Package ‘zzedc’

December 18, 2025

Type Package

Title Electronic Data Capture System for Clinical Trials

Version 1.0.0

Author Ronald G. Thomas [aut, cre]

Maintainer Ronald G. Thomas <rgthomas@ucsd.edu>

Description A comprehensive 'Shiny' application for electronic data capture (EDC) in clinical trials. Features include secure user authentication, data entry forms, quality control reports, data visualization, and flexible export capabilities. Built with modern 'bslib' components for responsive design and professional appearance. Supports the zzcollab framework for clinical research workflows.

License GPL-3 + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Depends R (>= 4.0.0), zzedc.validation (>= 1.0.0)

Imports methods, shiny (>= 1.7.0), bslib (>= 0.4.0), bsicons (>= 0.1.0), shinyjs (>= 2.1.0), DT (>= 0.20), ggplot2 (>= 3.4.0), plotly (>= 4.10.0), dplyr (>= 1.0.0), jsonlite (>= 1.7.0), digest (>= 0.6.0), writexl (>= 1.4.0), RSQLite (>= 2.2.0), pool (>= 0.1.6), config (>= 0.3.1), lubridate (>= 1.8.0), stringr (>= 1.4.0), httr (>= 1.4.0), shinyalert (>= 3.0.0), googlesheets4 (>= 1.0.0)

Suggests testthat (>= 3.0.0), knitr, rmarkdown, here, yaml, shinyTime, shinyWidgets, shinysignature, haven

URL https://github.com/rgt47/zzedc

BugReports https://github.com/rgt47/zzedc/issues

Config/testthat/edition 3

VignetteBuilder knitr

Contents

`zzedc-package` *ZZedc: Electronic Data Capture System for Clinical Trials*

Description

The zzedc package provides a comprehensive Shiny application for electronic data capture (EDC) in clinical trials. It features secure authentication, data entry forms, quality control reports, visualization tools, and export capabilities, all built with modern bslib components.

Details

Key Features:

Authentication & Security:

- Role-based user authentication
- Encrypted data storage
- Audit trail capabilities

Data Entry:

- Customizable data entry forms
- Real-time validation
- Progress tracking

Reporting:

- Basic data summaries
- Data quality reports
- Statistical analysis reports

Data Management:

- Interactive data explorer
- Missing data analysis
- Data visualization tools

Export & Integration:

- Multiple export formats (CSV, Excel, JSON, PDF, HTML)
- Batch export capabilities
- Export templates and scheduling

Getting Started:

To launch the EDC application:

```
library(zzedc)
launch_zzedc()
```

The application will open in your default web browser with a modern, responsive interface powered by Bootstrap 5.

Default Credentials:

For testing purposes, the following credentials are available:

- Username: `ww`, Password: `pw`
- Username: `q`, Password: `pw`
- Username: `w`, Password: `pw`

Note: Change these credentials before deploying to production.

Package Dependencies

This package builds on several excellent R packages including shiny, bslib, DT, ggplot2, plotly, and others to provide a comprehensive EDC solution.

Author(s)

Ronald G. Thomas

See Also

Useful links:

- `https://github.com/rgt47/zzedc`
- Report bugs at `https://github.com/rgt47/zzedc/issues`

check_aws_kms_status *Check AWS KMS Status and Permissions*

Description

Provides comprehensive diagnostic information about AWS KMS setup, credentials, region configuration, and IAM permissions. Useful for troubleshooting setup issues.

Usage

```
check_aws_kms_status()
```

Details

Performs comprehensive AWS KMS diagnostic checks:

1. Credentials: Checks if AWS credentials are available and valid
2. Region: Detects AWS region from environment or config
3. Identity: Retrieves AWS caller identity (account, user/role)
4. Connectivity: Tests connection to Secrets Manager
5. Secret: Checks if default secret exists
6. Permissions: Tests each required IAM permission
7. Recommendations: Suggests fixes for any issues found

Value

List with detailed AWS KMS status:

- configured: Logical TRUE if fully configured
- region: AWS region being used
- credentials_available: Logical TRUE if AWS credentials found
- sts_identity: AWS caller identity (account, user/role, ARN)
- secret_manager_access: Logical TRUE if can reach Secrets Manager
- default_secret_exists: Logical TRUE if "zzedc/db-encryption-key" exists
- permissions: List with individual permission test results
- recommendations: Character vector of setup recommendations
- errors: Character vector of any errors encountered
- status_message: Human-readable overall status

Examples

```
## Not run:
  status <- check_aws_kms_status()

  if (status$configured) {
    cat("AWS KMS fully configured\n")
  } else {
    cat("Issues found:\n")
    cat("Errors:", status$errors, "\n")
  }

## End(Not run)
```

`complete_wizard_setup` *Complete Setup Wizard Orchestration*

Description

Orchestrates all setup steps for initializing a new ZZedc instance

Usage

```
complete_wizard_setup(config_list, base_path = "~/zzedc_instance")
```

Arguments

`config_list`	Complete configuration from wizard
`base_path`	Base directory for new installation

Value

List with overall success status and detailed results

`connect_encrypted_db` *Connect to Encrypted Database*

Description

Main wrapper function for encrypted database connections. Transparently handles encryption at the connection layer.

Usage

```
connect_encrypted_db(db_path = NULL, aws_kms_key_id = NULL)
```

Arguments

`db_path`	Character: Path to database file (optional, uses get_db_path if NULL)
`aws_kms_key_id`	Character: AWS KMS key ID for production (optional)

Details

This function:

1. Gets database path (from parameter or environment)
2. Retrieves encryption key (from environment or AWS KMS)
3. Connects to SQLite with encryption key
4. Returns standard DBI connection object

Encryption is transparent - all existing SQL queries work unchanged.

Value

DBI SQLite connection object with encryption enabled

Examples

```
## Not run:
  # Development (environment variable):
  Sys.setenv(DB_ENCRYPTION_KEY = "a1b2c3d4...")
  conn <- connect_encrypted_db()

  # Production (AWS KMS):
  conn <- connect_encrypted_db(aws_kms_key_id = "arn:aws:kms:...")

  # Use connection normally
  result <- DBI::dbGetQuery(conn, "SELECT * FROM subjects")
  DBI::dbDisconnect(conn)

## End(Not run)
```

`create_audit_report` *Create Audit Report*

Description

Generate compliance audit report for specified period.

Usage

```
create_audit_report(
  start_date,
  end_date,
  report_type = "summary",
  db_path = NULL
)
```

Arguments

`start_date`	Date: Report start date
`end_date`	Date: Report end date
`report_type`	Character: Type of report ("summary", "detailed", "compliance")
`db_path`	Character: Database path (optional)

Value

List with report data and statistics

Examples

```
## Not run:
  report <- create_audit_report(
    start_date = Sys.Date() - 30,
    end_date = Sys.Date(),
    report_type = "summary"
  )

## End(Not run)
```

`create_error_display` *Display form validation errors to user*

Description

Creates user-friendly error messages from validation results

Usage

```
create_error_display(validation_result)
```

Arguments

`validation_result`	Result from validate_form()

Value

HTML list of error messages

`create_export_manifest` *Create Export Manifest*

Description

Creates JSON manifest with export metadata and verification information.

Usage

```
create_export_manifest(export_file_path, metadata = list())
```

Arguments

`export_file_path`	Character: Path to exported data file
`metadata`	List: Additional metadata to include

Details

Manifest includes:

- File metadata (size, hash, row/column counts)
- Export metadata (who, when, why)
- Verification information (hash for integrity checking)
- Confidentiality classification

Value

Character string with path to manifest file

Examples

```
## Not run:
  manifest_path <- create_export_manifest(
    export_file_path = "./exports/export_subjects_20251218_123456.csv",
    metadata = list(
      study_id = "TOY-TRIAL-001",
      export_reason = "DSMB Review",
      exported_by = "john_doe"
    )
  )

## End(Not run)
```

`create_launch_script` *Create Launch Script for New ZZedc Instance*

Description

Creates a customized launch script file that users can run to start the application

Usage

```
create_launch_script(config_list, output_path)
```

Arguments

`config_list`	Configuration from wizard
`output_path`	Path where launch script will be written

Value

List with success status

`create_paginated_reactive` *Create reactive paginated data*

Description

Returns reactive expression that manages paginated data view

Usage

```
create_paginated_reactive(
  data_source,
  page_size = 25,
  search_reactive = NULL,
  sort_reactive = NULL
)
```

Arguments

`data_source`	Reactive data.frame
`page_size`	Number of rows per page
`search_reactive`	Optional reactive search term
`sort_reactive`	Optional reactive sort specification (list with $by and $direction)

Value

Reactive expression returning list with paginated data and metadata

`create_pagination_ui` *Create pagination UI controls*

Description

Generates navigation buttons for pagination

Usage

```
create_pagination_ui(pagination, input_id = "data")
```

Arguments

`pagination`	Pagination info from paginate_data()
`input_id`	Namespace ID for pagination inputs

Value

HTML div with pagination controls

`create_wizard_config` *Create Config File from Wizard Configuration*

Description

Creates configuration file with all required application settings

Usage

```
create_wizard_config(config_list, config_path, security_salt)
```

Arguments

`config_list`	List containing wizard configuration
`config_path`	Path where config.yml will be written
`security_salt`	The security salt for hashing

Value

List with success status and messages

`create_wizard_database` *Create ZZedc Database from Wizard Configuration*

Description

Creates a complete database with all required tables

Usage

```
create_wizard_database(config_list, db_path)
```

Arguments

`config_list`	List containing wizard configuration (from wizard_state$system_config)
`db_path`	Path where database file will be created

Value

List with success status and messages

Examples

```
## Not run:
config <- list(
  study_name = "My Study",
  protocol_id = "PROTO-001",
  admin_username = "admin",
  admin_password = "MyPass123!",
  security_salt = "abc123..."
)
create_wizard_database(config, "~/my_study.db")

## End(Not run)
```

`create_wizard_directories` *Create Directories for New ZZedc Instance*

Description

Creates the directory structure needed for a new ZZedc installation

Usage

```
create_wizard_directories(base_path)
```

Arguments

`base_path`	Base directory where subdirectories will be created

Value

List with success status

`detect_setup_status` *Detect Configuration Status*

Description

Comprehensive check of setup status

Usage

```
detect_setup_status()
```

Value

List with status information

`enable_session_timeout` *Enable session timeout monitoring*

Description

Should be called in server() function to activate timeout checking. Monitors user inactivity and logs out after configured timeout period.

Usage

```
enable_session_timeout(
  session,
  user_input,
  timeout_config,
  on_timeout_callback = NULL
)
```

Arguments

`session`	Shiny session object
`user_input`	reactiveValues object containing user session state
`timeout_config`	List with timeout_minutes (from config$auth$session_timeout_minutes)
`on_timeout_callback`	Function to call when timeout occurs (default: logs out user)

`error_response` *Create standardized error response*

Description

Returns a consistent error response structure.

Usage

```
error_response(message, code = NULL)
```

Arguments

`message`	Character - error message
`code`	Character - error code (optional)

Value

List with success=FALSE and message

`execute_init_setup` *Execute Initialization Setup*

Description

Common code for both interactive and config modes

Usage

```
execute_init_setup(config, project_dir = ".")
```

Arguments

`config`	Configuration list
`project_dir`	Directory where project will be created

Value

List with setup results

`export_audit_log` *Export audit log to file*

Description

Saves audit log to CSV with hash verification included.

Usage

```
export_audit_log(audit_log, filepath, include_verification = TRUE)
```

Arguments

`audit_log`	reactiveVal or data.frame containing audit records
`filepath`	Character - path to save audit log
`include_verification`	Logical - include verification summary?

`export_audit_report` *Export Audit Report*

Description

Export audit report to file for compliance documentation.

Usage

```
export_audit_report(
  start_date,
  end_date,
  output_file,
  format = "csv",
  db_path = NULL
)
```

Arguments

`start_date`	Date: Report start date
`end_date`	Date: Report end date
`output_file`	Character: Path for output file
`format`	Character: Export format ("csv", "xlsx", "json")
`db_path`	Character: Database path (optional)

Value

Character string with path to exported report

Examples

```
## Not run:
  file_path <- export_audit_report(
    start_date = Sys.Date() - 90,
    end_date = Sys.Date(),
    output_file = "./audit_Q1_2025.csv"
  )

## End(Not run)
```

export_encrypted_data *Export Encrypted Data with Integrity Verification*

Description

Exports query results to multiple formats with cryptographic integrity hashing.

Usage

```
export_encrypted_data(
  query,
  format = "csv",
  password = NULL,
  include_hash = TRUE,
  export_dir = "./exports",
  db_path = NULL
)
```

Arguments

`query`	Character: SQL SELECT query to export
`format`	Character: Output format ("csv", "xlsx", "json", default: "csv")
`password`	Character: Optional password for additional encryption
`include_hash`	Logical: Add SHA-256 hash file? (default: TRUE)
`export_dir`	Character: Directory for exports (default: "./exports")
`db_path`	Character: Path to database (optional, uses default if NULL)

Details

This function:

1. Connects to encrypted database
2. Executes provided SQL query
3. Generates SHA-256 hash of export content
4. Exports to requested format (CSV, XLSX, JSON)
5. Stores hash in separate verification file
6. Logs export to audit trail
7. Returns file path

Formats:

- csv: Comma-separated values (default)
- xlsx: Microsoft Excel workbook
- json: JSON array format

Hash files (.sha256) enable verification using verify_exported_data()

Value

Character string with path to exported file

Examples

```
## Not run:
  # Export all subjects
  file_path <- export_encrypted_data(
    query = "SELECT * FROM subjects",
    format = "csv"
  )

  # Export with password protection
  file_path <- export_encrypted_data(
    query = "SELECT * FROM mmse_assessments WHERE visit_label = 'Baseline'",
    format = "xlsx",
    password = "secure_password"
  )

## End(Not run)
```

`export_to_file` *Export data to file*

Description

Writes export data to specified file format. Supports 9 formats: CSV, XLSX, JSON, SAS, SPSS, STATA, RDS, PDF, HTML

Usage

```
export_to_file(data, filepath, format, options = NULL)
```

Arguments

`data`	Data to export (data.frame or list)
`filepath`	Path to write export file
`format`	Export format (csv, xlsx, json, sas, spss, stata, rds, pdf, html)
`options`	List of format-specific options

Value

List with success status and file info

`filter_data_by_search` *Filter data by search term*

Description

Searches all columns for matching values

Usage

```
filter_data_by_search(data, search_term, columns = NULL)
```

Arguments

`data`	data.frame to search
`search_term`	Text to search for (case-insensitive)
`columns`	Column names to search in (NULL = all columns)

Value

Filtered data.frame

`generate_db_key` *Generate a random database encryption key*

Description

Creates a cryptographically secure 256-bit random key for SQLCipher database encryption. Returns as a 64-character hexadecimal string.

Usage

```
generate_db_key()
```

Details

Key generation:

- Uses openssl::rand_bytes() for cryptographic security
- 256-bit key = 32 bytes = 64 hex characters
- Never user-provided (best practice: auto-generated)
- Store result in environment variable or AWS Secrets Manager

Value

Character string: 64-hex-character encryption key (256-bit)

Examples

```
## Not run:
  key <- generate_db_key()
  Sys.setenv(DB_ENCRYPTION_KEY = key)

## End(Not run)
```

---

generate_export_filename

Generate safe export filename

Description

Creates a safe, properly formatted filename for export

Usage

```
generate_export_filename(base_name = NULL, data_source, format)
```

Arguments

`base_name`	Base filename (user-provided or default)
`data_source`	Data source identifier
`format`	Export format (csv, xlsx, json, sas, spss, stata, rds, pdf, html)

Value

Safe filename with extension

---

generate_security_salt

Generate Security Salt

Description

Creates a random 32-character salt for password hashing

Usage

```
generate_security_salt()
```

Value

Character string of random salt

get_audit_trail *Get Audit Trail*

Description

Retrieve audit trail with optional filtering and verification.

Usage

```
get_audit_trail(filters = list(), include_chain = FALSE, db_path = NULL)
```

Arguments

filters	List: Filter criteria (user, table_name, event_type, date_from, date_to)
include_chain	Logical: Include hash chain info? (default: FALSE)
db_path	Character: Database path (optional)

Value

Data frame with audit trail records

Examples

```
## Not run:
  # Get all events for a user
  trail <- get_audit_trail(filters = list(user = "jane_smith"))

  # Get updates to subjects table
  trail <- get_audit_trail(
    filters = list(
      table_name = "subjects",
      event_type = "UPDATE"
    )
  )

## End(Not run)
```

get_db_path *Get Database Path*

Description

Retrieves database file path from environment or default location. Creates directory if needed.

Usage

```
get_db_path()
```

Details

Priority:

1. Environment variable ZZEDC_DB_PATH
2. Default: "./data/zzedc.db"

Directory is created automatically if it doesn't exist.

Value

Character string with absolute path to database file

Examples

```
## Not run:
  db_path <- get_db_path()
  # Returns: "/path/to/data/zzedc.db"

## End(Not run)
```

`get_encryption_key` *Get database encryption key from environment or AWS KMS*

Description

Retrieves encryption key with automatic fallback:

1. Try AWS KMS (if credentials and key_id provided)
2. Try environment variable DB_ENCRYPTION_KEY
3. Error if neither available

Usage

```
get_encryption_key(aws_kms_key_id = NULL)
```

Arguments

`aws_kms_key_id`	Character: AWS KMS key ID (optional)

Details

Priority order:

1. AWS KMS (if aws_kms_key_id provided or USE_AWS_KMS=true)
   - Requires paws package
   - Requires AWS credentials (~/.aws/credentials or env vars)
   - Requires AWS IAM permissions for Secrets Manager
2. Environment variable DB_ENCRYPTION_KEY
   - Set with: Sys.setenv(DB_ENCRYPTION_KEY = "...")
   - Best for development
3. Error if neither available
   - Helpful error message with setup instructions

Value

Character: 64-char hex encryption key

Examples

```
## Not run:
  # Development (environment variable):
  Sys.setenv(DB_ENCRYPTION_KEY = "a1b2c3d4...")
  key <- get_encryption_key()

  # Production (AWS KMS):
  key <- get_encryption_key(aws_kms_key_id = "arn:aws:kms:...")

## End(Not run)
```

`get_encryption_key_from_aws_kms`
Retrieve encryption key from AWS Secrets Manager

Description

Retrieves stored encryption key from AWS Secrets Manager. Requires paws package and AWS credentials.

Usage

```
get_encryption_key_from_aws_kms(key_id = NULL)
```

Arguments

`key_id`	Character: AWS Secrets Manager secret name (optional)

Details

Default secret name: "zzedc/db-encryption-key"

Requirements:

- paws package installed: install.packages("paws")
- AWS credentials configured: ~/.aws/credentials or environment variables
- AWS IAM permissions: secretsmanager:GetSecretValue

Value

Character: Decrypted encryption key (64 hex chars)

Examples

```
## Not run:
  # Requires AWS credentials
  key <- get_encryption_key_from_aws_kms("zzedc/db-encryption-key")

## End(Not run)
```

get_export_history *Get Export Activity History*

Description

Retrieves audit trail of all data exports with optional filtering.

Usage

```
get_export_history(filters = list())
```

Arguments

filters	List: Filter criteria (date_from, date_to, user, format, status)

Details

Returns data frame with columns:

- export_id: Unique export identifier
- export_date: When export occurred
- user_id: Who performed export
- file_path: Location of exported file
- query: SQL query that was exported
- format: Export format used
- file_size: Size of exported file
- hash_verified: Whether hash verification was done
- status: Export status (success/failure)

Value

Data frame with export history records

Examples

```
## Not run:
  # Get all exports in last 7 days
  history <- get_export_history(
    filters = list(date_from = Sys.Date() - 7)
  )

  # Get exports by user
  history <- get_export_history(
    filters = list(user = "jane_smith")
  )

## End(Not run)
```

get_instrument_field *Get instrument field by name*

Description

Retrieves a single field definition from an instrument.

Usage

```
get_instrument_field(
  instrument_name,
  field_name,
  instruments_dir = "instruments/"
)
```

Arguments

`instrument_name`	Name of instrument
`field_name`	Name of field to retrieve
`instruments_dir`	Path to instruments directory

Value

List with field properties, or NULL if not found

get_page_summary *Generate page summary statistics*

Description

Calculates column-wise statistics for paginated data

Usage

```
get_page_summary(page_data, numeric_cols = NULL)
```

Arguments

`page_data`	data.frame with current page data
`numeric_cols`	Column names to compute stats for

Value

data.frame with summary statistics

`get_setup_instructions` *Get Setup Instructions*

Description

Returns helpful instructions for completing setup

Usage

```
get_setup_instructions()
```

Value

Character string with setup instructions

`handle_error` *Handle errors with logging and user notification*

Description

Wraps expression evaluation with error handling, logging, and user feedback.

Usage

```
handle_error(
  expr,
  error_title = "Error",
  show_user = TRUE,
  log_file = NULL,
  return_value = NULL
)
```

Arguments

`expr`	Expression to evaluate
`error_title`	Character - title for user error message
`show_user`	Logical - show error modal to user?
`log_file`	Character - file to log errors (optional)
`return_value`	Value to return if error occurs (default: NULL)

Value

Result of expr if successful, return_value if error

Examples

```
## Not run:
result <- handle_error({
  authenticate_user(username, password)
}, error_title = "Authentication Failed")

## End(Not run)
```

---

import_instrument *Import instrument as new form*

Description

Imports a pre-built instrument into the project as a new form. Creates form record in database and returns form metadata.

Usage

```
import_instrument(
  instrument_name,
  form_name = NULL,
  form_description = NULL,
  db_conn = NULL,
  instruments_dir = "instruments/"
)
```

Arguments

instrument_name	Name of instrument to import (e.g., "phq9")
form_name	New form name (defaults to instrument name if not provided)
form_description	Description of form for display
db_conn	Database connection (RSQLite::SQLiteConnection)
instruments_dir	Path to instruments directory

Value

List containing:

- success: Logical, operation successful?
- form_id: ID of newly created form
- form_name: Name of created form
- fields_imported: Number of fields added
- message: Status message
- errors: Character vector of any errors encountered

Examples

```
## Not run:
result <- import_instrument(
  instrument_name = "phq9",
  form_name = "baseline_depression",
  form_description = "PHQ-9 administered at baseline visit",
  db_conn = conn
)

## End(Not run)
```

`init` *Initialize ZZedc Project*

Description

Create a new ZZedc project locally or on a server. Two modes available:

- Interactive: User-friendly prompts in R console (recommended for novices)
- Config: Read from YAML configuration file (recommended for DevOps/AWS)

Usage

```
init(mode = "interactive", config_file = NULL, project_dir = ".")
```

Arguments

`mode`	Character. Either "interactive" (default) or "config"
`config_file`	Character. Path to configuration YAML file (required if mode="config")
`project_dir`	Character. Directory where project will be created (default: current directory)

Details

Interactive Mode:
Guides users through setup with prompts in the R console:

```
zzedc::init()
# Will ask for:
# - Study name
# - Protocol ID
# - PI information
# - Admin account details
# - Security settings
```

Config File Mode:
Reads configuration from YAML file (non-interactive):

```
zzedc::init(mode = "config", config_file = "zzedc_config.yml")
# Silently creates project from config
# Useful for automation, Docker, AWS
```

Value

Invisibly returns list with setup results and project location

Examples

```
## Not run:
# Interactive mode (novice user)
zzedc::init()

# Config file mode (DevOps)
zzedc::init(mode = "config", config_file = "aws_config.yml")

## End(Not run)
```

---

initialize_encrypted_database

Initialize Encrypted Database

Description

Creates a new encrypted database with complete schema.

Usage

```
initialize_encrypted_database(db_path = NULL, overwrite = FALSE)
```

Arguments

`db_path`	Character: Path for new database (optional, uses get_db_path if NULL)
`overwrite`	Logical: Overwrite existing database? (default: FALSE)

Details

This function:

1. Checks if database exists (fails if overwrite=FALSE)
2. Generates random 256-bit encryption key
3. Creates encrypted database connection
4. Creates base tables (study_info, subjects, etc.)
5. Stores encryption key in environment variable
6. Verifies encryption is working

Value

List with initialization results:

- success: Logical TRUE if successful
- path: Absolute path to created database
- key_stored: Logical TRUE if encryption key stored
- message: Status message

Examples

```
## Not run:
  result <- initialize_encrypted_database(
    db_path = "./data/new_study.db",
    overwrite = FALSE
  )
  if (result$success) {
    cat("Database created at:", result$path, "\n")
  }

## End(Not run)
```

`init_audit_log` *Initialize audit log*

Description

Creates a reactive audit log storage with immutable properties.

Usage

```
init_audit_log()
```

Value

reactiveVal containing tibble of audit records

Examples

```
## Not run:
audit_log <- init_audit_log()
log_audit_event(audit_log, "user1", "LOGIN", "authentication", status = "success")

## End(Not run)
```

`init_audit_logging` *Initialize Audit Logging System*

Description

Creates database tables and indexes for comprehensive audit logging.

Usage

```
init_audit_logging(db_path = NULL)
```

Arguments

`db_path`	Character: Database path (optional, uses default if NULL)

Details

Creates three tables:

1. audit_log - Main audit trail with all operations
2. audit_events - Specific event types and details
3. audit_chain - Hash-chained records for tamper detection

Implements hash-chaining where each record's hash depends on the previous record's hash, making tampering detectable.

Value

List with initialization results:

- success: Logical TRUE if successful
- tables_created: Number of tables created
- message: Status message

Examples

```
## Not run:
  result <- init_audit_logging()
  if (result$success) {
    cat("Audit logging initialized\n")
  }

## End(Not run)
```

`init_from_config` *Config File Mode: Non-Interactive Setup*

Description

Config File Mode: Non-Interactive Setup

Usage

```
init_from_config(config_file, project_dir = ".")
```

Arguments

`config_file`	Path to YAML configuration file
`project_dir`	Directory where project will be created

Value

List with setup results

`init_interactive` *Interactive Mode: Guided Setup*

Description

Interactive Mode: Guided Setup

Usage

```
init_interactive(project_dir = ".")
```

Arguments

`project_dir`	Directory where project will be created

Value

List with setup results

`init_session_timeout` *Initialize session timeout tracking*

Description

Creates reactive values to track user activity and session state.

Usage

```
init_session_timeout()
```

Value

List containing reactive objects for session management

`is_configured` *Check if ZZedc is Already Configured*

Description

Detect if ZZedc is being launched for the first time by checking for required configuration files. Returns TRUE if the system is fully configured.

Usage

```
is_configured(db_path = "./data/zzedc.db", config_path = "./config.yml")
```

Arguments

`db_path`	Path to database file
`config_path`	Path to config file

Value

Logical. TRUE if fully configured, FALSE otherwise

`launch_setup_if_needed` *Launch Setup Mode if Needed*

Description

Checks if ZZedc is configured. If not, shows setup options. Called from app startup to intercept first-time users.

Usage

```
launch_setup_if_needed(
  db_path = "./data/zzedc.db",
  config_path = "./config.yml"
)
```

Arguments

`db_path`	Path to database file
`config_path`	Path to config file

Value

Invisibly returns setup status

`launch_zzedc` *Launch the ZZedc Shiny Application*

Description

This function launches the interactive 'Shiny' application for electronic data capture (EDC) in clinical trials.

Usage

```
launch_zzedc(..., launch.browser = TRUE, host = "127.0.0.1", port = NULL)
```

Arguments

`...`	Additional arguments passed to `runApp`
`launch.browser`	Logical, whether to launch the app in browser. Default is `TRUE`.
`host`	Character string of IP address to listen on. Default is "127.0.0.1".
`port`	Integer specifying the port to listen on. Default is `NULL` (random port).

Details

The application provides comprehensive electronic data capture for clinical trials with the following features:

- Secure user authentication with role-based access
- Data entry forms with validation and quality control
- Comprehensive reporting system (basic, quality, statistical)
- Advanced data exploration and visualization tools
- Flexible export capabilities with multiple formats
- Modern responsive design using Bootstrap 5 via bslib

Value

No return value, launches the Shiny application

Examples

```
## Not run:
# Launch the application
launch_zzedc()

# Launch on specific port
launch_zzedc(port = 3838)

# Launch without opening browser
launch_zzedc(launch.browser = FALSE)

## End(Not run)
```

`list_available_instruments`
List available instruments

Description

Returns names and metadata of all available pre-built instruments.

Usage

```
list_available_instruments(instruments_dir = "instruments/")
```

Arguments

`instruments_dir`	Path to instruments directory (default: "instruments/")

Value

data.frame with columns:

- name: Instrument ID (e.g., "phq9")
- full_name: Full instrument name
- items: Number of items
- description: Brief description

Examples

```
## Not run:
available <- list_available_instruments()
print(available)

## End(Not run)
```

`load_instrument_template` *Load instrument template from CSV*

Description

Loads a pre-built instrument template and returns as data.frame with validated structure.

Usage

```
load_instrument_template(instrument_name, instruments_dir = "instruments/")
```

Arguments

`instrument_name`	Name of instrument (e.g., "phq9", "gad7")
`instruments_dir`	Path to instruments directory

Value

data.frame with columns:

- field_name: Unique field identifier
- field_label: User-facing label
- field_type: Input type (text, numeric, select, etc.)
- validation_rules: JSON string with validation constraints
- description: Item description/instruction text
- required: Logical, is field required?

Examples

```
## Not run:
phq9_fields <- load_instrument_template("phq9")
head(phq9_fields)

## End(Not run)
```

`log_audit_event` *Log an audit event*

Description

Records an audit event with cryptographic chaining to previous record.

Records a database operation to the immutable audit trail.

Usage

```
log_audit_event(
  event_type,
  table_name,
  record_id = NULL,
  operation,
  details = NULL,
  user_id = NULL,
  db_path = NULL
)

log_audit_event(
  event_type,
  table_name,
  record_id = NULL,
  operation,
  details = NULL,
  user_id = NULL,
  db_path = NULL
)
```

Arguments

`event_type`	Character: Type of event (INSERT, UPDATE, DELETE, SELECT, EXPORT, etc.)
`table_name`	Character: Table affected
`record_id`	Character: Record ID (optional)
`operation`	Character: Description of operation
`details`	Character: Additional details (JSON format)
`user_id`	Character: User ID (optional)
`db_path`	Character: Database path (optional)
`audit_log`	reactiveVal object (from init_audit_log)

`action`	Character - action type (e.g., "LOGIN", "DATA_EXPORT", "FORM_SUBMISSION")
`resource`	Character - what was affected (e.g., "authentication", "subject_123", "report_export")
`old_value`	Character - previous value (for modifications)
`new_value`	Character - new value (for modifications)
`status`	Character - success/failure
`error_message`	Character - error details if status == "failure"

Details

Each audit event is:

1. Assigned unique audit_id
2. Timestamped with timezone
3. Hashed using SHA-256
4. Linked to previous record (hash-chaining)
5. Made immutable by hash dependency

Value

Invisibly returns the new record (including hash)

Logical TRUE if successfully logged

Examples

```
## Not run:
audit_log <- init_audit_log()
log_audit_event(
  audit_log,
  user_id = "john.doe",
  action = "LOGIN_ATTEMPT",
  resource = "authentication",
  status = "success"
)

## End(Not run)
## Not run:
  log_audit_event(
    event_type = "INSERT",
    table_name = "subjects",
    record_id = "S001",
    operation = "New subject enrolled",
    details = '{"age": 65, "gender": "M"}',
    user_id = "jane_smith"
  )

## End(Not run)
```

`log_export_activity` *Log Export Activity (Internal)*

Description

Records export activity to audit trail database table.

Usage

```
log_export_activity(file_path, query, format, hash_verified)
```

`log_export_event` *Log export event to audit trail*

Description

Records data export with details for compliance audit

Usage

```
log_export_event(user_id, data_source, format, rows, audit_log)
```

Arguments

`user_id`	User performing export
`data_source`	Source of exported data
`format`	Export format
`rows`	Number of rows exported
`audit_log`	Audit log reactiveVal

`notify_if_invalid` *Validate and notify*

Description

Checks a condition and shows notification if false.

Usage

```
notify_if_invalid(condition, message, type = "warning")
```

Arguments

`condition`	Logical - condition to check
`message`	Character - message to show if condition is FALSE
`type`	Character - notification type ("message", "warning", "error")

Value

Invisibly returns the condition

`paginate_data` *Create paginated data view*

Description

Prepares data for paginated display with server-side processing. Handles filtering, sorting, and pagination efficiently.

Usage

```
paginate_data(
  data,
  page_size = 25,
  search_term = NULL,
  sort_by = NULL,
  sort_direction = "asc",
  page_number = 1
)
```

Arguments

`data`	data.frame to paginate
`page_size`	Number of rows per page (default: 25)
`search_term`	Optional text to filter rows
`sort_by`	Column name to sort by
`sort_direction`	"asc" or "desc"
`page_number`	Current page (1-indexed)

Value

List containing:

- data: data.frame with rows for current page
- pagination: list with page info (total_pages, total_rows, current_page)
- summary: summary statistics

Examples

```
## Not run:
paginated <- paginate_data(
  large_dataset,
  page_size = 25,
  page_number = 1
)
display_data(paginated$data)
show_page_numbers(paginated$pagination$total_pages)

## End(Not run)
```

`prepare_all_files_export` *Prepare all files data for export*

Description

Prepare all files data for export

Usage

```
prepare_all_files_export(options = NULL)
```

Arguments

`options`	List of export options

Value

List of file data

`prepare_edc_export` *Prepare EDC data for export*

Description

Retrieves EDC data with optional metadata and filtering

Usage

```
prepare_edc_export(db_conn, options = NULL)
```

Arguments

`db_conn`	Database connection
`options`	List with include_metadata, include_timestamps, date_range, etc.

Value

data.frame with EDC export data

`prepare_export_data` *Prepare data for export*

Description

Retrieves and formats data based on export configuration. Pure function with no Shiny dependencies.

Usage

```
prepare_export_data(data_source, format, options = NULL, db_conn = NULL)
```

Arguments

`data_source`	Character indicating data source ("edc", "all_files", "reports", "sample")
`format`	Character specifying export format: • "csv": Comma-separated values • "xlsx": Excel workbook • "json": JSON format • "sas": SAS transport file (.xpt) • "spss": SPSS/PSPP format (.sav) • "stata": Stata format (.dta) • "rds": R serialized object (.rds) • "pdf": PDF document (requires template) • "html": HTML document (requires template)
`options`	List of export options (metadata, timestamps, date_range, etc.)
`db_conn`	Database connection (if data_source == "edc")

Value

List containing:

- data: data.frame or list with export data
- info: metadata about export (rows, columns, size estimate)
- warnings: any issues encountered

Examples

```
## Not run:
export_result <- prepare_export_data(
  data_source = "edc",
  format = "csv",
  options = list(include_metadata = TRUE, include_timestamps = TRUE)
)

## End(Not run)
```

`prepare_reports_export` *Prepare reports data for export*

Description

Prepare reports data for export

Usage

```
prepare_reports_export(options = NULL)
```

Arguments

`options`	List of export options

Value

List of report data

`prepare_sample_export` *Prepare sample data for export*

Description

Prepare sample data for export

Usage

```
prepare_sample_export(options = NULL)
```

Arguments

`options`	List of export options

Value

data.frame with sample data

`query_audit_log` *Query audit log*

Description

Filter audit log by various criteria.

Usage

```
query_audit_log(
  audit_log,
  user_id = NULL,
  action = NULL,
  resource = NULL,
  start_date = NULL,
  end_date = NULL
)
```

Arguments

`audit_log`	reactiveVal or data.frame containing audit records
`user_id`	Character - filter by user (optional)
`action`	Character - filter by action type (optional)
`resource`	Character - filter by resource (optional)
`start_date`	Date - start of date range (optional)
`end_date`	Date - end of date range (optional)

Value

Filtered data.frame

`renderPanel` *Render form panel with typed input fields*

Description

Generates appropriate input controls based on field metadata. Supports 15+ field types including text input, numeric fields, dates, times, email, selection lists, radio buttons, checkboxes, text areas, sliders, file uploads, and digital signatures for flexible data collection.

Usage

```
renderPanel(fields, field_metadata = NULL)
```

Arguments

`fields`	Character vector of field names OR list of field configurations
`field_metadata`	List containing field definitions with type, required, choices, etc.

Details

Field metadata format: `list(age = list(type = "numeric", required = TRUE, min = 0, max = 150), email = list(type = "email", required = TRUE), treatment = list(type = "select", choices = c("A", "B", "C")), visit_date = list(type = "date", required = TRUE), visit_time = list(type = "time", required = TRUE), pain_level = list(type = "slider", min = 0, max = 10, value = 5), symptoms = list(type = "checkbox_group", choices = c("Pain", "Fever", "Cough")))`

Value

List of Shiny input controls matching field types

Examples

```
## Not run:
metadata <- list(
  age = list(type = "numeric", required = TRUE, label = "Age (years)"),
  gender = list(
    type = "select",
    choices = c("M", "F"),
    label = "Gender"
  ),
  pregnancy_date = list(
    type = "date",
    label = "Pregnancy Due Date",
    show_if = "gender == 'F'"  # Branching logic
  ),
  visit_time = list(type = "time", required = TRUE, label = "Visit Time")
)
renderPanel(names(metadata), metadata)

## End(Not run)
```

`rotate_encryption_key` *Rotate Database Encryption Key via AWS KMS*

Description

Rotates the active encryption key by archiving the old key and activating a new one. This function is used for planned key rotation in production environments.

Usage

```
rotate_encryption_key(new_key)
```

Arguments

`new_key`	Character: New 64-hex-character encryption key (from generate_db_key)

Details

Key Rotation Procedure:

1. Validate new key format (64 hex chars, lowercase)
2. Retrieve current key from AWS Secrets Manager
3. Archive current key with timestamp metadata
4. Store new key as active in AWS Secrets Manager
5. Return rotation confirmation with version IDs

After key rotation, the database must be re-encrypted with the new key.

Value

List with rotation status:

- success: Logical TRUE if rotation successful
- old_key_archived: Logical TRUE if old key saved
- new_key_active: Logical TRUE if new key now active
- timestamp: Rotation timestamp
- old_version_id: AWS version ID of archived old key
- new_version_id: AWS version ID of new key
- message: Human-readable status message
- error: Error message if rotation failed

Examples

```
## Not run:
  new_key <- generate_db_key()
  result <- rotate_encryption_key(new_key)
  if (result$success) {
    cat("Key rotation successful\n")
  }

## End(Not run)
```

`safe_reactive` *Safe reactive expression evaluation*

Description

Wraps reactive expression with error handling to prevent app crashes from reactive errors.

Usage

```
safe_reactive(expr, on_error = NULL, on_empty = NULL)
```

Arguments

`expr`	Expression to evaluate reactively
`on_error`	Function to call on error (or value to return)
`on_empty`	Function or value for empty results

Value

Reactive expression result or error value

`save_validated_form` *Save validated form data to database*

Description

Saves validated and cleaned form data with audit logging

Usage

```
save_validated_form(conn, table_name, cleaned_data, user_id, audit_log = NULL)
```

Arguments

`conn`	Database connection
`table_name`	Table to insert into
`cleaned_data`	Validated data from validate_form()
`user_id`	User submitting the form
`audit_log`	Optional audit log to record submission

Value

List with success status and record ID

`selective_field_export` *Selective Field Encryption on Export*

Description

Exports data with selective field-level encryption.

Usage

```
selective_field_export(
  data_df,
  fields_to_encrypt = NULL,
  format = "csv",
  export_dir = "./exports"
)
```

Arguments

`data_df`	Data frame: Data to export
`fields_to_encrypt`	Character vector: Column names to encrypt
`format`	Character: Output format ("csv", "xlsx", "json")
`export_dir`	Character: Directory for exports (default: "./exports")

Details

Encrypts specified columns while leaving others plaintext. Creates metadata file describing encryption scheme.

Value

Character string with path to exported file

Examples

```
## Not run:
  # Encrypt PII fields
  file_path <- selective_field_export(
    data_df = subjects_data,
    fields_to_encrypt = c("subject_id", "age"),
    format = "csv"
  )

## End(Not run)
```

`setup_aws_kms` *Setup AWS KMS Integration for ZZedc*

Description

Initializes and validates AWS KMS configuration for production key management. Checks AWS credentials, region, and permissions before returning setup status.

Usage

```
setup_aws_kms()
```

Details

AWS KMS Setup Requirements:

1. AWS credentials configured:
   - ~/.aws/credentials file, OR
   - AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables
2. AWS region configured:
   - AWS_REGION environment variable, OR

    - ~/.aws/config file with default region

3. IAM permissions required:
    - secretsmanager:CreateSecret (for initial setup)
    - secretsmanager:GetSecretValue (for key retrieval)
    - secretsmanager:PutSecretValue (for key rotation)
    - secretsmanager:DeleteSecretVersion (for archiving)

Default secret name: "zzedc/db-encryption-key"

Value

List with AWS KMS configuration status:

- aws_configured: Logical TRUE if AWS KMS properly configured
- region: AWS region (from config or env var)
- credentials_found: Logical TRUE if AWS credentials available
- secret_exists: Logical TRUE if default secret exists
- permissions: List with permission check results
- errors: Character vector of any setup errors
- message: Human-readable status message

Examples

```
## Not run:
  status <- setup_aws_kms()
  if (status$aws_configured) {
    cat("AWS KMS ready for key management\n")
  } else {
    cat("Setup errors:", status$errors, "\n")
  }

## End(Not run)
```

`setup_form_validation` *Create server-side form validation observer*

Description

Sets up reactive validation that updates UI in real-time

Usage

```
setup_form_validation(
  session,
  form_fields,
  field_metadata,
  error_container_id = "form_errors"
)
```

Arguments

`session`	Shiny session object
`form_fields`	Character vector of form field names
`field_metadata`	List with field validation rules
`error_container_id`	ID of element to display errors

`setup_pagination_observers`

Setup pagination observers

Description

Creates reactive observers to handle pagination navigation

Usage

```
setup_pagination_observers(
  session,
  data_reactive,
  current_page,
  input_id = "data"
)
```

Arguments

`session`	Shiny session object
`data_reactive`	Reactive expression returning current data
`current_page`	Reactive value holding current page number
`input_id`	Namespace ID for pagination inputs

`set_encryption_for_existing_db`

Enable Encryption on Existing Database

Description

Converts an existing unencrypted database to use encryption.

Usage

```
set_encryption_for_existing_db(db_path, new_key = NULL)
```

Arguments

`db_path`	Character: Path to existing database
`new_key`	Character: Encryption key to use (optional, generates if NULL)

Details

Process:

1. Verify database exists
2. Create backup copy
3. Generate or use provided encryption key
4. Enable encryption on database
5. Verify encryption working
6. Store key in environment or AWS KMS

Important: This enables encryption on the database file but does NOT re-encrypt existing data. New data written will be encrypted. For full re-encryption, use a database migration tool.

Value

List with encryption setup results:

- success: Logical TRUE if successful
- encrypted: Logical TRUE if now encrypted
- backup_created: Logical TRUE if backup saved
- key_stored: Logical TRUE if key securely stored
- message: Status message

Examples

```
## Not run:
  result <- set_encryption_for_existing_db(
    db_path = "./data/existing.db",
    new_key = generate_db_key()
  )
  if (result$success) {
    cat("Encryption enabled!\n")
  }

## End(Not run)
```

`sort_data` *Sort data frame*

Description

Sorts data by specified column

Usage

```
sort_data(data, sort_column, direction = "asc")
```

Arguments

`data`	data.frame to sort
`sort_column`	Column name
`direction`	"asc" or "desc"

Value

Sorted data.frame

`success_response` *Create standardized success response*

Description

Returns a consistent success response structure.

Usage

```
success_response(message = "Success", data = NULL)
```

Arguments

`message`	Character - success message
`data`	List - data to return

Value

List with success=TRUE and message

`test_encryption` *Test database encryption*

Description

Creates a test encrypted database, writes data, reads it back, and verifies encryption is actually being applied (file is binary, not plaintext).

Usage

```
test_encryption(db_path, key)
```

Arguments

`db_path`	Character: Path to test database file (will be created and deleted)
`key`	Character: Encryption key to test

Details

Test procedure:

1. Connect to database with key
2. Write test data
3. Disconnect
4. Verify file is encrypted (random bytes, not readable text)
5. Reconnect with correct key -> data readable
6. Verify data integrity
7. Cleanup temporary database

This function verifies that SQLCipher is properly compiled into RSQLite.

Value

Logical TRUE if all tests pass, otherwise stops with error

Examples

```
## Not run:
  key <- generate_db_key()
  test_encryption(tempfile(fileext = ".db"), key)

## End(Not run)
```

`update_session_activity` *Manual session activity update*

Description

Explicitly update last activity time. Useful when activity isn't captured through normal input changes.

Usage

```
update_session_activity(session_tracker)
```

Arguments

`session_tracker`	Object returned from init_session_timeout()

`validate_field_value` *Validate individual field value*

Description

Type-specific validation for a single form field

Usage

```
validate_field_value(field_name, value, type, rules)
```

Arguments

`field_name`	Character name of field
`value`	Value to validate
`type`	Field type (text, numeric, date, email, select, checkbox)
`rules`	List of validation rules

Value

List with valid, message, cleaned_value, warning

`validate_filename` *Validate and Sanitize Filename*

Description

Sanitizes a filename by removing or replacing problematic characters, preventing path traversal attacks, and limiting length

Usage

```
validate_filename(filename, max_length = 100)
```

Arguments

`filename`	Character string to sanitize
`max_length`	Maximum filename length (default 100)

Value

Sanitized filename safe for use in file operations

validate_form *Validate entire form submission*

Description

Validates all form fields against metadata rules. Returns detailed validation results suitable for user feedback.

Usage

```
validate_form(form_data, field_metadata)
```

Arguments

`form_data`	List or data.frame with submitted form values
`field_metadata`	List defining validation rules for each field

Value

List with:

- valid: logical, TRUE if all validations passed
- errors: named list of field-specific error messages
- warnings: named list of field-specific warnings
- cleaned_data: validated and cleaned data

Examples

```
## Not run:
metadata <- list(
  age = list(type = "numeric", required = TRUE, min = 18, max = 120),
  email = list(type = "email", required = TRUE)
)

result <- validate_form(
  list(age = 25, email = "user@example.com"),
  metadata
)

if (result$valid) {
  # Process the cleaned data
  save_record(result$cleaned_data)
} else {
  # Show errors to user
  show_validation_errors(result$errors)
}

## End(Not run)
```

`validate_instrument_csv` *Validate instrument CSV structure*

Description

Checks that a CSV file has correct structure for import. Used for validation before accepting user uploads.

Usage

```
validate_instrument_csv(filepath)
```

Arguments

`filepath`	Path to CSV file to validate

Value

List containing:

- valid: Logical, structure is valid?
- errors: Character vector of validation errors
- warnings: Character vector of warnings
- field_count: Number of fields in file

`verify_audit_integrity` *Verify Audit Trail Integrity*

Description

Verify integrity of audit trail using hash-chain validation.

Usage

```
verify_audit_integrity(start_id = NULL, end_id = NULL, db_path = NULL)
```

Arguments

`start_id`	Integer: First audit_id to verify (optional)
`end_id`	Integer: Last audit_id to verify (optional)
`db_path`	Character: Database path (optional)

Value

List with verification results

Examples

```
## Not run:
  verification <- verify_audit_integrity()
  if (verification$valid) {
    cat("Audit trail integrity verified\n")
  }

## End(Not run)
```

---

verify_audit_log_integrity

Verify audit log integrity

Description

Validates that audit log hasn't been tampered with by checking hash chain. Returns TRUE if all hashes are correctly chained, FALSE otherwise.

Usage

```
verify_audit_log_integrity(audit_log)
```

Arguments

`audit_log`	reactiveVal or data.frame containing audit records

Value

Logical - TRUE if log integrity verified, FALSE if tampering detected

---

verify_database_encryption

Verify Database Encryption

Description

Comprehensive verification that database encryption is working correctly.

Usage

```
verify_database_encryption(db_path = NULL)
```

Arguments

`db_path`	Character: Database to verify (optional, uses get_db_path if NULL)

Details

Verifies:

1. Database file is binary (not plaintext)
2. Can connect with encryption key
3. Can write and read data
4. Data is encrypted in file (no readable text)

Value

List with verification results:

- encrypted: Logical TRUE if encryption working
- file_is_binary: Logical TRUE if file content is binary (encrypted)
- connection_works: Logical TRUE if can connect and query
- data_intact: Logical TRUE if data readable after encryption
- message: Detailed status message

Examples

```
## Not run:
  verification <- verify_database_encryption()
  if (verification$encrypted) {
    cat("Database encryption verified!\n")
  } else {
    cat("Encryption issues:", verification$message, "\n")
  }

## End(Not run)
```

`verify_db_key` *Verify database encryption key format*

Description

Validates that a key is properly formatted for SQLCipher. Checks length, characters, and format.

Usage

```
verify_db_key(key)
```

Arguments

`key`	Character string: The key to validate

Details

Valid format requirements:

- Exactly 64 hexadecimal characters (256 bits)
- All lowercase a-f and 0-9
- Single string (length 1)

Invalid keys will stop execution with descriptive error message.

Value

Logical TRUE if valid, otherwise stops with error message

Examples

```
## Not run:
  key <- generate_db_key()
  verify_db_key(key)  # Returns TRUE

## End(Not run)
```

`verify_exported_data` *Verify Exported Data Integrity*

Description

Verifies exported data has not been tampered with using SHA-256 hash.

Usage

```
verify_exported_data(file_path, hash_file = NULL)
```

Arguments

`file_path`	Character: Path to exported data file
`hash_file`	Character: Path to hash file (auto-detected if NULL)

Details

Compares SHA-256 hash of current file content with stored hash. Detects:

- File tampering or corruption
- Missing hash file
- Hash mismatches

Value

List with verification results:

- valid: Logical TRUE if hash matches
- file_hash: Computed hash of file
- stored_hash: Hash read from hash file
- message: Detailed verification message

Examples

```
## Not run:
  verification <- verify_exported_data(
    file_path = "./exports/export_subjects_20251218_123456.csv"
  )
  if (verification$valid) {
    cat("Data integrity verified!\n")
  }

## End(Not run)
```

`%||%` *Null Coalesce Operator*

Description

Null Coalesce Operator

Usage

```
x %||% y
```

Arguments

x	First value
y	Default value if x is NULL

Value

x if not NULL, otherwise y

Index