultaneously-active connections (defaults to `concurrency`).

Details

Parallel mode applies to the HTTP backend only; the browser backend always runs sequentially. `delay` and `Crawl-delay` are applied *between* batches.

Value

The crawler, invisibly.

Examples

```
crawler("https://example.com") |> cr_parallel(concurrency = 8)
```

`cr_persist` *Persist a crawl to a run directory (and resume it)*

Description

Wires a crawler to a directory on disk so a crawl is **reproducible and resumable**. It persists:

Usage

```
cr_persist(crawler, dir, dataset = c("jsonl", "duckdb", "memory"))
```

Arguments

`crawler`	A Crawler.
`dir`	Run directory (created if needed).
`dataset`	Dataset backend to use: `"jsonl"` (default), `"duckdb"` or `"memory"` (not persisted).

Details

- the request queue state (`queue.rds`) — pending requests, seen keys and handled count, checkpointed during `cr_run()`;
- the dataset, via a persistent Dataset backend (`dataset.jsonl` or `dataset.duckdb`);
- binary content saved by `ctx$save_body()` (under `kv/`);
- a run manifest (`manifest.rds`, plus `manifest.json` when **jsonlite** is available).

If a queue state already exists in `dir`, the crawl **resumes**: the saved pending/seen/handled state is restored, so `cr_run()` continues where it left off and already-fetched URLs are not fetched again.

Call `cr_persist()` before `cr_run()`. For the `"duckdb"` backend, collect results with `cr_collect()` before `cr_close()`.

Value

The crawler, invisibly.

Examples

```
## Not run:
crawler("https://example.com") |>
  cr_persist("runs/example", dataset = "duckdb") |>
  cr_on_html(\(ctx) ctx$push_data(list(url = ctx$request$url))) |>
  cr_run() |>
  cr_collect()
# Re-running the same pipeline resumes from runs/example.

## End(Not run)
```

cr_run *Run a crawl*

Description

Drains the request queue, fetching each request, dispatching it to the matching handler and collecting pushed records, until the queue is empty or the `max_requests` limit is reached.

Usage

```
cr_run(crawler)
```

Arguments

`crawler`	A configured Crawler.

Value

The crawler, invisibly (its dataset now holds the results).

Examples

```
## Not run:
crawler("https://example.com") |>
  cr_on_html(\(ctx) ctx$push_data(list(url = ctx$request$url))) |>
  cr_run() |>
  cr_collect()

## End(Not run)
```

cr_store *Configure the key-value store for binary content*

Description

Sets the directory used by `ctx$save_body()` to persist raw responses (PDFs, images, snapshots).

Usage

```
cr_store(crawler, path)
```

Arguments

`crawler`	A Crawler.
`path`	Target directory. Created if it does not exist.

Value

The crawler, invisibly.

`cr_stream` *Enable the streaming scheduler*

Description

A continuous-pool alternative to `cr_parallel()`'s synchronous batches. The streaming engine keeps requests in flight at all times (via async promises, `httr2::req_perform_promise()`): the moment one request finishes, its handler runs and the next request is pulled from the queue to refill the slot. Under heterogeneous response latency this avoids the "wait for the slowest in the batch" stall and improves throughput.

Usage

```
cr_stream(crawler, concurrency = 8L, adaptive = FALSE, min = 1L, max = NULL)
```

Arguments

`crawler`	A Crawler.
`concurrency`	Number of requests to keep in flight (the fixed target, or the starting point is `min` when `adaptive = TRUE`).
`adaptive`	If `TRUE`, adapt the in-flight target within `[min, max]`.
`min, max`	Bounds for the adaptive target. `max` defaults to `concurrency`.

Details

With `adaptive = TRUE` the in-flight target adapts at run time (AIMD on back-pressure, like `cr_autoscale()`), within `[min, max]`.

Launches are paced **per host**: a host is not hit again until `delay` / `robots.txt` `Crawl-delay` has elapsed, while different hosts run in parallel. With `delay = 0` and no `Crawl-delay`, pacing is a no-op.

Requires the **promises** and **later** packages, and the HTTP backend.

Value

The crawler, invisibly.

Examples

```
crawler("https://example.com") |> cr_stream(concurrency = 10)
crawler("https://example.com") |> cr_stream(adaptive = TRUE, min = 2, max = 16)
```

`cr_use_browser` *Use the headless-browser fetch backend*

Description

Switches the crawler to render pages with a headless Chrome/Chromium via the **chromote** package — for JavaScript-heavy sites where the plain HTTP backend would see an empty shell. Handlers work exactly as with `cr_use_http()` (`ctx$page`, `enqueue_links()`, ...), and additionally gain `ctx$screenshot()`.

Usage

```
cr_use_browser(crawler, wait = 0, wait_selector = NULL)
```

Arguments

`crawler`	A Crawler.
`wait`	Seconds to wait after page load before capturing the DOM (useful for late-rendering content).
`wait_selector`	Optional CSS selector to wait for before capturing.

Details

Requires **chromote** and a Chrome/Chromium installation. PDF extraction still requires the HTTP backend.

Value

The crawler, invisibly.

Examples

```
## Not run:
crawler("https://example.com") |>
  cr_use_browser(wait_selector = ".results") |>
  cr_on_html(\(ctx) {
    ctx$push_data(list(url = ctx$request$url))
    ctx$screenshot()
  }) |>
  cr_run()

## End(Not run)
```

cr_use_http *Use the HTTP fetch backend*

Description

Selects the plain HTTP backend (powered by `httr2`), suitable for static HTML, XML, RSS and document endpoints. This is the default mode.

Usage

```
cr_use_http(crawler)
```

Arguments

`crawler`	A Crawler.

Value

The crawler, invisibly.

crawler *Create a crawler*

Description

Constructs a new Crawler seeded with `start_urls`. The result is designed to be piped through the `cr_*` configuration verbs and finally `cr_run()`.

Usage

```
crawler(start_urls = character(), ...)
```

Arguments

`start_urls`	Character vector of seed URLs to enqueue at depth 0.
`...`	Options forwarded to `cr_options()` (e.g. `max_requests`, `delay`, `log_level`).

Value

A Crawler object.

Examples

```
cr <- crawler("https://example.com", max_requests = 10)
cr
```

`Crawler-class` *Crawler*

Description

The stateful object at the center of crawlee. It holds the request queue, the dataset, the registered handlers and the run configuration. You will rarely create one with `Crawler$new()` directly; use `crawler()` and the `cr_*` verbs, which return the crawler invisibly so they compose with the native pipe (`|>`).

Public fields

`options` Named list of run options.

`queue` The RequestQueue.

`dataset` The Dataset.

`handlers` Named list of label-specific handlers.

`defaults` Named list of default handlers by content kind (`html`, `pdf`, `any`).

`kv` Lazily-created KeyValueStore for binary content.

`mode` Fetch mode, `"http"` (default) or `"browser"`.

`stats` Named list of run statistics.

Methods

Public methods:

- `Crawler$new()`
- `Crawler$set_options()`
- `Crawler$set_handler()`
- `Crawler$get_kv()`
- `Crawler$set_persist_dir()`
- `Crawler$close()`
- `Crawler$run()`
- `Crawler$clone()`

`Crawler$new()`: Create a crawler.

Usage:

```
Crawler$new(start_urls = character(), ...)
```

Arguments:

`start_urls` Character vector of seed URLs.

`...` Options forwarded to `cr_options()`.

`Crawler$set_options()`: Update one or more options.

Usage:

```
Crawler$set_options(...)
```

Arguments:

`...` Named options to override.

`Crawler$set_handler()`: Register a handler for a content label or kind.

Usage:

```
Crawler$set_handler(handler, label = NULL, kind = "html")
```

Arguments:

`handler` A function of one argument, the handler context.

`label` Optional label; `NULL` registers a default handler.

`kind` Content kind for the default handler (`"html"`, `"pdf"`, `"any"`). Ignored when `label` is given.

`Crawler$get_kv()`: Get (lazily creating) the key-value store for binaries.

Usage:

```
Crawler$get_kv()
```

Returns: A KeyValueStore.

`Crawler$set_persist_dir()`: Set the run directory where the manifest is written.

Usage:

```
Crawler$set_persist_dir(dir)
```

Arguments:

`dir` A directory path.

`Crawler$close()`: Release resources (browser session, DuckDB connection).

Usage:

```
Crawler$close()
```

`Crawler$run()`: Run the crawl until the queue drains or a limit is hit.

Usage:

```
Crawler$run()
```

`Crawler$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Dataset *Dataset*

Description

An append-only structured store for the records produced by handlers via `ctx$push_data()`. Three backends are available:

- `"memory"` (default): records accumulate in memory.
- `"jsonl"`: each record is appended as a line of JSON to a file — schema-flexible, append-only and resumable across runs.
- `"duckdb"`: records are appended to a table in a DuckDB database, ready for SQL analysis.

Collect everything as a single tibble with `cr_collect()`.

Public fields

`backend` Name of the storage backend.

`path` Path for persistent backends.

Methods

Public methods:

- `Dataset$new()`
- `Dataset$push()`
- `Dataset$collect()`
- `Dataset$count()`
- `Dataset$close()`
- `Dataset$clone()`

`Dataset$new()`: Create a dataset.

Usage:

```
Dataset$new(backend = "memory", path = NULL, table = "dataset")
```

Arguments:

`backend` One of `"memory"`, `"jsonl"`, `"duckdb"`.

`path` File (jsonl) or database (duckdb) path; required for the persistent backends.

`table` Table name for the `"duckdb"` backend.

`Dataset$push()`: Append one or more records.

Usage:

```
Dataset$push(data)
```

Arguments:

`data` A data frame / tibble or a named list (coerced to one row).

`Dataset$collect()`: Collect all records as a single tibble.

Usage:

```
Dataset$collect()
```

Returns: A tibble (empty if nothing was stored).

`Dataset$count()`: Number of records (rows) stored.

Usage:

```
Dataset$count()
```

Returns: Integer scalar.

`Dataset$close()`: Close any open backend resources (e.g. the DuckDB connection). Safe to call multiple times.

Usage:

```
Dataset$close()
```

`Dataset$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`KeyValueStore` *Key-value store*

Description

A simple on-disk store for binary or text content keyed by an arbitrary string (typically a URL). It backs `ctx$save_body()`, letting handlers persist raw responses — PDFs, images, snapshots — alongside the structured Dataset. Keys are sanitised into safe file names.

Public fields

`dir` Directory backing the store.

Methods

Public methods:

- `KeyValueStore$new()`
- `KeyValueStore$set_raw()`
- `KeyValueStore$set_text()`
- `KeyValueStore$get_raw()`
- `KeyValueStore$path_of()`
- `KeyValueStore$keys()`

- KeyValueStore$clone()

`KeyValueStore$new()`: Create a store.

Usage:

```
KeyValueStore$new(dir = NULL)
```

Arguments:

`dir` Target directory; defaults to a `crawlee-store` folder in the session's temporary directory. Created if it does not exist.

`KeyValueStore$set_raw()`: Store raw bytes under `key`.

Usage:

```
KeyValueStore$set_raw(key, raw)
```

Arguments:

`key` Character key.

`raw` A raw vector.

Returns: The file path, invisibly.

`KeyValueStore$set_text()`: Store text under `key`.

Usage:

```
KeyValueStore$set_text(key, text)
```

Arguments:

`key` Character key.

`text` A character vector (written one element per line).

Returns: The file path, invisibly.

`KeyValueStore$get_raw()`: Retrieve raw bytes for `key`, or `NULL` if absent.

Usage:

```
KeyValueStore$get_raw(key)
```

Arguments:

`key` Character key.

`KeyValueStore$path_of()`: Full path for `key` (whether or not it exists).

Usage:

```
KeyValueStore$path_of(key)
```

Arguments:

`key` Character key.

`KeyValueStore$keys()`: List stored keys (file names).

Usage:

```
KeyValueStore$keys()
```

`KeyValueStore$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

RequestQueue *Request queue*

Description

A deduplicating, FIFO-with-priority request queue, the in-memory engine behind every `crawler()`. Requests are keyed by a normalised `unique_key` (see `cr_normalize_url()`) so the same URL is never enqueued twice. The queue tracks which requests have been handled, which makes a crawl resumable: when given a `path`, its state (pending requests, seen keys, handled count) can be saved to and restored from disk — see `cr_persist()`.

This class is exported mainly for advanced use and introspection; most users interact with it indirectly through the `cr_*` verbs.

Methods

Public methods:

- `RequestQueue$new()`
- `RequestQueue$add()`
- `RequestQueue$pop()`
- `RequestQueue$reschedule()`
- `RequestQueue$mark_handled()`
- `RequestQueue$pending_count()`
- `RequestQueue$handled()`
- `RequestQueue$is_empty()`
- `RequestQueue$set_path()`
- `RequestQueue$has_saved_state()`
- `RequestQueue$save()`
- `RequestQueue$restore()`
- `RequestQueue$clone()`

`RequestQueue$new()`**:** Create a new, empty request queue.

Usage:

```
RequestQueue$new(path = NULL)
```

Arguments:

`path` Optional path to an `.rds` file backing the queue state.

`RequestQueue$add()`**:** Add a request to the queue.

Usage:

```
RequestQueue$add(
  url,
  label = NULL,
  depth = 0L,
  user_data = list(),
```

```
  method = "GET",
  force_unique = FALSE
)
```

Arguments:

`url` Character scalar URL.

`label` Optional handler label routing this request.

`depth` Integer crawl depth (distance from a start URL).

`user_data` Optional named list carried with the request.

`method` HTTP method, defaults to `"GET"`.

`force_unique` If `TRUE`, skip deduplication.

Returns: Invisibly, `TRUE` if added, `FALSE` if a duplicate.

`RequestQueue$pop()`: Pop the next request from the front of the queue.

Usage:

```
RequestQueue$pop()
```

Returns: A request list, or `NULL` when the queue is empty.

`RequestQueue$reschedule()`: Re-queue a request for another attempt, incrementing its retry counter.

Usage:

```
RequestQueue$reschedule(request)
```

Arguments:

`request` A request list previously obtained from `pop()`.

`RequestQueue$mark_handled()`: Mark a request as successfully handled.

Usage:

```
RequestQueue$mark_handled()
```

`RequestQueue$pending_count()`: Number of requests waiting to be processed.

Usage:

```
RequestQueue$pending_count()
```

Returns: Integer scalar.

`RequestQueue$handled()`: Number of requests handled so far.

Usage:

```
RequestQueue$handled()
```

Returns: Integer scalar.

`RequestQueue$is_empty()`: Whether the queue has no pending requests.

Usage:

```
RequestQueue$is_empty()
```

Returns: Logical scalar.

`RequestQueue$set_path()`: Set (or clear) the persistence path.

Usage:

```
RequestQueue$set_path(path)
```

Arguments:

`path` Path to an `.rds` file, or `NULL`.

`RequestQueue$has_saved_state()`: Whether a persisted state file exists at the queue's path.

Usage:

```
RequestQueue$has_saved_state()
```

Returns: Logical scalar.

`RequestQueue$save()`: Persist the queue state to its `path` (a no-op without one).

Usage:

```
RequestQueue$save()
```

`RequestQueue$restore()`: Replace the in-memory state with the one persisted at `path`.

Usage:

```
RequestQueue$restore()
```

`RequestQueue$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Index