Auth Service Documentation

Release v1.0.0

Дымников Михаил (dym-dino)

Apr 06, 2025

CONTENTS

CHAPTER

ONE

APP

ICCEM backend API

1.1 logger

Logger Module

Provides logging utilities for the application.

1.1.1 logging

Logging package for Python. Based on PEP 282 and comments thereto in comp.lang.python.

To use, simply 'import logging' and log away!

class app.logger.logging.BufferingFormatter(linefmt=None)

Bases: object

A formatter suitable for formatting a number of records.

format(records)

Format the specified records and return the result as a string.

formatFooter(records)

Return the footer string for the specified records.

formatHeader(records)

Return the header string for the specified records.

class app.logger.logging.FileHandler(filename, mode='a', encoding=None, delay=False, errors=None)

Bases: StreamHandler

A handler class which writes formatted logging records to disk files.

close()

Closes the stream.

emit(record)

Emit a record.

If the stream was not opened because 'delay' was specified in the constructor, open it before calling the superclass's emit.

If stream is not open, current mode is 'w' and _closed=True, record will not be emitted (see Issue #42378).

class app.logger.logging.Filter(name='')

Bases: object

Filter instances are used to perform arbitrary filtering of LogRecords.

Loggers and Handlers can optionally use Filter instances to filter records as desired. The base filter class only allows events which are below a certain point in the logger hierarchy. For example, a filter initialized with "A.B" will allow events logged by loggers "A.B", "A.B.C", "A.B.C.D", "A.B.D" etc. but not "A.BB", "B.A.B" etc. If initialized with the empty string, all events are passed.

filter(record)

Determine if the specified record is to be logged.

Returns True if the record should be logged, or False otherwise. If deemed appropriate, the record may be modified in-place.

class app.logger.logging.Formatter(fmt=None, datefmt=None, style='%', validate=True, * (Keyword-only parameters separator (PEP 3102)), defaults=None)

Bases: object

Formatter instances are used to convert a LogRecord to text.

Formatters need to know how a LogRecord is constructed. They are responsible for converting a LogRecord to (usually) a string which can be interpreted by either a human or an external system. The base Formatter allows a formatting string to be specified. If none is supplied, the style-dependent default value, "%(message)s", "{message}", or "${message}", is used.

The Formatter can be initialized with a format string which makes use of knowledge of the LogRecord attributes - e.g. the default value mentioned above makes use of the fact that the user's message and arguments are pre- formatted into a LogRecord's message attribute. Currently, the useful attributes in a LogRecord are described by:

%(name)s Name of the logger (logging channel) %(levelno)s Numeric logging level for the message (DEBUG, INFO,

WARNING, ERROR, CRITICAL)

%(levelname)s Text logging level for the message ("DEBUG", "INFO",
"WARNING", "ERROR", "CRITICAL")

%(pathname)s Full pathname of the source file where the logging
call was issued (if available)

%(filename)s Filename portion of pathname %(module)s Module (name portion of filename) %(lineno)d Source line number where the logging call was issued

(if available)

%(funcName)s Function name %(created)f Time when the LogRecord was created (time.time()

return value)

%(asctime)s Textual time when the LogRecord was created %(msecs)d Millisecond portion of the creation time %(relativeCreated)d Time in milliseconds when the LogRecord was created,

relative to the time the logging module was loaded (typically at application startup time)

%(thread)d Thread ID (if available) %(threadName)s Thread name (if available) %(taskName)s Task name (if available) %(process)d Process ID (if available) %(message)s The result of record.getMessage(), computed just as

the record is emitted

converter()

localtime([seconds]) -> (tm_year,tm_mon,tm_mday,tm_hour,tm_min,
tm_sec,tm_wday,tm_yday,tm_isdst)

Convert seconds since the Epoch to a time tuple expressing local time. When 'seconds' is not passed in, convert the current time instead.

default_msec_format = '%s,%03d'

default_time_format = '%Y-%m-%d %H:%M:%S'

format(record)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

formatException(ei)

Format and return the specified exception information as a string.

This default implementation just uses traceback.print_exception()

formatMessage(record)

formatStack(stack_info)

This method is provided as an extension point for specialized formatting of stack information.

The input data is a string as returned from a call to traceback.print_stack(), but with the last trailing newline removed.

The base implementation just returns the value passed in.

formatTime(record, datefmt=None)

Return the creation time of the specified LogRecord as formatted text.

This method should be called from format() by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behaviour is as follows: if datefmt (a string) is specified, it is used with time.strftime() to format the creation time of the record. Otherwise, an ISO8601-like (or RFC 3339-like) format is used. The resulting string is returned. This function uses a user-configurable function to convert the creation time to a tuple. By default, time.localtime() is used; to change this for a particular formatter instance, set the 'converter' attribute to a function with the same signature as time.localtime() or time.gmtime(). To change it for all formatters, for example if you want all logging times to be shown in GMT, set the 'converter' attribute in the Formatter class.

usesTime()

Check if the format uses the creation time of the record.

class app.logger.logging.Handler(level=0)

Bases: Filterer

Handler instances dispatch logging events to specific destinations.

The base handler class. Acts as a placeholder which defines the Handler interface. Handlers can optionally use Formatter instances to format records as desired. By default, no formatter is specified; in this case, the 'raw' message as determined by record.message is logged.

acquire()

Acquire the I/O thread lock.

close()

Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

createLock()

Acquire a thread lock for serializing access to the underlying I/O.

emit(record)

Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

flush()

Ensure all logging output has been flushed.

This version does nothing and is intended to be implemented by subclasses.

format(record)

Format the specified record.

If a formatter is set, use it. Otherwise, use the default formatter for the module.

get_name()

handle(record)

Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock.

Returns an instance of the log record that was emitted if it passed all filters, otherwise a false value is returned.

handleError(record)

Handle errors which occur during an emit() call.

This method should be called from handlers when an exception is encountered during an emit() call. If raiseExceptions is false, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The record which was being processed is passed in to this method.

property name

release()

Release the I/O thread lock.

setFormatter(fmt)

Set the formatter for this handler.

setLevel(level)

Set the logging level of this handler. level must be an int or a str.

set_name(name)

class app.logger.logging.LogRecord(name, level, pathname, lineno, msg, args, exc_info, func=None, sinfo=None, **kwargs)

Bases: object

A LogRecord instance represents an event being logged.

LogRecord instances are created every time something is logged. They contain all the information pertinent to the event being logged. The main information passed in is in msg and args, which are combined using str(msg) % args to create the message field of the record. The record also includes information such as when the record was created, the source line where the logging call was made, and any exception information to be logged.

getMessage()

Return the message for this LogRecord.

Return the message for this LogRecord after merging any user-supplied arguments with the message.

class app.logger.logging.Logger(name, level=0)

Bases: Filterer

Instances of the Logger class represent a single logging channel. A "logging channel" indicates an area of an application. Exactly how an "area" is defined is up to the application developer. Since an application can have any number of areas, logging channels are identified by a unique string. Application areas can be nested (e.g. an area of "input processing" might include sub-areas "read CSV files", "read XLS files" and "read Gnumeric files"). To cater for this natural nesting, channel names are organized into a namespace hierarchy where levels are separated by periods, much like the Java or Python package namespace. So in the instance given above, channel names might be "input" for the upper level, and "input.csv", "input.xls" and "input.gnu" for the sub-levels. There is no arbitrary limit to the depth of nesting.

addHandler(hdlr)

Add the specified handler to this logger.

callHandlers(record)

Pass a record to all relevant handlers.

Loop through all handlers for this logger and its parents in the logger hierarchy. If no handler was found, output a one-off error message to sys.stderr. Stop searching up the hierarchy whenever a logger with the "propagate" attribute set to zero is found - that will be the last logger whose handlers are called.

critical(msg, *args, **kwargs)

Log 'msg % args' with severity 'CRITICAL'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.critical("Houston, we have a %s", "major disaster", exc_info=True)

debug(msg, *args, **kwargs)

Log 'msg % args' with severity 'DEBUG'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.debug("Houston, we have a %s", "thorny problem", exc_info=True)

error(msg, *args, **kwargs)

Log 'msg % args' with severity 'ERROR'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.error("Houston, we have a %s", "major problem", exc_info=True)

exception(msg, *args, exc_info=True, **kwargs)

Convenience method for logging an ERROR with exception information.

fatal(msg, *args, **kwargs)

Don't use this method, use critical() instead.

findCaller(stack_info=False, stacklevel=1)

Find the stack frame of the caller so that we can note the source file name, line number and function name.

getChild(suffix)

Get a logger which is a descendant to this one.

This is a convenience method, such that

logging.getLogger('abc').getChild('def.ghi')

is the same as

logging.getLogger('abc.def.ghi')

It's useful, for example, when the parent logger is named using __name__ rather than a literal string.

getChildren()

getEffectiveLevel()

Get the effective level for this logger.

Loop through this logger and its parents in the logger hierarchy, looking for a non-zero logging level. Return the first one found.

handle(record)

Call the handlers for the specified record.

This method is used for unpickled records received from a socket, as well as those created locally. Logger-level filtering is applied.

hasHandlers()

See if this logger has any handlers configured.

Loop through all handlers for this logger and its parents in the logger hierarchy. Return True if a handler was found, else False. Stop searching up the hierarchy whenever a logger with the "propagate" attribute set to zero is found - that will be the last logger which is checked for the existence of handlers.

info(msg, *args, **kwargs)

Log 'msg % args' with severity 'INFO'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.info("Houston, we have a %s", "notable problem", exc_info=True)

isEnabledFor(level)

Is this logger enabled for level 'level'?

log(level, msg, *args, **kwargs)

Log 'msg % args' with the integer severity 'level'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.log(level, "We have a %s", "mysterious problem", exc_info=True)

makeRecord(name, level, fn, lno, msg, args, exc_info, func=None, extra=None, sinfo=None)

A factory method which can be overridden in subclasses to create specialized LogRecords.

manager = <logging.Manager object>

removeHandler(hdlr)

Remove the specified handler from this logger.

root = <RootLogger root (WARNING)>

setLevel(level)

Set the logging level of this logger. level must be an int or a str.

warn(msg, *args, **kwargs)

warning(msg, *args, **kwargs)

Log 'msg % args' with severity 'WARNING'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.warning("Houston, we have a %s", "bit of a problem", exc_info=True)

class app.logger.logging.LoggerAdapter(logger, extra=None)

Bases: object

An adapter for loggers which makes it easier to specify contextual information in logging output.

critical(msg, *args, **kwargs)

Delegate a critical call to the underlying logger.

debug(msg, *args, **kwargs)

Delegate a debug call to the underlying logger.

error(msg, *args, **kwargs)

Delegate an error call to the underlying logger.

exception(msg, *args, exc_info=True, **kwargs)

Delegate an exception call to the underlying logger.

getEffectiveLevel()

Get the effective level for the underlying logger.

hasHandlers()

See if the underlying logger has any handlers.

info(msg, *args, **kwargs)

Delegate an info call to the underlying logger.

isEnabledFor(level)

Is this logger enabled for level 'level'?

log(level, msg, *args, **kwargs)

Delegate a log call to the underlying logger, after adding contextual information from this adapter instance.

property manager

property name

process(msg, kwargs)

Process the logging message and keyword arguments passed in to a logging call to insert contextual information. You can either manipulate the message itself, the keyword args or both. Return the message and kwargs modified (or not) to suit your needs.

Normally, you'll only need to override this one method in a LoggerAdapter subclass for your specific needs.

setLevel(level)

Set the specified level on the underlying logger.

warn(msg, *args, **kwargs)

warning(msg, *args, **kwargs)

Delegate a warning call to the underlying logger.

class app.logger.logging.NullHandler(level=0)

Bases: Handler

This handler does nothing. It's intended to be used to avoid the "No handlers could be found for logger XXX" one-off warning. This is important for library code, which may contain code to log events. If a user of the library does not configure logging, the one-off warning might be produced; to avoid this, the library developer simply needs to instantiate a NullHandler and add it to the top-level logger of the library module or package.

createLock()

Acquire a thread lock for serializing access to the underlying I/O.

emit(record)

Stub.

handle(record)

Stub.

class app.logger.logging.StreamHandler(stream=None)

Bases: Handler

A handler class which writes logging records, appropriately formatted, to a stream. Note that this class does not close the stream, as sys.stdout or sys.stderr may be used.

emit(record)

Emit a record.

If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception and appended to the stream. If the stream has an 'encoding' attribute, it is used to determine how to do the output to the stream.

flush()

Flushes the stream.

setStream(stream)

Sets the StreamHandler's stream to the specified value, if it is different.

Returns the old stream, if the stream was changed, or None if it wasn't.

terminator = '\n'

app.logger.logging.addLevelName(level, levelName)

Associate 'levelName' with 'level'.

This is used when converting levels to text during message formatting.

app.logger.logging.basicConfig(**kwargs)

Do basic configuration for the logging system.

This function does nothing if the root logger already has handlers configured, unless the keyword argument force is set to True. It is a convenience method intended for use by simple scripts to do one-shot configuration of the logging package.

The default behaviour is to create a StreamHandler which writes to sys.stderr, set a formatter using the BASIC_FORMAT format string, and add the handler to the root logger.

A number of optional keyword arguments may be specified, which can alter the default behaviour.

filename Specifies that a FileHandler be created, using the specified
filename, rather than a StreamHandler.

filemode Specifies the mode to open the file, if filename is specified
(if filemode is unspecified, it defaults to 'a').

format Use the specified format string for the handler. datefmt Use the specified date/time format. style If a format string is specified, use this to specify the

type of format string (possible values '%', '{', '$', for %-formatting, str.format() and string. Template - defaults to '%').

level Set the root logger level to the specified level. stream Use the specified stream to initialize the StreamHandler. Note

that this argument is incompatible with 'filename' - if both are present, 'stream' is ignored.

handlers If specified, this should be an iterable of already created
handlers, which will be added to the root logger. Any handler in the list which does not have a formatter assigned will be assigned the formatter created in this function.

force If this keyword is specified as true, any existing handlers
attached to the root logger are removed and closed, before carrying out the configuration as specified by the other arguments.

encoding If specified together with a filename, this encoding is passed to
the created FileHandler, causing it to be used when the file is opened.

errors If specified together with a filename, this value is passed to the
created FileHandler, causing it to be used when the file is opened in text mode. If not specified, the default value is backslashreplace.

Note that you could specify a stream created using open(filename, mode) rather than passing the filename and mode in. However, it should be remembered that StreamHandler does not close its stream (since it may be using sys.stdout or sys.stderr), whereas FileHandler closes its stream when the handler is closed.

Changed in version 3.2: Added the style parameter.

Changed in version 3.3: Added the handlers parameter. A ValueError is now thrown for incompatible arguments (e.g. handlers specified together with filename/filemode, or filename/filemode specified together with stream, or handlers specified together with stream.

Changed in version 3.8: Added the force parameter.

Changed in version 3.9: Added the encoding and errors parameters.

app.logger.logging.captureWarnings(capture)

If capture is true, redirect all warnings to the logging package. If capture is False, ensure that warnings are not redirected to logging but to their original destinations.

app.logger.logging.critical(msg, *args, **kwargs)

Log a message with severity 'CRITICAL' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

app.logger.logging.debug(msg, *args, **kwargs)

Log a message with severity 'DEBUG' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

app.logger.logging.disable(level=50)

Disable all logging calls of severity 'level' and below.

app.logger.logging.error(msg, *args, **kwargs)

Log a message with severity 'ERROR' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

app.logger.logging.exception(msg, *args, exc_info=True, **kwargs)

Log a message with severity 'ERROR' on the root logger, with exception information. If the logger has no handlers, basicConfig() is called to add a console handler with a pre-defined format.

app.logger.logging.fatal(msg, *args, **kwargs)

Don't use this function, use critical() instead.

app.logger.logging.getLevelName(level)

Return the textual or numeric representation of logging level 'level'.

If the level is one of the predefined levels (CRITICAL, ERROR, WARNING, INFO, DEBUG) then you get the corresponding string. If you have associated levels with names using addLevelName then the name you have associated with 'level' is returned.

If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned.

If a string representation of the level is passed in, the corresponding numeric value is returned.

If no matching numeric or string value is passed in, the string 'Level %s' % level is returned.

app.logger.logging.getLogger(name=None)

Return a logger with the specified name, creating it if necessary.

If no name is specified, return the root logger.

app.logger.logging.getLoggerClass()

Return the class to be used when instantiating a logger.

app.logger.logging.info(msg, *args, **kwargs)

Log a message with severity 'INFO' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

app.logger.logging.log(level, msg, *args, **kwargs)

Log 'msg % args' with the integer severity 'level' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

app.logger.logging.makeLogRecord(dict)

Make a LogRecord whose attributes are defined by the specified dictionary, This function is useful for converting a logging event received over a socket connection (which is sent as a dictionary) into a LogRecord instance.

app.logger.logging.setLoggerClass(klass)

Set the class to be used when instantiating a logger. The class should define __init__() such that only a name argument is required, and the __init__() should call Logger.__init__()

app.logger.logging.shutdown(handlerList=[<weakref at 0x000002ADB930B5B0; to '_StderrHandler'>,
<weakref at 0x000002ADB6A73FB0; to 'NewLineStreamHandler'>, <weakref
at 0x000002ADB8ECA250; to 'WarningStreamHandler'>, <weakref at
0x000002ADBA33EBB0; to 'StreamHandler'>, <weakref at
0x000002ADBABCC4F0; to 'NullHandler'>, <weakref at
0x000002ADBAC83560; to 'StreamHandler'>, <weakref at
0x000002ADBAC65C60; to 'NullHandler'>, <weakref at
0x000002ADBACE4D60; to 'NullHandler'>, <weakref at
0x000002ADBC5AFC90; to 'MemoryHandler'>, <weakref at
0x000002ADBD731030; to 'EmptyLogger'>, <weakref at
0x000002ADBD731350; to 'StreamHandler'>, <weakref at
0x000002ADBD7A2C00; to 'StreamHandler'>, <weakref at
0x000002ADBD7A2C50; to 'PostgresLogHandler'>, <weakref at
0x000002ADBD7AB740; to 'StreamHandler'>, <weakref at
0x000002ADBD7AB830; to 'StreamHandler'>, <weakref at
0x000002ADBD7A8400; to 'StreamHandler'>, <weakref at
0x000002ADBD7AB880; to 'StreamHandler'>, <weakref at
0x000002ADBD7ABC40; to 'StreamHandler'>, <weakref at
0x000002ADBD7C4040; to 'StreamHandler'>, <weakref at
0x000002ADBD7C4400; to 'StreamHandler'>, <weakref at
0x000002ADBD7C4810; to 'StreamHandler'>, <weakref at
0x000002ADBD7C4D10; to 'StreamHandler'>, <weakref at
0x000002ADBD731120; to 'StreamHandler'>, <weakref at
0x000002ADBD7ABFB0; to 'StreamHandler'>, <weakref at
0x000002ADBD7A82C0; to 'StreamHandler'>, <weakref at
0x000002ADBD7A8590; to 'StreamHandler'>, <weakref at
0x000002ADBD7C4EF0; to 'StreamHandler'>, <weakref at
0x000002ADBD7C4630; to 'StreamHandler'>, <weakref at
0x000002ADBD7C4310; to 'StreamHandler'>, <weakref at
0x000002ADBD7C53A0; to 'StreamHandler'>, <weakref at
0x000002ADBD7C5800; to 'StreamHandler'>, <weakref at
0x000002ADBD7C5DA0; to 'StreamHandler'>, <weakref at
0x000002ADBD7C6340; to 'StreamHandler'>, <weakref at
0x000002ADBD7A8680; to 'StreamHandler'>, <weakref at
0x000002ADBD7ABD80; to 'StreamHandler'>, <weakref at
0x000002ADBD7C5F80; to 'StreamHandler'>, <weakref at
0x000002ADBD7C5530; to 'StreamHandler'>, <weakref at
0x000002ADBD7C52B0; to 'StreamHandler'>, <weakref at
0x000002ADBD7C4A90; to 'StreamHandler'>, <weakref at
0x000002ADBD7C58A0; to 'StreamHandler'>, <weakref at
0x000002ADBD7C6700; to 'StreamHandler'>, <weakref at
0x000002ADBD7C6B10; to 'StreamHandler'>, <weakref at
0x000002ADBD7C6FC0; to 'StreamHandler'>, <weakref at
0x000002ADBD7C7510; to 'StreamHandler'>, <weakref at
0x000002ADBD7C7560; to 'PostgresLogHandler'>, <weakref at
0x000002ADBD7ABE70; to 'StreamHandler'>, <weakref at
0x000002ADBD7ABBF0; to 'StreamHandler'>, <weakref at
0x000002ADBD7AB6A0; to 'StreamHandler'>, <weakref at
0x000002ADBD800A90; to 'StreamHandler'>, <weakref at
0x000002ADBD8010D0; to 'StreamHandler'>, <weakref at
0x000002ADBD801620; to 'StreamHandler'>, <weakref at
0x000002ADBD801490; to 'StreamHandler'>, <weakref at
0x000002ADBD800540; to 'StreamHandler'>, <weakref at
0x000002ADBD730270; to 'StreamHandler'>, <weakref at
0x000002ADBDA41940; to 'StreamHandler'>, <weakref at
0x000002ADBDA42FC0; to 'StreamHandler'>, <weakref at
0x000002ADBDA43F60; to 'StreamHandler'>, <weakref at
0x000002ADBD8743B0; to 'StreamHandler'>, <weakref at
0x000002ADBD874770; to 'StreamHandler'>, <weakref at
0x000002ADBD801120; to 'StreamHandler'>, <weakref at

Perform any cleanup actions in the logging system (e.g. flushing buffers).

Should be called at application exit.

app.logger.logging.warn(msg, *args, **kwargs)

app.logger.logging.warning(msg, *args, **kwargs)

Log a message with severity 'WARNING' on the root logger. If the logger has no handlers, call basicConfig() to add a console handler with a pre-defined format.

app.logger.logging.getLogRecordFactory()

Return the factory to be used when instantiating a log record.

app.logger.logging.setLogRecordFactory(factory)

Set the factory to be used when instantiating a log record.

Parameters
factory – A callable which will be called to instantiate

a log record.

app.logger.logging.getLevelNamesMapping()

app.logger.logging.getHandlerByName(name)

Get a handler with the specified name, or None if there isn't one with that name.

app.logger.logging.getHandlerNames()

Return all known handler names as an immutable set.

1.2 models

Models Module

Defines data models used across the application.

1.2.1 result

Uniform Response

Defines a Pydantic model for uniform requests response.

class app.models.result.UniformResponse(*, status_code: int = 200, result: str, data: Any = None)

Bases: BaseModel

Uniform API response structure.

status_code

HTTP status code of the response (default is 200).

Type
int

result

A message describing the outcome of the operation.

Type
str

data
: Additional data returned by the API.

  Type
  : Any

status_code: int

result: str

data: Any

1.2.2 token

Token Request

Defines a Pydantic model for token authentication requests.

class app.models.token.RefreshRequest(*, refresh_token: str)

Bases: BaseModel

refresh_token: str

class app.models.token.TokenRequest(*, secret_hash: str, role: str, login: str | None = None, password: str | None = None)

Bases: BaseModel

Model representing a token request for authentication.

secret_hash
: A secret hash used for authentication.

  Type
  : str

role
: The user's role, e.g., "admin" or "operator".

  Type
  : str

login
: The login identifier for the user, if applicable.

  Type
  : Optional[str]

password
: The user's password, if applicable.

  Type
  : Optional[str]

secret_hash: str

role: str

login: str | None

password: str | None

1.2.3 user

Users Models

Defines SQLAlchemy and Pydantic models for users entities.

class app.models.user.User(**kwargs)

Bases: Base

id

login

password

role

permissions

class app.models.user.UserCreate(*, login: str, password: str, role: str, permissions: List[str] = None)

Bases: UsersBase

class app.models.user.UserInDB(*, login: str, password: str, role: str, permissions: List[str] = None, id: str | None = None)

Bases: UsersBase

id: str | None

class app.models.user.UsersBase(*, login: str, password: str, role: str, permissions: List[str] = None)

Bases: BaseModel

login: str

password: str

role: str

permissions: List[str]

1.3 routers

Routers Package

Contains API route definitions for various modules.

1.3.1 admin

Admin Router

Configures the FastAPI router with admin token verification dependency and includes texts and users routes.

users

User Router

Handles creation, deletion, and listing of user accounts.

async app.routers.admin.users.create_user(user_data: UserCreate, db: Session = Depends(get_db))

Create a new user account.

- Checks for login uniqueness.
- Hashes the password.
- Accepts custom roles and permissions.

Returns a sanitized version of the created user object (without real password).

async app.routers.admin.users.update_permissions(user_id: str, payload: dict, db: Session = Depends(get_db))

Update permissions for an existing user.

- Requires the user ID.
- Expects a JSON body with a list of permissions.
- Returns updated user info without real password.

async app.routers.admin.users.remove_user(user_id: str, db: Session = Depends(get_db))

Delete an existing user by their unique ID.

- Returns a 404 error if user doesn't exist.
- Requires admin or operator rights.

Returns success message on deletion.

app.routers.admin.users.get_users(db: Session = Depends(get_db), limit: int = Query(10), offset: int = Query(0), search: str | None = Query(None), role: str | None = Query(None))

Retrieve a paginated list of users.

- Supports filtering by login (partial match) and user role.
- Requires the "view_users" permission.
- Returns up to 100 users per page.

1.3.2 public

Admin Router

Configures the FastAPI router with admin token verification dependency and includes texts and users routes.

ping

Public Auth

Provides user-accessible endpoints such as profile check.

async app.routers.public.ping.get_current_user(payload=Depends(verify_token))

Retrieve the current authenticated user's payload.

- Requires a valid Bearer access token in the Authorization header.
- Returns the decoded JWT token payload containing:
  - sub: subject (typically username)
  - role: user role
  - permissions: list of permissions

- exp: expiration timestamp
- type: token type (should be 'access')

Useful for user-facing applications to confirm identity and role.

1.3.3 auth

Auth Token

Provides endpoints for token generation and refresh.

async app.routers.auth.get_token(request: Request, payload: TokenRequest, db: Session = Depends(get_db))

Generates access and refresh tokens based on role and credentials.

- If role is user, only the secret hash is required.
- If role is admin or operator, login and password are also required.
- Automatically creates the user if they don't exist.
- Returns access and refresh tokens.

async app.routers.auth.refresh_token(request: Request, payload: RefreshRequest)

Issues a new access token using a valid refresh token.

- Requires a valid JWT refresh token in the request body.
- Verifies that the token is of type 'refresh'.
- Returns a new access token.

1.4 services

Services Package

Provides auxiliary services for the application.

1.4.1 backup

Automatic Backups

Schedules PostgreSQL backups using pg_dump.

app.services.backup.run_backup()

Creates a compressed PostgreSQL backup using pg_dump. Keeps only the last 10 backups in BACKUP_PATH.

1.5 tests

1.5.1 conftest

Test setup for FastAPI application using PostgreSQL.

app.tests.conftest.setup_test_database()

app.tests.conftest.db()

app.tests.conftest.override_get_db(db)

app.tests.conftest.client(override_get_db)

app.tests.conftest.admin_token()

app.tests.conftest.user_token()

1.5.2 test_auth

Tests for token generation and refresh endpoints.

app.tests.test_auth.test_get_token_user(client: TestClient)

Test generating access and refresh tokens for a user.

- Sends a POST request to /token with role 'user', login, password, and valid secret_hash.
- Expects a 200 OK response.
- Asserts that both access_token and refresh_token are present in the response.

app.tests.test_auth.test_get_token_admin_create_and_login(client: TestClient)

Test token generation for an admin with user auto-creation and authentication.

- Sends a POST request with admin role, login, password, and secret_hash.
- Expects user creation on first call and authentication on the second.
- Verifies both calls return valid tokens.

app.tests.test_auth.test_invalid_secret_key(client: TestClient)

Test access denial when an invalid secret_hash is used.

- Sends a POST request to /token with an incorrect secret_hash.
- Expects 403 Forbidden response.
- Verifies the error message mentions invalid token secret.

app.tests.test_auth.test_refresh_token(client: TestClient)

Test refreshing an access token using a valid refresh token.

- Sends a POST request to /token to obtain tokens.
- Uses the refresh_token in a POST request to /refresh.
- Expects 200 OK and a new access_token in the response.

app.tests.test_auth.test_invalid_refresh_token_type(client: TestClient, user_token: str)

Test error when using an access token instead of a refresh token for refreshing.

- Sends an access_token in a POST request to /refresh.
- Expects 403 Forbidden response.
- Checks for error message "Expected refresh token".

1.5.3 test_public

Tests for the /me public endpoint.

app.tests.test_public.test_get_current_user_payload(client: TestClient, user_token: str)

Test successful retrieval of current user payload using valid JWT.

- Sends a GET request to /me with a valid Authorization header.
- Expects a 200 OK response.

- Asserts that the returned payload contains the correct role.

app.tests.test_public.test_get_current_user_no_token(client: TestClient)

Test error when accessing /me endpoint without Authorization header.

- Sends a GET request to /me without any token.
- Expects 401 Unauthorized response.
- Verifies that the response includes 'Missing Authorization header'.

1.5.4 test_users

Tests for user creation, deletion, listing, and access control.

app.tests.test_users.test_create_user(client: TestClient, admin_token: str)

Test user creation via /create_user endpoint.

- Sends a POST request with login, password, role, and permissions.
- Expects a 200 OK response and correct user data in response body.

app.tests.test_users.test_create_user_duplicate(client: TestClient, admin_token: str)

Test duplicate user creation.

- Creates a user with a specific login.
- Attempts to create the same user again.
- Expects a 400 Bad Request with a message about duplicate login.

app.tests.test_users.test_get_users_list(client: TestClient, admin_token: str)

Test retrieving a paginated list of users.

- Creates 15 users.
- Sends a GET request with limit and offset parameters.
- Expects a list of users with length <= limit.

app.tests.test_users.test_remove_user(client: TestClient, admin_token: str)

Test user deletion by ID.

- Creates a user.
- Deletes the user using their ID.
- Expects a 200 OK response with confirmation message.

app.tests.test_users.test_remove_nonexistent_user(client: TestClient, admin_token: str)

Test deletion of a non-existent user.

- Sends a DELETE request with a random UUID.
- Expects a 404 Not Found response.

app.tests.test_users.test_update_user_permissions(client: TestClient, admin_token: str)

Test updating user permissions.

- Creates a user with initial permissions.
- Sends a PUT request to update their permissions.
- Expects a 200 OK response and updated permission list in the result.

1.6 utils

Utils Module

Provides utility functions and helpers used across the application

1.6.1 auth

Auth Module

Provides utility functions and helpers used across the application

hash

Hash Utility

Provides a function for generating a SHA-256 hash from a given string.

app.utils.auth.hash.hash_str(s: str) → str

Generates a SHA-256 hash for the provided string.

Parameters
s (str) – The input string to hash.

Returns
The hexadecimal SHA-256 hash of the input string.

Return type
str

jwt_handler

JWT Handler

Provides functions for creating and verifying JWT tokens for authentication, role validation, and permission-based access control.

app.utils.auth.jwt_handler.get_token(authorization: str = Depends(APIKeyHeader)) → str

Extracts token from Authorization header.

app.utils.auth.jwt_handler.create_access_token(data: Dict[str, Any], expires_delta: timedelta = datetime.timedelta(seconds=900)) → str

Creates a JWT access token.

app.utils.auth.jwt_handler.create_refresh_token(data: Dict[str, Any], expires_delta: timedelta = datetime.timedelta(days=7)) → str

Creates a JWT refresh token.

app.utils.auth.jwt_handler.verify_token(token: str = Depends(get_token), expected_type: str = 'access') → Dict[str, Any]

Verifies a JWT token (access or refresh) and returns its payload.

app.utils.auth.jwt_handler.verify_refresh_token(token: str = Depends(get_token)) → Dict[str, Any]

Verifies a JWT refresh token and returns its payload.

app.utils.auth.jwt_handler.require_permission(permission: str)

Dependency that checks if the user has a specific permission in token payload. Usage:

@router.get("/secure", dependencies=[Depends(require_permission("read_users"))])

static_protection

Static protections

Secured code docs route with login and password

class app.utils.auth.static_protection.ProtectedStaticFiles(*, directory: str | PathLike[str] | None = None, packages: List[str | Tuple[str, str]] | None = None, html: bool = False, check_dir: bool = True, follow_symlink: bool = False)

Bases: StaticFiles

async get_response(path: str, scope)

Returns an HTTP response, given the incoming path, method and request headers.

swagger_auth

Swagger Auth

Module for protecting access to Swagger documentation using HTTP Basic authentication.

app.utils.auth.swagger_auth.get_swagger_basic_auth(credentials: HTTPBasicCredentials = Depends(HTTPBasic)) → str

Dependency for HTTP Basic authentication to access Swagger documentation.

Parameters
credentials (HTTPBasicCredentials) – User-provided credentials.

Returns
The authenticated username.

Return type
str

Raises
HTTPException – If the provided credentials are invalid.

1.6.2 base_handler

Base Handler

Decorator for uniform API response formatting, and error handling for both async and sync functions.

app.utils.base_handler.handle_exception(func_name: str, args: Any, kwargs: Any, e: Exception) → UniformResponse

app.utils.base_handler.base_handler(func: F) → Callable[[...], Any]

Wraps an endpoint function with unified exception handler and response formatter.

Constants

app.utils.base_handler.F

1.6.3 rate_limit

Rate Limiting

Provides global rate limiting configuration using slowapi.

1.7 config

Config Settings

Module for loading application configuration settings from a .env file.

class app.config.Settings

Bases: object

SECRET_KEY: str = 'default-secret'

ADMIN_SWAGGER_PASSWORD: str = 'admin_pass'

ADMIN_SWAGGER_LOGIN: str = 'admin_login'

POSTGRES_DB: str = 'auth_db'

POSTGRES_USER: str = 'auth_user'

POSTGRES_PASSWORD: str = 'password'

POSTGRES_PORT: str = '5432'

POSTGRES_HOST: str = 'localhost'

POSTGRES_URL: str = 'postgresql://auth_user:password@localhost:5432/auth_db'

BACKUP_PATH: str = '/app/backups'

DOCS_DIR: Path = WindowsPath('C:/Users/miald/storage/Progarmming/dino-projects/PythonProject/docs/html')

LOG_TO_DB: bool = True

TESTING: bool = True

1.8 database

PostgreSQL Connection

Connects to PostgreSQL database using SQLAlchemy.

app.database.get_db()

1.9 main

Main Application

Creates a FastAPI instance, connects routers, applies rate limiter, CORS, docs, and startup tasks.

app.main.get_documentation(credentials: str = Depends(get_swagger_basic_auth))

app.main.get_redoc_documentation(credentials: str = Depends(get_swagger_basic_auth))

app.main.ping()

async app.main.start_scheduler_if_not_running()

async app.main.startup_event()