# EEZZ

*Release 1.0*

Albert Zedlitz

Feb 19, 2024

# Contents:

EEZZ provides a bidirectional user interface for Python to an HTML browser using WEB-sockets, which allows you to develop software in two galvanic seperated threads.

# 1 eezz

## 1.1 eezz package

### 1.1.1 Submodules

### 1.1.2 module eezz.blueserv

The module bueserv handles the bluetooth features of EEZZ. and implements the following classes

- `eezz.table.TBluetooth`: TTable for listing bluetooth devices in range
- `eezz.table.TBluetoothService`: Communicates with bluetooth-service EEZZ on mobile device

`class eezz.blueserv.TBluetoothService(`*address*`)`

Bases: `object`

The TBluetoothService handles a connection for a specific mobile given by address. The class is defined as dataclass, so that the call parameters become properties.

**Variables**

- `eezz_service` – The service GUID of the eezz App
- `m_lock` – Lock communication for a single request/response cycle
- `bt_socket` – The communication socket
- `bt_service` – The services associated with the eezz App
- `connected` – Indicates that the service is active

`address: str`

Property - The address of the bluetooth device

`open_connection()`

Open a bluetooth connection

`send_request(`*command: str*, *args: list*`)` → dict

A request is send to the device, waiting for a response. The protocol use EEZZ-JSON structure: send -> {message: str, args: list} receive -> {return: dict { code: int, value: str}, …}

**Parameters**

- `command` – The command to execute
- `args` – The arguments for the given command

**Returns**
JSON structure send by device

`shutdown()`

Shutdown interrupts open connections, stops the port-select and closes all open sockets.

`class eezz.blueserv.TBluetooth(`*None*`)`

Bases: `TTable`

The bluetooth class manages bluetooth devices in range A scan_thread is started to keep looking for new devices. TBluetooth service is a singleton to manage this consistently

**Variables**

- `column_names` – Constant list [‘Address’, ‘Name’]
- `title` – Constant title = bluetooth devices
- `bt_table_changed` – Condition: Signals table change events

`find_devices()` → None

This method is called frequently by thread *self.scan:bluetooth* to keep track of devices in range. The table is checked for new devices or devices which went out of range. Only if the list changes the condition TTable.async_condition is triggered with notify_all.

`get_visible_rows(`*get_all: bool = False*`)` → List[*TTableRow*]

Return the visible rows at the current cursor

**Parameters**
`get_all` – A bool value to overwrite the visible_items for the current call

**Returns**
A list of visible row items

### 1.1.3 module eezz.database

This module handles the database access and implements the following classes

- *`eezz.database.TDatabaseTable`*: Create database from scratch. Encapsulate database access.
- *`eezz.database.TDatabaseColumn`*: Extends the TTableColumn by parameters, which are relevant only for database access

The TDatabaseTable allows flexible usage of database and buffer access of data by switching seamlessly and hence make standard access very easy and performant. The database is created in sqlite3 in the first call of the class.

`class eezz.database.TDatabaseTable(`*column_names*, *title*`)`

Bases: `TTable`

General database management Purpose of this class is a sophisticate work with database using an internal cache. All database operations are mapped to TTable. The column descriptor is used to generate the database table. The database results are restricted to the visible scope Any sort of data is launched to the database Only the first select statement is executed. For a new buffer, set member is_synchron to False The property column_names and title are inherited from TTable.

**Variables**

- `statement_select` – (str): Select statement, inserting limit and offset according to TTable settings: `select <TTable.column_names> from <TTable.title>... limit <TTable.visible_items>... offset <TTable.offset>... where...`
- `statement_count` – (str): Evaluates the number of elements in the database `select count (*) ...`
- `statement_create` – (str): Create statement for database table: `create <TTable.title> <[List of TTable.column_names]> ... primary keys <[list of TDatabaseColumn.primary_key]>`
- `is_synchron` – (bool): If True, select data from cache, else select from database
- `column_descr` – (List[TDatabaseColumn]): Properties for each column
- `virtual_len` – (int): The number of entries in the database

`append`(*table_row: list*, *attrs: dict = None*, *row_type: str = 'body'*, *row_id: str = ''*, *exists_ok: bool = True*) → *TTableRow*

Append data to the internal table, creating a unique row-key The row key is generated using the primary key values as comma separated list. You select from list as (no spaces) do_select(row_id = 'key_value1,key_value2,...')

**Parameters**

- `exists_ok` – If set to True, do not raise exception, just ignore the appending silently
- `table_row` (*List [ Any ]*) – A list of values as row to insert
- `attrs` (*dict*) – Optional attributes for this row
- `row_type` – Row type used to trigger template output for HTML
- `row_id` (*SHA256 hash of primary key values*) – Unique row-id, calculated internal, if not set

**Returns**

List of rows with length of TTable.visible_items

**Return type**

List[*TTableRow*]

`commit`()

Write all new entries to database, which have been added using method *append()*

`db_create`() → None

Create the table on the database using *eezz.table.TTable.column_names* and *eezz.table.TTable.title*

`do_select`(*filters: dict*, *get_all: bool = False*) → List[*TTableRow*]

Works on local buffer, as long as the scope is not changed. If send to database the syntax of the values have to be adjusted to splite3-like[1]

**Parameters**

- `get_all` – Ignore TTable.visible_items for this call
- `filters` – Dictionary with column names as key and corresponding regular expression values to filter

`get_visible_rows(`*get_ all=False*`)`

Get visible rows. Works on local buffer for `eezz.database.TDatabaseTable.is_synchron`

**Parameters**

`get_all` – Ignore TTable.visible_items for this call

`navigate(`*where_ togo: TNavigation = TNavigation.NEXT*, *position: int = 0*`)` → None

Navigate in block mode

**Parameters**

- `where_togo` (`TNavigation`) – Navigation direction
- `position` – Use database access if position > 0, disabling absolute positioning for database cursor and make it easy to distinguish different access types.

`prepare_statements()`

Generate a set of consistent database statements, used to select and navigate in database and to sync with TTable buffers

`class eezz.database.TDatabaseColumn(`*primary_ key*, *options*, *alias*`)`

Bases: `TTableColumn`

Extension for column descriptor TTableColumn

`alias: str = ''`

Property - Name for the column in prepared statements `insert <column>... values(:<alias>, ...)`

`options: str = ''`

Property - Database option for column creation (e.g. not null). `create table ... column text not null`

`primary_key: bool = False`

Property - Makes a column a primary key. In TTable the row-id is calculated as SHA256 hash on primary key values

### 1.1.4 module eezz.document

This module implements the following classes

- `eezz.document.TManifest`: Document header representation. The header is a dictionary with a given structure and a defined set of keys and sub-keys. The manifest defines the database table and access. The manifest is the structure, which is signed and which is used to identify and verify the document.
- `eezz.document.TDocument`: A document consists of more than one file and the manifest. Part of the document is encrypted. The document key could be used in combination with a mobile device to decrypt the file.

The document module allows download of files in chunks and encryption/decryption. The method `eezz.document.TDocument.zip()` creates a partial encrypted archive with a signed header. the manifest. The method unzip will check this header before unpacking. There is a rudimentary idea implemented to trade self-consistent multi media files, using eezz server as transaction platform.

`class eezz.document.TDocuments(`*path*, *name*`)`

Bases: `TDatabaseTable`

Manages documents There are two ways to start the document:

[1] https://www.sqlitetutorial.net/sqlite-like/

1. Create a document using prepare_download, handle_download and create As a result the document is zipped in TAR format with a signed manifest. The key for decryption is stored on the mobile device and on EEZZ
2. Open a document, reading the manifest. Noe you could check if you have the key on your mobile device, or you buy the key from EEZZ

**Variables**

- `path` (*Path*) – Document file name
- `name` (*str*) – Document name
- `files_list` (*List [TFile]*) – List of files
- `key` (*bytes*) – Document key
- `vector` (*bytes*) – Document vector
- `manifest` (*TManifest*) – Document header

`add_document_to_device()` → dict

`column_names: List[str] = None`

List of column names is calculated for this class

`create_document(`*name: str*, *nr_files: int*, *queue: Queue[TEezzFile]*`)` → None

After all files downloaded, The document header is registered on eezz server and signed All files are zipped together with this header.

**Parameters**

- `name` – Name of the document on disk
- `nr_files` – Number of files in the queue
- `queue` – The process queue

`decrypt_key_with_device(`*encrypted_key: bytes*`)` → bytes | None

Decrypt the document key

**Parameters**

`encrypted_key` – The encrypted key

**Returns**

The decrypted key, vector pair

`eezz_buy_document(`*transaction_key: bytes*`)`

Buy transaction to get the document key.

**Parameters**

`transaction_key` – With the method `eezz.document.TDocument.eezz_get_document_key()` you get the key, if you are owner or you get a transaction_key, which could be used in this method to buy the key. Once you own the key, you could store it in local database. The document key is encrypted with the mobile device key.

`eezz_get_document_key(`*buy_request=False*`)` → dict

Get the document key from EEZZ server

**Parameters**

`buy_request` (*bool = False*) – If True, a transaction key is created, if the called is not yet owner

**Returns**

`eezz_register_document()` → dict

Registers the document header to EEZZ and returns it signed with the EEZZ key. The signed header is stored as manifest in the final document.

`encrypt_key_with_device(`*key: bytes*, *vector: bytes*`)` → bytes | None

Encrypt the document key

**Parameters**

- `key` – Document key
- `vector` – Document vector

**Returns**

encrypted document key

`get_device_key()` → bytes | None

`handle_download(`*request: dict*, *raw_ data: Any*`)` → dict

Handle file download

**Parameters**

- `request` – Download reqeust
- `raw_data` – Data chunk to write

**Returns**

Update response

`prepare_download(`*request: dict*`)`

Prepares the download of several files to include to an EEZZ document. The preparation puts all file descriptors into a queue and waits to all documents until the last download. This triggers the creation of the EEZZ document

**Parameters**

`request` – The json format of a WEB socket request

`read_document_header(`*source: Path*`)` → bool

If a customer finds an EEZZ document, this method opens the zipped content and verifies the header. With the verified header, the document could be unzipped.

**Parameters**

`source` –

**Returns**

`unzip(`*source: Path*, *manifest_ only=False*, *document_ key: bytes = None*`)` → dict

Unzip a file and return the Manifest in JSON format. Unzip needs the document key. If the key is not available, unzip the preview and the manifest, store the result into database for further processing

**Parameters**

- `source` –
- `manifest_only` – Extract the header
- `document_key` – If set, try to decrypt the document on the fly

**Returns**

The header

`zip`(*destination: Path*, *manifest: dict*, *files: List[TFile]*)

Zip the given files and the manifest to an EEZZ document.

**Parameters**

- `destination` – Path to the EEZZ document. Has to be like <directory>/<filename>
- `manifest` – Description and header
- `files` – Files included for the document

## 1.1.5 module eezz.filesrv

This module implements the following classes:

- **TFile**: Takes a chunk of data and merges it to a file
- **TEezzFile**: Extends TFile and implements encryption and decryption for transmitted data
- **TFileMode**: Enum file-mode for TEezzFiles

This module supports a download of big files in chunks and ensures, that the incoming fragments are put together in the correct order again. Furthermore, a hash is calculated for each chunk, so that the data consistency of a file could be ensured during reading.

`class eezz.filesrv.TEezzFile`(**, file_type: str, destination: Path, size: int, chunk_size: int, key: bytes, vector: bytes, response: Queue = None, hash_chain: dict = None*)

Bases: *TFile*

Derived from TFile, this class allows encryption and decryption using AES key. After finishing the transfer, the instance is pushed into the response queue, which allows to implement a supervisor thread, which blocks on the queue reading

**Parameters**

- `key` (*Crypto.Random.new(16)*) – AES key for cypher
- `vector` (*Crypto.Random.new(16)*) – AES vector for cypher
- `response` (*Queue.queue*) – Queue to get the final state of an instance
- `hash_chain` (*List[SHA256.hexdigest]*) – A list of hash values for each chunk

`decrypt`(*raw_data: Any*, *sequence_nr: int*) → str

Decrypt the incoming stream

**Parameters**

- `raw_data` – Data chunk of the steam
- `sequence_nr` – Sequence number in the stream

**Returns**

Hash value of the chunk

**Return type**

SHA256.hexdigest

`encrypt`(*raw_data: Any*, *sequence_nr: int*) → str

Encrypt the incoming data stream

**Parameters**

- `raw_data` – Data chunk of the stream
- `sequence_nr` – Sequence number in the stream

**Returns**

Hash value of the chunk

**Return type**

SHA256.hexdigest

`read`(*source: BufferedReader*, *hash_list: List[str] = None*) → None

Read an encrypted file from source input stream and create an decrypted version

**Parameters**

- `source` – Input stream
- `hash_list` (*List [SHA256.hexdigest ]*) – A hash list to check the stream

`write`(*raw_data: Any*, *sequence_nr: int*, *mode:* TFileMode *= TFileMode.ENCRYPT*) → str

Write a chunk of data

**Parameters**

- `raw_data` – The data chunk to write
- `sequence_nr` – Sequence of the data chunk in the stream
- `mode` – The mode used to en- or decrypt the data or pass through

**Returns**

The hash value of the data after encryption/before decryption

**Return type**

SHA256.hexdigest

`class eezz.filesrv.TFile`(***, *file_type: str*, *destination: Path*, *size: int*, *chunk_size: int*)

Bases: `object`

Class to be used as file download handler. It accepts chunks of data in separate calls

**Parameters**

- `file_type` – User defined file type
- `destination` – Path to store the file
- `size` – The size of the file
- `chunk_size` – Fixed size for each chunk of data, except the last element

`write`(*raw_data: Any*, *sequence_nr: int*, *mode:* TFileMode *= TFileMode.NORMAL*) → str

Write constant chunks of raw data to file. Only the last chunk might be smaller. The sequence number is passed along, because we cannot guarantee, that elements received in the same order as they are send.

**Parameters**

- `raw_data` – Raw chunk of data
- `sequence_nr` – Sequence number to insert chunks at the right place

- `mode` – Ignored: set signature for derived classes

**Returns**
Empty string: set signature for derived classes

`enum eezz.filesrv.TFileMode(`*value*`)`

Bases: `Enum`

File mode: Determine how to handle incoming stream

**Parameters**

- `NORMAL` – Write through
- `ENCRYPT` – Encrypt and write
- `ENCRYPT` – Decrypt and write

Valid values are as follows:

`NORMAL = <TFileMode.NORMAL: 0>`

`ENCRYPT = <TFileMode.ENCRYPT: 1>`

`DECRYPT = <TFileMode.DECRYPT: 2>`

`eezz.filesrv.test_file_reader()`

Test the TFile interfaces :meta private:

### 1.1.6 module eezz.http_agent

- **THttpAgent**: Handle WEB-Socket requests

The interaction with the JavaScript via WEB-Socket includes generation of HTML parts for user interface updates

`class eezz.http_agent.THttpAgent`

Bases: `TWebSocketAgent`

Agent handles WEB socket events

`compile_data(`*a_parser: Lark*, *a_tag_list: list*, *a_id: str*, *a_query: dict = None*`)` → None

Compile data-eezz-json to data-eezz-compile, create tag attributes and generate tag-id to manage incoming requests

**Parameters**

- `a_parser` – The Lark parser to compile EEZZ to json
- `a_tag_list` – HTML-Tag to compile
- `a_id` – The ID of the tag to be identified for update
- `a_query` – The query of the HTML request

**Returns**
None

`do_get(`*a_resource: Path | str*, *a_query: dict*`)` → str

Response to an HTML GET command

The agent reads the source, compiles the data-eezz sections and adds the web-socket component It returns the enriched document

**Parameters**

- `a_resource` (*pathlib.Path*) – The path to the HTML document, containing EEZZ extensions
- `a_query` – The query string of the URL

**Returns**

The compiled version of the HTML file

`format_attributes(`*a_key: str*, *a_value: str*, *a_fmt_funct: Callable*`)` → str

Eval template tag-attributes, diving deep into data-eezz-json

**Parameters**

- `a_key` – Thw key string to pick the items in a HTML tag
- `a_value` – The dictionary in string format to be formatted
- `a_fmt_funct` – The function to be called to format the values

**Returns**

The formatted string

`generate_html_cells(`*a_tag: Tag*, *a_cell:* TTableCell`)` → Tag

Generate HTML cells Input for the lamda is a string and output is formatted according to the TTableCell object

**Parameters**

- `a_tag` – The parent tag to generate the table cells
- `a_cell` – The template cell to format to HTML

**Returns**

The formatted HTML tag

`generate_html_grid(`*a_tag: Tag*`)` → dict

Besides the table, supported display is grid (via class clzz_grid or select

`generate_html_grid_item(`*a_tag: Tag*, *a_row:* TTableRow, *a_header:* TTableRow`)` → Tag

Generates elements of the same kind, derived from a template and update content according the row values

`generate_html_rows(`*a_html_cells: list*, *a_tag: Tag*, *a_row:* TTableRow`)` → Tag

This operation add fixed cells to the table. Cells which are not included as template for table data are used to add a constant info to the row

**Parameters**

- `a_html_cells` – A list of cells to build up a row
- `a_tag` – The parent containing the templates for the row
- `a_row` – The table row values to insert

**Returns**

The row with values rendered to HZML

`generate_html_table(`*a_table_tag: Tag*`)` → dict

Generates a table structure in four steps

1. Get the column order and the viewport
2. Get the row templates

3. Evaluate the table cells
4. Send the result separated by table main elements

**Parameters**
`a_table_tag` – The parent table tag to produce the output

`handle_download(`*request_data: dict*, *raw_data: Any*`)` → str

Handle file downloads: The browser slices the file into chunks and the agent has to re-arrange the stream using the file name and the sequence number

**Parameters**

- `request_data` – The request data are encoded in dictionary format
- `raw_data` – The rae data chunk to download

**Returns**
Progress information to the update destination of the event

`handle_request(`*request_data: dict*`)` → str | None

Handle WEB socket requests

- **initialize**: The browser sends the complete HTML for analysis.
- **call**: The request issues a method call and the result is send back to the browser

**Parameters**
`request_data` – The request send by the browser

**Returns**
Response in JSON stream, containing valid HTML parts for the browser

`setup_download(`*request_data: dict*`)` → str

This method is called before a download of files starts

### 1.1.7 module eezz.seccom

Copyright (C) 2015 www.EEZZ.biz (haftungsbeschränkt)

TSecureSocket: Implements secure communication with eezz server using RSA and AES encryption

`class eezz.seccom.TSecureSocket`

Bases: `object`

`send_request(`*a_action*, *a_header=None*, *a_data=None*`)`

### 1.1.8 module eezz.server

EezzServer: High speed application development and high speed execution based on HTML5

Copyright (C) 2015 Albert Zedlitz

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

`class eezz.server.THttpHandler(`*request*, *client_address*, *server*`)`

Bases: `SimpleHTTPRequestHandler`

HTTP handler for incoming requests

`do_GET()`

handle GET request

`do_POST()`

handle POST request

`handle_request()`

handle GET and POST requests

`shutdown(`*args: int = 0*`)`

`class eezz.server.TWebServer(`*a_server_address*, *a_http_handler*, *a_web_socket*`)`

Bases: `HTTPServer`

WEB Server encapsulate the WEB socket implementation

`shutdown()`

Stops the serve_forever loop.

Blocks until the loop has finished. This must be called while serve_forever() is running in another thread, or it will deadlock.

`eezz.server.shutdown_function(`*handler:* THttpHandler`)`

### 1.1.9 module eezz.service

This module implements the following classes:

- **TGlobalService**: Container for global environment
- **TService**: A singleton for TGlobalService
- **TServiceCompiler**: A Lark compiler for HTML EEZZ extensions
- **TTranslate**: Extract translation info from HTML to create a POT file
- **TQuery**: Class representing the query of an HTML request

`class eezz.service.TGlobal`

Bases: `object`

`classmethod get_instance(`*cls_type*`)`

`instances: dict = {}`

`class eezz.service.TQuery(`*query: dict*`)`

Bases: `object`

Transfer the HTTP query to class attributes

**Parameters**

`query` – The query string in dictionary format

`class eezz.service.TService(`**, root_path: Path = None, document_path: Path = None, application_path: Path = None, public_path: Path = None, resource_path: Path = None, locales_path: Path = None, host: str = 'localhost', websocket_addr: int = 8100, global_objects: dict = None, translate: bool = False, async_methods: Dict[Callable, Thread] = None, private_key: RsaKey = None, public_key: RsaKey = None, database_path: Path = None, eezz_service_id: str = None*`)`

Bases: `object`

Container for global environment

`application_path: Path = None`

Path to applications using the browser interface

`assign_object(`*obj_id: str, description: str, attrs: dict, a_tag: Tag = None*`)` → None

assign_object Assigns an object to an HTML tag

**Raises**

- `IndexError` – description systax does not match
- `AttributeError` – Class not found

**Parameters**

- `obj_id` – Unique object-id
- `description` – Path to the class: <directory>.<module>.<class>
- `attrs` – Attributes for the constructor
- `a_tag` – Parent tag which handles an instance of this object

`async_methods: Dict[Callable, Thread] = None`

`database_path: Path = None`

`document_path: Path = None`

Path to EEZZ documents

`eezz_service_id: str = None`

`classmethod get_instance(`*class_type=None*`)`

`get_method(`*obj_id: str, a_method_name: str*`)` → tuple

Get a method by name for a given object

**Raises**

`AttributeError` – Class has no method with the given name

**Parameters**

- `obj_id` – Unique hash-ID for object as stored in *`eezz.service.TService.assign_object()`*
- `a_method_name` –

**Returns**
tuple(object, method, parent-tag)

`get_object(`*obj_id: str*`)` → Any

Get the object for a given ID

**Parameters**
`obj_id` – Unique hash-ID for object as stored in `eezz.service.TGlobalService.assign_object()`

**Returns**
The assigned object

`global_objects: dict = None`

`host: str = 'localhost'`

`locales_path: Path = None`

`private_key: RsaKey = None`

`public_key: RsaKey = None`

`public_path: Path = None`

`resource_path: Path = None`

`root_path: Path = None`

Root path for the HTTP server

`classmethod set_instance(`*instance*`)`

`singletons: ClassVar[dict] = {}`

`translate: bool = False`

`websocket_addr: int = 8100`

`class eezz.service.TServiceCompiler(`*a_tag: Tag*, *a_id: str = ''*, *a_query: dict = None*`)`

Bases: `Transformer`

Transforms the parser tree into a list of dictionaries The transformer output is in json format

**Parameters**

- `a_tag` (*`BeautifulSoup4.Tag`*) – The parent tag
- `a_id` – A unique object id
- `a_query` – The URL query part

`static assignment(`*item*`)`

Parse 'assignment' expression: `variable = value`

`static download(`*item*`)`

Parse 'download' section

`static escaped_str(`*item*`)`

Parse an escaped string

`static format_string(`*item*`)`

Create a format string: `{value}`

static format_value(*item*)

Create a format string: {key.value}

funct_assignment(*item*)

Parse ‘function’ section

static list_arguments(*item*)

Accumulate arguments for function call

static list_updates(*item*)

Accumulate ‘update’ statements

post_init(*item*)

Parse ‘post-init’ section for function assignment

static qualified_string(*item*)

Parse a qualified string: part1.part2.part3

static simple_str(*item*)

Parse a string token

table_assignment(*item*)

Parse ‘assign’ section, assigning a Python object to an HTML-Tag The table assignment uses TQuery to format arguments In case the arguments are not all present, the format is broken and process continues with default

template_section(*item*)

Create tag attributes

static update_item(*item*)

Parse ‘update’ expression

static update_section(*item*)

Parse ‘update’ section

class eezz.service.TTranslate

Bases: object

static generate_pot(*a_soup*, *a_title*)

Generate a POT file from HTML file

**Parameters**

- a_soup – The HTML page for translation
- a_title – The file name for the POT file

eezz.service.test_parser(*source: str*)

### 1.1.10 module eezz.session

This module implements the following classes

- *eezz.session.TSession*: User session management

The class is created by a call to the local browser site with SID and user-name as query parameter. This is done automatically in the EEZZ environment during login in autostart. The session is stored as singleton in the global storage of the HTTP server.

`class eezz.session.TSession(`*sid*, *name*`)`

Bases: `TTable`

TSession implements the interface to Windows users. In a first step the user connects to the HTTP server with SID and NAME. Within the connect call a thread is started to sync with bluetooth device search, with the intention to connect to the EEZZ-App on this device. Pairing is supported with standard UI interfaces to select the device from GUI and register the user. After this, the user could choose to store the password on the device to allow automatic lock and unlock feature of the EEZZ Windows installation.

**Variables**

- `desktop_connected` (*bool*) – Connected to the desktop user
- `device_connected` (*bool*) – Connected device and desktop user
- `paired_device` (`TTableRow`) – Data of connected device
- `bt_service` (`TBluetoothService`) – Bluetooth communication protocol
- `bt_devices` (`TBluetooth`) – Table listing bluetooth devices in range
- `mb_devices` (`TMobileDevices`) – Table with paired devices

`connect(`*local_user: dict*`)`

Connect a Windows user to EEZZ interface. This method is called using html: http://localhost:<port>/eezzyfree?sid=<user-sid>,name=<user-name>

**Parameters**
`local_user` – The user to connect to GUI

`get_user_pwd()` → dict

Called by external process to unlock workstation

**Returns**
The password to unlock the workstation

`handle_bt_devices()`

Interact with the bluetooth search *eezz.blueserv.TBluetooth.find_devices()*. This method is called as thread target in :py:meth:eezz.session.TSession.connect` and keeps loop as long as the connection to desktop user is established

`name: str = None`

Property - Windows login-user name

`pair_device(`*address: str*, *password: str*`)` → bool

Stores the user password on mobile device. The password is encrypted and the key is stored in the Windows registry. This method is called by the user interface - The user has to be connected to eezz, which is automatically done using the TaskBar tool - The address has to be selected via user interface

**Parameters**

- `address` –
- `password` – Password will be encrypted and stored on device for unlock workstation

**Returns**
EEZZ Confirmation message as dict

`read_windows_registry()`

Read user data from windows registry

`register_user(`*password: str*, *alias: str*, *fname: str*, *lname: str*, *email: str*, *iban: str = ''*`)` → dict

Register user on EEZZ server. The request is send to the mobile device, which enriches the data and then forwards it to the eezz server page.

**Parameters**

- `alias` – Display name of the user
- `fname` – First name
- `lname` – Last name
- `email` – E-Mail address
- `iban` – Payment account
- `password` – Password for the service. Only the hash value is stored, not the password itself

**Returns**
Status message

`send_bt_request(`*command: str*, *args: list*`)` → dict

`sid:  str = None`

Property - Windows login-user SID

### 1.1.11 module eezz.table

This module implements the following classes:

- *eezz.table.TTableCell*: Defines properties of a table cell
- *eezz.table.TTableRow*: Defines properties of a table row, containing a list of TTableCells
- *eezz.table.TTableColumn*: Defines properties of a table column
- *eezz.table.TTable*: Defines properties of a table, containing a list of TTableRows
- `eezz.table.TTableInsertException`: Exception on checking the row-id, which has to be unique

TTable is used for formatted ASCII output of a table structure. It allows to access the table data for further processing e.g. for HTML output. The class handles an internal read cursor, which allows to navigate in the list of rows and to read a fixed amount of rows.

TTable is a list of TTableRow objects, each of which is a list of TCell objects. The TTableColumn holds the as well column names as types and is used to organize sort and filter. A TTableCell object could hold a TTable object for recursive tree structures.

Besides this the following enumerations are used

- *eezz.table.TNavigation*: Enumeration for method *eezz.table.TTable.navigate()*
- *eezz.table.TSort*: Enumeration for method *eezz.table.TTable.do_sort()*

class eezz.table.TTable(*column_names*, *title*)

Bases: UserList

The table is derived from User-list to enable sort and list management This is a dataclass, so the arguments become properties

**Variables**

- column_names_map (*Dict[str, TTableCell]*) – Map names for output to re-arrange order
- column_names_alias (*Dict[str, str]*) – Map alias names to column names. This could be used to translate the table header without changing the select statements.
- column_names_filter (*List[int]*) – Map columns for output, allows selecting a subset and rearanging, without touching the internal structure of the table
- column_descr (*List[TTableColumn]*) – Contains all attributes of a column like type and width
- table_index (*Dict[str, TTableRow]*) – Managing an index for row-id
- visible_items (*int*) – Number of visible items: default is 20
- offset (*int*) – Cursor position in data set
- header_row (TTableRow) – Header row
- apply_filter_column (*bool*) – Choose between a filtered setup or the original header
- format_types (*Dict[str, Callable]*) – Maps a column type to a formatter for ASCII output

### Examples

Table instance:

```
>>> from table import TTable
>>> my_table = TTable(column_names=['FileName', 'Size'], title='Directory')
>>> for file in Path('.').iterdir():
>>>     my_table.append(table_row=[file, file.stat().st_size])
>>> my_table.print()
Table: Directory
| FileName     | Size |
| .idea        | 4096 |
| directory.py | 1699 |
| __init__.py  |   37 |
| __pycache__  | 4096 |
```

This is a possible extension of a format for type iban, breaking the string into chunks of 4:

```
>>> iban = 'de1212341234123412'
>>> format_types['iban'] = lambda x_size, x_val: ' '.join(['{}' for x in range(6)]).
↪format(* re.findall('.{1,4}', iban)})
de12 1234 1234 1234 1234 12
```

`append`(*table_row: list*, *attrs: dict = None*, *row_type: str = 'body'*, *row_id: str = ''*, *exists_ok=True*) → *TTableRow*

Append a row into the table This procedure also defines the column type and the width

**Parameters**

- `exists_ok` – Try to append, but do not throw exception, if key exists
- `table_row` – List of values
- `attrs` – Customizable attributes
- `row_type` – Row type used for output filter
- `row_id` – Unique row id

**Raises**

`TableInsertException` – Exception if row-id already exists

`column_names: List[str]`

Property - List of column names

`do_select`(*filters: dict | str*, *get_all: bool = False*) → List[*TTableRow*]

Select table rows using column values pairs, return at maximum visible_items. The value could be any valid regular expression.

**Parameters**

- `filters` (*Dict[column_name, value] or qualified string*) – dictionary with column-name/value pairs or qualified string: row-id[.row-id]*, in which case the algorithm will search recursivly in TTableRow.child structure
- `get_all` – If True select more than visible_items

**Returns**

List of selected rows

**Return type**

List[*TTableRow*]

Example:

```
>>> rows  = my_table.do_select(filters={'FileName': '__*'})
>>> my_table.print(rows)
| FileName     | Size |
| __init__.py  |   37 |
| __pycache__  | 4096 |
```

`do_sort`(*column: int | str*, *reverse: bool = False*) → None

Toggle sort on a given column index

**Parameters**

- `column` – The column to sort for
- `reverse` – Sort direction reversed

`filter_clear`()

Clear the filters and return to original output filter_columns(*column_names: Dict[str, str]*) → None

The column_names is a dictionary with a set of keys as subset of TTable.column_names. The values are translated names to display in output. The order of the keys represents the order in the output. The filter is used to generate customized output. This function could also be used to reduce the number of visible columns

**Parameters**

column_names (*Dict[column_name: alias_name]*) – Map new names to a column, e.g. after translation

Example:

```
>>> my_table.filter_columns(column_names={'Size':'Größe', 'FileName': 'Datei'})
>>> my_table.print()
Table: Directory
| Größe | Datei        |
| 4096  | .idea        |
| 1886  | directory.py |
|   37  | __init__.py  |
| 4096  | __pycache__  |
```

get_header_row() → *TTableRow*

Returns the header row.

**Returns**

The header of the table

**Return type**

*TTableRow*

get_visible_rows(*get_all: bool = False*) → List[*TTableRow*]

Return the visible rows at the current cursor

**Parameters**

get_all – A bool value to overwrite the visible_items for the current call

**Returns**

A list of visible row items

navigate(*where_togo:* TNavigation *= TNavigation.NEXT*, *position: int = 0*) → None

Navigate in block mode

**Parameters**

- where_togo (TNavigation) – Navigation direction
- position – Position for absolute navigation, ignored in any other case

print(*rows: List[*TTableRow*] | None = None*) → None

Print ASCII formatted table

**Parameters**

rows – Optional parameter to print selected rows. If not set, print the visible rows.

title: str = 'Table'

Property - Table title name

class eezz.table.TTableCell(*name*, *value*)

Bases: object

The cell is the smallest unit of a table. This class is a dataclass, so all parameters become properties

**Variables**

- `width` (*int*) – Width of the cell content
- `value` (*int*) – Display value of the cell
- `index` (*int*) – Index of this cell in the column
- `type` (*str*) – Type of the value (class name), derived from runtime environment
- `attrs` (*dict*) – User defined attributes

`name: str`

Property - Name of the column

`value: Any`

Property - Value of the cell

`class eezz.table.TTableRow(`*cells*`)`

Bases: `object`

This structure is created for each row in a table. It allows also to specify a sub-structure table. This class is a dataclass, so all parameters become properties TTable row implements methods to access values like an array

- __getitem__ : value = row[column-name]
- __setitem__ : row[column-name] = value

**Variables**

- `cells_filter` (*List [str]*) – A list of cells with filtered attributes. Used for example for translation or re-ordering.
- `column_descr` (*List [str]*) – The column descriptor holds the name of the column
- `index` (*int*) – Unique address for the column
- `row_id` (*str*) – Unique row id for the entire table
- `child` (`TTable`) – The row could handle recursive data structures
- `type` (*str*) – Customizable type used for triggering template output
- `attrs` (*dict*) – Customizable row attributes

`cells: List[TTableCell] | List[str]`

Property - A list of strings are converted to a list of TTableCells

`get_values_list()` → list

Get all values in a row as a list

**Returns**
value of each cell

**Return type**
List[any]

`class eezz.table.TTableColumn(`*header*, *attrs*`)`

Bases: `object`

Summarize the cell properties in a column, which includes sorting and formatting. This class is a dataclass, so all parameters become properties.

**Variables**

- `index` (*int*) – Stable address the column, even if filtered or translated
- `width` (*int*) – Width to fit the largest element in the column
- `filter` (*str*) – Visible name for output
- `sort` (`TSort`) – Sort direction
- `type` (*str*) – Value type (class name)

`attrs: dict = None`

Property - Customizable attributes of the column

`header: str`

Property - Name of the column

`class eezz.table.TNavigation(`*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

Bases: `Enum`

Elements to describe navigation events for method *`eezz.table.TTable.navigate()`*. The navigation is organized in chunks of rows given by property *TTable.visible_items*:

`ABS = (0, 'Request an absolute position in the dataset')`

`LAST = (4, 'Set the cursor to show the last chunk of rows')`

`NEXT = (1, 'Set the cursor to show the next chunk of rows')`

`PREV = (2, 'Set the cursor to show the previous chunk of rows')`

`TOP = (3, 'Set the cursor to the first row')`

`class eezz.table.TSort(`*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

Bases: `Enum`

Sorting control enum to define sort on columns

`ASCENDING = 1`

`DESCENDING = 2`

`NONE = 0`

### 1.1.12 module eezz.websocket

This module implements the following classes

- *`eezz.websocket.TWebSocketAgent`*: The abstract class has to be implemented by the user to drive the TWebSocketClient
- *`eezz.websocket.TWebSocketException`*: The exception for errors on low level interface
- *`eezz.websocket.TWebSocketClient`*: This class interacts with the TWebSocketAgent and HTML frontend
- *`eezz.websocket.TWebSocket`*: Low level access to the socket interface
- *`eezz.websocket.TAsyncHandler`*: This class is used to interact with user defined methods

The TWebSocket implements the protocol according to rfc 6455[2]

[2] https://tools.ietf.org/html/rfc6455

`class eezz.websocket.TAsyncHandler(`*method: Callable*, *args: dict*, *socket_server:* TWebSocketClient, *request: dict*, *description: str*`)`

Bases: `Thread`

Execute method in background task

**Parameters**

- `method` (*Callable*) – The method to be executed
- `args` (*Dict [name , value ]*) – The arguments for this method as key/value pairs plus meta-arguments with the reserved key _meta, here with a loop request: `{'_meta': {'loop': 100,...}}`. The loop continues until the user method returns None
- `socket_server` (TWebSocketClient) – The server to send the result
- `request` (*dict [eezz-lark-key:value ]*) – The request, which is waiting for the method to return
- `description` – The name of the thread

`run()`

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

`class eezz.websocket.TWebSocket(`*a_web_address: tuple*, *a_agent_class: type[*TWebSocketAgent*]*`)`

Bases: `Thread`

Manage connections to the WEB socket interface. TWebSocket implements the socket of a http.server.HTTPServer

**Parameters**

- `a_web_address` (*Tupel [host , address ]*) – The connection information
- `a_agent_class` (*type [*TWebSocketAgent*]*) – The implementation of the EEZZ protocol

`run()`

Wait for incoming requests

`shutdown()`

Shutdown closes all sockets

`class eezz.websocket.TWebSocketAgent`

Bases: `object`

User has to implement this class to receive data. TWebSocketClient is called with the class type, leaving the TWebSocketClient to generate an instance

`abstract handle_download(`*description: str*, *raw_data: Any*`)` → str

handle download expects a json structure, describing the file and the data

`abstract handle_request(`*request_data: Any*`)` → str

handle request expects a json structure

`abstract setup_download(`*request_data: dict*`)` → str

This method is called before a download of files starts `shutdown()`

Implement shutdown to release allocated resources

`class eezz.websocket.TWebSocketClient(`*a_client_addr: tuple*, *a_agent: type[TWebSocketAgent]*`)`

Bases: `object`

Implements a WEB socket service thread. This class is created for each WebSocket connection

**Parameters**

- `a_client_addr` (*Tuple[host, address]*) – The communication socket to the web-browser
- `a_agent` (*type[TWebSocketAgent]*) – The agent class handle to handle incoming request

`gen_handshake(`*a_data: str*`)`

Upgrade HTTP connection to WEB-socket

**Parameters**

`a_data` – Upgrade request data

**Returns**

`gen_key()`

Generates a key to establish a secure connection

**Returns**

Base64 representation of the calculated hash

`handle_aync_request(`*request: dict*`)` → None

This method is called after each method call request by user interface. The idea of an async call is, that a user method is unpredictable long-lasting and could block the entire communication channel. The environment takes care, that the same method is not executed as long as prior execution lasts.

**Parameters**

`request` (*dict*) – The original request to execute after EEZZ function call

`handle_request()` → None

Receives an request and send a response The given method is executed async, so there will be no blocking calls. After the call the result is collected.

`read_frame(`*x_opcode*, *a_mask_vector*, *a_payload_len*`)`

Read one frame

**Parameters**

- `x_opcode` – The opcode describes the data type
- `a_mask_vector` – The mask is used to decrypt and encrypt the data stream
- `a_payload_len` – The length of the data block

**Returns**

The buffer with the data

`read_frame_header()`

Interpret the incoming data stream, starting with analysis of the first bytes

**Returns**

A tuple of all attributes, which enable the program to read the payload: final(byte), opcode(data-type), mask(encryption), len(payload size) `read_websocket()` → bytes

Read a chunk of data from stream

**Returns**

The chunk of data coming from browser

`shutdown()`

`upgrade()`

Upgrade HTTP connection to WEB socket

`write_frame(`*a_data: bytes*, *a_opcode: hex = 1*, *a_final: hex = 128*, *a_mask_vector: list | None = None*`)` → None

Write single frame

**Parameters**

- `a_data` (*bytes*) – Data to send to browser
- `a_opcode` (*byte*) – Opcode defines the kind of data
- `a_final` (*bool*) – Indicates if all data are written to stream
- `a_mask_vector` (*List[byte, byte, byte, byte]*) – Mask to use for secure communication

`exception eezz.websocket.TWebSocketException(`*a_value*`)`

Bases: `Exception`

Exception class for this module

### 1.1.13 module eezz.mobile

This module implements

- *eezz.mobile.TMobileDevices*: Database access to TUser

The database table TUser holds the mobile device information per user

`class eezz.mobile.TMobileDevices`

Bases: `TDatabaseTable`

Manage mobile device data for auto-login and document-key management

### 1.1.14 Module contents

# 2 Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index