

Package: EpiSODIC (via r-universe)

September 30, 2026

Type Package

Title Epidemiological Signal Observation, Detection, Identification,
and Classification

Version 0.24.0

Description An automated surveillance system for infectious disease epidemiologists that detects aberrations in laboratory-confirmed infections, records epidemiologist's formal assessment with a full audit trail, and produces outbreak reports for clinical colleagues and infection prevention nurses. Applies evidence-based statistical methods such as Improved Farrington (2013) <[doi:10.1002/sim.5595](https://doi.org/10.1002/sim.5595)> and 'EpiEstim' (Cori et al. 2013) <[doi:10.1093/aje/kwt133](https://doi.org/10.1093/aje/kwt133)>. Ships with a synthetic demo data set requiring no external data or infrastructure.

License GPL-2

Encoding UTF-8

Roxygen list(markdown = TRUE)

Depends R (>= 4.1.0)

Imports AMR (>= 3.0.0), DBI, digest, ggplot2 (>= 3.5.0), mem, rlang, RSQLite, shiny (>= 1.14.0), sodium, surveillance, yaml

Suggests AzureAuth, AzureGraph, bslib, cli, commonmark, curl, EpiEstim, htmltools, htr2, jsonlite, knitr, Microsoft365R, pkgdown, quarto, RMariaDB, rmarkdown, sf, testthat (>= 3.0.0), withr, zip

Config/testthat/edition 3

VignetteBuilder knitr

URL <https://certe-medical-epidemiology.github.io/EpiSODIC>,
<https://github.com/certe-medical-epidemiology/EpiSODIC>

BugReports <https://github.com/certe-medical-epidemiology/EpiSODIC/issues>

Config/roxygen2/version 8.1.0

Config/pak/sysreqs cmake make libicu-dev libsodium-dev libuv1-dev zlib1g-dev

Repository <https://certe-medical-epidemiology.r-universe.dev>

Date/Publication 2026-09-30 10:40:58 UTC

RemoteUrl <https://github.com/certe-medical-epidemiology/EpiSODIC>

RemoteRef HEAD

RemoteSha 13d45390ccaf2352fea7e649d4577db13b54ce34

Contents

episodic_add_manual_cluster	3
episodic_add_user	5
episodic_auth	7
episodic_case_data	7
episodic_charts	11
episodic_check_cases	12
episodic_check_denominators	14
episodic_check_institution_activity	15
episodic_config_export	16
episodic_db_connect	17
episodic_db_create	18
episodic_db_dsn_mariadb	18
episodic_db_migrate	19
episodic_db_open	21
episodic_db_truncate	22
episodic_demo	23
episodic_geo	26
episodic_mail_template_preview	27
episodic_notifications	28
episodic_notify_test	29
episodic_palette	29
episodic_report_render	31
episodic_run_app	33
episodic_run_cron	34
episodic_scheduled_reports	37
episodic_setup_microsoft365	38
episodic_synthetic_cases	39
episodic_synthetic_cases_calibration	41
episodic_synthetic_denominators	42
episodic_synthetic_ground_truth	43
episodic_synthetic_institution_activity	44
episodic_ui_italicise_taxon	46
episodic_validate_comparator	47
episodic_validate_detection	49
episodic_validate_rethreshold	52

Index

54

episodic_add_manual_cluster

Add One or More Manual (External) Clusters

Description

Adds clusters that were never detected by EpiSODIC's own pipeline - output from another algorithm or system, fed in as already-parsed R values (an operator's own code is responsible for reading whatever JSON/CSV that other system produces). A manual cluster is never connected to this instance's own case data: it gets a real `episodic_stream` identity (so it appears in the dashboard's normal filtering, sorting and geography panels, exactly like a detected cluster), but its case-level detail, if you supply any, is stored separately in `episodic_cluster_manual_case` and never touches `episodic_case`. It is therefore also never picked up by `episodic_run_cron()`'s reconciliation or automatic closure - a manual cluster only closes when an epidemiologist records that verdict.

Usage

```
episodic_add_manual_cluster(
  db_path = Sys.getenv("EPISODIC_DB", unset = NA),
  user_id,
  pathogen,
  level,
  first_day,
  last_day,
  n_cases = NA,
  care_line = NA,
  region_code = NA,
  institution_id = NA,
  ward = NA,
  expected = NA,
  excess = NA,
  ratio = NA,
  detector_agreement = 1L,
  priority_score = NA,
  case_dates = NULL,
  pc = NULL,
  sex = NULL,
  age = NULL,
  note = NA
)
```

Arguments

<code>db_path</code>	Path to an existing SQLite database, or a <code>mysql://DSN</code> (see <code>episodic_db_dsn_mariadb()</code>). Defaults to the <code>EPISODIC_DB</code> environment variable.
<code>user_id</code>	The <code>episodic_app_user</code> id this batch is attributed to

	required, since provenance ("who added this, and when") is exactly what a hand-added cluster most needs. Recorded as the author of the seed note when note is supplied.
pathogen	Raw pathogen string, exactly as it would appear in <code>episodic_case\$pathogen</code> - used verbatim, not matched against any fixed list.
level	One of the five lattice levels: "pathogen_ward", "pathogen_institution", "pathogen_area", "pathogen_province", "pathogen_region" (see the "The lattice: streams, not one big feed" section of Detection and reconciliation).
first_day, last_day	The cluster's episode window (Date or "YYYY-MM-DD" text).
n_cases	The case count. Optional if <code>case_dates</code> (or any other per-cluster case-level detail) is supplied for that cluster, in which case it defaults to that detail's own length; required otherwise, since there is then nothing else to derive it from.
care_line, region_code, institution_id, ward	The remaining <code>episodic_stream</code> identity fields, exactly as <code>episodic_stream_key()</code> expects them. <code>institution_id</code> must already exist in <code>episodic_institution</code> (e.g. resolved earlier via <code>episodic_institutions_resolve()</code>) - this function does not create institutions.
expected, excess, ratio	Optional pass-through metrics, for external algorithms that do produce a baseline comparison. NA (the default) when the source has no such concept - exactly as for a detected cluster whose detector cannot produce one.
detector_agreement	How many independent sources agree on this cluster. Defaults to 1L (a single external source). Scored on the same scale as a detected cluster's, i.e. out of the four built-in detectors, so 2L here carries the weight of two detectors agreeing and the two kinds of cluster rank against each other honestly.
priority_score	Optional override. When NA (the default), it is computed with the same <code>episodic_priority_score()</code> used for detected clusters, so manual and detected clusters sort comparably - from <code>expected/excess/ratio</code> , <code>detector_agreement</code> , and, when per-cluster case-level detail is supplied, growth slope and spatial concentration computed from it exactly as the cron pipeline would.
case_dates, pc, sex, age	Optional per-cluster case-level detail, each a <code>list()</code> of length <code>n</code> (one vector per cluster, all of equal length within a cluster) - <code>case_dates</code> a vector of sample dates, <code>pc</code> postcodes, <code>sex</code> "M"/"F"/"U"/NA, <code>age</code> in years. Feeds the dossier's epi curve, demography and geography panels for this cluster. Leave NULL (the default) for a source that only supplies an aggregate count - the cluster is still created, its panels just have nothing to draw.
note	Optional markdown text to seed the cluster's note with (see the dossier's notes panel), attributed to <code>user_id</code> .

Details

Every argument is vectorised: pass a single value to add one cluster, or vectors/lists of length `n` to add `n` clusters in one call, each getting its own new cluster id. Arguments of length 1 are recycled

against the longest argument you supply; per-cluster case-level detail (case_dates/pc/sex/age) is never recycled - if you supply it, it must be a `list()` of exactly the target length, one element (a vector) per cluster.

The whole call is one transaction: either every requested cluster is created, or (on any validation or database error) none are.

Value

An integer vector of the newly created `cluster_id` values, one per cluster, in the same order as the arguments.

Examples

```
## Not run:
episodic_add_manual_cluster(
  user_id = 1L,
  pathogen = "Measles virus",
  level = "pathogen_area",
  first_day = "2025-01-10",
  last_day = "2025-01-20",
  region_code = "GR",
  case_dates = list(as.Date(c("2025-01-10", "2025-01-14", "2025-01-20"))),
  pc = list(c("9711AA", "9711AB", "9712CD")),
  note = "Reported by municipal health service, outside our own lab data."
)

## End(Not run)
```

episodic_add_user

Create an Account for a New Epidemiologist or Viewer

Description

There is no self-service registration: an account is added either from the Settings screen (by an `is_admin` account) or with this function at the R console. The password you supply is temporary - the new account is flagged to require a password change, so the account holder chooses their own password the first time they sign in.

Usage

```
episodic_add_user(
  db_path = Sys.getenv("EPISODIC_DB", unset = NA),
  username,
  full_name,
  email,
  password,
  role = "epidemiologist",
  is_admin = FALSE,
  must_change = TRUE
)
```

Arguments

<code>db_path</code>	Path to the EpiSODIC database: an existing SQLite file, or a MariaDB/MySQL DSN (see episodic_db_dsn_mariadb()). Defaults to the EPISODIC_DB environment variable.
<code>username, full_name, email</code>	The new account's sign-in name, display name, and email address.
<code>password</code>	A temporary plaintext password, never stored or logged as-is: it is hashed before it reaches the database, and the account holder is required to replace it on first sign-in.
<code>role</code>	Either "epidemiologist" (can classify and close clusters, in addition to everything a viewer can do) or "viewer" (read-only - can see everything a signed-in epidemiologist sees, including patient-level line lists, but cannot record an assessment). Both roles require sign-in; there is no anonymous access to patient detail.
<code>is_admin</code>	Whether the new account may also see the Settings screen (manage notification channels, other accounts, and export the configuration). Independent of role - an admin is still either an epidemiologist or a viewer for everything outside Settings.
<code>must_change</code>	Whether the account holder is required to replace password the first time they sign in. TRUE (the default) is right for every real account: the password you type here has been in your shell history and quite possibly in a message to the person it belongs to. FALSE is for a throwaway account in a throwaway database, which is what episodic_demo() creates.

Details

At the console, you only need to run this once per person. It opens the database, adds the account, and closes the connection again, so it is meant to be run interactively rather than from application code.

Value

Invisibly, the new account's `user_id`.

Examples

```
db_path <- tempfile(fileext = ".sqlite")
con <- episodic_db_create(db_path)
DBI::dbDisconnect(con)

user_id <- episodic_add_user(
  db_path,
  username = "jdoe", full_name = "Jane Doe",
  email = "jane@example.org", password = "temporary-password"
)
user_id

file.remove(db_path)
```

episodic_auth*How sign-in works*

Description

EpiSODIC's dashboard is used by a small board of epidemiologists (the "epidemiologist" role - by definition, they assess clusters and classify them) and by "viewer" accounts for anyone who needs to see cluster detail, including patient-level data, without classifying anything themselves. Both roles sign in with a username and password; there is no self-service registration - an administrator creates each account with `episodic_add_user()`, and the new user sets their own password on first sign-in.

Details

Passwords are hashed (never stored in plain text). Both outcomes are recorded: a successful sign-in as a login event on the account, a refused one - with which of unknown username, wrong password or deactivated account it was, and the username as typed - in `episodic_app_login_failure`. Both appear on the dashboard's Activity screen, to signed-in readers only, where the sign-in category can be filtered to on its own.

That record is an audit trail, not a defence. This login exists to attribute who assessed what, not to keep an attacker out: EpiSODIC does not implement TLS, account lockout, or rate limiting, so it should always be deployed behind your own organisation's network controls (VPN, internal network, or a reverse proxy that terminates TLS). What the failure log gives you is the ability to *notice* - a burst of refused attempts on one account is visible on the Activity screen the same day it happens.

episodic_case_data*Connect your own laboratory data*

Description

EpiSODIC does not connect to any laboratory or hospital system itself. Instead, you extract and transform your own positive-result data beforehand - with whatever tooling you already use - and hand the result to `episodic_run_cron()` as a plain data frame (a `tibble` is equally fine) in the shape described here. This keeps EpiSODIC decoupled from any specific laboratory information system: whatever your source system is, your own extract step is the only place that needs to know about it.

Details

A data set is the normal case, and the one to reach for. If producing the data only makes sense at run time - a live database query, for instance - you can pass a zero-argument function returning such a data set instead; EpiSODIC accepts either.

Only confirmed-positive results belong here - there is no outcome column, so do not include negative results. If you also want a denominator (tests performed, for a positivity rate), supply that separately as pre-aggregated counts; see [episodic_synthetic_denominators\(\)](#).

The only case data shipped with the package is what the synthetic generator ([episodic_synthetic_cases\(\)](#)) returns for the bundled demo - a useful template for the shape your own data should have.

Required columns

Column	Data type	NA allowed	Accepts (see below for more info)
source_key	character	no	any string, unique within the data set
lab_number	character	no	any string, your lab's own specimen/culture number
patient_key	character	no	any string, stable per patient
sample_date	Date or character	no	Format YYYY-MM-DD only
receipt_date	Date or character	yes	Format YYYY-MM-DD only
pathogen	character	no	free text, spelled the same every run
care_line	character	yes	one of: "first", "second", "third", "other", "unknown"
institution_key	character	no	any string, stable per institution
institution_display_name	character	no	any string
institution_type	character	no	one of: "hospital", "ltc_institution", "gp_municipality"
municipality	character	yes	any string
ward	character	yes	any string, spelled the same every run
specialism	character	yes	any string
pc	character	yes	postcode digits as text, or anything that connects to your system
sex	character	yes	one of: "M", "F", "U"
age	numeric	yes	whole years, not an age group

Every column is required to be *present*; "empty allowed" says whether its values may be NA. Dates may be a real Date column or ISO 8601 text, and nothing else: "01-01-2025" is rejected rather than read as the first of January in the year 1. Numbers stored as text, dates stored as date-times, and text stored as a factor are all common extract mistakes, and all named for what they are by the check below.

source_key Character, required, no NA, unique within the data set as a whole - across every pathogen and every patient, not just within one of them. A stable identifier for this *result*, so re-running the same extract later cannot create duplicate cases. Your laboratory's own specimen or culture number rarely satisfies this by itself: the same culture can yield more than one reported result (a multiplex panel positive for two organisms, two isolates from one culture with different antibiograms), and a bare specimen number then repeats across rows. Where that is true of your source data, build **source_key** yourself by combining whichever of your own columns are unique per *row* - many laboratory systems need all four of **lab_number**, **patient_key**, a test/panel code, and a strain or isolate number to pin down one row, e.g. `paste(lab_number, patient_key, test_code, isolate_number)` in your own extract step (only **lab_number** and **patient_key** reach EpiSODIC itself; a test code or isolate number that exists only in your source system is fine to fold into **source_key** and drop otherwise). **source_key** itself is never shown in the dashboard - it exists only so a re-run stays idempotent - which is exactly why it is safe to make it an artificial combination rather than a real-world identifier.

- lab_number** Character, required, no NA. Your laboratory's own identifier for the specimen or culture this result came from (an accession, specimen or culture number) - not pseudonymised, and not required to be unique: two rows legitimately share one lab_number when a single culture produced more than one reported result (see source_key). Shown alongside patient_key in the dashboard's case tables, since - unlike patient_key - it is not a patient identifier and carries no reason to hide it.
- patient_key** Character, required, no NA. A pseudonymised patient identifier, the same for the same patient across results. Deduplication and episode grouping key on it: without it every positive becomes its own case. Do pseudonymise it before it reaches EpiSODIC - a national identification number or hospital number must not be passed through - but the pseudonym itself is shown in the dashboard's case tables, same as lab_number.
- sample_date** Date, or character in YYYY-MM-DD (ISO 8601) form. Required, no NA. The date the sample was taken, and the anchor every detector, trend and report is built against. If your system falls back to a receipt date when the sample date is unfilled, apply that fallback in your own extract step, not here.
- receipt_date** Date, or character in YYYY-MM-DD form; NA allowed. The date the laboratory received the sample. Stored for provenance only - deliberately never used to measure reporting delay, since a lab's receipt-date field can itself be standing in for a missing sample date.
- pathogen** Character, required, no NA. The pathogen name exactly as your laboratory reports it (e.g. "Escherichia coli", "Influenza A", "Norovirus"). Free text, deliberately unconstrained: used verbatim and never matched against a taxonomy, since detection has to work for anything a lab can report, viral or not. Spelling must be stable across runs - "Influenza A" and "influenza a" are two different pathogens as far as detection is concerned. Names matching in the default config ([inst/config/episodic_default_pathogen_config.csv](#)) pick up that pathogen's episode length, incubation window and serial interval; anything else falls back to the schema defaults (30-day episode, 14 case-free days, no Rt, no MEM). The same positive may appear under more than one pathogen value where that is epidemiologically useful - an ETEC positive reported as both "Escherichia coli" and "ETEC", so each is monitored separately.
- care_line** Character; NA allowed. One of "first" (primary care), "second" (secondary care), "third" (tertiary care), "other", or "unknown". Anything else is rejected. NA is read as "unknown" and stored that way, so you need not map missing values yourself: an empty care_line, an R NA and a database NULL all mean the same thing here, and the dashboard shows all three as "unknown".
- institution_key** Character, required, no NA. A stable identifier for the reporting institution. Hashed (SHA-1) on the way in, so a later rename does not fracture the institution's history, and so the raw key never reaches the database.
- institution_display_name** Character, required, no NA. The human-readable institution name shown in the dashboard.
- institution_type** Character, required. Exactly one of "hospital", "ltc_institution" (long-term care), "gp_municipality", "ooh_service" (out-of-hours service), or "other". Anything else is rejected. This decides how the institution is handled: "hospital" institutions are monitored as first-class entities and are the only ones eligible for patient-day normalisation, while a "gp_municipality" is collapsed to its municipality.
- municipality** Character; NA allowed. The institution's municipality. Required in practice for "gp_municipality" rows, since that is what they are collapsed to.

- ward Character; NA allowed. The ward within the institution. Only meaningful for "hospital" and "ltc_institution" rows; leave NA otherwise. This is the unit L1 (ward-level) detection watches, so an inconsistent spelling splits one ward into two streams.
- specialism Character; NA allowed. The requesting clinical specialism. Context for the epidemiologist, not a detection input.
- pc Character; NA allowed. The *patient's* postcode area, not the institution's - it is what the geography panel and area-level (L3) detection use, in whatever form your country's postcodes take. Store it as character, not as a number, so a leading zero survives. With EPISODIC_GEO_DATA configured, it must match that file's pc column exactly for a case to reach the map.
- sex Character; NA allowed. Exactly one of "M", "F", or "U" (unknown) when present. Anything else is rejected, so map your own coding (1/2, "male"/"female") in your extract step.
- age Integer; NA allowed. Age in whole years at sampling. Not age group - the dashboard bands it itself. Use `AMR::age()` to determine the age based on a patient's date of birth and the date of specimen receipt at the lab.

Check your data before you run anything

Do not find out from an empty dashboard. Run `episodic_check_cases()` on your extract - it needs no database, changes nothing, and reports every problem it finds at once, with the rows and values involved and what to do about each:

```
cases <- my_extract_and_transform_function()
episodic_check_cases(cases)
```

It also reports what is merely worth a look: one pathogen spelled two ways (two streams instead of one), no ward on any hospital row (no ward-level detection), a `patient_key` that never repeats (no deduplication), postcodes the map cannot place, sample dates in the future. `episodic_check_cases(..., stop_on_problem = TRUE)` runs the same checks but throws an error, for use in a script. `episodic_run_cron()` runs that before every run, and refuses to start on data it cannot use, naming what to fix.

Examples

```
# A minimal but complete extract: two results, one patient, all sixteen
# columns present, and the ones your laboratory does not record left NA.
cases <- data.frame(
  source_key = c("LAB-1", "LAB-2"),
  lab_number = c("LAB-1", "LAB-2"),
  patient_key = c("PAT-1", "PAT-1"),
  sample_date = c("2025-01-06", "2025-01-09"),
  receipt_date = c("2025-01-07", "2025-01-10"),
  pathogen = "Norovirus",
  care_line = "second",
  institution_key = "HOSP-01",
  institution_display_name = "Example Hospital",
  institution_type = "hospital",
  municipality = "Groningen",
  ward = "Ward A2",
  specialism = "Internal medicine",
  pc = "9713",
```

```
sex = c("F", "F"),
age = c(71L, 71L),
stringsAsFactors = FALSE
)
episodic_check_cases(cases)
```

episodic_charts

Draw the Dashboard Charts (Epidemic Curve, Trend, and Rt)

Description

These functions build the three time-series charts used throughout the EpiSODIC dashboard and outbreak reports: an epidemic curve, a trend chart against the expected baseline, and an effective reproduction number (R_t) chart. Each takes the small, already-summarised data frame the app itself works with, so they are also useful for reproducing a dossier's chart in your own report or presentation. All three return a static [ggplot2::ggplot](#) object that can be printed, saved with [ggplot2::ggsave\(\)](#), or further customised with additional ggplot2 layers.

Usage

```
episodic_ui_epi_curve_chart(
  curve,
  lang = Sys.getenv("EPISODIC_LANGUAGE"),
  accent = NULL
)

episodic_ui_trend_chart(
  trend,
  lang = Sys.getenv("EPISODIC_LANGUAGE"),
  accent = NULL
)

episodic_ui_rt_chart(rt, lang = Sys.getenv("EPISODIC_LANGUAGE"), accent = NULL)
```

Arguments

curve	A data frame with one row per day: <code>sample_date</code> (Date), <code>n_cases</code> (case count), and <code>incomplete</code> (logical, TRUE for the most recent day(s) where reporting is still catching up - these are drawn at reduced opacity as a visual reminder not to over-interpret a downturn that is really just a reporting lag). Optionally <code>nowcast_low</code> , <code>nowcast_mid</code> and <code>nowcast_high</code> : the 5%, 50% and 95% quantiles of a day's expected final count once reporting is complete, drawn as a line with an open point at the median on the days that have them (NA elsewhere).
lang	Language for axis labels: "en", "ar", "nl", "fr", "de", "hi", "zh", or "es", or a regional variant of one ("en-US", "es-419"). Defaults to the EPISODIC_LANGUAGE environment variable, falling back to "en" if that is unset.

accent	Fill colour for the bars. Defaults to the palette's primary, which every chart in the dashboard uses for its main series; the status colours are kept for marks that mean something (thresholds, positivity).
trend	A data frame with one row per week: week_start (Date), n_cases (observed count), expected (the Farrington baseline), and upperbound (the alarm threshold, shown as a shaded band).
rt	A data frame with one row per estimation window: window_end (Date), mean (point estimate of R_t), and lower/upper (95% credible interval). A dashed reference line is drawn at $R_t = 1$, the threshold between a shrinking and a growing outbreak. The y axis is on a log2 scale, labelled in R_t itself, so that a halving and a doubling sit the same distance either side of that line.

Value

A `ggplot2::ggplot` object.

Examples

```
curve <- data.frame(
  sample_date = seq(as.Date("2025-01-01"), by = "day", length.out = 14),
  n_cases = c(1, 0, 2, 1, 3, 2, 4, 3, 5, 2, 1, 0, 1, 2),
  incomplete = c(rep(FALSE, 12), TRUE, TRUE)
)
episodic_ui_epi_curve_chart(curve, lang = "en")
trend <- data.frame(
  week_start = seq(as.Date("2025-01-06"), by = "week", length.out = 8),
  n_cases = c(2, 3, 1, 4, 6, 5, 3, 2),
  expected = c(2, 2, 2, 2, 2, 2, 2, 2),
  upperbound = c(4, 4, 4, 4, 4, 4, 4, 4)
)
episodic_ui_trend_chart(trend, lang = "en")
rt <- data.frame(
  window_end = seq(as.Date("2025-01-08"), by = "day", length.out = 5),
  mean = c(1.4, 1.3, 1.1, 0.9, 0.8),
  lower = c(1.0, 0.9, 0.8, 0.6, 0.5),
  upper = c(1.8, 1.7, 1.4, 1.2, 1.1)
)
episodic_ui_rt_chart(rt)
```

episodic_check_cases *Check Your Case Data Before You Hand It to EpiSODIC*

Description

Runs every check the [episodic_case_data](#) requirements imply over your extract and reports *everything* it finds in one go - what is wrong, how many rows are affected, which rows those are, what the offending values look like, and what to do about each one. Nothing is written, nothing is changed, and no database is needed: this is the call to make while you are still building your extract step, and the first call to make when a run did not produce what you expected.

Usage

```
episodic_check_cases(cases, stop_on_problem = FALSE)
```

Arguments

cases Your case data: a data frame or tibble in the shape [episodic_case_data](#) describes, or a zero-argument function returning one. Anything else is itself reported as a problem rather than throwing - the point of this function is that it always answers.

stop_on_problem A [logical](#) to indicate whether an error must be thrown if any problem is found. Default is FALSE.

Details

Use `episodic_check_cases()` plainly when you want to *look*, or `episodic_check_cases(..., stop_on_problem = TRUE)` when you want a script to stop. The latter is what [episodic_run_cron\(\)](#) itself calls before a run.

None of this applies to a cluster added with [episodic_add_manual_cluster\(\)](#): it is never connected to your own case data, so it never goes through these requirements (or any other check here) at all.

Value

A data frame of findings with class `episodic_case_check` and one row per finding, with columns `severity` ("problem" or "advice"), `issue`, `column`, `n_rows`, `rows`, `values`, `message` and `fix`. Zero rows means the data set passed every check. A summary of what was read (rows, columns, date range, counts) is attached as the "summary" attribute and shown when printing.

What it reports

Two kinds of finding, deliberately kept apart:

Problems Things EpiSODIC cannot work around: a missing or unexpected column, an empty value in a column that must be filled, a value outside a fixed set, a date that does not read as a date, a duplicated `source_key`. A run refuses to start while any of these stand.

Advice Things that are *allowed* but rarely intended, and that quietly cost you signal: no ward on any hospital row (so ward-level detection has nothing to group on), one pathogen spelled two ways (so its cases are split over two streams), a `patient_key` that is unique per row (so deduplication and episode grouping cannot do anything), postcodes that the geography panel cannot place, sample dates in the future. A run proceeds regardless; these are for you to judge.

The value prints as a report; it is also a plain data frame, one row per finding, so you can work with it programmatically (see the examples).

See Also

[episodic_case_data](#) for what each column means and which values it accepts.

Examples

```
cases <- episodic_synthetic_cases(
  start_date = as.Date("2025-01-01"), end_date = as.Date("2025-01-31")
)
episodic_check_cases(cases)

# a typical extract mistake: dates written day-first, sex as words
broken <- cases
broken$sample_date <- format(as.Date(broken$sample_date), "%d-%m-%Y")
broken$sex <- ifelse(broken$sex == "M", "male", "female")
report <- episodic_check_cases(broken)
report

# it is a data frame too
report$column[report$severity == "problem"]
```

episodic_check_denominators

Check Your Positivity Data Before You Hand It to EpiSODIC

Description

The same purpose as [episodic_check_cases\(\)](#), for the optional denominators feed: every check [episodic_validate_denominators\(\)](#) would throw on, reported instead of thrown - what is wrong, how many rows, which ones, and what to do about it - plus a couple of things that are allowed but worth a look. Nothing is written, no database is needed. Run this while building your extract step, or when a run's positivity panels come out empty or look wrong.

Usage

```
episodic_check_denominators(denominators)
```

Arguments

denominators Your positivity data: a data frame or tibble with pathogen, sample_date, care_line, area_code and n_tests (see [episodic_synthetic_denominators\(\)](#) for the shape), or a zero-argument function returning one.

Value

A data frame of findings with class `episodic_case_check`, the same shape [episodic_check_cases\(\)](#) returns. Zero rows means the data set passed every check.

See Also

[episodic_check_cases\(\)](#) for the case data feed, [episodic_check_institution_activity\(\)](#) for the hospital activity feed.

Examples

```
denom <- episodic_synthetic_denominators(  
  start_date = as.Date("2025-01-01"), end_date = as.Date("2025-03-31")  
)  
episodic_check_denominators(denom)
```

episodic_check_institution_activity

Check Your Hospital Activity Data Before You Hand It to EpiSODIC

Description

The same purpose as [episodic_check_cases\(\)](#), for the optional institution_activity feed: every check [episodic_run_cron\(\)](#) would refuse a run over, reported instead of thrown, plus a couple of things worth a look. Nothing is written, no database is needed.

Usage

```
episodic_check_institution_activity(institution_activity)
```

Arguments

institution_activity

Your hospital activity data: a data frame or tibble with institution_key, period_start, period_end, patient_days (nullable admissions, n_beds, source) - see [episodic_synthetic_institution_activity\(\)](#) for the shape - or a zero-argument function returning one.

Details

Unlike the case and denominator feeds, an institution_key that matches no institution in your case data is not a problem here - it is deliberately allowed, and only ever counted and warned about at run time (see [episodic_institution_activity_load\(\)](#)), since the two feeds need not be perfectly synchronised and this function has no case data to compare against in the first place.

Value

A data frame of findings with class episodic_case_check, the same shape [episodic_check_cases\(\)](#) returns. Zero rows means the data set passed every check.

See Also

[episodic_check_cases\(\)](#) for the case data feed, [episodic_check_denominators\(\)](#) for the positivity feed.

Examples

```
institutions <- data.frame(
  institution_key = "HOSP-1", institution_type = "hospital", n_beds = 320
)
activity <- episodic_synthetic_institution_activity(
  institutions,
  start_date = as.Date("2025-01-01"), end_date = as.Date("2025-03-31")
)
episodic_check_institution_activity(activity)
```

```
episodic_config_export
```

Export the Resolved Configuration as a Zip File

Description

Bundles the fully-resolved configuration (shipped defaults, your EPISODIC_CONFIG YAML overlay, and - if db_path points at a database - any Settings-screen notifications override) together with the pathogen configuration and, if set, the colour palette override, into a single zip file. This is the Settings screen's "export configuration" button; it is also just a plain function you can call from the console to snapshot an instance's configuration for backup or migration.

Usage

```
episodic_config_export(
  db_path = Sys.getenv("EPISODIC_DB", unset = NA),
  episodic_config_path = Sys.getenv("EPISODIC_CONFIG", unset = NA),
  output_dir = NULL,
  include_secrets = FALSE
)
```

Arguments

db_path	Path to the EpiSODIC database: an existing SQLite file, or a MariaDB/MySQL DSN (see episodic_db_dsn_mariadb()). Defaults to the EPISODIC_DB environment variable. Only used to read a Settings-screen notifications override, if one exists - NA/unset skips this and exports the YAML-resolved configuration only.
episodic_config_path	The config path.
output_dir	Directory to write the zip into. Defaults to what <code>episodic_report_output_dir()</code> resolves for a config_exports/ subdirectory (mirroring where episodic_report_render() writes outbreak reports), or a temporary directory if db_path is not set.
include_secrets	If FALSE (the default), notification secrets (SMTP/webhook/client passwords) are replaced with "***" in the exported configuration - the same masking the

Settings screen applies on-screen. Pass TRUE only when you specifically intend the export to be usable to restore working notification channels elsewhere, and can handle the file with the same care as the secrets themselves.

Value

Invisibly, the path to the written zip file.

Examples

```
## Not run:
episodic_config_export(db_path = Sys.getenv("EPISODIC_DB"))

## End(Not run)
```

episodic_db_connect	<i>Connect to an Existing EpiSODIC Database</i>
---------------------	---

Description

Opens a connection to a database you have already set up with [episodic_db_create\(\)](#), with the settings EpiSODIC needs enabled (for SQLite: WAL journal mode, a busy timeout, and foreign key enforcement). Remember to disconnect with [DBI::dbDisconnect\(\)](#) when you are done.

Usage

```
episodic_db_connect(path, check_schema_version = TRUE)
```

Arguments

path	Path to an existing SQLite file, or a mysql:// DSN (see episodic_db_dsn_mariadb()) pointing at an existing MariaDB/MySQL database.
check_schema_version	Whether to refuse a database whose schema version is not the one this build of EpiSODIC expects, with an error naming episodic_db_migrate() as the fix. TRUE (the default) everywhere except inside episodic_db_migrate() itself, which by definition has to open a database that is out of date.

Value

An open [DBI::DBIConnection](#).

Examples

```
db_path <- tempfile(fileext = ".sqlite")
con <- episodic_db_create(db_path)
DBI::dbDisconnect(con)
con <- episodic_db_connect(db_path)
DBI::dbDisconnect(con)
file.remove(db_path)
```

episodic_db_create	<i>Set Up a New EpiSODIC Database</i>
--------------------	---------------------------------------

Description

Run this once, when setting up a new EpiSODIC instance: it creates a new database at path and builds all the required tables. Refuses to run against a database that already has tables in it, so it cannot accidentally overwrite existing surveillance data - use [episodic_db_connect\(\)](#) to open a database you have already set up.

Usage

```
episodic_db_create(path, overwrite = FALSE)
```

Arguments

path	Path to a SQLite file to create, or a <code>mysql://</code> DSN (see episodic_db_dsn_mariadb()) pointing at a MariaDB/MySQL database. The schema does not have to be empty: EpiSODIC only ever enumerates and touches its own tables, so it can share one with another application. It does have to be free of <i>EpiSODIC</i> 's tables, unless <code>overwrite = TRUE</code> .
overwrite	If TRUE, delete an existing SQLite file (or drop all tables in an existing MariaDB/MySQL database) first. Use with care - this destroys any data already there.

Value

(Invisibly) an open [DBI::DBIConnection](#) to the new database. You are responsible for disconnecting it (with [DBI::dbDisconnect\(\)](#)) when done.

Examples

```
db_path <- tempfile(fileext = ".sqlite")
con <- episodic_db_create(db_path)
DBI::dbDisconnect(con)
file.remove(db_path)
```

episodic_db_dsn_mariadb	<i>Connect EpiSODIC to a MariaDB or MySQL Server</i>
-------------------------	--

Description

EpiSODIC stores its data in either a SQLite file (the default, and all you need for a single-server deployment) or a MariaDB/MySQL database. This function builds the connection string ("DSN") for the latter, so you never have to hand-assemble one or worry about special characters in your password breaking it. Use the resulting string as `db_path` anywhere EpiSODIC expects one, or store it in the `EPISODIC_DB` environment variable. `episodic_db_dsn_mysql()` is an identical alias - use whichever name matches the server you run.

Usage

```
episodic_db_dsn_mariadb(host, dbname, user, password, port = 3306L)
```

```
episodic_db_dsn_mysql(host, dbname, user, password, port = 3306L)
```

Arguments

<code>host</code>	Server hostname or IP address.
<code>dbname</code>	Database (schema) name.
<code>user, password</code>	Credentials.
<code>port</code>	TCP port. Defaults to 3306.

Value

A single string, ready to pass as `db_path` or to `Sys.setenv(EPISODIC_DB = ...)`.

Examples

```
episodic_db_dsn_mariadb(
  host = "db.internal", dbname = "episodic",
  user = "episodic_app", password = "s3cr3t!"
)
```

<code>episodic_db_migrate</code>	<i>Bring an Existing Database up to the Current Schema</i>
----------------------------------	--

Description

EpiSODIC records which version of its schema a database was built with, and refuses to open one it does not recognise rather than failing later with an unexplained SQL error (or, worse, not failing and reading a column that has since come to mean something else). When you upgrade the package and its schema has moved on, this is what brings your existing database along - in one transaction per step, so a failed migration leaves the database exactly as it was.

Usage

```
episodic_db_migrate(
  db_path = Sys.getenv("EPISODIC_DB", unset = NA),
  backup = TRUE
)
```

Arguments

db_path	Path to an existing SQLite database, or a MariaDB/MySQL DSN (see episodic_db_dsn_mariadb()). Defaults to the EPISODIC_DB environment variable.
backup	Whether to copy a SQLite database beside itself before the first step. Ignored for MariaDB/MySQL.

Details

You rarely need to call this yourself: [episodic_run_cron\(\)](#) does it at the start of every run that finds the database behind the installed package, unless `database.auto_migrate` is set to `false` in your configuration. Call it directly where schema changes go through change control rather than through the application - with that setting off, and from an account that holds the privileges to change the schema.

Nothing is ever dropped or rewritten: migrations add tables, add columns, and backfill values. Your surveillance history, your assessments and your audit trail are carried forward untouched. A SQLite database is copied beside itself before the first step (`<file>.schema-<from>-to-<to>.bak`); a MariaDB or MySQL database is not, since a server's backups belong to whoever runs the server.

Each step runs inside a transaction, which under SQLite covers the schema changes themselves. MariaDB and MySQL commit implicitly on every CREATE/ALTER, so there a failed step can leave its schema change in place with no version row recorded; every migration is written to skip what it finds already done, so simply running this again is the correct response.

Two processes can safely be asked to migrate the same database at once - a scheduled run and a manual call, or two servers sharing one MariaDB schema. One migrates; the other waits for it (up to ten minutes on MariaDB/MySQL, five seconds per step on SQLite), finds the work done, and carries on.

A database created by EpiSODIC 0.12.x or earlier carries no version at all, since the version table postdates it. Such a database is adopted at version 1 (which is the shape it already has) rather than refused.

Value

Invisibly, the schema version the database is at afterwards.

Examples

```
db_path <- tempfile(fileext = ".sqlite")
con <- episodic_db_create(db_path)
DBI::dbDisconnect(con)

# already current: reports so and changes nothing
episodic_db_migrate(db_path)
```

```
file.remove(db_path)
```

episodic_db_open

Connect Using the EPISODIC_DB Environment Variable

Description

Like [episodic_db_connect\(\)](#), but falls back to the EPISODIC_DB environment variable when you do not pass a path explicitly - handy for one-off console use, e.g. [episodic_add_user\(\)](#) uses it internally so adding an account needs only a username and password, not a connection you build yourself first.

Usage

```
episodic_db_open(
  db_path = Sys.getenv("EPISODIC_DB", unset = NA),
  check_schema_version = TRUE
)
```

Arguments

db_path	Path to an existing SQLite database, or a mysql:// DSN (see episodic_db_dsn_mariadb()). Defaults to the EPISODIC_DB environment variable.
check_schema_version	Whether to refuse a database whose schema version is not the one this build of EpiSODIC expects, with an error naming episodic_db_migrate() as the fix. TRUE (the default) everywhere except inside episodic_db_migrate() itself, which by definition has to open a database that is out of date.

Value

An open [DBI::DBIConnection](#); you are responsible for disconnecting it.

Examples

```
db_path <- tempfile(fileext = ".sqlite")
con <- episodic_db_create(db_path)
DBI::dbDisconnect(con)
Sys.setenv(EPISODIC_DB = db_path)
con <- episodic_db_open()
DBI::dbDisconnect(con)
file.remove(db_path)
```

episodic_db_truncate *Empty Every EpiSODIC Table, Keeping the Schema Itself*

Description

A hard reset back to "freshly created, no data" - every row in every EpiSODIC table is deleted, but the tables, indexes and constraints stay exactly as `episodic_db_create()` built them, so the database is immediately ready for a new `episodic_run_cron()` without a schema migration. This includes `episodic_app_user`: dashboard accounts are data too, and are deleted along with everything else - there is nothing this function leaves behind to sign in with afterwards.

Usage

```
episodic_db_truncate(path)
```

Arguments

path	Path to an existing SQLite file, or a mysql://DSN (see <code>episodic_db_dsn_mariadb()</code>) pointing at an existing MariaDB/MySQL database.
------	---

Details

Deliberately hard to trigger by accident:

Interactive only Refuses outright under `interactive()` FALSE - a script, a cron job, or any other unattended context can never reach the confirmation prompt, so it can never reach the deletion either.

Typed confirmation Prints exactly what is about to be deleted (every table, and its row count) and then requires you to type the database's own name back at a prompt. A bare yes/no is answerable on autopilot; typing the name back is not.

Value

Invisibly, the character vector of tables that were truncated - `character(0)` if you cancelled, or if the database had no EpiSODIC tables to begin with.

Examples

```
## Not run:
episodic_db_truncate(db_path)

## End(Not run)
```

episodic_demo

Try EpiSODIC With Synthetic Outbreak Data

Description

The fastest way to see what EpiSODIC does: this single call creates a fresh database, generates several years of synthetic laboratory data, runs detection over it, creates a demo epidemiologist account, and opens the dashboard - all without needing access to any real laboratory system or an instance configuration file. It needs nothing but the package: the shipped defaults, plus the few files it writes beside the database (see below).

Usage

```
episodic_demo(
  db_path = tempfile(fileext = ".sqlite"),
  username = "demo",
  full_name = "Demo User",
  email = "demo@example.org",
  password = "demo",
  launch = TRUE,
  run_date = NULL,
  replay_days = 14L,
  lang = Sys.getenv("EPISODIC_LANGUAGE"),
  cases = NULL,
  denominators = NULL,
  overwrite = FALSE,
  ...
)
```

Arguments

db_path	Path to the SQLite database to create. Defaults to a temporary file, so repeated calls never collide and nothing is left behind once the R session ends. A database that already exists is refused, and a MariaDB/MySQL DSN is refused outright: this call generates synthetic cases and runs detection over them, which is contamination anywhere but a throwaway database.
username, full_name, email, password	Credentials for the demo epidemiologist account this creates, so you can sign in and classify a cluster right away. These are placeholder values - change them for anything beyond a local demo.
launch	If TRUE (default), opens the dashboard afterwards (see episodic_run_app()); this call blocks until you close it. Set to FALSE to only build the demo database and return its path, e.g. for scripting or screenshots.
run_date	The date to run detection as of. Left NULL (the default) it is chosen for you, and which way depends on whose data this is: for the bundled synthetic data, the end of the last complete week, so the week the statistical detectors test is a full

	one however far into the week you happen to run the demo; for a cases extract you supply yourself, the last day that extract covers, and the demo says so when it does it. That second rule is the demo's alone. Detection is bounded to a lookback window around <code>run_date</code>, so a run dated today against an extract from last year correctly finds nothing - which is right for a scheduled run and useless for someone trying the system out on a historical export. <code>episodic_run_cron()</code> therefore keeps dating its runs from the system date, as a real surveillance run must.
<code>replay_days</code>	For the bundled synthetic data, how many of its last days are replayed as daily runs, each loading only the cases received by its date, after a first run over everything before them. A scheduled instance learns its reporting delays this way, one run at a time, and the delays are what the reporting completeness, the nowcast and the end-of-outbreak probability are computed from: with a single run there are none to learn from. Each replayed day adds a few seconds; 0 builds the demo in one run, without them. A cases extract you supply is always run once, as of <code>run_date</code> : its receipt dates are not necessarily the dates its cases were reported.
<code>lang</code>	Dashboard language when <code>launch = TRUE</code> : "en", "ar", "nl", "fr", "de", "hi", "zh", or "es", or a regional variant of one ("en-US", "es-419"). Defaults to the <code>EPISODIC_LANGUAGE</code> environment variable, falling back to "en" if that is unset.
<code>cases, denominators</code>	The data to generate the demo from - normally data frames (or tibbles), passed on unchanged to <code>episodic_run_cron()</code>. Default to several years of synthetic data; generate a narrower date range yourself (see <code>episodic_synthetic_cases()</code>) and pass it here for a quicker demo. Trying the demo with your own extract is a good way to see EpiSODIC work end to end: cases is checked against the <code>episodic_case_data</code> requirements first, so data it cannot use stops here with an explanation of what to fix, rather than opening a dashboard with nothing in it. Run <code>episodic_check_cases()</code> on your extract yourself to see the same findings, plus the advisory ones.
<code>overwrite</code>	If TRUE, an existing demo database at <code>db_path</code> and the three configuration files beside it are deleted and rebuilt. FALSE (the default) refuses instead. This deletes whatever is at that path, so it is deliberately not something the demo decides for you.
<code>...</code>	Arguments passed on to <code>episodic_run_app()</code> .

Value

Invisibly, `db_path`.

What the demo writes beside the database

EpiSODIC has no built-in geography - no default map, and no built-in rule for turning a postcode into a province - because a default for either would be a default for one country. The demo therefore configures its own, exactly the way a real deployment does: two files named after `db_path` (`<name>-config.yaml` and `<name>-pc-province.csv`), plus the Netherlands postcode geometry bundled with the package. A third, `<name>-pathogens.csv`, gives the transmissible pathogens in the synthetic data an offspring distribution (`end_r`, `end_k`), which ships empty, so the outbreak

dossier can show its end-of-outbreak probability. Its values are illustrative, chosen for the demo, not estimates for any setting.

They are left in place, so a database built with `launch = FALSE` can be re-opened later with the same geography by setting `EPISODIC_DB`, `EPISODIC_CONFIG`, `EPISODIC_PC_PROVINCE_MAP`, `EPISODIC_PATHOGEN_CONFIG` and `EPISODIC_GEO_DATA` back to them - the paths are printed when the demo finishes. Without them the database still opens; it simply shows no map and no provinces.

Check your data before you run anything

Do not find out from an empty dashboard. Run `episodic_check_cases()` on your extract - it needs no database, changes nothing, and reports every problem it finds at once, with the rows and values involved and what to do about each:

```
cases <- my_extract_and_transform_function()
episodic_check_cases(cases)
```

It also reports what is merely worth a look: one pathogen spelled two ways (two streams instead of one), no ward on any hospital row (no ward-level detection), a `patient_key` that never repeats (no deduplication), postcodes the map cannot place, sample dates in the future. `episodic_check_cases(..., stop_on_problem = TRUE)` runs the same checks but throws an error, for use in a script. `episodic_run_cron()` runs that before every run, and refuses to start on data it cannot use, naming what to fix.

See Also

`episodic_check_cases()` to see what EpiSODIC makes of your own extract first, and `episodic_case_data` for the shape it expects.

Examples

```
## Not run:
# launches a blocking, interactive Shiny session against several years
# of freshly-generated synthetic data
episodic_demo()

## End(Not run)

# non-interactive: populate a database and stop there, e.g. for scripting
cases <- episodic_synthetic_cases(
  start_date = as.Date("2025-01-01"), end_date = as.Date("2025-03-31")
)
db_path <- episodic_demo(launch = FALSE, cases = cases, denominators = NULL)
file.remove(db_path)
```

episodic_geo

Show Clusters on a Map

Description

The dashboard can plot cluster case counts on a choropleth map by postcode (or any other geographic unit you use), provided the optional `sf` package is installed and geographic reference data is available. Without either, the dashboard still works fine - it just falls back to a bar chart of case counts by area instead of a map.

Usage

```
episodic_geo_overlay_resolve(
  path = Sys.getenv("EPISODIC_GEO_DATA_OVERLAY", unset = NA)
)

episodic_geo_source_resolve(path = Sys.getenv("EPISODIC_GEO_DATA", unset = NA))
```

Arguments

<code>path</code>	Path to an <code>.rds</code> file holding an <code>sf</code> object with <code>pc</code> and <code>geometry</code> columns. Defaults to the <code>EPISODIC_GEO_DATA</code> environment variable. With none set, there is no map and the dashboard shows its bar-chart fallback instead.
-------------------	---

Details

There is no default map. Point the `EPISODIC_GEO_DATA` environment variable at your own `.rds` file (an `sf` object with a `pc` column matching your case data's area codes, and a `geometry` column) to map your own region. The Dutch postcode geometry bundled with the package is used only by `episodic_demo()`.

You can optionally add a second, purely visual layer of outlines - e.g. province or municipality borders - drawn on top of the choropleth for orientation, via `EPISODIC_GEO_DATA_OVERLAY` and `episodic_geo_overlay_resolve()`. This layer carries no case counts, so it only needs a `geometry` column.

Value

An `sf` object, or `NULL` if `sf` is not installed, no path is set, or the file is missing or invalid.

Why there is no default map

A default map is a default *country*. Falling back to the Netherlands PC4 geometry bundled with the package would give a laboratory anywhere else a silent, confident map of the Netherlands with its own case counts joined onto whichever Dutch postcodes happened to share a number - postcode-like codes are four digits in a great many countries - and a plausible-looking wrong answer is worse than no answer. So there is no default: with nothing configured the map is simply absent, and the dashboard falls back to a bar chart of case counts by area, which is correct everywhere.

`episodic_demo()` sets `EPISODIC_GEO_DATA` to the bundled Netherlands geometry itself, so the demo still shows a map.

Examples

```
# NULL when unset (or when the sf package is not installed)
episodic_geo_overlay_resolve(path = NA)
# NULL when unset (or when the sf package is not installed)
episodic_geo_source_resolve(path = NA)
```

```
episodic_mail_template_preview
      Preview an Email Template
```

Description

Renders the body of one of EpiSODIC's emails against example data and writes it to an HTML file to open in a browser, so an email template can be checked without waiting for a detection run to send one.

Usage

```
episodic_mail_template_preview(
  kind = c("new_clusters", "report"),
  template_path = NULL,
  output_file = tempfile(fileext = ".html"),
  lang = Sys.getenv("EPISODIC_LANGUAGE")
)
```

Arguments

<code>kind</code>	"new_clusters" or "report".
<code>template_path</code>	Path to the .qmd template to preview. Defaults to the one the matching environment variable names; with neither, the built-in body is written, which is what the email would go out with.
<code>output_file</code>	Where to write the HTML.
<code>lang</code>	Language to render in. Defaults to <code>EPISODIC_LANGUAGE</code> .

Details

Two emails can be given a template of their own, each with its own environment variable pointing at a Quarto (.qmd) file:

- `EPISODIC_MAIL_TEMPLATE_NEW_CLUSTERS` for the alert sent when a run detects new clusters;
- `EPISODIC_MAIL_TEMPLATE_REPORT` for the email a scheduled outbreak report is attached to.

A template changes the HTML body sent through the email channels (smtp, sendmail, microsoft365); the subject line, the ntfy, Teams and Slack messages and the run-failure email are unchanged. Two starting points ship with the package, in `system.file("mail", package = "EpiSODIC")`. Every template receives the instance palette (`episodic_palette()`) and the inline styles built from it, so it can carry the dashboard's own colours and font. See `vignette("notifications")` for the data a template receives.

A template that cannot be used when an email is due never stops the email: it goes out with the built-in body, and the run log says why. This function, by contrast, stops with the error, which is what a preview is for.

Value

The path to the written HTML file, invisibly.

Examples

```
## Not run:
# the shipped starting point for the new-clusters alert
episodic_mail_template_preview(
  "new_clusters",
  template_path = system.file(
    "mail",
    "episodic_default_mail_new_clusters.qmd",
    package = "EpiSODIC"
  )
)

## End(Not run)
```

episodic_notifications

How notifications work

Description

When `episodic_run_cron()` detects new clusters or fails, it can notify your team through one or more channels: ntfy, SMTP, sendmail, Microsoft 365, Teams (Power Automate Workflow web-hook), or Slack. Notifications are configured in the instance YAML configuration file (the same file `EPISODIC_CONFIG` points at), under a `notifications` key. The notification settings are deliberately excluded from the detection `config_hash`, so changing them never alters reproducibility and secrets never reach the `config_snapshot` stored on each run.

Details

Notifications are dispatched *after* the detection transaction has committed: a failed notification never rolls back a successful run. Errors during dispatch are logged via `episodic_trace()` and never stop the run from finishing.

episodic_notify_test	<i>Send a Test Notification Through All Configured Channels</i>
----------------------	---

Description

Validates the notification configuration and sends a test message through every enabled channel. Run this interactively after setting up your instance configuration to verify that notifications work end to end.

Usage

```
episodic_notify_test(  
  episodic_config_path = Sys.getenv("EPISODIC_CONFIG", unset = NA)  
)
```

Arguments

episodic_config_path
The config path.

Value

Invisibly, a named logical vector: TRUE for each channel that succeeded, FALSE for each that failed.

Examples

```
## Not run:  
# After configuring notifications in your EPISODIC_CONFIG YAML:  
episodic_notify_test()  
  
## End(Not run)
```

episodic_palette	<i>The Dashboard's Colour Palette and Typography</i>
------------------	--

Description

Returns the colours and typography used throughout the EpiSODIC dashboard and charts, as a named list. Useful if you want to match your own plots or reports to the house style, or check what colour a given status uses.

Usage

```
episodic_palette()
```

Details

The palette ships with an organisation-neutral default (`inst/config/episodic_default_style.yaml`). To use your own institute's colours or fonts instead, point the EPISODIC_STYLE environment variable at a YAML file that overrides only the roles you want to change - anything you do not set keeps its shipped default.

The same palette governs everything EpiSODIC draws: the dashboard, the outbreak reports rendered by `episodic_report_render()` (their charts, text, headings, links and tables) and the emails it sends. The process doing the render resolves it and hands it to the template, so a report or an email always carries the colours and font of the instance that produced it.

Each value in the EPISODIC_STYLE file is checked: a colour must be a quoted hex colour ("`#RGB`", "`#RRGGBB`" or "`#RRGGBBAA`"), font a CSS font-family stack and `font_size_base` a CSS length (px, pt, rem or em). A value that fails keeps the shipped value for that role only, and a file that does not exist or cannot be read keeps the whole shipped palette. Neither stops the dashboard, a report or an email: the problem is shown on the Info screen's reference-data panel, written to every detection run's log, and raised once as a warning.

This is independent of `episodic_config_resolve()` on purpose: colours and typography never affect the `config_hash` recorded with a detection run, since they have no bearing on reproducibility.

Value

A named list. The greyscale neutrals are ink (default text), muted (secondary text), faint (tertiary text), border, `bg_subtle`, `bg`, and surface. The semantic roles are primary, secondary, tertiary, success, warning, and danger, each with `_dark/``_light/``_tint` variants where used. The severity scale is `severity_baseline`, `severity_low`, `severity_medium`, `severity_high` and `severity_very_high`. `font` and `font_size_base` hold the app's typography, not a colour.

Default font and colours

These are all the default values, and all can be changed using a custom YAML file.

```
font:          '"IBM Plex Sans", ui-sans-serif, system-ui, "Segoe UI", sans-serif'
font_size_base:  "13px"
ink:           "#222222"
muted:         "#495057"
faint:         "#ADB5BD"
border:        "#DEE2E6"
bg_subtle:     "#EBEBEB"
bg:            "#F8F9FA"
surface:       "#FFFFFF"
primary:       "#008CBA"
primary_dark:  "#005F7A"
primary_light: "#66BAD6"
primary_tint:  "#D9EDF5"
secondary:     "#333333"
secondary_dark: "#1A1A1A"
tertiary:      "#20C997"
tertiary_dark: "#168D6A"
```

```

success:          "#43AC6A"
success_dark:     "#2F784A"
warning:          "#F8AC59"
warning_dark:     "#AE783E"
danger:           "#F36A5A"
danger_dark:      "#AA4A3F"
severity_baseline: "#ADB5BD"
severity_low:     "#43AC6A"
severity_medium:  "#E99002"
severity_high:    "#F04124"
severity_very_high: "#222222"

```

Of note:

- `primary_dark` is the background colour of the navigation bar.
- `severity_baseline` to `severity_very_high` are one ordinal scale, shared by an epidemiologist's verdicts and the Moving Epidemic Method's intensity bands, so a step has one colour on every screen. Text on them is set to whichever of ink, white or black reads.
- `font` is a CSS font-family stack, and `font_size_base` is the app's base font size.
 - Every other font size in the dashboard is set in rem relative to it, so changing `font_size_base` scales the whole app's type proportionally (useful when swapping in a font that reads naturally smaller or larger than the default at the same pixel size).
 - Changing `font` only changes the CSS declaration; if it names a webfont rather than a system font, delivering that font (a self-hosted @font-face or a link to its provider) is the operator's own concern.

Examples

```

pal <- episodic_palette()
pal$primary
pal$danger
pal$font
pal$font_size_base

```

episodic_report_render

Render an Outbreak Report for Clinical Colleagues

Description

Produces a self-contained HTML outbreak report for one cluster - the document you send to a treating physician, an infection prevention nurse, or a clinical colleague (e.g., a clinical microbiologist) who has neither an EpiSODIC account nor R installed. The report includes the epidemic curve, trend chart, the narrative summary shown in the dashboard, and (optionally) the case line list.

Usage

```
episodic_report_render(
  con,
  cluster_id,
  output_dir,
  user_id = NA,
  include_linelist = TRUE,
  small_count_threshold = NULL,
  episodic_config_path = Sys.getenv("EPISODIC_CONFIG", unset = NA),
  lang = Sys.getenv("EPISODIC_LANGUAGE"),
  qmd_path = Sys.getenv("EPISODIC_QUARTO_REPORT", unset = NA)
)
```

Arguments

<code>con</code>	A DBI::DBConnection .
<code>cluster_id</code>	The cluster to report on.
<code>output_dir</code>	Directory the rendered HTML is written to. Created if it does not exist.
<code>user_id</code>	The id of the user requesting the report, or NA for an automated (cron) render.
<code>include_linelist</code>	If TRUE (default), the case line list is included in the report. Set to FALSE for a summary-only report, e.g. when sending outside your own organisation.
<code>small_count_threshold</code>	Small counts in the geography/institution breakdown tables are suppressed (shown as "<threshold") below this value, to avoid identifying individuals in a small population. Defaults to <code>config\$report\$small_count_threshold</code> .
<code>episodic_config_path</code>	The config path.
<code>lang</code>	Report language: "en", "ar", "nl", "fr", "de", "hi", "zh", or "es", or a regional variant of one ("en-US", "es-419"). Defaults to the EPISODIC_LANGUAGE environment variable, falling back to "en" if that is unset.
<code>qmd_path</code>	Path to the Quarto template to render. Defaults to the EPISODIC_QUARTO_REPORT environment variable, falling back to the shipped template if that is unset.

Details

Every render is kept, versioned, and logged to the database, including exactly which cases it contained - so what was sent out on a given date stays a fully recoverable record, even if the underlying data changes later.

By default the report uses EpiSODIC's own report template. If your organisation needs its own layout or branding, set the EPISODIC_QUARTO_REPORT environment variable to your own .qmd file; the default `inst/report/episodic_default_report.qmd` is a good starting point to copy and adapt.

The report is styled from the instance palette (`episodic_palette()`), the same colours and font as the dashboard: its charts are drawn in it, and an `episodic.scss` theme built from it is written

beside the template, which the shipped template names as its theme. A custom template receives the palette as `palette` in its data and can name the same theme file.

Rendering requires **Quarto** to be installed separately (both the quarto R package and the Quarto command-line tool) - this function raises an informative error if it is not found.

Value

Invisibly, a list with `file_path` (where the HTML was written), `file_sha256`, `version_no`, and `report_id`.

Examples

```
## Not run:
# needs both the quarto R package and the Quarto CLI installed, plus a
# database with at least one detected cluster - see episodic_demo() for
# a populated one, and the app's rail for a cluster_id to render
db_path <- episodic_demo(launch = FALSE)
con <- episodic_db_connect(db_path)
episodic_report_render(con, cluster_id = 1, output_dir = tempdir())
DBI::dbDisconnect(con)

## End(Not run)
```

episodic_run_app

Open the EpiSODIC Dashboard

Description

Launches the Shiny dashboard against a database already populated by `episodic_run_cron()`: an overview of monitored surveillance streams and each cluster's dossier, with charts, its narrative interpretation, and an assessment form. Anyone can browse the dashboard; signing in is only required to record an assessment or classify a cluster.

Usage

```
episodic_run_app(
  db_path = Sys.getenv("EPISODIC_DB", unset = NA),
  lang = Sys.getenv("EPISODIC_LANGUAGE"),
  ...
)
```

Arguments

<code>db_path</code>	Path to the EpiSODIC database: a SQLite file, or a MariaDB/MySQL DSN (see <code>episodic_db_dsn_mariadb()</code>). Defaults to the EPISODIC_DB environment variable.
----------------------	---

lang	Dashboard language, fixed for the whole running app - there is no in-app language switcher. One of "en", "ar", "nl", "fr", "de", "hi", "zh", or "es", or a regional variant of one ("en-US", "es-419"). Defaults to the EPISODIC_LANGUAGE environment variable, falling back to "en" if that is unset.
...	Passed on to <code>shiny::runApp()</code> , e.g. port or host.

Details

If you just want to explore EpiSODIC without setting anything up first, use `episodic_demo()` instead - it populates a database with synthetic data and calls this function for you.

Value

Invisible; called for its side effect of starting the app. This call blocks until the app is stopped.

Examples

```
## Not run:
# opens a blocking, interactive dashboard session; see episodic_demo()
# for a one-call version that also creates a populated demo database
episodic_run_app("/path/to/episodic.sqlite")

## End(Not run)
```

episodic_run_cron	<i>Run One Surveillance Detection Cycle</i>
-------------------	---

Description

This is the function you schedule to run regularly (e.g. daily, via cron): it pulls in new laboratory data, checks every monitored stream for statistical aberrations with the configured detectors, reconciles the results into cluster dossiers for the board to assess, and records everything in the database. A run either completes in full or leaves no trace at all - it runs inside a single database transaction, so a failed run is always safe to simply retry.

Usage

```
episodic_run_cron(
  cases,
  denominators = NULL,
  institution_activity = NULL,
  episodic_config_path = Sys.getenv("EPISODIC_CONFIG", unset = NA),
  pathogen_config_path = Sys.getenv("EPISODIC_PATHOGEN_CONFIG", unset = NA),
  db_path = Sys.getenv("EPISODIC_DB"),
  host = Sys.info()[["nodename"]],
  account = Sys.info()[["user"]],
  run_date = Sys.Date(),
  backfill = NULL,
```

```

    debug = FALSE
  )

```

Arguments

cases	Your laboratory data: a data frame or tibble in the shape episodic_case_data describes, or a zero-argument function that returns one. Defaults to the bundled synthetic generator, useful for demos and testing but not real surveillance.
denominators	Optional: your testing-volume data, in the same form as cases - normally a data set, a function if it has to be produced at run time (see episodic_synthetic_denominators() for the expected shape). Leave as NULL (the default) if you have none to supply - positivity panels simply stay blank.
institution_activity	Optional: your hospital patient-days data (see episodic_synthetic_institution_activity() for the expected shape), normally as a data set, or as a function taking the current institutions table. Its <code>institution_key</code> is the same identifier your case data uses - EpiSODIC hashes both on load, so a key taken from the institutions table this passes a function is already hashed and will match nothing. Leave as NULL (the default) if you have none - detection falls back to raw case counts.
episodic_config_path	The config path.
pathogen_config_path	Path to an operator's own per-pathogen CSV, overlaid row-by-row on the shipped <code>inst/config/episodic_default_pathogen_config.csv</code> . Defaults to the <code>EPISODIC_PATHOGEN_CONFIG</code> environment variable; if unset, only the shipped defaults are used. Set to a path that does not exist, this stops with an error. See <code>vignette("tuning-detection")</code> for the column reference.
db_path	Path to the EpiSODIC database: a SQLite file (created automatically if it does not exist yet) or a MariaDB/MySQL DSN (see episodic_db_dsn_mariadb()).
host, account	Recorded with the run for audit purposes; default to the current machine and account.
run_date	The date to treat as "today". Defaults to the system date. Every detector anchors on it: <code>same_place</code> and <code>rare_trigger</code> report only hits inside their <code>lookback_days</code> of it, and Farrington tests only weeks that are complete on or before it. Detection against a historical extract is therefore one run per date with a <code>run_date</code> inside the extract's own window, not one run dated today holding the whole archive - a run dated today reports what is current, and the archive is not. <code>episodic_validate_detection()</code> replays exactly that way.
backfill	Whether this run reports the whole case history it is given rather than only what falls inside the detectors' lookback windows. NULL, the default, decides from the database: a run with no completed run behind it is a backfill, so a first import of years of history arrives as years of clusters (the settled ones closed in that same run by <code>reconciliation.stale_open_days</code> , and so straight into the Archive) rather than as an empty dashboard. Every run after it is bounded again, so nothing is ever reported twice. Pass FALSE to bound the first run too, which is what a prospective replay wants: <code>episodic_validate_detection()</code> measures detection delay, and a first replay run that reported its whole baseline at once

	would measure the import instead. TRUE forces it on, for a database whose history was imported by something other than a run.
debug	If TRUE, print a good deal more than the phase-by-phase progress this function always writes: <code>sessionInfo()</code> , the versions of every package whose own behaviour a run depends on, memory snapshots at the start and end, and per-stream detail inside the detection loop - which stream, how many cases, whether it cleared the eligibility gate, what each detector did with it. Useful when a run's <i>results</i> are not what an operator expects and the question is which stream, for which phase, they diverged at. Noisy; leave off for routine scheduled runs.

Details

EpiSODIC never connects to your laboratory system directly. You extract and transform your own data beforehand, and hand it over as a plain data frame or tibble: cases for the laboratory results themselves (see [episodic_case_data](#) for the required columns and their allowed values), and optionally denominators and `institution_activity` for testing volume and hospital activity. A data set is the normal case; if producing the data only makes sense at run time (a live database query, for instance), a zero-argument function returning one is accepted just as well.

The exact detection settings used are recorded with the run, so any past result can always be traced back to the configuration that produced it.

A database whose schema is behind the installed package is brought up to date before the run starts, the way `episodic_db_migrate()` would, so upgrading an instance is upgrading the package. Set `database.auto_migrate` to false in your configuration where schema changes go through a DBA instead; the run then refuses such a database and records why. See `vignette("deployment")`, "Upgrading EpiSODIC".

So is what each feed delivered. Before the run writes anything, your case data goes through `episodic_check_cases()`. Structural problems - a missing column, a value outside the allowed set, a date that does not read as a date - stop the run with an error naming every offending column, its values and the rows they are in, and are recorded on the run as well, so the reason is visible both where the run was started and in the dashboard's activity screen. Advisory findings are mentioned once and the run proceeds. Run `episodic_check_cases()` on your extract yourself to see all of it without starting a run at all. A run that fails later, for any other reason, records the reason and warns rather than returning quietly. Rows that are merely unmatched are counted rather than dropped in silence: institution activity whose `institution_key` matches no known institution is skipped with a warning, its count recorded, and the run finishes "partial" instead of "success". Both are complete runs the dashboard reads from; "partial" says go and look at why rows were skipped. `episodic_detection_run` carries the counts (`n_cases_supplied`, `n_cases_inserted`, `n_activity_skipped`, and the rest).

Value

Invisibly, the `run_id` of the completed run. The run's row in `episodic_detection_run` holds its status, the per-feed load counts, and `error_text` if it failed. Case data that does not satisfy the requirements throws instead of returning - the run row is still written, with `status = "failed"` and the same message in `error_text`.

Check your data before you run anything

Do not find out from an empty dashboard. Run `episodic_check_cases()` on your extract - it needs no database, changes nothing, and reports every problem it finds at once, with the rows and values involved and what to do about each:

```
cases <- my_extract_and_transform_function()
episodic_check_cases(cases)
```

It also reports what is merely worth a look: one pathogen spelled two ways (two streams instead of one), no ward on any hospital row (no ward-level detection), a patient_key that never repeats (no deduplication), postcodes the map cannot place, sample dates in the future. `episodic_check_cases(..., stop_on_problem = TRUE)` runs the same checks but throws an error, for use in a script. `episodic_run_cron()` runs that before every run, and refuses to start on data it cannot use, naming what to fix.

See Also

`episodic_check_cases()` to see what EpiSODIC makes of your extract before you schedule anything, and `episodic_case_data` for the requirements it checks against.

Examples

```
db_path <- tempfile(fileext = ".sqlite")
cases <- episodic_synthetic_cases(
  start_date = as.Date("2025-01-01"), end_date = as.Date("2025-03-31")
)
run_id <- episodic_run_cron(db_path = db_path, cases = cases)
file.remove(db_path)
```

episodic_scheduled_reports

How scheduled reports work

Description

A scheduled report is a standing subscription on one cluster: every `interval_days`, `episodic_run_cron()` emails the outbreak report - the same self-contained HTML `episodic_report_render()` produces on demand - to a list of addresses, as an attachment. It exists for people who need to follow a cluster without an EpiSODIC account, or with only a viewer one: a ward manager, a department head, a referring physician. This is deliberately not the same thing as `episodic_notifications`: those alert *epidemiologists* that something needs assessing; a scheduled report keeps *everyone else* informed once it is being watched, on a cadence they chose, without needing to open the dashboard at all.

Details

An epidemiologist sets a schedule from the cluster's Reports panel: an interval in days, one or more recipient addresses, which already-configured, email-capable notification channel to send through (smtp, sendmail or microsoft365 - see [episodic_notifications](#)), and whether to include the case line list (excluded by default, since these recipients are often outside the organisation - see vignette("deployment") for how a custom Quarto template can include it anyway). Every setting is stored as an event, the same event-sourced shape as a cluster note: setting a new schedule (even just changing the recipient list) starts its cadence over, and the very next cron run sends the first report under the new settings so the change is confirmed immediately rather than silently taking effect days later.

Each cron run checks every cluster's current schedule and sends whichever are due - due the first time under a schedule's current settings, or `interval_days` after the last *successful* send under those settings. A failed send (a broken SMTP relay, say) does not postpone the next attempt: the very next cron run tries again, and every attempt, sent or failed, is logged to `episodic_report_subscription_send`. A schedule stops automatically, after one final report, the run a cluster closes, merges, or is suppressed - further updates about an inactive cluster would be noise rather than news. An epidemiologist can also cancel a schedule outright at any time.

episodic_setup_microsoft365

Set Up Microsoft 365 Authentication for Notifications

Description

Runs an interactive Azure AD login and caches the refresh token for subsequent unattended use by [episodic_run_cron\(\)](#). You only need to run this once per server, unless the token expires. Requires the Microsoft365R and AzureGraph packages.

Usage

```
episodic_setup_microsoft365(
  tenant_id,
  client_id = NULL,
  scopes = c("Mail.Send", "User.Read", "openid", "offline_access")
)
```

Arguments

<code>tenant_id</code>	Your Azure AD tenant ID.
<code>client_id</code>	Your Azure AD app registration's client ID. If NULL, uses the Microsoft365R package default.
<code>scopes</code>	Scopes to request. The default requests <code>Mail.Send</code> .

Details

If your Azure AD app registration has a client secret configured for application-level (daemon) access, put the `client_secret` in the instance YAML instead and skip this function entirely: the cron will use the client-credentials flow directly.

If your staff already authenticate to Microsoft 365 through a shared login on this server (for example by having called `Microsoft365R::get_business_outlook()` interactively at some point, or by running this function once), you can skip `client_id` entirely in the instance YAML: `episodic_notify_microsoft365()` will reuse whatever cached Azure AD token it finds for the configured `tenant_id` in `AzureAuth::AzureR_dir()`, without any client ID, client secret, or new interactive login.

Value

Invisible TRUE on success.

Examples

```
## Not run:
episodic_setup_microsoft365(tenant_id = "your-tenant-id")

## End(Not run)
```

`episodic_synthetic_cases`*Generate Synthetic Outbreak Data*

Description

Produces several years of laboratory surveillance data for a fictional northern-Netherlands region - eight hospitals, twenty long-term care institutions and the region's municipalities - and injects six outbreaks for the detectors to find. This is what powers [episodic_demo\(\)](#) and the package's test suite, and it doubles as a worked example of the shape your own data should have (see [episodic_case_data](#)).

Usage

```
episodic_synthetic_cases(
  start_date = end_date - 5 * 365,
  end_date = Sys.Date(),
  seed = 1,
  outbreaks = TRUE,
  outbreak_offsets = NULL
)
```

Arguments

start_date	First sample date to generate. Defaults to five years before end_date - Farrington needs four of them before it will compare anything against anything.
end_date	Last sample date to generate. Defaults to today, so a demo built from this always shows current surveillance rather than whatever year the package was released in.
seed	RNG seed, for reproducible demo data.
outbreaks	Which outbreaks to inject: TRUE for all six (the default), FALSE for none, or a character vector of outbreak identifiers ("RARE", "WARD", "LTC", "PS", "PROP", "WAVE"). A history with none injected is what a negative control needs: every alarm raised on it is a false one, which is the only clean way to measure an alarm rate. Each outbreak draws from the random stream whether or not it is kept, so a given seed always produces the same cases for a given outbreak regardless of which others were asked for.
outbreak_offsets	Days to move outbreaks further back from end_date. All six are anchored to end_date, so by default they sit in the last few months of whatever window is generated - fine for a demo, wrong for a prospective evaluation, which needs them dispersed across the window instead. Either a single number applied to all of them, or a named vector giving days per outbreak identifier (any not named stays at 0). NULL, the default, leaves every outbreak anchored to end_date.

Value

A data frame satisfying `episodic_check_cases()`, carrying the ground truth of what was injected as an attribute - see `episodic_synthetic_ground_truth()`.

What it puts in

A seasonal Poisson baseline for eight endemic pathogens, deliberately thin per place: a demo whose baseline keeps tripping the rule-based detectors by coincidence buries the outbreaks it is meant to demonstrate. Influenza A and RSV come in short winter waves of varying size and timing, as they do in life, so the Moving Epidemic Method has seasons to fit its thresholds and intensity bands to; the others rise and fall gently through the year. Patients recur - roughly one case in six is a repeat positive from a patient already in the data - so deduplication and episode grouping have something to do.

On top of that, six outbreaks, sized from a single case to a regional wave, each shaped for a different detector:

Outbreak	Shape	Found by
Invasive meningococcal disease	one case	rare_trigger
Ward cluster	3 cases, one ward, 12 days	same_place
Nursing home outbreak	8 cases, one institution, 9 days	same_place
Point source	14 cases, one ward, days apart	same_place
Propagated	15 cases, four generations, 90 days	same_place, Rt
Regional wave	144 cases, diffuse, 6 rising weeks	Farrington, MEM

The regional wave is deliberately spread one case to a place: no institution sees enough of it for a rule-based detector to notice, and only the statistical baseline comparison finds it. That contrast - a diffuse signal no amount of local vigilance would catch - is half the reason the statistical detectors exist.

Outbreaks are anchored to `end_date` and clipped to the window you ask for, so a short window returns a partial one rather than cases outside the range you asked for.

What was injected, as data

The result carries its own ground truth: which outbreaks were injected, where and when each of them ran, and exactly which cases belong to which. Read it with [episodic_synthetic_ground_truth\(\)](#). Injected cases also carry a PT-OUTBREAK-* patient key, but that is a convenience for a human reading a line list and not an interface: anything measuring detection against what was injected takes the ground truth, never a pattern match on a key.

See Also

[episodic_synthetic_ground_truth\(\)](#) for what was injected, [episodic_check_cases\(\)](#) to see what the requirements make of it, and [episodic_synthetic_cases_calibration\(\)](#) for many more clusters than a demo wants, to tune a configuration against.

Examples

```
cases <- episodic_synthetic_cases(  
  start_date = as.Date("2025-01-01"), end_date = as.Date("2025-03-31")  
)  
nrow(cases)  
head(cases)
```

episodic_synthetic_cases_calibration

Generate Synthetic Data at Tunable Cluster Volume

Description

[episodic_synthetic_cases\(\)](#) injects six outbreaks in total - enough to show every detector working, and few enough that a demo dashboard reads like a real morning's work. That is the wrong shape for tuning a configuration against (e.g. deciding how many dossiers your board can realistically review per month). This function fills that gap: on top of the same baseline and the same six outbreaks, it adds many independent case clusters for one chosen pathogen, at a rate you control, so you can see how detection volume responds as you adjust `n_bumps_per_month` or your own configuration. Every case it adds carries a PT-VOL-* patient key so it is always identifiable as synthetic tuning data, never mistaken for anything else.

Usage

```
episodic_synthetic_cases_calibration(
  start_date = end_date - 5 * 365,
  end_date = Sys.Date(),
  pathogen = "Clostridioides difficile",
  n_bumps_per_month = 3,
  seed = 1
)
```

Arguments

start_date, end_date	The window to generate over; defaults as in episodic_synthetic_cases() .
pathogen	Which pathogen to generate the extra clusters for. Defaults to <i>Clostridioides difficile</i> , a plausible example of an endemic pathogen that produces frequent clusters at a busy institution.
n_bumps_per_month	Average number of independent case clusters generated per calendar month. Raise or lower this to see how detection volume responds.
seed	RNG seed, for reproducible runs.

Value

A data frame satisfying [episodic_check_cases\(\)](#), including everything [episodic_synthetic_cases\(\)](#) produces (background baseline, the six demo outbreaks) plus the extra volume.

Examples

```
cases <- episodic_synthetic_cases_calibration(
  start_date = as.Date("2025-01-01"), end_date = as.Date("2025-06-30"),
  n_bumps_per_month = 4
)
sum(startsWith(cases$patient_key, "PT-VOL-"))
```

```
episodic_synthetic_denominators
```

Add a Testing-Volume (Positivity) Feed

Description

Case counts alone cannot distinguish a rise in infections from a rise in testing. If you can supply how many tests were performed - even as a weekly aggregate, not per-test detail - EpiSODIC can show a positivity rate alongside the case count, which is often the more meaningful signal. This feed is entirely optional: skip it and positivity panels simply stay blank.

Usage

```
episodic_synthetic_denominators(
  start_date = end_date - 5 * 365,
  end_date = Sys.Date(),
  seed = 1
)
```

Arguments

start_date, end_date	The period to generate weekly rows for. Defaults to the five years up to today, matching episodic_synthetic_cases() .
seed	RNG seed, for reproducible demo data.

Details

This function is a synthetic example showing the expected shape: weekly counts of a multiplex GI PCR panel that also reports Norovirus. Use it as a template for your own data, which you pass to [episodic_run_cron\(\)](#) as denominators - a data frame or tibble with the same five columns: pathogen, sample_date (week start), care_line, area_code (may be NA), and n_tests.

Value

A data frame with pathogen, sample_date (week start), care_line, area_code, n_tests.

Examples

```
denom <- episodic_synthetic_denominators(
  start_date = as.Date("2025-01-01"), end_date = as.Date("2025-03-31")
)
head(denom)
```

episodic_synthetic_ground_truth

Read Back What Was Injected Into Synthetic Data

Description

[episodic_synthetic_cases\(\)](#) injects outbreaks into an otherwise endemic history. This reads back exactly what it injected: one row per outbreak saying where and when it ran and how large it was, and the case-level membership that goes with it. It is what makes a detection result measurable against known truth rather than against an epidemiologist's verdict on it - see [episodic_validate_detection\(\)](#).

Usage

```
episodic_synthetic_ground_truth(cases)
```

Arguments

`cases` A data frame from `episodic_synthetic_cases()`.

Details

Injected cases also carry a recognisable PT-OUTBREAK-* patient key, and it is tempting to recover membership by matching on it. Do not: that couples whatever does the matching to a naming convention nobody has undertaken to keep, and it fails silently, as a smaller set of outbreak cases rather than an error, the first time the convention changes.

Value

A list of two data frames:

`outbreaks` One row per injected outbreak: `outbreak_id`, `label`, `pathogen`, `institution_key` and `ward` (both NA where the outbreak was not confined to one), `first_day`, `peak_day`, `last_day`, `n_cases`, and the `expected_channel` and `expected_level` the shape was designed to exercise.

`cases` One row per injected case: `outbreak_id` and the `source_key` it carries in the case data, which is what joins it to `episodic_case` once a run has loaded it.

Both are empty (zero rows, same columns) when nothing was injected.

See Also

`episodic_synthetic_cases()`, `episodic_validate_detection()`

Examples

```
cases <- episodic_synthetic_cases(
  start_date = as.Date("2025-01-01"), end_date = as.Date("2025-03-31")
)
truth <- episodic_synthetic_ground_truth(cases)
truth$outbreaks[, c("outbreak_id", "pathogen", "n_cases")]
```

`episodic_synthetic_institution_activity`

Add a Hospital Activity Feed (Patient-Days)

Description

Raw case counts at a hospital can rise simply because the hospital is busier, not because infection risk has increased. If you can supply weekly patient-days (or admissions, or bed counts) per hospital, EpiSODIC can normalise case counts against activity for a more reliable signal. This feed is entirely optional: without it, detection falls back to raw counts, which is a reasonable default, not a broken one.

Usage

```
episodic_synthetic_institution_activity(
  institutions = episodic_synthetic_institutions(),
  start_date = end_date - 5 * 365,
  end_date = Sys.Date(),
  seed = 1
)
```

Arguments

institutions A data frame (or tibble) of institutions from **your own** registry, with `institution_key`, `institution_type` and `n_beds`; filtered internally to hospitals only. Defaults to the synthetic registry `episodic_synthetic_cases()` draws on, so the two feeds line up with no arguments at all.

`institution_key` must be the same identifier your case feed uses, not the one `episodic_db_institutions()` returns: EpiSODIC stores a hash of your key rather than the key itself, and both feeds are hashed on load. Handing this function the database's own table produces a feed whose keys are already hashed, and none of it will match.

start_date, end_date The period to generate weekly rows for. Defaults to the five years up to today, matching `episodic_synthetic_cases()`.

seed RNG seed, for reproducible demo data.

Details

This function is a synthetic example showing the expected shape: weekly patient-days per hospital, modelled as bed count times occupancy, with a realistic winter peak. Use it as a template for your own data, which you pass to `episodic_run_cron()` as `institution_activity` - normally a data frame or tibble.

Value

A data frame with `institution_key`, `period_start`, `period_end`, `patient_days`, `n_beds`, `source`.

Examples

```
institutions <- data.frame(
  institution_key = "HOSP-1", institution_type = "hospital", n_beds = 320
)
activity <- episodic_synthetic_institution_activity(
  institutions,
  start_date = as.Date("2025-01-01"), end_date = as.Date("2025-03-31")
)
head(activity)
```

`episodic_ui_italicise_taxon`*Format Text for Outbreak Reports and the Dashboard*

Description

Small HTML formatting helpers used when building dossier text, both in the dashboard and in the Quarto outbreak report template. Both are exported so that a custom report template of your own (set via `EPISODIC_QUARTO_REPORT`) can use the same formatting as the shipped one. Input text is always HTML-escaped first, so the result is safe to pass on to `shiny::HTML()`.

Usage

```
episodic_ui_italicise_taxon(pathogen)

episodic_ui_code_join(detectors, sep = ", ")
```

Arguments

<code>pathogen</code>	A character vector of pathogen display names.
<code>detectors</code>	A character vector of detector names.
<code>sep</code>	Separator between entries.

Details

`episodic_ui_italicise_taxon()` italicises pathogen names that AMR recognises as a taxonomic binomial (e.g. *Escherichia coli*), following standard microbiological convention. Names it does not recognise (e.g. "Influenza A", a virus type rather than a species) are left as-is.

`episodic_ui_code_join()` renders a vector of detector names (e.g. "farrington", "same_place") as inline `<code>`, joined into one string - useful when listing which detectors flagged a cluster.

Value

A character vector (or, for `episodic_ui_code_join()`, a single string), safe to pass to `shiny::HTML()`.

Examples

```
episodic_ui_italicise_taxon(c("Escherichia coli", "Influenza A"))
episodic_ui_code_join(c("farrington", "same_place"))
```

episodic_validate_comparator
<i>Measure a Trivial Rule Against the Same Known Truth</i>

Description

A sensitivity of 0.8 and one false alarm per two thousand stream-weeks mean very little on their own. They mean a great deal next to what a rule anyone could write in an afternoon gets on the same data. This runs such a rule over exactly the same generated history, week by week, and reports it through the same matching and the same metrics as [episodic_validate_detection\(\)](#), so the two sit in one table.

Usage

```
episodic_validate_comparator(  
  method = c("same_place", "shewhart"),  
  seeds = 1,  
  end_date = Sys.Date(),  
  history_years = 1,  
  evaluation_weeks = 4,  
  min_recall = 0.5,  
  min_precision = 0.5,  
  outbreaks = TRUE,  
  outbreak_offsets = NULL,  
  n_cases = 3,  
  k_days = 7,  
  case_free_days = 14,  
  baseline_weeks = 52,  
  sd_multiplier = 2,  
  quiet = TRUE  
)
```

Arguments

method	Which rule to run.
seeds	RNG seeds to replicate over. Seeds are the replicates: one realisation of a stochastic generator is an anecdote, so the summary reports medians and interquartile ranges across them.
end_date	The last day of generated history, and the last run the replay makes. Defaults to today, which means two runs on different days are not comparable; pin it for anything whose numbers are going to be quoted.
history_years	Years of endemic history before the evaluation window, for the detectors to build a baseline from.
evaluation_weeks	Weekly runs to replay.

`min_recall, min_precision`
 The matching thresholds described above.

`outbreaks, outbreak_offsets`
 Passed to `episodic_synthetic_cases()`. `outbreaks = FALSE` generates history with nothing seeded in it, which is the negative control: every alarm raised on it is a false one.

`n_cases, k_days, case_free_days`
 The "same_place" rule's parameters: how many cases, within how many days, and how long a quiet gap ends one alarm and starts the next.

`baseline_weeks, sd_multiplier`
 The "shewhart" rule's parameters.

`quiet`
 If TRUE (the default), the per-run progress `episodic_run_cron()` writes is suppressed; a replay is hundreds of lines of it. Warnings and errors are not suppressed.

Details

Two rules are available:

"same_place" Any `n_cases` cases of one pathogen at one place within `k_days`. A place is a ward inside a hospital and the institution itself everywhere else, which is what `episodic_detect_same_place()` watches. Cases at one place more than `case_free_days` apart belong to separate alarms.

"shewhart" A plain control limit on weekly counts: the whole catchment's count for one pathogen in the week just completed, against the mean plus `sd_multiplier` standard deviations of the `baseline_weeks` weeks before it. Consecutive alarming weeks are one alarm, not one each.

Neither reconciles, scores, suppresses or ages anything, which is the point: they are the floor a four-detector design has to clear to be worth its complexity. Their false-alarm denominator is their own - places watched, or pathogens watched - and not EpiSODIC's stream count, so read denominator before comparing two rates.

Value

An `episodic_validation` object, in the same shape `episodic_validate_detection()` returns, with `meta$method` naming the rule and `meta$config_hash` absent - a rule this simple has no configuration to hash.

See Also

`episodic_validate_detection()`

Examples

```
naive <- episodic_validate_comparator()
naive$summary[naive$summary$metric == "sensitivity", ]
```

episodic_validate_detection

Measure Detection Against Known Truth

Description

The dashboard's Performance screen measures this instance against its own epidemiologists' verdicts. That is the right operational metric and a circular one for a paper: it reports what the board thought of what the system showed them, and an outbreak the detectors miss never becomes a cluster for anyone to have an opinion about, so sensitivity cannot be measured that way at all.

Usage

```
episodic_validate_detection(
  seeds = 1,
  end_date = Sys.Date(),
  history_years = 1,
  evaluation_weeks = 4,
  min_recall = 0.5,
  min_precision = 0.5,
  outbreaks = TRUE,
  outbreak_offsets = NULL,
  detectors = episodic_validation_detectors(),
  config = NULL,
  pathogen_config = NULL,
  quiet = TRUE
)

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

Arguments

seeds	RNG seeds to replicate over. Seeds are the replicates: one realisation of a stochastic generator is an anecdote, so the summary reports medians and interquartile ranges across them.
end_date	The last day of generated history, and the last run the replay makes. Defaults to today, which means two runs on different days are not comparable; pin it for anything whose numbers are going to be quoted.
history_years	Years of endemic history before the evaluation window, for the detectors to build a baseline from.
evaluation_weeks	Weekly runs to replay.
min_recall, min_precision	The matching thresholds described above.

outbreaks, outbreak_offsets	Passed to <code>episodic_synthetic_cases()</code> . <code>outbreaks = FALSE</code> generates history with nothing seeded in it, which is the negative control: every alarm raised on it is a false one.
detectors	Which detectors to switch on, for a drop-one analysis. Any of "farrington", "mem", "same_place" and "rare_trigger"; the rest are switched off in the instance configuration this writes, so <code>config_hash</code> records which ran.
config	Further instance configuration to write, as a nested list (e.g. <code>list(farrington = list(alpha = 0.01))</code>), merged over the shipped defaults and validated the same way an operator's file is. It may not set any detector's enabled key - that is what <code>detectors</code> is for, and two mechanisms for one setting is one too many.
pathogen_config	Optional per-pathogen overrides: a data frame (or a path to a CSV) with a pathogen column and any subset of the columns from <code>inst/config/episodic_default_pathogen_config</code> . Merged row-by-row over the shipped defaults the same way <code>EPIODIC_PATHOGEN_CONFIG</code> is.
quiet	If TRUE (the default), the per-run progress <code>episodic_run_cron()</code> writes is suppressed; a replay is hundreds of lines of it. Warnings and errors are not suppressed.
x	An <code>episodic_validation</code> object.
...	Ignored.

Details

This measures the detectors against outbreaks that are known to be there because they were put there. It generates synthetic history with `episodic_synthetic_cases()`, replays it week by week through `episodic_run_cron()` against a throwaway database, and compares the clusters that came out against the ground truth that went in.

Value

An `episodic_validation` object: a list of

outbreaks One row per (seed, seeded outbreak): whether it was detected, by which cluster and which detector, how much of it that cluster held, how late (three ways - see below), and how much of it was still to come at that moment.

clusters One row per (seed, cluster raised): its case-set precision, whether it counts as a true positive, its priority score, and which detectors fired on it.

runs One row per (seed, run): streams watched, clusters raised, and how many of those survived lattice suppression.

overlap Every (run, cluster, outbreak) that shared a case, which is the raw material both tables above are derived from.

truth What was injected: the outbreak table and the case-level membership, per seed. Kept with the result so the matching thresholds can be varied afterwards (`episodic_validate_rethreshold()`) without replaying anything.

`time_to_detection` The Kaplan-Meier curve of days to detection over the outbreaks that began inside the evaluation window, with the ones nothing found kept in as right-censored observations rather than dropped.

`summary` The headline numbers, one row per metric and group.

`meta` The package version, the resolved `config_hash`, the thresholds, the seeds, and the denominators every figure rests on.

How a cluster is matched to an outbreak

On the cases themselves, in both directions, never on overlapping dates. An endemic winter cluster spanning the same fortnight as a seeded outbreak overlaps it perfectly in time and shares not one case with it.

- An **outbreak is detected** when some cluster's final membership holds at least `min_recall` of the outbreak's cases.
- A **cluster is a true positive** when at least `min_precision` of its own cases belong to one seeded outbreak. Anything else is a false alarm.

Both thresholds are arguments because both are choices, and a result that only holds at one particular pair of values is not a result. Sweep them and see (`episodic_validate_rethreshold()` does it without replaying anything; `data-raw/validation/` does it over a grid).

Three delays, not one

The day something first appeared on the board that turned out to be this outbreak (`delay_from_open`), the day what was on the board was mostly this outbreak (`delay_from_first`), and the day it held most of the whole outbreak, cases still to come included, are three different claims. They can be days apart. All three are reported, because picking one and calling it "the delay" would be choosing a number rather than measuring one.

Why it replays week by week

A single run over the whole history says nothing about timeliness: it sees the outbreak's last case at the same moment as its first. So the replay steps `run_date` forward one week at a time and hands each run only the cases sampled on or before that date - what a laboratory extract taken that day would actually have contained.

Two consequences worth knowing before reading any number out of this. Farrington needs $(\text{farrington\$b} + 1) * 52$ weeks of history and MEM needs two seasons, so an evaluation window that starts before the baseline requirement is met measures the warm-up rather than the method: keep `history_years` at 4 or more if the statistical detectors are meant to be running. And all six seeded outbreaks are anchored to the end of the generated window unless `outbreak_offsets` disperses them, so a prospective evaluation wants them dispersed.

Cost

Every run refits Farrington for every eligible stream, so the cost is roughly (seeds x weeks) full detection runs. The defaults are deliberately small - one seed, a short window, a single year of history - so this returns in seconds and can be run from an example. They are a demonstration of the machinery, not a study: at one year of history the statistical detectors have no baseline to fit against

and report nothing, and one seed is one realisation. The study lives in `data-raw/validation/`, is measured in minutes to hours, and is not run by the test suite, by R CMD check, or by anything a user installs.

See Also

[episodic_synthetic_ground_truth\(\)](#) for what is being measured against, and [episodic_run_cron\(\)](#) for the detection run this replays.

Examples

```
result <- episodic_validate_detection()
result$summary[result$summary$metric == "sensitivity", ]
```

episodic_validate_rethreshold

Re-Match a Validation Result at Different Thresholds

Description

`min_recall` and `min_precision` are choices, and a result that only holds at one particular pair of them is not a result. This applies a different pair to a replay that has already been run, so a sweep costs arithmetic rather than another few hours of detection.

Usage

```
episodic_validate_rethreshold(
  result,
  min_recall = result$meta$min_recall,
  min_precision = result$meta$min_precision
)
```

Arguments

`result` An episodic_validation object, from [episodic_validate_detection\(\)](#) or [episodic_validate_comparator\(\)](#).

`min_recall, min_precision` The thresholds to apply instead.

Details

It re-derives everything the thresholds touch and nothing else: which outbreaks count as detected, when, how badly they fragmented, and which clusters count as true positives. The clusters that were raised, the runs that raised them and the cases they held are what they were - those are measurements, and no threshold changes them.

Value

An `episodic_validation` object of the same shape, with `meta$min_recall` and `meta$min_precision` updated and `meta$rethresholded_from` recording what it was re-matched from.

See Also

[episodic_validate_detection\(\)](#)

Examples

```
result <- episodic_validate_detection()
strict <- episodic_validate_rethreshold(result, min_recall = 0.8)
strict$summary[strict$summary$metric == "sensitivity", ]
```

Index

AMR::age(), [10](#)

DBI::dbDisconnect(), [17](#), [18](#)
DBI::DBIConnection, [17](#), [18](#), [21](#), [32](#)

episodic_add_manual_cluster, [3](#)
episodic_add_manual_cluster(), [13](#)
episodic_add_user, [5](#)
episodic_add_user(), [7](#), [21](#)
episodic_auth, [7](#)
episodic_case_data, [7](#), [12](#), [13](#), [24](#), [25](#),
[35–37](#), [39](#)
episodic_charts, [11](#)
episodic_check_cases, [12](#)
episodic_check_cases(), [10](#), [14](#), [15](#), [24](#), [25](#),
[36](#), [37](#), [40–42](#)
episodic_check_denominators, [14](#)
episodic_check_institution_activity,
[15](#)
episodic_config_export, [16](#)
episodic_db_connect, [17](#)
episodic_db_connect(), [18](#), [21](#)
episodic_db_create, [18](#)
episodic_db_create(), [17](#), [22](#)
episodic_db_dsn_mariadb, [18](#)
episodic_db_dsn_mariadb(), [3](#), [6](#), [16–18](#),
[20–22](#), [33](#), [35](#)
episodic_db_dsn_mysql
(episodic_db_dsn_mariadb), [18](#)
episodic_db_migrate, [19](#)
episodic_db_migrate(), [17](#), [21](#), [36](#)
episodic_db_open, [21](#)
episodic_db_truncate, [22](#)
episodic_demo, [23](#)
episodic_demo(), [6](#), [34](#), [39](#)
episodic_geo, [26](#)
episodic_geo_overlay_resolve
(episodic_geo), [26](#)
episodic_geo_overlay_resolve(), [26](#)
episodic_geo_source_resolve
(episodic_geo), [26](#)
episodic_mail_template_preview, [27](#)
episodic_notifications, [28](#), [37](#), [38](#)
episodic_notify_test, [29](#)
episodic_palette, [29](#)
episodic_palette(), [32](#)
episodic_report_render, [31](#)
episodic_report_render(), [16](#), [30](#), [37](#)
episodic_run_app, [33](#)
episodic_run_app(), [23](#), [24](#)
episodic_run_cron, [34](#)
episodic_run_cron(), [3](#), [7](#), [10](#), [13](#), [15](#), [20](#),
[22](#), [24](#), [25](#), [28](#), [33](#), [37](#), [38](#), [43](#), [45](#), [48](#),
[50](#), [52](#)
episodic_scheduled_reports, [37](#)
episodic_setup_microsoft365, [38](#)
episodic_synthetic_cases, [39](#)
episodic_synthetic_cases(), [8](#), [24](#), [41–45](#),
[48](#), [50](#)
episodic_synthetic_cases_calibration,
[41](#)
episodic_synthetic_cases_calibration(),
[41](#)
episodic_synthetic_denominators, [42](#)
episodic_synthetic_denominators(), [8](#),
[14](#), [35](#)
episodic_synthetic_ground_truth, [43](#)
episodic_synthetic_ground_truth(), [40](#),
[41](#), [52](#)
episodic_synthetic_institution_activity,
[44](#)
episodic_synthetic_institution_activity(),
[15](#), [35](#)
episodic_ui_code_join
(episodic_ui_italicise_taxon),
[46](#)
episodic_ui_epi_curve_chart
(episodic_charts), [11](#)

`episodic_ui_italicise_taxon`, [46](#)
`episodic_ui_rt_chart` (`episodic_charts`),
 [11](#)
`episodic_ui_trend_chart`
 (`episodic_charts`), [11](#)
`episodic_validate_comparator`, [47](#)
`episodic_validate_comparator()`, [52](#)
`episodic_validate_detection`, [49](#)
`episodic_validate_detection()`, [43](#), [44](#),
 [47](#), [48](#), [52](#), [53](#)
`episodic_validate_rethreshold`, [52](#)
`episodic_validate_rethreshold()`, [50](#), [51](#)

`ggplot2::ggplot`, [11](#), [12](#)
`ggplot2::ggsave()`, [11](#)

`interactive()`, [22](#)

`logical`, [13](#)

`print.episodic_validation`
 (`episodic_validate_detection`),
 [49](#)

`shiny::HTML()`, [46](#)
`shiny::runApp()`, [34](#)