Package 'finman'

July 16, 2025

Type Package

Title Finance Account Management

Version 0.0.1

Description Core logic for managing master and child accounts recursively, including tracking balances, transactions, and dues across nested accounts.

URL https://github.com/statisticsguru1/personal-finance-manager/tree/refactor-main-account-docs, https://statisticsguru1.github.io/portfolio/

BugReports https://github.com/statisticsguru1/personal-finance-manager/issues

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

Imports R6,
tidyverse,
uuid,
lubridate,
jose,
filelock,
withr,
jsonlite,
rlang

Suggests testthat (>= 3.0.0),
covr,
lintr,
pkgdown,
knitr,
rmarkdown

Config/testthat/edition 3

License file LICENSE

VignetteBuilder knitr

R topics documented:

`build_plugin_args` *Build Plugin Arguments for Storage Backend Functions*

Description

Constructs a list of arguments for storage plugin functions (e.g., ‘load_from_file‘, ‘save_to_mongo‘). It merges the base arguments passed directly with plugin-specific configuration from environment variables.

Usage

```
build_plugin_args(backend, mode = "load", ...)
```

Arguments

backend	A string indicating the backend (e.g., "file", "mongo", "gdrive").
mode	The operation mode, one of "load", "save", "file_exists", or "remove".
...	Additional arguments (e.g., ‘user_id‘, ‘file_name‘, etc.).

Details

Supports backends such as local file system, MongoDB, and Google Drive.

Value

A named list of arguments suitable for ‘do.call()‘ when calling a plugin function.

Examples

```
Sys.setenv(ACCOUNT_BASE_DIR = "user_accounts")
args <- build_plugin_args(
  "file", "load",
  user_id = "user1",
  file_name = "account_tree.Rds"
)
# Returns something like:
# list(user_id = "user1", file_name = "account_tree.Rds",
```

```
#      base_dir = "user_accounts")
```

`ChildAccount` *ChildAccount Class*

Description

An extension of the `MainAccount` class used to model specialized sub-accounts such as goals, needs, or debt repayment accounts. Child accounts inherit all the core functionality of `MainAccount` while adding features like allocation percentage, account status, and priority levels for fund distribution.

Details

- Inherits from `MainAccount`.
- Can be part of a parent account and participate in fund distribution.
- Tracks its own balance, transaction history, and priority for receiving funds.

Methods

`initialize(name, allocation, status, parent, path, priority)` Constructor method.

`deposit(amount, transaction_number, by, channel, date)` Overridden deposit method with status check.

`change_status(status)` Updates the status of the account.

`get_account_status()` Returns and prints the account's status.

`get_priority()` Returns the priority level of the account.

`set_priority(priority)` Sets a new priority level.

Super class

`finman::MainAccount` -> `ChildAccount`

Public fields

`allocation` Numeric. Share of distributed income (0-1 scale).

`status` Character. Indicates whether the account is "active" or "closed".

`parent` Optional. Reference to the parent account (if hierarchical).

`path` Character. Logical path to the account (used for organizing accounts).

`priority` Numeric. Determines order of distribution among children (higher = more).

Methods

Public methods:

- ChildAccount$new()
- ChildAccount$deposit()
- ChildAccount$change_status()
- ChildAccount$get_account_status()
- ChildAccount$get_priority()
- ChildAccount$set_priority()
- ChildAccount$clone()

Method `new()`**:** Initializes a new ChildAccount object by setting the name, allocation, status, parent, path, and priority. Inherits initialization logic from the MainAccount class.

Usage:

```
ChildAccount$new(
  name,
  allocation = 0,
  status = "active",
  parent = NULL,
  path = NULL,
  priority = 0
)
```

Arguments:

`name` Character. The name of the child account.

`allocation` Numeric. Allocation weight for distributing funds (default is 0).

`status` Character. Status of the account: "active", "inactive", or "closed" (default is "active").

`parent` MainAccount or NULL. Optional parent account reference.

`path` Character or NULL. Path or label to identify account lineage.

`priority` Numeric. Priority value used when distributing small amounts (default is 0).

Examples:

```
# Create a basic ChildAccount with default values
acc <- ChildAccount$new(name = "Emergency Fund")

# Create a ChildAccount with custom allocation and priority
acc2 <- ChildAccount$new(
  name = "Education",
  allocation = 0.3,
  status = "active",
  path = "main_account/education",
  priority = 2
)

# View the account status
acc2$get_account_status()
```

Method `deposit()`**:** Deposits funds into a ChildAccount if the account is active. Records the transaction, updates the balance, and distributes the funds to any nested child accounts, if applicable.

Usage:

```
ChildAccount$deposit(
  amount,
  transaction_number = NULL,
  by = "User",
  channel = NULL,
  date = Sys.time()
)
```

Arguments:

`amount` Numeric. Amount of money to deposit. Must be greater than 0.

`transaction_number` Character or NULL. Optional transaction ID. If NULL, an ID is generated automatically.

`by` Character. Identifier for who made the deposit (default is "User").

`channel` Character or NULL. Channel through which the deposit is made (e.g., "Mobile", "Bank").

`date` POSIXct. Timestamp of the transaction (default is current time).

Details: This method overrides the `deposit()` method from the MainAccount class. It includes a status check to ensure the account is active before proceeding. If the account is inactive or closed, the deposit is blocked and a message is printed.

Returns: No return value. Updates the account state and prints a summary.

Examples:

```
# Create an active child account
acc <- ChildAccount$new(name = "Savings", allocation = 0.5)

# Deposit funds into the account
acc$deposit(amount = 100, channel = "Mobile")

# Attempting to deposit into an inactive account
acc$change_status("inactive")
acc$deposit(amount = 50, channel = "Mobile")  # Will not proceed
```

Method `change_status()`**:** Changes the status of the ChildAccount. If the new status is "closed", the account must have a zero balance; otherwise, an error is thrown.

Usage:

```
ChildAccount$change_status(status)
```

Arguments:

`status` Character. The desired new status of the account. Acceptable values include "active", "inactive", or "closed".

Details: If the account is already "closed", a message is printed and no changes are made. If closing is requested and the balance is not zero, an error is raised to ensure proper fund handling before deactivation.

Returns: No return value. Modifies the account's status in place and prints a message.

Examples:

```
acc <- ChildAccount$new(name = "Emergency Fund", allocation = 0.3)
acc$change_status("inactive")  # Changes status to inactive
acc$change_status("active")    # Re-activates the account

# Attempting to close with non-zero balance triggers error
acc$deposit(100, channel = "Mobile")
```

```
\dontrun{
acc$change_status("closed")    # Will raise an error
}

# Withdraw funds then close
acc$withdraw(100, channel = "Transfer")
acc$change_status("closed")    # Successful closure
```

Method `get_account_status()`**:** Retrieves and prints the current status of the child account.

Usage:

```
ChildAccount$get_account_status()
```

Returns: Character. The current status of the account: typically "active", "inactive", or "closed".

Examples:

```
acc <- ChildAccount$new(name = "School Fees", allocation = 0.4)
acc$get_account_status()
# Output: "School Fees is active"
```

Method `get_priority()`**:** Returns the current priority level assigned to the child account.

Usage:

```
ChildAccount$get_priority()
```

Returns: Numeric. The priority value used in fund distribution (higher values indicate higher priority).

Examples:

```
acc <- ChildAccount$new(name = "Emergency Fund", priority = 3)
acc$get_priority()
# [1] 3
```

Method `set_priority()`**:** Updates the priority level of the child account. Higher priority values indicate a stronger preference for receiving funds during distribution.

Usage:

```
ChildAccount$set_priority(priority)
```

Arguments:

`priority` Numeric. The new priority value to assign to the account.

Returns: No return value. Prints a message confirming the new priority.

Examples:

```
acc <- ChildAccount$new(name = "Education Fund", priority = 1)
acc$set_priority(5)
# Priority for Education Fund set to 5
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

`MainAccount`

Examples

```
library(R6)
library(uuid)
library(tidyverse)
# Create a basic ChildAccount instance
child <- ChildAccount$new(
  name = "Emergency Fund",
  allocation = 0.3,
  priority = 2
)

# Check initial status and priority
child$get_account_status()
child$get_priority()

# Deposit into the child account
child$deposit(
  amount = 1000,
  channel = "Bank Transfer"
)

# Change account status to inactive
child$change_status("inactive")

# Try another deposit (won't proceed if inactive)
child$deposit(
  amount = 500,
  channel = "Bank Transfer"
)

# Close the account after setting balance to zero
child$withdraw(
  amount = child$balance,
  channel = "Transfer to Main"
)
child$change_status("closed")


## ------------------------------------------------
## Method `ChildAccount$new`
## ------------------------------------------------

# Create a basic ChildAccount with default values
acc <- ChildAccount$new(name = "Emergency Fund")

# Create a ChildAccount with custom allocation and priority
acc2 <- ChildAccount$new(
  name = "Education",
  allocation = 0.3,
  status = "active",
  path = "main_account/education",
  priority = 2
)

# View the account status
acc2$get_account_status()
```

```
## ------------------------------------------------
## Method `ChildAccount$deposit`
## ------------------------------------------------

# Create an active child account
acc <- ChildAccount$new(name = "Savings", allocation = 0.5)

# Deposit funds into the account
acc$deposit(amount = 100, channel = "Mobile")

# Attempting to deposit into an inactive account
acc$change_status("inactive")
acc$deposit(amount = 50, channel = "Mobile")  # Will not proceed

## ------------------------------------------------
## Method `ChildAccount$change_status`
## ------------------------------------------------

acc <- ChildAccount$new(name = "Emergency Fund", allocation = 0.3)
acc$change_status("inactive")  # Changes status to inactive
acc$change_status("active")    # Re-activates the account

# Attempting to close with non-zero balance triggers error
acc$deposit(100, channel = "Mobile")
## Not run:
acc$change_status("closed")    # Will raise an error

## End(Not run)

# Withdraw funds then close
acc$withdraw(100, channel = "Transfer")
acc$change_status("closed")    # Successful closure

## ------------------------------------------------
## Method `ChildAccount$get_account_status`
## ------------------------------------------------

acc <- ChildAccount$new(name = "School Fees", allocation = 0.4)
acc$get_account_status()
# Output: "School Fees is active"

## ------------------------------------------------
## Method `ChildAccount$get_priority`
## ------------------------------------------------

acc <- ChildAccount$new(name = "Emergency Fund", priority = 3)
acc$get_priority()
# [1] 3

## ------------------------------------------------
## Method `ChildAccount$set_priority`
## ------------------------------------------------

acc <- ChildAccount$new(name = "Education Fund", priority = 1)
acc$set_priority(5)
# Priority for Education Fund set to 5
```

create_user_account_base *Create a New User Account.*

Description

Initializes a 'MainAccount' object and saves it using the configured backend.

Usage

```
create_user_account_base(
  user_id,
  base_dir = Sys.getenv("ACCOUNT_BASE_DIR", "user_accounts"),
  initial_balance = 0
)
```

Arguments

user_id	A validated user ID (must contain only letters, digits, and underscores).
base_dir	Deprecated. Use ACCOUNT_BACKEND instead.
initial_balance	Optional numeric value specifying the starting balance for the main account. Default is '0'.

Value

Invisibly returns 'TRUE' if the account was created successfully.

file_exists_file *Check if a user file exists on the local file system*

Description

Check if a user file exists on the local file system

Usage

```
file_exists_file(user_id, file_name, base_dir)
```

Arguments

user_id	The user ID
file_name	The file name to check
base_dir	The base directory for user data

Value

Logical TRUE/FALSE

GrandchildAccount *GrandAccount Class*

Description

Extends ChildAccount to model low-level accounts such as bills, loans, and targeted savings. Adds time-based logic, due tracking, and automatic closure/reactivation to ensure intelligent fund allocation.

Details

This class introduces behavior tailored to two main categories:

1. Periodic Accounts (e.g., Bills, Rent, Fixed Savings):

- Require recurring payments before a due_date.
- If fully funded before the due date, the account is marked "inactive" and any surplus is returned to the parent.
- Upon reaching the next due date, the account reactivates and begins tracking the next cycle's funding needs.

2. Open-Ended Accounts (e.g., Long-Term Debts, Target Savings):

- Do not rely on due_date or cycles.
- Once the fixed_amount target is met, the account is permanently closed and surplus funds are redirected.
- This prevents over-allocation to already satisfied targets.

These behaviors:

- Guard against poor user allocation strategies by reallocating excess funds from fully funded accounts.
- Adapt automatically to variable incomes, ensuring flexible prioritization (e.g., for freelancers).
- Allow non-expert users to benefit from dynamic, self-adjusting savings and debt repayment logic over time.

Methods

initialize(...) Constructor. Sets allocation, type, due date, fixed target, etc.

deposit(...) Handles reactivation on due date and closes account when target met. Returns surplus to parent account for redistribution.

withdraw(...) Withdraws funds and adjusts period tracking accordingly.

get_/set_ **methods** Get/set values for due date, amount, account type, freq, periods.

Super classes

finman::MainAccount -> finman::ChildAccount -> GrandchildAccount

Public fields

`status` Character. "active", "inactive", or "closed".

`due_date` POSIXct or NULL. When funding is due (for bills, etc.).

`amount_due` Numeric. Amount left to fully fund the account.

`fixed_amount` Numeric. Fixed target for each funding cycle.

`account_type` Character. e.g., "Bill", "Debt", "FixedSaving".

`freq` Numeric or NULL. Cycle length in days (for recurring accounts).

`num_periods` Numeric. Number of unpaid cycles.

`Track_dues_and_balance` Data frame. History of balance and dues.

Methods

Public methods:

- `GrandchildAccount$new()`
- `GrandchildAccount$get_due_date()`
- `GrandchildAccount$set_due_date()`
- `GrandchildAccount$get_fixed_amount()`
- `GrandchildAccount$set_fixed_amount()`
- `GrandchildAccount$get_account_type()`
- `GrandchildAccount$set_account_type()`
- `GrandchildAccount$deposit()`
- `GrandchildAccount$withdraw()`
- `GrandchildAccount$get_account_freq()`
- `GrandchildAccount$set_account_freq()`
- `GrandchildAccount$get_account_periods()`
- `GrandchildAccount$set_account_periods()`
- `GrandchildAccount$clone()`

Method `new()`**:** Initializes a new `GrandchildAccount` instance with attributes and tracking suitable for both periodic (e.g., bills, rent, fixed savings) and open-ended (e.g., long-term debts, target savings) accounts. Inherits from `ChildAccount` and sets up account-specific parameters like due dates, target amounts, and funding cycles.

Usage:

```
GrandchildAccount$new(
  name,
  allocation = 0,
  priority = 0,
  fixed_amount = 0,
  due_date = NULL,
  account_type = "Expense",
  freq = NULL,
  status = "active"
)
```

Arguments:

`name` Character. Name or label of the account (e.g., "Rent", "Car Loan").

`allocation` Numeric. Proportion of parent funds to allocate to this account. Used during distribution logic. Defaults to 0.

`priority` Numeric. Priority weight for redistribution of residual funds, especially in cases of overflow or unmet allocations. Defaults to 0.

`fixed_amount` Numeric. Target amount required to fully fund the account per period or in total. Used for both bills and savings goals.

`due_date` POSIXct or NULL. Optional due date indicating when the next funding cycle is expected (for recurring accounts).

`account_type` Character. Type of the account: e.g., "Bill", "Debt", "FixedSaving", or "Expense". Influences reactivation and closure behavior. Defaults to "Expense".

`freq` Numeric or NULL. Frequency in days for periodic accounts to recur. Required for automated reactivation logic.

`status` Character. "active", "inactive", or "closed".

Details: The constructor also initializes a data frame `Track_dues_and_balance` to monitor the account's funding status over time. Each row logs the current due amount and balance upon deposit or withdrawal.
For accounts of type "Bill", "FixedSaving", or "Debt" (with a `due_date`), the constructor sets up fields that support automated activation, deactivation, and fund tracking per cycle.

Examples:

```
# Initialize a rent account due every 30 days with a fixed monthly cost
library(R6)
library(uuid)
library(tidyverse)
rent <- GrandchildAccount$new(
  name = "Rent",
  allocation = 0.3,
  priority = 2,
  fixed_amount = 75000,
  due_date = Sys.Date() + 30,
  account_type = "Bill",
  freq = 30
)

# Initialize a target savings account without a due date
car_saving <- GrandchildAccount$new(
  name = "Car Fund",
  allocation = 0.2,
  fixed_amount = 500000,
  account_type = "FixedSaving"
)
```

Method `get_due_date()`**:** Retrieves the current due date of the account. This is typically used for periodic accounts such as bills or fixed savings that require funding on a recurring schedule.

Usage:

```
GrandchildAccount$get_due_date()
```

Returns: A POSIXct object representing the due date, or `NULL` if no due date is set.

Examples:

```
rent <- GrandchildAccount$new(
  name = "Rent",
  fixed_amount = 75000,
  due_date = Sys.Date() + 30
)
rent$get_due_date()
```

Method `set_due_date()`**:** Sets a new due date for the account. This is useful for accounts with periodic funding requirements, such as rent, bills, or fixed savings.

Usage:

```
GrandchildAccount$set_due_date(due_date)
```

Arguments:

`due_date` POSIXct. The new due date to assign to the account.

Returns: None. Updates the account's `due_date` field and prints a confirmation message.

Examples:

```
bill <- GrandchildAccount$new(name = "Electricity", fixed_amount = 5000)
bill$set_due_date(Sys.Date() + 15)
```

Method `get_fixed_amount()`**:** Retrieves the fixed amount assigned to the account. This is typically used in accounts like bills, fixed savings, or loan payments where a specific amount is expected periodically.

Usage:

```
GrandchildAccount$get_fixed_amount()
```

Returns: Numeric. The fixed amount required by the account.

Examples:

```
rent <- GrandchildAccount$new(name = "Rent", fixed_amount = 75000)
rent$get_fixed_amount()
#> [1] 75000
```

Method `set_fixed_amount()`**:** Sets a new fixed amount for the account. This value represents the expected periodic contribution or payment (e.g., monthly rent, loan installment). It also recalculates the current amount due based on the number of unpaid periods and the account balance.

Usage:

```
GrandchildAccount$set_fixed_amount(fixed_amount)
```

Arguments:

`fixed_amount` Numeric. The new fixed amount to be assigned to the account.

Returns: None. Updates internal fields and prints a confirmation message.

Examples:

```
rent <- GrandchildAccount$new(name = "Rent", fixed_amount = 50000)
rent$set_fixed_amount(75000)
#> Fixed amount for Rent set to 75000
```

Method `get_account_type()`**:** Sets a new fixed amount for the account. This value represents the expected periodic contribution or payment (e.g., monthly rent, loan installment). It also recalculates the current amount due based on the number of unpaid periods and the account balance.

Usage:

```
GrandchildAccount$get_account_type()
```

Arguments:

`fixed_amount` Numeric. The new fixed amount to be assigned to the account.

Returns: None. Updates internal fields and prints a confirmation message.

Examples:

```
rent <- GrandchildAccount$new(name = "Rent", fixed_amount = 50000)
rent$set_fixed_amount(75000)
#> Fixed amount for Rent set to 75000
```

Method `set_account_type()`**:** Sets the type of the account, which influences how it behaves with respect to funding, reactivation, and closure policies. This field is central to determining whether the account is recurring, fixed, or target-based.

Usage:

```
GrandchildAccount$set_account_type(account_type)
```

Arguments:

`account_type` Character. One of the supported types such as `"Bill"`, `"Debt"`, `"FixedSaving"`, `"NonFixedSaving"`, or `"Expense"`. Determines how due dates, funding limits, and surplus reallocation are handled.

Details: - `"Bill"` or `"FixedSaving"`: These accounts are period-based and are reactivated upon due dates. - `"Debt"` or target savings: Once fully funded, they are closed and not reopened. - `"Expense"` or `"NonFixedSaving"`: Do not enforce due dates or strict funding targets.

Returns: None. Sets the `account_type` field and prints a confirmation.

Examples:

```
rent <- GrandchildAccount$new(name = "Rent", fixed_amount = 75000)
rent$set_account_type("Bill")
#> Account type for Rent set to Bill
```

Method `deposit()`**:** Handles incoming deposits for grandchild accounts, including complex behavior for fixed-amount accounts such as Bills, FixedSavings, and Debts. It intelligently manages due dates, period increments, surplus reallocation, and account status updates.

Usage:

```
GrandchildAccount$deposit(
  amount,
  transaction_number = NULL,
  by = "User",
  channel = NULL,
  date = Sys.time()
)
```

Arguments:

`amount` Numeric. The amount of money being deposited into the account.

`transaction_number` Character or NULL. Optional unique identifier for the transaction. If NULL, a new one is generated.

`by` Character. The party initiating the transaction (default is "User").

`channel` Character or NULL. The method or channel used for the transaction (e.g., "Mobile Money").

`date` POSIXct. The timestamp for the transaction (defaults to current system time).

Details: This method supports two core behaviors depending on the type of account:
1. Period-based Accounts (‘Bill‘, ‘FixedSaving‘) - If the due date has passed, the system automatically increments the number of unpaid periods and extends the due date. - If the deposited amount fully covers the required amount across all unpaid periods, the account is marked as ‘"inactive"‘ (temporarily closed). - Any surplus is redirected to the parent account for reallocation, ensuring no money is trapped in overfunded accounts. - The transaction is logged in ‘Track_dues_and_balance‘ to track financial health over time.

2. Non-period Accounts (‘LongTermDebt‘, ‘TargetSaving‘) - When the required amount is met, the account is permanently closed, and will not reactivate. This design ensures funds are focused on accounts that still need attention.
This mechanism encourages automatic reallocation of excess funds to critical needs without requiring users to micromanage their allocations — useful especially for users with fluctuating income.

Returns: None. Internally updates the account status, balance, due amounts, and transaction history.

Examples:

```
# main account
main<- MainAccount$new("main")

# child account
child <- ChildAccount$new(
  name = "Emergency Fund",
  allocation = 0.3,
  priority = 2
)

# Grand child account
bill <- GrandchildAccount$new(
  name = "Rent",
  fixed_amount = 75000,
  account_type = "Bill",
  due_date = Sys.Date(),
  freq = 30
)
# attach grand child to parent
main$add_child_account(child)
child$add_child_account(bill)
bill$deposit(75000,channel="ABSA")

# Example with surplus being returned to parent:
bill$deposit(80000,channel="ABSA")

# Example with underpayment:
bill$deposit(20000,channel="ABSA")  # Remains active, shows updated due
```

Method `withdraw()`**:** Handles withdrawal requests from a grandchild account. The method ensures sufficient balance is available, updates the internal transaction tracking, and compensates for partial withdrawals in fixed-amount accounts by adjusting the effective number of periods.

Usage:

```
GrandchildAccount$withdraw(
  amount,
  transaction_number = NULL,
  by = "User",
  channel = NULL,
  date = Sys.time()
)
```

Arguments:

`amount` Numeric. The amount to withdraw from the account.

`transaction_number` Character or NULL. Optional transaction reference ID.

`by` Character. The party initiating the withdrawal (default is "User").

`channel` Character or NULL. Source or medium of the transaction (e.g., "Mobile Money").

`date` POSIXct. The date and time of withdrawal (default is current system time).

Details: The method performs the following steps:
- Checks whether the account balance is sufficient for the requested withdrawal. - If sufficient, it processes the withdrawal via the parent class method. - It then logs the updated balance and remaining amount due into the 'Track_dues_and_balance' history.
For accounts with a 'fixed_amount' (e.g., Bills, FixedSavings, Debts):
- Partial withdrawals are treated as funding reversals and reduce the number of fulfilled periods.
- This ensures that the system maintains accurate state about what's left to fulfill for the account, without modifying the 'fixed_amount' itself.
This mechanism is crucial in dynamic environments where users may occasionally retrieve money from priority accounts — e.g., for emergencies — and helps the system readjust allocation logic accordingly.

Returns: None. The internal state of the account (balance, period count, logs) is updated.

Examples:

```
# Withdraw a partial amount from a fully funded rent account
rent <- GrandchildAccount$new("Rent", fixed_amount = 75000,
account_type = "Bill")
rent$deposit(75000,channel="ABSA")
# Now equivalent to 0.53 of the rent period remaining.
rent$withdraw(35000,channel="ABSA")
```

Method `get_account_freq()`**:** Retrieves the recurrence frequency of the account, typically used for accounts with periodic obligations such as bills, fixed savings, or loans.

Usage:

```
GrandchildAccount$get_account_freq()
```

Details: This frequency determines how often the account expects funding. For instance, a rent account with a monthly cycle would have a frequency of 30 (days), while a weekly expense might have 7.
This field is primarily used in due date updates and period tracking, especially for auto-reactivating accounts like Bills or Fixed Savings after their due dates lapse.

Returns: The frequency of recurrence, as stored in the account (e.g., number of days, or a character label like "monthly").

Examples:

```
acc <- GrandchildAccount$new(
  "Rent",
  fixed_amount = 1000,
  freq = 30
)
acc$get_account_freq()
# [1] 30
```

Method `set_account_freq()`**:** Sets the recurrence frequency of the account, which defines how often the account expects to be funded.

Usage:

```
GrandchildAccount$set_account_freq(account_freq)
```

Arguments:

`account_freq` Numeric or character value representing the recurrence frequency. For example, use '30' for a monthly bill or '"weekly"' if implementing a custom handler.

Details: This frequency value is crucial for managing due dates and determining when a new period starts (e.g., when a rent account should reactivate after a month).It is used in conjunction with the due date to trigger reactivation and allocation adjustments for fixed-type accounts such as Bills, Fixed Savings, and Loans. Changing the frequency may affect how missed or overdue periods are computed going forward.

Examples:

```
acc <- GrandchildAccount$new("Water Bill", fixed_amount = 500)
acc$set_account_freq(30)
# Frequency for Water Bill set to 30
```

Method `get_account_periods()`**:** Retrieves the number of unpaid or active periods associated with the account.

Usage:

```
GrandchildAccount$get_account_periods()
```

Details: This is particularly relevant for fixed-type accounts like Bills, Fixed Savings, or Debts with a defined frequency. The number of periods ('num_periods') represents how many cycles have passed without full funding. It increases when due dates pass without adequate deposits and decreases when partial withdrawals are made from already-funded periods.
For example, if a rent account expects funding every 30 days and misses two cycles, 'num_periods' will be 3 (including the current one), and the system will attempt to fund all missed cycles.

Returns: Numeric value indicating how many periods are currently pending or tracked for the account.

Examples:

```
acc <- GrandchildAccount$new(
  "Internet Bill",
  fixed_amount = 2500,
  freq = 30
 )
acc$get_account_periods()
# [1] 1
```

Method `set_account_periods()`**:** Manually sets the number of unpaid or active periods for the account.

Usage:

```
GrandchildAccount$set_account_periods(periods)
```

Arguments:

`periods` Numeric value representing the number of periods to assign.

Details: This method allows manual control of how many cycles (e.g., months or days) are currently due or tracked for the account. It can be used in administrative corrections or simulations of time passage in budgeting models.
Use with caution: setting 'num_periods' directly may desynchronize with the actual due date logic unless adjustments are consistently maintained.

Examples:

```
acc <- GrandchildAccount$new("Loan Payment", fixed_amount = 10000,
freq = 30)
acc$set_account_periods(3)
# Output: Loan Payment has 3 period(s)
```

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

ChildAccount, MainAccount

Examples

```
## ------------------------------------------------
## Method `GrandchildAccount$new`
## ------------------------------------------------

# Initialize a rent account due every 30 days with a fixed monthly cost
library(R6)
library(uuid)
library(tidyverse)
rent <- GrandchildAccount$new(
  name = "Rent",
  allocation = 0.3,
  priority = 2,
  fixed_amount = 75000,
  due_date = Sys.Date() + 30,
  account_type = "Bill",
  freq = 30
)

# Initialize a target savings account without a due date
car_saving <- GrandchildAccount$new(
  name = "Car Fund",
  allocation = 0.2,
  fixed_amount = 500000,
  account_type = "FixedSaving"
)

## ------------------------------------------------
## Method `GrandchildAccount$get_due_date`
## ------------------------------------------------

rent <- GrandchildAccount$new(
  name = "Rent",
  fixed_amount = 75000,
  due_date = Sys.Date() + 30
)
rent$get_due_date()

## ------------------------------------------------
```

```
## Method `GrandchildAccount$set_due_date`
## ------------------------------------------------

bill <- GrandchildAccount$new(name = "Electricity", fixed_amount = 5000)
bill$set_due_date(Sys.Date() + 15)

## ------------------------------------------------
## Method `GrandchildAccount$get_fixed_amount`
## ------------------------------------------------

rent <- GrandchildAccount$new(name = "Rent", fixed_amount = 75000)
rent$get_fixed_amount()
#> [1] 75000

## ------------------------------------------------
## Method `GrandchildAccount$set_fixed_amount`
## ------------------------------------------------

rent <- GrandchildAccount$new(name = "Rent", fixed_amount = 50000)
rent$set_fixed_amount(75000)
#> Fixed amount for Rent set to 75000

## ------------------------------------------------
## Method `GrandchildAccount$get_account_type`
## ------------------------------------------------

rent <- GrandchildAccount$new(name = "Rent", fixed_amount = 50000)
rent$set_fixed_amount(75000)
#> Fixed amount for Rent set to 75000


## ------------------------------------------------
## Method `GrandchildAccount$set_account_type`
## ------------------------------------------------

rent <- GrandchildAccount$new(name = "Rent", fixed_amount = 75000)
rent$set_account_type("Bill")
#> Account type for Rent set to Bill

## ------------------------------------------------
## Method `GrandchildAccount$deposit`
## ------------------------------------------------

# main account
main<- MainAccount$new("main")

# child account
child <- ChildAccount$new(
  name = "Emergency Fund",
  allocation = 0.3,
  priority = 2
)

# Grand child account
bill <- GrandchildAccount$new(
  name = "Rent",
  fixed_amount = 75000,
```

```
  account_type = "Bill",
  due_date = Sys.Date(),
  freq = 30
)
# attach grand child to parent
main$add_child_account(child)
child$add_child_account(bill)
bill$deposit(75000,channel="ABSA")

# Example with surplus being returned to parent:
bill$deposit(80000,channel="ABSA")

# Example with underpayment:
bill$deposit(20000,channel="ABSA")  # Remains active, shows updated due

## ------------------------------------------------
## Method `GrandchildAccount$withdraw`
## ------------------------------------------------

# Withdraw a partial amount from a fully funded rent account
rent <- GrandchildAccount$new("Rent", fixed_amount = 75000,
account_type = "Bill")
rent$deposit(75000,channel="ABSA")
# Now equivalent to 0.53 of the rent period remaining.
rent$withdraw(35000,channel="ABSA")


## ------------------------------------------------
## Method `GrandchildAccount$get_account_freq`
## ------------------------------------------------

acc <- GrandchildAccount$new(
  "Rent",
  fixed_amount = 1000,
  freq = 30
)
acc$get_account_freq()
# [1] 30

## ------------------------------------------------
## Method `GrandchildAccount$set_account_freq`
## ------------------------------------------------

acc <- GrandchildAccount$new("Water Bill", fixed_amount = 500)
acc$set_account_freq(30)
# Frequency for Water Bill set to 30

## ------------------------------------------------
## Method `GrandchildAccount$get_account_periods`
## ------------------------------------------------

acc <- GrandchildAccount$new(
  "Internet Bill",
  fixed_amount = 2500,
  freq = 30
 )
acc$get_account_periods()
```

```
# [1] 1

## ------------------------------------------------
## Method `GrandchildAccount$set_account_periods`
## ------------------------------------------------

acc <- GrandchildAccount$new("Loan Payment", fixed_amount = 10000,
freq = 30)
acc$set_account_periods(3)
# Output: Loan Payment has 3 period(s)
```

`is_valid_user_id` *Validate a User ID Format*

Description

Checks whether a given user ID is valid and safe for use in file paths. A valid user ID consists of only alphanumeric characters and underscores. This function is used to prevent unsafe input that could lead to directory traversal attacks or file system misuse.

Usage

```
is_valid_user_id(user_id)
```

Arguments

`user_id`	A character string representing the user ID.

Value

A logical value: ‘TRUE‘ if the user ID is valid, ‘FALSE‘ otherwise.

Examples

```
is_valid_user_id("user123")       # TRUE
is_valid_user_id("user-abc")      # FALSE
is_valid_user_id("../../etc/")    # FALSE
```

`load_from_file` *Load a User File from Local Filesystem*

Description

Loads a user-specific file (e.g., account tree, lock file) from the localfilesystem.The file format is determined automatically based on its extension.

Usage

```
load_from_file(user_id, file_name, base_dir)
```

Arguments

user_id	A character string representing the user's unique identifier.
file_name	The name of the file to load (e.g., '"account_tree.Rds"', '"data.json"').
base_dir	The base directory where user data is stored. This should point to the top-level folder containing all user subdirectories.

Details

This function is intended to be called indirectly through a generic loader such as load_user_file, which determines the appropriate backend to use. The file extension determines the parser, and unsupported file types will raise an error.

Value

The loaded object, parsed according to the file extension:

- '.Rds' files are loaded with readRDS()
- '.json' files are loaded with jsonlite::fromJSON()
- '.csv' files are loaded with read.csv()

some depedencies:

See Also

load_user_file, readRDS, fromJSON, read.csv

Examples

```
## Not run:
  load_from_file("user123", "account_tree.Rds", base_dir = "user_accounts")

## End(Not run)
```

---

load_user_file *Load a User's File from the Storage Backend*

Description

Loads a file associated with a user from the configured storage backend. The backend is determined by the 'ACCOUNT_BACKEND' environment variable (default is '"file"' for local storage), and supports loading various file types like '.Rds', '.json', and '.csv' via the appropriate plugin.

Usage

```
load_user_file(user_id, file_name = "account_tree.Rds")
```

Arguments

user_id	A string representing the unique user ID.
file_name	The name of the file to load (e.g., '"account_tree.Rds"'). This should be a base name relative to the user-specific storage root.

Details

This function provides a unified interface to load user-specific data, regardless of where or how it is stored (e.g., local file system, cloud, or database). The actual loading logic is delegated to a plugin based on the backend, which is configured using environment variables.

The function constructs the appropriate arguments for the selected backend using ‘build_plugin_args()‘ and dispatches the call using ‘do.call()‘. The file type determines how the file is parsed (e.g., ‘.Rds‘ via ‘readRDS()‘, ‘.json‘ via ‘jsonlite::fromJSON()‘).

Value

The R object loaded from the specified file.

See Also

[save_user_file()], [build_plugin_args()], and plugin implementations like ‘load_from_file()‘.

Examples

```
## Not run:
Sys.setenv(ACCOUNT_BACKEND = "file")
load_user_file("user123", "account_tree.Rds")
load_user_file("user123", "transactions.csv")

## End(Not run)
```

MainAccount *MainAccount Class*

Description

The ‘MainAccount‘ R6 class represents the top-level virtual account in a hierarchical financial structure. It receives all income and is responsible for distributing funds to its child accounts based on allocation rules.

This class is the core of the budgeting engine and manages:

- Income reception and logging.
- Transaction tracking via a structured data frame.
- A list of linked child accounts.
- Auto-generated UUID for identification.
- A balance that reflects both manual and system transactions.

Details

The account includes the following core attributes:

uuid A unique identifier for the account, auto-generated.

name Human-readable name for the account.

balance Current balance in the account.

transactions A ‘data.frame‘ tracking transaction logs, including custom system transactions.

transaction_counter Used internally to create unique transaction IDs.

child_accounts A list containing attached child accounts.

total_allocation Tracks the total allocation distributed to children.

path Logical traversal path from the root (used to locate accounts hierarchically).

Hierarchy

This is the top-level class in a hierarchy that includes:

- **MainAccount**: Root of the virtual budget system.
- **ChildAccount**: Draws a portion of income from the MainAccount.
- **GrandChildAccount**: Represents specific budget goals (e.g., rent, savings) and has due dates, frequency, and priority logic.

Public fields

`uuid` Auto-generated unique identifier.

`name` Account name.

`balance` Current balance.

`transactions` A 'data.frame' of transaction logs.

`transaction_counter` Counter used to generate unique transaction IDs.

`child_accounts` List of child accounts.

`total_allocation` Numeric. Sum of allocated funds to children.

`path` Character vector representing the account's hierarchy path.

some imports

Methods

Public methods:

- `MainAccount$new()`
- `MainAccount$generate_transaction_id()`
- `MainAccount$is_duplicate_transaction()`
- `MainAccount$deposit()`
- `MainAccount$distribute_to_children()`
- `MainAccount$add_child_account()`
- `MainAccount$set_child_allocation()`
- `MainAccount$withdraw()`
- `MainAccount$get_balance()`
- `MainAccount$get_transactions()`
- `MainAccount$list_child_accounts()`
- `MainAccount$find_account()`
- `MainAccount$find_account_by_uuid()`
- `MainAccount$move_balance()`
- `MainAccount$list_all_accounts()`
- `MainAccount$compute_total_balance()`
- `MainAccount$compute_total_due()`
- `MainAccount$compute_total_due_within_n_days()`

- MainAccount$spending()
- MainAccount$total_income()
- MainAccount$allocated_amount()
- MainAccount$income_utilization()
- MainAccount$walking_amount()
- MainAccount$clone()

Method `new()`**:** Constructor for the 'MainAccount' class. Initializes a new main account with a unique identifier, user-defined name, zero balance, and an empty transaction data frame. Also sets the default account path to "main_account".

Usage:

```
MainAccount$new(name, balance = 0)
```

Arguments:

`name` Character. The name of the main account. Used for identification in the app UI and reports.

`balance` Numeric. Initial balance for the account.

Examples:

```
\dontrun{
main_acc <- MainAccount$new(name = "My Main Account")
print(main_acc$uuid)
print(main_acc$balance)
print(main_acc$transactions)
}
```

Method `generate_transaction_id()`**:** Generates a unique system transaction ID by appending an incrementing counter to a fixed prefix ("sys"). The counter is then incremented for future calls.

Usage:

```
MainAccount$generate_transaction_id()
```

Returns: A character string representing the generated transaction ID.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new(name = "Salary Pool")
  txn_id1 <- main_acc$generate_transaction_id()
  txn_id2 <- main_acc$generate_transaction_id()
  print(txn_id1)  # e.g., "sys1"
  print(txn_id2)  # e.g., "sys2"
}
```

Method `is_duplicate_transaction()`**:** Checks if a given transaction number already exists in the transaction log. This is used to prevent duplicate transaction entries.

Usage:

```
MainAccount$is_duplicate_transaction(transaction_number)
```

Arguments:

`transaction_number` Character. The transaction ID to be checked.

Returns: Logical. `TRUE` if the transaction number exists, `FALSE` otherwise.

Examples:

```
\dontrun{
  # Create a new main account
  main_acc <- MainAccount$new(name = "Salary Pool")

  # Manually add a transaction with ID "sys1"
  main_acc$transactions <- data.frame(
    Type = "Income",
    By = "User",
    TransactionID = "sys1",
    Channel = "Bank",
    Amount = 5000,
    Balance = 5000,
    amount_due = 0,
    overall_balance = 5000,
    Date = Sys.time(),
    stringsAsFactors = FALSE
  )

  # Check for duplicate
  main_acc$is_duplicate_transaction("sys1")  # Returns TRUE
  main_acc$is_duplicate_transaction("sys2")  # Returns FALSE
}
```

Method `deposit()`**:** Deposits a specified amount into the account and distributes it to child accounts based on allocation rules. This method also records the transaction in the internal ledger.

Usage:

```
MainAccount$deposit(
  amount,
  transaction_number = NULL,
  by = "User",
  channel = NULL,
  date = Sys.time()
)
```

Arguments:

`amount` Numeric. The amount of money to deposit. Must be greater than zero.

`transaction_number` Optional character. A unique identifier for this transaction. If not provided, the system generates one automatically.

`by` Character. Identifier of the depositor (default is ‘"User"‘).

`channel` Character. The source of funds (e.g., ‘"ABSA"‘, ‘"MPESA"‘). Required.

`date` POSIXct or character. The timestamp of the transaction (defaults to current time).

Returns: No return value. The method updates the account balance, transaction log, and distributes funds to child accounts.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new(name = "Salary Pool")
  main_acc$deposit(
    amount = 1000,
    channel = "Bank Transfer"
  )
}
```

Method `distribute_to_children()`**:** Distributes a given amount from the an account to active child accounts based on their allocation weights and priorities. If the amount is too small (less than 0.10), it is routed entirely to the highest-priority child.

This method is automatically called after a deposit into a parent account. It performs internal withdrawals and instructs child accounts to deposit their corresponding shares.

Usage:

```
MainAccount$distribute_to_children(amount, transaction, by = "System")
```

Arguments:

`amount` Numeric. Total amount available for distribution.

`transaction` Character. The transaction ID associated with the distribution.

`by` Character. Identifier of the actor performing the transfer (default is '"System"').

Returns: No return value. This method updates balances and transaction logs in both the main account and all affected child accounts.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new(name = "Salary Pool")
  child1 <- ChildAccount$new(name = "Food Fund", allocation = 0.6)
  child2 <- ChildAccount$new(name = "Savings", allocation = 0.4)
  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  main_acc$deposit(amount = 1000, channel = "Bank")
  # This will trigger distribute_to_children internally.
}
```

Method `add_child_account()`**:** Adds a 'ChildAccount' object to the list of child accounts of the account. It checks for valid allocation percentages (must not exceed 100 hierarchical path references, and updates total allocation.

Usage:

```
MainAccount$add_child_account(child_account)
```

Arguments:

`child_account` An object of class 'ChildAccount'or 'GrandChildAccount', representing the account to be added as a subordinate of the current main account.

Details: If the child's allocation is zero, it will be automatically marked as inactive. The method also updates the logical 'path' of the child and attaches a 'parent' reference to maintain the hierarchy.

Returns: None. This method modifies the object in-place.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new(name = "Master")
  child_acc <- ChildAccount$new(name = "Savings", allocation = 0.4)
  main_acc$add_child_account(child_acc)
}
```

Method `set_child_allocation()`**:** Updates the allocation percentage for a specified child account. Ensures that the total allocation across all children does not exceed 100

Usage:

```
MainAccount$set_child_allocation(child_account_name, new_allocation)
```

Arguments:

`child_account_name` Character. The name of the child account whose allocation is to be updated.

`new_allocation` Numeric. The new allocation proportion (between 0 and 1).

Details: If the allocation is set to zero, the child account is marked as inactive. The function automatically updates the total allocation tracker.

Returns: None. This method modifies the object in-place.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Main")
  child <- ChildAccount$new(name = "Emergency", allocation = 0.2)
  main_acc$add_child_account(child)
  main_acc$set_child_allocation("Emergency", 0.3)
}
```

Method `withdraw()`**:** Withdraws a specified amount from the account and logs the transaction. A transaction number is generated automatically if not provided. Withdrawals require a valid channel and sufficient balance.

Usage:

```
MainAccount$withdraw(
  amount,
  transaction_number = NULL,
  by = "User",
  channel = NULL,
  date = Sys.time()
)
```

Arguments:

`amount` Numeric. The amount to withdraw. Must be greater than zero and #' not exceed the current balance.

`transaction_number` Optional character. Custom transaction ID. If NULL, a system-generated ID will be used.

`by` Character. The entity initiating the withdrawal. Default is '"User"'.

`channel` Character. The withdrawal channel (e.g., "Bank Transfer"). Required.

`date` POSIXct. Timestamp for the transaction. Defaults to 'Sys.time()'.

Returns: None. Modifies the object's internal state by reducing the balance and appending a new transaction to the log.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Primary Pool")
  main_acc$deposit(amount = 1000, channel = "Mobile", by = "User")
  main_acc$withdraw(amount = 200, channel = "ATM", by = "User")
}
```

Method `get_balance()`**:** Returns the current balance of the account and prints it to the console.

Usage:

```
MainAccount$get_balance()
```

Returns: Numeric. The current account balance.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Primary Pool")
  main_acc$deposit(amount = 500, channel = "Bank Transfer")
  main_acc$get_balance()
}
```

Method `get_transactions()`**:** Retrieves and displays the transaction history for the account. If no transactions are found, a message is printed. Otherwise, the transaction log is displayed in the console.

This method is inherited by child and grandchild account classes.

Usage:

```
MainAccount$get_transactions()
```

Returns: A data frame containing the account's transaction history.

Examples:

```
\dontrun{
  acc <- MainAccount$new("Main Budget")
  acc$deposit(500, channel = "M-Pesa")
  acc$get_transactions()
}
```

Method `list_child_accounts()`**:** Lists all direct child accounts attached to this account. If no child accounts are found, a message is printed.

This method is inherited by both 'ChildAccount' and 'GrandChildAccount', allowing recursive visibility into nested account structures.

Usage:

```
MainAccount$list_child_accounts()
```

Returns: Invisibly returns a character vector of child account names.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Main Budget")
  child <- ChildAccount$new("Bills", allocation = 0.5)
  main_acc$add_child_account(child)
  main_acc$list_child_accounts()
}
# Recursively find all accounts by name
```

Method `find_account()`**:** Recursively searches the current account, its children, and its parent chain to collect **all** accounts with a given name. This version differs from the original by not stopping at the first match—it returns a list of **all** matches instead.

It avoids infinite recursion by tracking visited account paths.

Usage:

```
MainAccount$find_account(target_name, visited_paths = NULL, matches = NULL)
```

Arguments:

`target_name` Character. The name of the account(s) to locate.

`visited_paths` (Internal use only) Tracks visited paths to avoid cycles.

`matches` (Internal use only) A list to accumulate matches during recursion.

Returns: A list of account objects matching the given name. If no matches are found, returns an empty list.

Examples:

```
\dontrun{
  main <- MainAccount$new("Main")
  savings1 <- ChildAccount$new("Savings", allocation = 0.5)
  savings2 <- ChildAccount$new("Savings", allocation = 0.3)
  main$add_child_account(savings1)
  main$add_child_account(savings2)
  found <- main$find_account("Savings")
  length(found)  # 2
  found[[1]]$uuid
}
```

Method `find_account_by_uuid()`**:** Recursively searches for an account by its unique UUID. The method traverses downward through all child accounts and upward to the parent, if needed, while preventing circular recursion by tracking visited paths. This is especially useful when account names are not unique but UUIDs are.

Usage:

```
MainAccount$find_account_by_uuid(target_uuid, visited_paths = NULL)
```

Arguments:

`target_uuid` Character. The UUID of the account to find.

`visited_paths` Internal use only. A list used to track visited paths and prevent infinite loops in cyclic or nested account structures.

Returns: Returns the account object whose UUID matches the target, or `NULL` if no match is found.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Root")
  groceries <- ChildAccount$new("Groceries", allocation = 0.3)
  main_acc$add_child_account(groceries)
  found <- main_acc$find_account_by_uuid(groceries$uuid)
  if (!is.null(found)) cat("Found UUID:", found$uuid)
}
# Move balance to another account (by UUID)
```

Method `move_balance()`**:** Moves a specified amount from the current account to another account identified by its **UUID**. This is intended for internal transfers within the account tree. It reuses ‘withdraw()‘ and ‘deposit()‘ logic and logs the transfer using the "Internal Transfer" channel.

The target account is resolved using ‘find_account_by_uuid()‘, not by name. This ensures unambiguous targeting even when accounts share the same name.

Usage:

```
MainAccount$move_balance(target_account_uuid, amount)
```

Arguments:

`target_account_uuid` Character. The UUID of the account to which the funds will be moved.

`amount` Numeric. The amount to transfer. Must be less than or equal to the current account's balance.

Returns: No return value. Side effects include balance updates and transaction logs for both the source and target accounts.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Main")
  savings <- ChildAccount$new("Savings", allocation = 0.5)
  emergency <- ChildAccount$new("Emergency", allocation = 0.5)
  main_acc$add_child_account(savings)
  main_acc$add_child_account(emergency)

  # Initial deposit
  main_acc$deposit(1000, channel = "Bank")

  # Move 200 to savings using UUID
  main_acc$move_balance(savings$uuid, 200)
}
```

Method `list_all_accounts()`**:** Recursively lists the names of all accounts in the hierarchy, both upward (towards ancestors) and downward (towards descendants), starting from the current account. This method avoids revisiting any account by tracking visited paths, preventing infinite loops in case of circular references.

Usage:

```
MainAccount$list_all_accounts(visited_paths = NULL)
```

Arguments:

`visited_paths` Optional list. Used internally for recursion to avoid revisiting accounts. Should generally be left as 'NULL' by the user.

Returns: A character vector of account names found across the full reachable tree (both children and ancestors) from the current account.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Main")
  savings <- ChildAccount$new("Savings", allocation = 0.4)
  emergency <- ChildAccount$new("Emergency", allocation = 0.6)

  main_acc$add_child_account(savings)
  main_acc$add_child_account(emergency)

  # List all accounts from the root
  main_acc$list_all_accounts()
  # Output: "Main" "Savings" "Emergency"

  # List all accounts from a child node (will include parents)
  savings$list_all_accounts()
  # Output: "Savings" "Main" "Emergency"
}
```

Method `compute_total_balance()`**:** Recursively computes the total balance held by the current account and all of its child accounts. This method includes the balance of the account on which it's called and traverses down the tree to sum balances of all active descendants.

Usage:

```
MainAccount$compute_total_balance()
```

Returns: Numeric. The total aggregated balance of this account and its entire descendant subtree.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Main")
  child1 <- ChildAccount$new("Child1", allocation = 0.5)
  child2 <- ChildAccount$new("Child2", allocation = 0.5)

  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  main_acc$deposit(100, channel = "Bank", transaction_number = "txn1")
  # This distributes 100 into child1 and child2 based on allocation

  # Check total balance recursively
  total <- main_acc$compute_total_balance()
  print(total)
  # Should return 100 (main + children)
}
```

Method `compute_total_due()`**:** Recursively computes the total amount due for the current account and all of its descendant child accounts. If 'amount_due' is not defined in an account, it defaults to zero.

This is useful for aggregating outstanding dues across the entire hierarchical account structure (e.g., main → child → grandchild).

Usage:

```
MainAccount$compute_total_due()
```

Returns: Numeric. The total due amount for this account and all its descendants.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Main")
  child1 <- ChildAccount$new("Child1", allocation = 0.6)
  child2 <- ChildAccount$new("Child2", allocation = 0.4)

  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  # Manually set dues
  main_acc$amount_due <- 50
  child1$amount_due <- 20
  child2$amount_due <- 30

  total_due <- main_acc$compute_total_due()
  print(total_due)
  # Should return 100 (50 + 20 + 30)
}
```

Method `compute_total_due_within_n_days()`**:** Recursively computes the total amount due for the current account and all child accounts where the due date is within the next `n` days from the current system time.

This method is useful for identifying upcoming payments or obligations in a multi-account hierarchy and prioritizing them based on urgency.

Usage:

```
MainAccount$compute_total_due_within_n_days(n)
```

Arguments:

`n` Integer. The number of days from today within which dues should be considered. Dues without a due date are ignored.

Returns: Numeric. The total due amount within the next `n` days across the account hierarchy.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Main")
  child1 <- ChildAccount$new("Child1", allocation = 0.6)
  child2 <- ChildAccount$new("Child2", allocation = 0.4)

  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  # Assign dues and due dates
  main_acc$amount_due <- 100
  main_acc$due_date <- Sys.time() + 2 * 24 * 60 * 60  # Due in 2 days

  child1$amount_due <- 50
  child1$due_date <- Sys.time() + 5 * 24 * 60 * 60    # Due in 5 days

  child2$amount_due <- 70
  child2$due_date <- Sys.time() + 10 * 24 * 60 * 60   # Due in 10 days

  # Compute total dues within next 7 days
  total_due_7_days <- main_acc$compute_total_due_within_n_days(7)
  print(total_due_7_days)
  # Should return 100 + 50 = 150
}
```

Method `spending()`**:** Recursively computes the total spending (i.e., user-initiated withdrawals) for the current account and all child accounts within a specified date range.

Spending is defined as withdrawals made by the user (‘By == "User"‘), excluding system-initiated or internal transfers. The function includes both the current account and all of its descendants in the calculation.

Usage:

```
MainAccount$spending(daterange = c(Sys.Date() - 365000, Sys.Date()))
```

Arguments:

`daterange` A Date or POSIXct vector of length 2. The start and end dates for the period over which to compute spending. Defaults to the entire timeline.

Returns: Numeric. The total spending amount over the specified date range.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Main")
  child1 <- ChildAccount$new("Child1", allocation = 0.5)
```

```
  child2 <- ChildAccount$new("Child2", allocation = 0.5)

  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  # Simulate some deposits and withdrawals
  main_acc$deposit(500, "T1", By = "User", channel = "Cash")
  main_acc$withdraw(200, By = "User", channel = "Spending",
  date = Sys.time() - 10)
  child1$deposit(300, "T2", By = "User", channel = "Mobile")
  child1$withdraw(100, By = "User", channel = "Shopping",
  date = Sys.time() - 5)

  # Get total user spending in last 30 days
  main_acc$spending(c(Sys.Date() - 30, Sys.Date()))
  # Should return 200 + 100 = 300
}
```

Method `total_income()`**:** Recursively computes the total income (i.e., user-initiated deposits) for the current account and all of its child accounts within a specified date range.

This function sums up all "Deposit" transactions where the ‘By‘ field is set to ‘"User"‘. It includes both the current account and all of its descendants in the income calculation.

Usage:

```
MainAccount$total_income(daterange = c(Sys.Date() - 365000, Sys.Date()))
```

Arguments:

`daterange` A vector of two Dates or POSIXct objects specifying the start and end dates for the income calculation. Defaults to the entire timeline.

Returns: A numeric value representing the total income across all relevant accounts within the specified date range.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Main")
  child1 <- ChildAccount$new("Child1", allocation = 0.5)
  child2 <- ChildAccount$new("Child2", allocation = 0.5)

  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  # Simulate some deposits
  main_acc$deposit(500, "TX01", By = "User", channel = "Cash",
  date = Sys.time() - 7)
  child1$deposit(300, "TX02", By = "User", channel = "Mobile",
  date = Sys.time() - 3)

  # Get total income in last 10 days
  main_acc$total_income(c(Sys.Date() - 10, Sys.Date()))
  # Should return 500 + 300 = 800
}
```

Method `allocated_amount()`**:** Calculates the total allocated amount to this account and its child accounts over a specified date range.

This includes **all deposits** (from both ‘"User"‘ and ‘"System"‘) into the current account, **plus** all **user-initiated** deposits into child and deeper-level descendant accounts. It provides insight into how much funding (regardless of origin) has been allocated directly or indirectly to this node in the account tree.

Usage:

```
MainAccount$allocated_amount(daterange = c(Sys.Date() - 365000, Sys.Date()))
```

Arguments:

`daterange` A vector of two ‘Date‘ or ‘POSIXct‘ objects specifying the start and end dates for the deposit aggregation. Defaults to a very wide range.

Returns: A numeric value representing the total allocated amount across the account and its descendants within the specified date range.

Examples:

```
\dontrun{
  main_acc <- MainAccount$new("Main")
  child1 <- ChildAccount$new("Child1", allocation = 0.5)
  child2 <- ChildAccount$new("Child2", allocation = 0.5)

  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  main_acc$deposit(1000, "TX001", By = "System", channel = "Bank",
  date = Sys.time() - 5)
  child1$deposit(300, "TX002", By = "User", channel = "Mobile",
  date = Sys.time() - 3)
  child2$deposit(200, "TX003", By = "User", channel = "Cash",
  date = Sys.time() - 2)

  # Get total allocated amount within last 7 days
  main_acc$allocated_amount(c(Sys.Date() - 7, Sys.Date()))
  # Expected output: 1000 (System) + 300 + 200 = 1500
}
```

Method `income_utilization()`**:** Computes the **income utilization ratio** over a specified date range.

This is calculated as the ratio of total user withdrawals (‘spending‘) to the total allocated income (‘allocated_amount‘). It measures how effectively funds are being used relative to the total amount deposited (both by user and system).

A value close to 1 indicates high utilization (most funds spent), while a valueclose to 0 indicates low spending relative to funds received.

Usage:

```
MainAccount$income_utilization(daterange = c(Sys.Date() - 365000, Sys.Date()))
```

Arguments:

`daterange` A vector of two ‘Date‘ or ‘POSIXct‘ values defining the date range for the calculation. Defaults to a large historical window.

Returns: A numeric value (between 0 and potentially >1) representing the income utilization ratio. If no income has been allocated, a small epsilon is added to the denominator to avoid division by zero.

Examples:

```
\dontrun{
  account <- MainAccount$new("Parent")
  account$deposit(1000, "TX001", By = "System", channel = "Bank")
  account$withdraw(200, By = "User", channel = "Mobile")

  account$income_utilization()
  # Expected output: 200 / 1000 = 0.2
}
```

Method `walking_amount()`**:** Computes the **latest recorded amount** (either ‘amount_due‘ or ‘balance‘) from the ‘Track_dues_and_balance‘ tracker for the current account and all its child accounts within a specified date range.

It retrieves the latest entry in the given date range, and recursively sums the values for all child accounts.

Usage:

```
MainAccount$walking_amount(
  amt_type = "amount_due",
  daterange = c(Sys.Date() - 365000, Sys.Date())
)
```

Arguments:

`amt_type` A string specifying the metric to return: - ‘"amount_due"‘: Return the last tracked ‘Amount_due‘. - ‘"Balance"‘: Return the last tracked ‘Balance‘.

`daterange` A vector of two dates (of class ‘Date‘ or ‘POSIXct‘) specifying the time window. Default: wide historical range.

Returns: A numeric value representing the sum of either latest ‘Amount_due‘ or ‘Balance‘ from this account and all its children.

Examples:

```
\dontrun{
account$walking_amount("amount_due", c(Sys.Date() - 30,
Sys.Date()))
account$walking_amount("Balance", c(Sys.Date() - 7, Sys.Date()))
}
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
library(R6)
library(uuid)
library(tidyverse)
# Create a main account

acc <- MainAccount$new(name = "Salary Pool", balance = 1000)

# Generate a system transaction ID
acc$generate_transaction_id()
```

```
# Check for duplicate transaction ID
acc$is_duplicate_transaction("sys1")

# Inspect balance
acc$balance

# Inspect UUID
acc$uuid


## ------------------------------------------------
## Method `MainAccount$new`
## ------------------------------------------------

## Not run:
main_acc <- MainAccount$new(name = "My Main Account")
print(main_acc$uuid)
print(main_acc$balance)
print(main_acc$transactions)

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$generate_transaction_id`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new(name = "Salary Pool")
  txn_id1 <- main_acc$generate_transaction_id()
  txn_id2 <- main_acc$generate_transaction_id()
  print(txn_id1)  # e.g., "sys1"
  print(txn_id2)  # e.g., "sys2"

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$is_duplicate_transaction`
## ------------------------------------------------

## Not run:
  # Create a new main account
  main_acc <- MainAccount$new(name = "Salary Pool")

  # Manually add a transaction with ID "sys1"
  main_acc$transactions <- data.frame(
    Type = "Income",
    By = "User",
    TransactionID = "sys1",
    Channel = "Bank",
    Amount = 5000,
    Balance = 5000,
    amount_due = 0,
    overall_balance = 5000,
    Date = Sys.time(),
    stringsAsFactors = FALSE
  )
```

```
  # Check for duplicate
  main_acc$is_duplicate_transaction("sys1")  # Returns TRUE
  main_acc$is_duplicate_transaction("sys2")  # Returns FALSE

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$deposit`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new(name = "Salary Pool")
  main_acc$deposit(
    amount = 1000,
    channel = "Bank Transfer"
  )

## End(Not run)


## ------------------------------------------------
## Method `MainAccount$distribute_to_children`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new(name = "Salary Pool")
  child1 <- ChildAccount$new(name = "Food Fund", allocation = 0.6)
  child2 <- ChildAccount$new(name = "Savings", allocation = 0.4)
  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  main_acc$deposit(amount = 1000, channel = "Bank")
  # This will trigger distribute_to_children internally.

## End(Not run)


## ------------------------------------------------
## Method `MainAccount$add_child_account`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new(name = "Master")
  child_acc <- ChildAccount$new(name = "Savings", allocation = 0.4)
  main_acc$add_child_account(child_acc)

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$set_child_allocation`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new("Main")
  child <- ChildAccount$new(name = "Emergency", allocation = 0.2)
  main_acc$add_child_account(child)
```

```
  main_acc$set_child_allocation("Emergency", 0.3)

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$withdraw`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new("Primary Pool")
  main_acc$deposit(amount = 1000, channel = "Mobile", by = "User")
  main_acc$withdraw(amount = 200, channel = "ATM", by = "User")

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$get_balance`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new("Primary Pool")
  main_acc$deposit(amount = 500, channel = "Bank Transfer")
  main_acc$get_balance()

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$get_transactions`
## ------------------------------------------------

## Not run:
  acc <- MainAccount$new("Main Budget")
  acc$deposit(500, channel = "M-Pesa")
  acc$get_transactions()

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$list_child_accounts`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new("Main Budget")
  child <- ChildAccount$new("Bills", allocation = 0.5)
  main_acc$add_child_account(child)
  main_acc$list_child_accounts()

## End(Not run)
# Recursively find all accounts by name


## ------------------------------------------------
## Method `MainAccount$find_account`
## ------------------------------------------------

## Not run:
  main <- MainAccount$new("Main")
```

```
  savings1 <- ChildAccount$new("Savings", allocation = 0.5)
  savings2 <- ChildAccount$new("Savings", allocation = 0.3)
  main$add_child_account(savings1)
  main$add_child_account(savings2)
  found <- main$find_account("Savings")
  length(found)  # 2
  found[[1]]$uuid

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$find_account_by_uuid`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new("Root")
  groceries <- ChildAccount$new("Groceries", allocation = 0.3)
  main_acc$add_child_account(groceries)
  found <- main_acc$find_account_by_uuid(groceries$uuid)
  if (!is.null(found)) cat("Found UUID:", found$uuid)

## End(Not run)
# Move balance to another account (by UUID)


## ------------------------------------------------
## Method `MainAccount$move_balance`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new("Main")
  savings <- ChildAccount$new("Savings", allocation = 0.5)
  emergency <- ChildAccount$new("Emergency", allocation = 0.5)
  main_acc$add_child_account(savings)
  main_acc$add_child_account(emergency)

  # Initial deposit
  main_acc$deposit(1000, channel = "Bank")

  # Move 200 to savings using UUID
  main_acc$move_balance(savings$uuid, 200)

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$list_all_accounts`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new("Main")
  savings <- ChildAccount$new("Savings", allocation = 0.4)
  emergency <- ChildAccount$new("Emergency", allocation = 0.6)

  main_acc$add_child_account(savings)
  main_acc$add_child_account(emergency)

  # List all accounts from the root
```

```
  main_acc$list_all_accounts()
  # Output: "Main" "Savings" "Emergency"

  # List all accounts from a child node (will include parents)
  savings$list_all_accounts()
  # Output: "Savings" "Main" "Emergency"

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$compute_total_balance`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new("Main")
  child1 <- ChildAccount$new("Child1", allocation = 0.5)
  child2 <- ChildAccount$new("Child2", allocation = 0.5)

  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  main_acc$deposit(100, channel = "Bank", transaction_number = "txn1")
  # This distributes 100 into child1 and child2 based on allocation

  # Check total balance recursively
  total <- main_acc$compute_total_balance()
  print(total)
  # Should return 100 (main + children)

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$compute_total_due`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new("Main")
  child1 <- ChildAccount$new("Child1", allocation = 0.6)
  child2 <- ChildAccount$new("Child2", allocation = 0.4)

  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  # Manually set dues
  main_acc$amount_due <- 50
  child1$amount_due <- 20
  child2$amount_due <- 30

  total_due <- main_acc$compute_total_due()
  print(total_due)
  # Should return 100 (50 + 20 + 30)

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$compute_total_due_within_n_days`
## ------------------------------------------------
```

```
## Not run:
  main_acc <- MainAccount$new("Main")
  child1 <- ChildAccount$new("Child1", allocation = 0.6)
  child2 <- ChildAccount$new("Child2", allocation = 0.4)

  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  # Assign dues and due dates
  main_acc$amount_due <- 100
  main_acc$due_date <- Sys.time() + 2 * 24 * 60 * 60  # Due in 2 days

  child1$amount_due <- 50
  child1$due_date <- Sys.time() + 5 * 24 * 60 * 60    # Due in 5 days

  child2$amount_due <- 70
  child2$due_date <- Sys.time() + 10 * 24 * 60 * 60   # Due in 10 days

  # Compute total dues within next 7 days
  total_due_7_days <- main_acc$compute_total_due_within_n_days(7)
  print(total_due_7_days)
  # Should return 100 + 50 = 150

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$spending`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new("Main")
  child1 <- ChildAccount$new("Child1", allocation = 0.5)
  child2 <- ChildAccount$new("Child2", allocation = 0.5)

  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  # Simulate some deposits and withdrawals
  main_acc$deposit(500, "T1", By = "User", channel = "Cash")
  main_acc$withdraw(200, By = "User", channel = "Spending",
  date = Sys.time() - 10)
  child1$deposit(300, "T2", By = "User", channel = "Mobile")
  child1$withdraw(100, By = "User", channel = "Shopping",
  date = Sys.time() - 5)

  # Get total user spending in last 30 days
  main_acc$spending(c(Sys.Date() - 30, Sys.Date()))
  # Should return 200 + 100 = 300

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$total_income`
## ------------------------------------------------

## Not run:
```

```
  main_acc <- MainAccount$new("Main")
  child1 <- ChildAccount$new("Child1", allocation = 0.5)
  child2 <- ChildAccount$new("Child2", allocation = 0.5)

  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  # Simulate some deposits
  main_acc$deposit(500, "TX01", By = "User", channel = "Cash",
  date = Sys.time() - 7)
  child1$deposit(300, "TX02", By = "User", channel = "Mobile",
  date = Sys.time() - 3)

  # Get total income in last 10 days
  main_acc$total_income(c(Sys.Date() - 10, Sys.Date()))
  # Should return 500 + 300 = 800

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$allocated_amount`
## ------------------------------------------------

## Not run:
  main_acc <- MainAccount$new("Main")
  child1 <- ChildAccount$new("Child1", allocation = 0.5)
  child2 <- ChildAccount$new("Child2", allocation = 0.5)

  main_acc$add_child_account(child1)
  main_acc$add_child_account(child2)

  main_acc$deposit(1000, "TX001", By = "System", channel = "Bank",
  date = Sys.time() - 5)
  child1$deposit(300, "TX002", By = "User", channel = "Mobile",
  date = Sys.time() - 3)
  child2$deposit(200, "TX003", By = "User", channel = "Cash",
  date = Sys.time() - 2)

  # Get total allocated amount within last 7 days
  main_acc$allocated_amount(c(Sys.Date() - 7, Sys.Date()))
  # Expected output: 1000 (System) + 300 + 200 = 1500

## End(Not run)

## ------------------------------------------------
## Method `MainAccount$income_utilization`
## ------------------------------------------------

## Not run:
  account <- MainAccount$new("Parent")
  account$deposit(1000, "TX001", By = "System", channel = "Bank")
  account$withdraw(200, By = "User", channel = "Mobile")

  account$income_utilization()
  # Expected output: 200 / 1000 = 0.2

## End(Not run)
```

```
## ------------------------------------------------
## Method `MainAccount$walking_amount`
## ------------------------------------------------

## Not run:
account$walking_amount("amount_due", c(Sys.Date() - 30,
Sys.Date()))
account$walking_amount("Balance", c(Sys.Date() - 7, Sys.Date()))

## End(Not run)
```

`remove_from_file` *Remove a File from the Local File System*

Description

Removes a specified file from a user's directory on the local file system. This is a plugin for the '"file"' backend, used via the generic interface 'remove_user_file()'.

Usage

```
remove_from_file(user_id, file_name, base_dir)
```

Arguments

`user_id`	A string specifying the unique user ID.
`file_name`	The name of the file to remove (e.g., '"account_tree.Rds"').
`base_dir`	The root directory containing all user folders. The full path is constructed as 'file.path(base_dir, user_id, file_name)'.

Details

If the file does not exist, a warning is issued but no error is thrown.

Value

Returns 'TRUE' if the file was successfully removed, or 'FALSE' if the file did not exist or could not be removed. A warning is issued if the file is not found.

See Also

[save_to_file()], [load_from_file()], [remove_user_file()]

Examples

```
## Not run:
remove_from_file("user123", "account_tree.Rds", base_dir = "user_data")

## End(Not run)
```

`remove_user_file` *Remove a User File via Configured Storage Backend*

Description

Deletes a specific user file using the appropriate backend plugin, based on the 'ACCOUNT_BACKEND' environment variable. This function abstracts the file removal logic, allowing different storage systems to handle file deletion (e.g., local file system, MongoDB, Google Drive).

Usage

```
remove_user_file(user_id, file_name = "account_tree.Rds")
```

Arguments

`user_id`	A character string representing the user ID.
`file_name`	Name of the file to remove. Defaults to '"account_tree.Rds"'.

Details

The actual removal behavior is implemented by the selected backend plugin (e.g., 'remove_from_file', 'remove_from_gdrive', etc.). Configuration is controlled via environment variables like 'ACCOUNT_BACKEND' and 'ACCOUNT_BASE_DIR'.

Value

Invisibly returns 'NULL'. Issues a warning if the file does not exist, or throws an error if the backend plugin is not found.

See Also

[save_user_file()], [load_user_file()], [build_plugin_args()]

Examples

```
## Not run:
  remove_user_file("user123")  # Deletes "account_tree.Rds"
  remove_user_file("user123", file_name = "data.json")

## End(Not run)
```

`save_to_file` *Save a User's File to the Local File System*

Description

Saves an R object to a user-specific directory on the local file system using a format inferred from the file extension. Supported formats include '.Rds', '.json', '.csv', and '.lock'.

Usage

```
save_to_file(user_id, object, file_name, base_dir)
```

Arguments

`user_id`	A character string specifying the unique user ID.
`object`	The R object to save. This can be a list, data frame, atomic vector, or any serializable R object.
`file_name`	The name of the file to save, including its extension (e.g., '"account_tree.Rds"', '"transactions.json"', '"session.lock"').
`base_dir`	The root directory where user-specific folders are stored. The object will be saved in 'file.path(base_dir, user_id, file_name)'.

Details

This function serves as the '"file"' backend plugin for saving user data. It is usually called via 'save_user_file()' and supports automatic directory creation for the target user.

The file format is determined from the extension in 'file_name': - **.Rds** — Saves the object using 'saveRDS()'. - **.json** — Saves the object using 'jsonlite::write_json()' with 'auto_unbox = TRUE'. - **.csv** — Saves using 'write.csv()' with 'row.names = FALSE'. - **.lock** — Treated as a text file; 'writeLines()' is used with the assumption that 'object' is a string or scalar.

If the file extension is not recognized, the function falls back to serializing the object with 'serialize()' and writes it as binary. A warning is issued to notify the developer and suggest extending the plugin to handle custom file types explicitly.

Value

Invisibly returns the path of the file after saving.

Note

This function is intended for internal use within the plugin system. For application-level usage, prefer calling 'save_user_file()'.

See Also

[save_user_file()], [load_from_file()], [build_plugin_args()]

Examples

```
## Not run:
save_to_file("user123", list(name = "Alice"), "profile.json",
             base_dir = "data")

save_to_file("user123", data.frame(income = c(100, 200)), "income.csv",
             base_dir = "data")

save_to_file("user123", MainAccount$new(name = "Main"), "account_tree.Rds",
             base_dir = "data")

# Save a lock file
save_to_file("user123", Sys.getpid(), "account_tree.lock",
             base_dir = "data")

# Save unknown file format (e.g., .yaml) — triggers warning
save_to_file("user123", list(config = TRUE), "custom.yaml",
             base_dir = "data")

## End(Not run)
```

`save_user_file` *Save a User-Specific File via Plugin Backend*

Description

Saves an R object to a user-specific location using the appropriate plugin backend (e.g., ‘"file"‘, ‘"mongo"‘, ‘"gdrive"‘). The format and destination are determined by the backend and file extension.

Usage

```
save_user_file(user_id, object, file_name = "account_tree.Rds")
```

Arguments

`user_id`	A character string representing the unique user ID.
`object`	The R object to save. This can be any R object appropriate for the file extension used (e.g., list, data.frame, custom class).
`file_name`	The file name, including the extension (e.g., ‘"account_tree.Rds"‘, ‘"meta.json"‘, ‘"transactions.csv"‘, ‘"account_tree.lock"‘).

Details

This function acts as a plugin launcher for saving user-related data, including account trees (‘.Rds‘), transaction records (‘.csv‘), metadata (‘.json‘), or lockfiles (‘.lock‘). It delegates the actual save operation to the corresponding backend plugin function (e.g., ‘save_to_file()‘).

The appropriate plugin is selected based on the ‘ACCOUNT_BACKEND‘ environment variable (default: ‘"file"‘). The function builds the arguments required by the plugin using [build_plugin_args()], then delegates the save operation.

An error is raised if no suitable save plugin is found.

Value

No return value. This function is invoked for its side effect of persisting a file via the selected backend plugin.

See Also

[build_plugin_args()], [load_user_file()], [remove_user_file()], [user_file_exists()]

Examples

```
## Not run:
  # Save an account tree to .Rds
  save_user_file("user123", MainAccount$new(name = "Main"))

  # Save a data frame to CSV
  save_user_file("user123", data.frame(a = 1:3), "transactions.csv")

  # Save a lock file (PID as a number)
  save_user_file("user123", Sys.getpid(), "account_tree.lock")

## End(Not run)
```

---

user_file_exists *Check If a User File Exists via Configured Storage Backend*

Description

Checks whether a specific file (e.g., ‘account_tree.Rds‘, lock file, etc.) exists for a given user by delegating to a backend-specific plugin.

Usage

```
user_file_exists(user_id, file_name = "account_tree.Rds")
```

Arguments

user_id	A string representing the unique user ID.
file_name	Name of the file to check, relative to the user’s folder. Defaults to ‘"account_tree.Rds"‘.

Details

This function supports checking files stored in different storage backends such as local disk, MongoDB, Google Drive, etc., as configured via the ‘ACCOUNT_BACKEND‘ environment variable.

Value

A logical value: ‘TRUE‘ if the file exists, ‘FALSE‘ otherwise.

Examples

```
## Not run:
  user_file_exists("user123") # Checks account_tree.Rds by default
  user_file_exists("user123", file_name = "account_tree.lock")

## End(Not run)
```

`verify_token` *Decode and Verify a JWT Token*

Description

Attempts to decode and verify a JSON Web Token (JWT) using an HMAC secret. Returns the decoded payload if valid, or ‘NULL‘ if verification fails.

Usage

```
verify_token(token, secret)
```

Arguments

`token`	A character string representing the JWT token (e.g., from an HTTP header).
`secret`	A raw or character vector used as the HMAC secret for verification. Defaults to the global ‘secret_key‘ variable, which should be securely set (e.g., via ‘Sys.getenv("JWT_SECRET")

Value

A list representing the decoded JWT payload if the token is valid; otherwise, ‘NULL‘ if decoding fails or the token is invalid/expired.

See Also

[jose::jwt_decode_hmac()], [Sys.getenv()]

Examples

```
## Not run:
library(jose)
token <- jwt_encode_hmac(list(user_id = "abc123"),
secret = charToRaw("my-secret"))
verify_token(token, secret = charToRaw("my-secret"))

## End(Not run)
```

`with_account_lock` *Acquire a Lock for a User's Account Tree using Plugin-Based File Access*

Description

Ensures exclusive access to a user's account tree by creating a lock file. Waits until the lock is released or a timeout is reached. Uses the plugin architecture to support multiple storage backends.

Usage

```
with_account_lock(user_id, expr, timeout = 1800)
```

Arguments

`user_id`	The user ID whose account tree should be locked.
`expr`	An expression to evaluate within the lock.
`timeout`	Maximum time in seconds to wait for the lock. Default is 1800.

Value

Returns the result of evaluating ‘expr‘.

Index