Graphical User Interface with HTML

Albert Zedlitz

November 2023

Abstract

As application developer the user interface is always a very special challenge. In quite a few systems you would face even more than one level of indirection, which in the simplest form are presented in XML, JSON and YAML, or in more complex scenarios comes with a huge library. In some cases you face even a special editor, which could hide some of the complexity of the underlying system, but also hides some the features, which gives you a hard time, if the output does not match your expectations.

In the following I want to present the *HTML* browser as my the first choice user interface for any application and stay with the basic browser features with zero installation on this level. In the following I will prove, that it's possible to start with a command line driven application and connect the user interface as a second step, without touching in the coding. The main tasks in this concept could be handled without deep diving into *JavaScript* or any of the available wrapping libraries.

My extensions works based on *WebSockets*, a bi-directional communication protocol for real-time data transfer. This opens up a wide range of possibilities, including interactive dashboards, remote monitoring, and dynamic front-end user experiences, all while retaining the power and flexibility of traditional command-line utilities.

1 Introduction

While graphical user interfaces are absolute necessary nowadays for any project, it also addresses several key issues that could face you, particularly if you are without specialized training in user interface (UI) design. Especially me as a back-end and systems-level close-to-the-hardware developer, I often struggle when tasked with creating intuitive, visually appealing UIs. My results in this case where almost always a poor user experiences, which negatively affected even my most powerful command-line applications.

In the process of establishing an user interface front end, I investigated and worked with many third-party libraries like *Qt* or *Microsoft Visual C++ (MSVC)*. However, incorporating these libraries into a project introduced several critical issues:

- **Platform Coverage:** Third-party libraries often have varying levels of support for different operating systems. *Qt*, for example, is known for its cross-platform capabilities, but it can add significant overhead to projects. On the other hand, *MSVC* is closely tied to the Windows platform, making it difficult to maintain compatibility across Linux, macOS, and other environments. Choosing a library can lead to trade-offs between compatibility, performance, and complexity.

- **License Models:** Many third-party libraries come with complex licensing models that developers have to consider. For instance, *Qt* offers both open-source and commercial licensing, which can impose constraints on how the software can be used, distributed, or modified. *MSVC*, as a proprietary tool, comes with its own set of licensing terms that can limit flexibility and impact project costs. Misunderstanding or overlooking these licensing requirements can lead to legal issues or unexpected expenses down the road

- **Project Binding:** In most cases your code is heavy intermixed by library calls to manage interface input and data transfer. This makes your code harder to maintain, extend or test. Many issues like strings, which are not translated or graphic elements which will not scaled properly are detected only very late in the project and are very hard to correct. There may be many layers before output, each of which need some expertise and maintenance. You can only hope, that the support will last as long as your project needs it.

As you could see, the integration of user interfaces presented more challenges to me than opportunities. That was the starting point, where I had the idea for a complete new approach. I wanted to attach my command-line applications to *HTML* outputs, using *WebSocket* technology.

In practice this new approach speed up time to customer, because I could separate the development to a second teams, without any interference. Driving tests became much easier and resulted into better quality and performance. The examples given in this article show, that you could achieve some nice results with minimal efforts.

2 The EEZZ Project

From TTable to HTML

Before getting into details, let's have a preview on the usage to motivate the further work. The EEZZ project, which implements this environment, consists of the following parts.

- The application interface to collect data (*eezz.table.TTable*)
- The web-socket handler (*eezz.websocket.TWebSocketClient*)
- The HTML rendering service (*eezz.http_agent.THttpAgent*)
- The JavaScript handler (*websocket.js*)
- The HTTP server (*eezz.server.TWebServer*)

The *HTTP* server has no special tasks to do and could be replaced by Apache or NGInx.

As an example how these elements are fit together, let's have a look at the source in Figure 1, which shows the basic setup after installation of the EEZZ extension and a small example project, which displays the content of a local directory written in *Python*. The output in the console window shows the project before any graphical user interface is active. This is the part for collecting data.

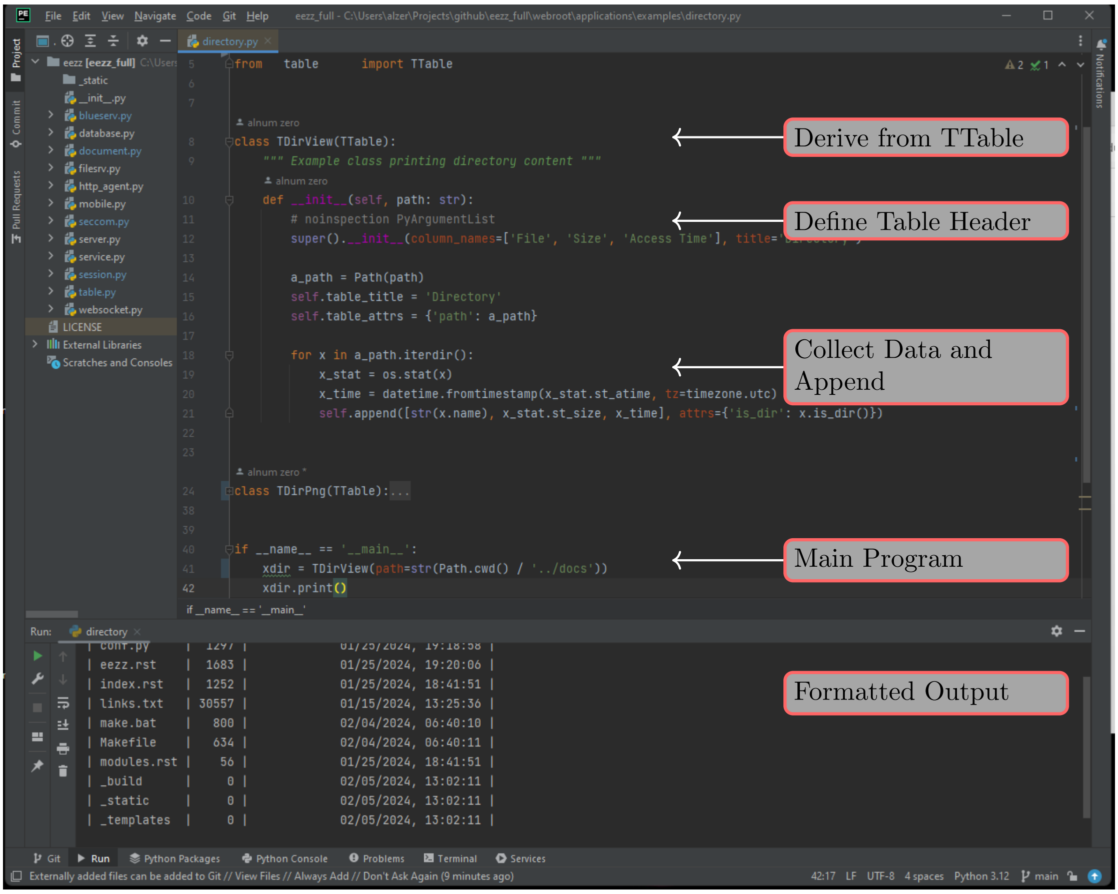


Figure 1: Example for Directory Listing

First thing you should notice is, that there is no hint whatsoever for any user interface. Except perhaps, that the project has to reside in the web-root directory of an *HTTP* server. A central point for the further development is the class *TTable*, which is already capable to print a nice ASCII-table to console, using the python data types for formatting. You could easily add your own data types with special format rules.

Regarding the code implementation project for a simple output-only scenario, this task is done and you could seal your work and go home. No further changes necessary for this part.

The central feature of *TTable* its ability to organize formatted output which is the basis for generating a valid *HTML* stream.

Prepare HTML Page for Output

The HTML code in listing 1 shows a setup to call your application and display the data in a browser window. You see a rudimentary HTML page with the special attribute *data-eezz* on some nodes. These eight lines are really all you need to this first try.

```html
<!DOCTYPE html>
<head><title>Title</title>
  <script data-eezz="template:websocket"></script></head>
<html><body>
  <h1>Simple Table</h1>
  <table data-eezz="assign:examples.directory.TDirView({query.path})"></table>
</body>
</html>
```

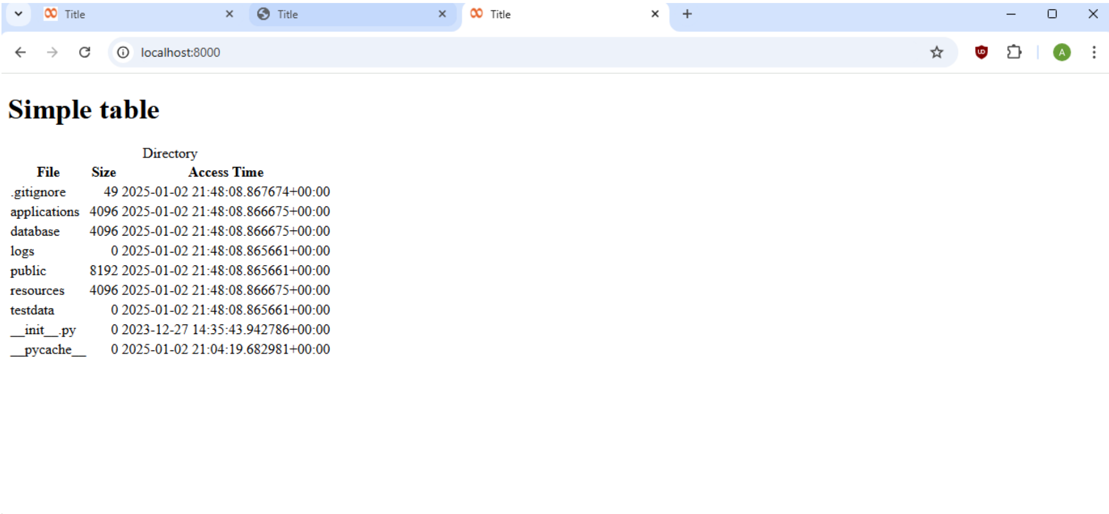
Listing 1: index.html for Directory Listing

The content of attribute *data-eezz* is compiled into statements, as described in the definition of the EEZZ syntax in Listing 2. In the header section, there is the "*template*" statement in the "*<script>*" node tag, which creates an include statement for the EEZZ web-socket implementation. For the "*<table>*" node tag, the "*data-eezz:assign*" statement connects the example *TDirView(TTable)* to the *HTML* and the rest in done by the framework.

With the curly brackets you have access to the following:

- URL query section (only for the assign): {*query.**}
- Attributes of *TTable*: {*table.**}
- Attributes of *TTableRow* (only for tr nodes): {*row.**}
- Attributes of *TTableCell* (only for td or th nodes): {*cell.**}

For the next steps you would need to deploy this file, configure and restart the HTTP server, in this case to listen on port 8000, and the result is available in your favorite browser.



Simple table

Directory

File	Size	Access Time
.gitignore	49	2025-01-02 21:48:08.867674+00:00
applications	4096	2025-01-02 21:48:08.866675+00:00
database	4096	2025-01-02 21:48:08.866675+00:00
logs	0	2025-01-02 21:48:08.865661+00:00
public	8192	2025-01-02 21:48:08.865661+00:00
resources	4096	2025-01-02 21:48:08.866675+00:00
testdata	0	2025-01-02 21:48:08.865661+00:00
__init__.py	0	2023-12-27 14:35:43.942786+00:00
__pycache__	0	2025-01-02 21:04:19.682981+00:00

Figure 2: Browser Output: http://localhost:8000

Nothing too spectacular, but this demonstrate the basic idea. In the following I will show the entire setup and all the possibilities to interact with the output for user defined formatting and scripting.

In each project software engineers and user interface experts should generate such a small *HTML* page, to define the interface as a contract. This defines the classes with all methods and arguments to be used. Now both could start working in their own team.

2.1 Conclusion and Perspective

The takeaway from this chapter is, that it is quite easy and convenient in the EEZZ environment to add a user interface to an existing command line driven application. Using a browser has also an advantage, that you could decide any time to switch to a network based solution. As developer, you could concentrate on programming in Python. No *SQL*, no GUI-packages, no *JavaScript* nor any of its wrappers or frameworks to interoperate with. We have a clear cut between UI related staff and the rest.

In the following sections I will show, that I'm not restricted on tables. There is a variety of possible layouts. For example tree views, grid-container and input-forms. It will extend this introduction of the basis concept, to get into depth. So I will only mention, that it's also possible to handle more complex elements like SVG, including libraries like *Chart.js*, introduce motion and more complex user interaction.

3 Behind the Scenes

3.1 EEZZ Grammar

The resources to build up page and to establish the connection to the *Python* code are located in a directory parallel to the custom pages. In here you find the *JavaScript* package *websocket.js*, a template for a table layout *template.html* and a grammar shown in in Listing 2 for the EEZZ interface.

The EEZZ interface uses only the user property namespace *data-eezz* on some *HTML* node elements. This has the advantage, that the *HTML* page could be designed independent from the application and that there is no intermixing with script and layout. There is also the possibility to work with design time data, so there should be no surprises, when matching the parts, application and user interface, together and there should be no need for sophisticated debugger to get things running.

```
?start              : list_statements

?list_statements    : [ statement  ("," statement  )* ]
?statement          : "event"     ":" function_call                ->  funct_assignment
                    | "assign"    ":" function_call                ->  table_assignment
                    | "update"    ":" list_updates                 -> update_section
                    | "onload"    ":" list_updates                 -> onload_section
                    | "post_init" ":" function_call                -> post_init
                    | "template"  ":" string ( "(" value_string ")" )?  ->  template_section
                    | string      ":" string                       ->  parameter_section
                    | qualified_string ":" value_string            -> setenv

list_updates        : [ update_item  ("," update_item  )* ]

update_item         : qualified_string "=" function_call        -> update_function
                    | qualified_string "=" update_string
                    | function_call                             -> update_task
                    | qualified_string

?update_string      : string
```

Listing 2: Extract of EEZZ Extension Grammar: eezz.lark

As you can see, the grammar description is quit small. At the end it is compiled to a json format, which is attached to the elements attribute *data-eezz-json*.

3.2 Templates

A template is a node, which is used for rendering table rows to *HTML* elements. So where are the templates in the first example shown in Listing 1?

The answer is, that I wanted to optimize the interaction with the back-end, which required all the table child elements to be explicit available during rendering. So I inject missing parts using the definition shown in Listing 3, which contains all the required templates.

```
<table data-eezz="template: table">
    <caption></caption>
    <thead>
        <tr data-eezz="template: row, match: header">
             <th class="clzz_th"
                 data-eezz="
                    template:   cell,
                    event:      do_sort(column={cell.index}),
                    update:     this.tbody">{cell.value}</th></tr>
    </thead>
    <tbody>
        <tr data-eezz="
            template: row, match: body,
            event: on_select(row = {row.row_id})">

            <td class='clzz_{cell.type}'
                data-eezz="template: cell">{cell.value}</td></tr>
    </tbody>
    <tfoot>
        <tr data-eezz="
            template:   reference(table),
            style:      visibility = {table.visible_navigation}">
            <td style="text-align:center" colspan="4">
                <img class = "clzz_navigation_img" src="navbar.png"
                        style  = "height:20px; width:160px"
                        usemap = "#table-navigation"/>
                <map class="clzz_navigation" name="table-navigation">
                    <area shape="rect" coords="  0, 0,  40, 20"
                        data-eezz="event:  navigate(where_togo = 3), update: this.tbody"/>
                    <area shape="rect" coords=" 40, 0,  80, 20"
```

Listing 3: Extract of template.html

There are row and cell templates. Row templates are used to design different layouts based on the value of *TTable.row_type* attribute, which is referenced in the page as *data-eezz:match*. Default values are *header* and *body* for *thead* and *tbody* table child elements respectively. The tfoot rows are not included in this template concept and are considered to be constant.

3.3 Events

The following events are implemented in the injected definition, but you could also define own event methods:

- *TTable.do_sort* for each column in the thead section
- *TTable.do_select* for each row in the tbody section

The first statement potentially alters the table content and needs an update directive, to make changes visible. The select statement could be handled without any changes in the data, so an update statement is optional.

All events are executed in a separate thread to prevent blocking the user interface, if there is a long running method. You could issue the statement "*process:sync*" to force execution synchronously.

3.4 Time Laps View

The time laps diagram below Figure 3 shows, that even the program flow is very simple. I put the TDirView from the example above in the rightmost row. Here you could imagine your python TTable class.

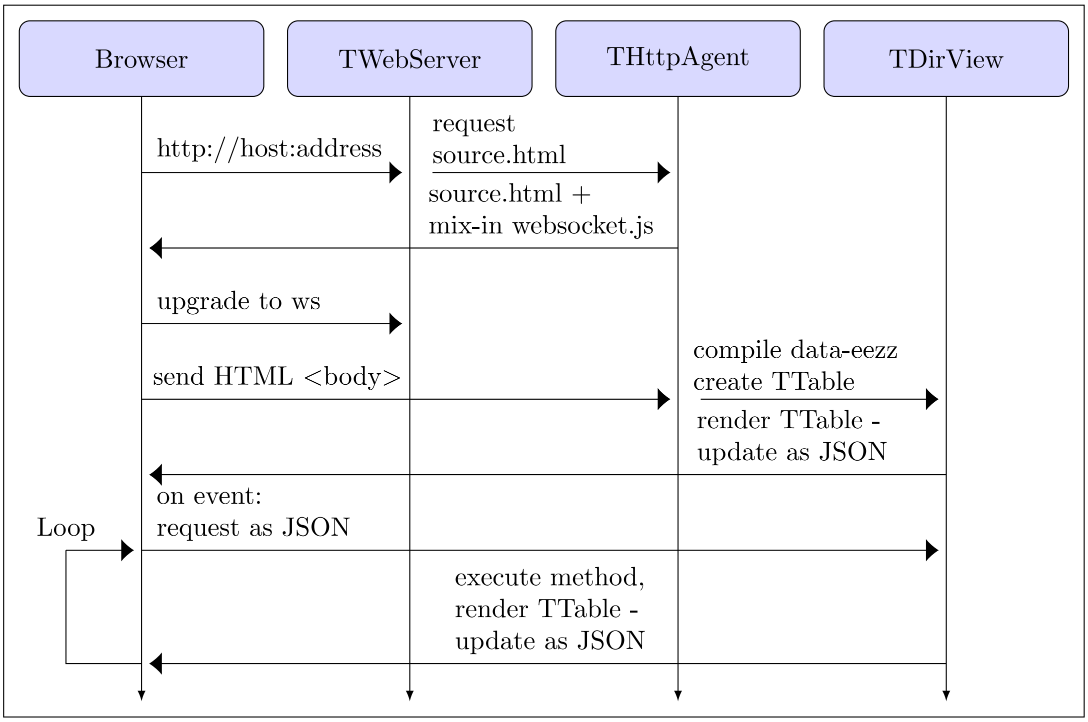


Figure 3: Time lapsed diagram

3.5 Conclusion

This chapter showed the basic concept of the EEZZ extension, the data flow and the default settings. It gave a short overview over the in- and output features and interaction between server and client.

The EEZZ extension allows you to connect *HTML* element attributes with the programming logic of a back-end system. It does not introduce another wrapper to *JavaScript* nor does it introduce a new programming language.

4 Features

There are some features, which I implemented for better efficiency in project development. The first addresses the handling of mass elements of the same kind. The second shows an alternative for polling data. And last not least I did some work to make life with database access easier.

4.1 Arrays as Value Type

It's possible to define an array for a cell. In this case the tree view would take the first element of the array for display. In a grid view on the other hand, it allows you to create a set of elements of the same kind. Take for example the calculator app in Listing 4. It defines two columns, one for the numbers and one for the operators. The *TCalc(TTable)* class defines two methods, one for each set of elements, both returning a byte-stream. The first is for collecting digits and the other executes math.

```
class TCalc(TTable):
    def __init__(self):
        self.number_input: int   = 0
        self.stack: list         = list()
        self.op:    str          = ''

        super().__init__(column_names=['numpad', 'op'], visible_items=1)
        self.append([[1,2,3,4,5,6,7,8,9,0], ['+','-','*','=']])

    def key_pad_input(self, key) -> bytes:
        self.number_input *= 10
        self.number_input += int(key)
        return f'{self.number_input}'.encode('utf8')

    def key_op_input(self, key) -> bytes:
        result: float = 1.0 * self.number_input

        self.stack.append(self.number_input)
```

Listing 4: Calculator

The HTML page references such cells with the detail property. The advantage of this approach is, that you need only two templates for 14 buttons.

```
        <div style="display: grid; grid-template-columns: repeat(3, 30px); gap:5px;">
          <input  type  ='button'
                  value ="{detail.value}"
                  data-eezz ="
                      template: cell(numpad.detail),
                      event :    get_header_row(),
                      update:    display.innerHTML = key_pad_input(key={detail.value})"/>
        </div>
        <div style="display: grid; grid-template-rows: repeat(5, 30px); gap:5px;">
          <input  type  ='button'
                  value ="{detail.value}"
                  data-eezz ="
                      template: cell(op.detail),
                      event:     get_header_row(),
                      update:    display.innerHTML = key_op_input(key={detail.value})"/>
```

Listing 5: Calculator Layout

As a result we have created a calculator as a working application as shown in Figure 4

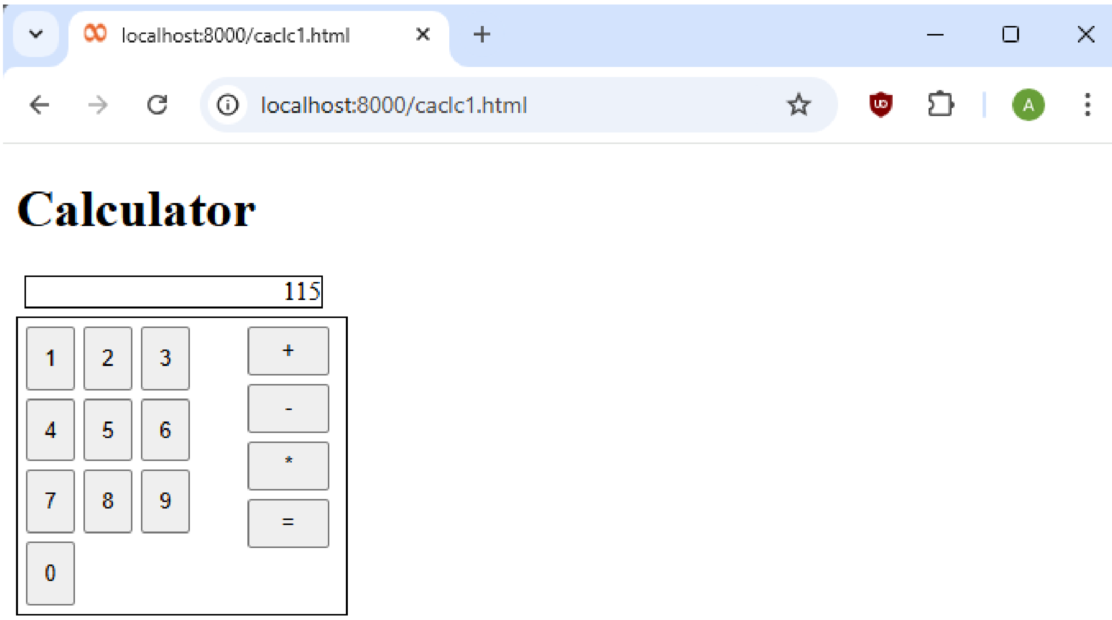


Figure 4: Calculator with Cell-Array

It would be the same setup for a boards used by Minesweeper, Chess or Sudoku. Another usage could be the transmission of coordinates for dynamic generated charts. This is also my preferred way to implement a dynamic selector element. The selected cell value is just one of the elements in the list.

4.2 Push Service

If the update function is placed in the assign statement, it is considered to be a background task. This enable the user to push any data any time to the browser:

```
    <div class='clzz_grid'
        id="WebCamClass"
        data-eezz='
            assign: examples.webcam.TCamera(),
            update: webcamimg.src = read_frame()'>
    </div>

    <img id='webcamimg'/>
```

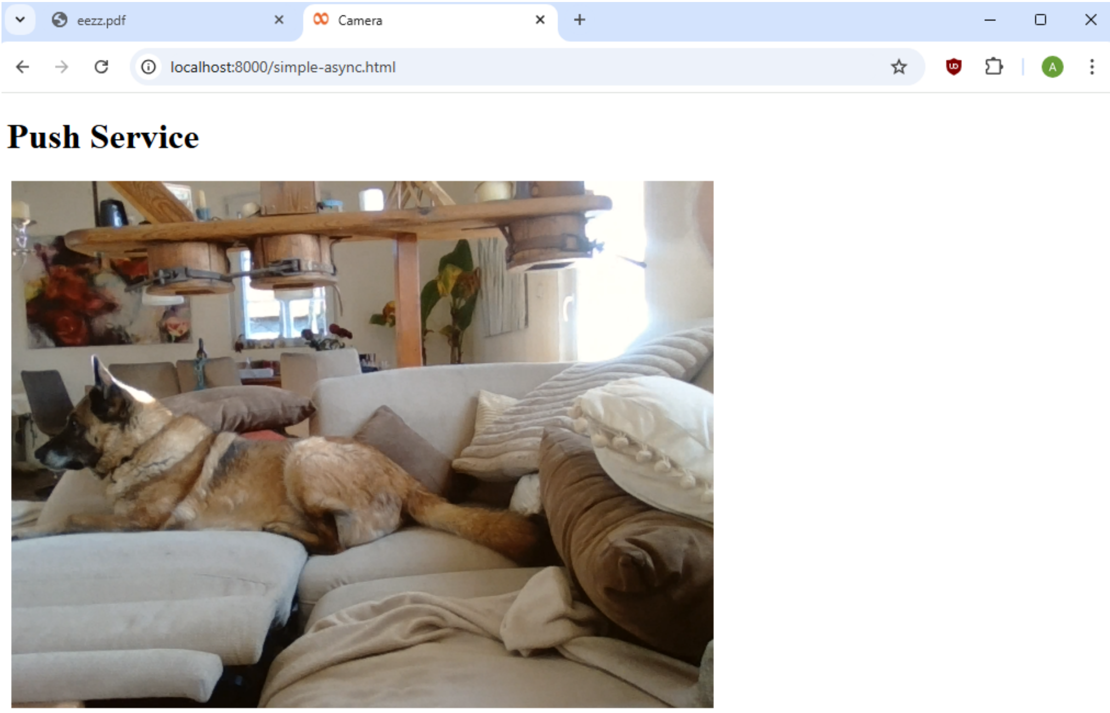
Listing 6: WebCam as Push Service

In the example in Listing 6 the *img* element on the page is changed whenever the function *read_frame* returns a picture. The implementation of this function is straight forward as shown in Listing 7.

```
class TCamera(TTable):
    def __init__(self):
        self.cam = cv2.VideoCapture(0)
        super().__init__(column_names=['camera'])

    def read_frame(self) -> bytes:
        time.sleep(5)
        ret, frame  = self.cam.read()
        ret, buffer = cv2.imencode('.png', frame)
        return BytesIO(buffer).getvalue()
```

Listing 7: WebCam Implementation

This is the entire project and the result is shown in Figure 5. I used *OpenCV* to generate a *PNG* stream buffer, waiting for some seconds between each frame and would now be able to monitor my room from remote, if I grant access to the URL.



;

Figure 5: WebCam as Push Service

4.3 Database Access

The derived class *TDatabaseTable(TTable)* manages database access. It encapsulates database access transparently, with the same properties and interfaces as the base class. Access to the database with this class will add the options *limit* and *offset* to any *SQL* command, to reduce the data flow for the entire application stack to the absolute minimum and still holding track on the entire result set and cache data. Most common mistakes in database handling could be avoided this way.

4.4 Logging

With one line you have access to the logger output of one request. You have to specify the update to the elements you want to monitor with the attribute: *data-eezz=update:eezz_log_table*. This is helpful, because you could focus on just one window, monitoring the performance and the execution.

```
<table id = "eezz_log_table" data-eezz="assign: eezz.websocket.TLogger()"></table>
```

Listing 8: Activate the Logger

5 Improve the Output

5.1 Using Meta Data

The Listing 9 shows how to access new incoming elements for rework. Just implement the abstract method *eezz.on_update*, which is called by the framework.

Let's start to improve the time format. You need no access to the generating *Python* program to achieve this. The *Python datetime* object is rendered to the *timestamp* attribute of the element. We could access the value directly and convert it to a local time.

```
    <script type="text/javascript">
        eezz.on_update = (a_element) => {
            x_element  = document.getElementById(a_element);
            x_list     = x_element.querySelectorAll("td[timestamp]");

            for (var inx in x_list) {
                var x_time       = x_list[inx];
                var x_timestamp  = x_time.getAttribute('timestamp');
                var date         = new Date(x_timestamp * 1000);
                x_time.innerHTML = date.toLocaleTimeString();
            }
        }

    </script>
```

Listing 9: Head of simple-update.html

As a result the time output could be changed, after update. This gives the designer the full access and utmost flexibility on any data, which is fetched from an application.

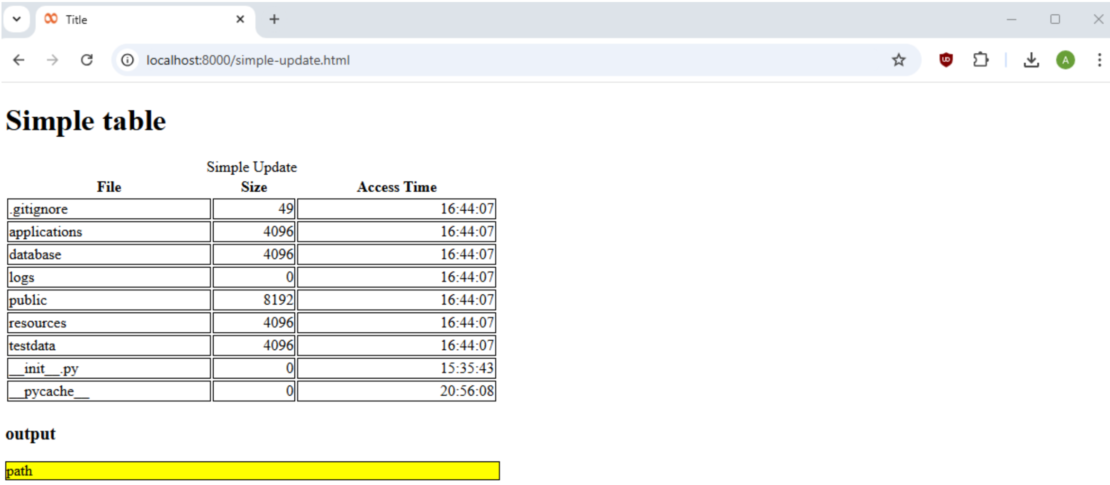


Figure 6: Browser Output: http://localhost:8000/simple-update.html

5.2 Matching Templates

The fragment in listing 10 shows, how to use the *data-eezz:match* attribute to specify the layout of a row, according to a given data type. If *data-eezz:match* is set to "*is_dir*" a directory icon is inserted at the start of the row and a file icon if the value is set to "*is_file*".

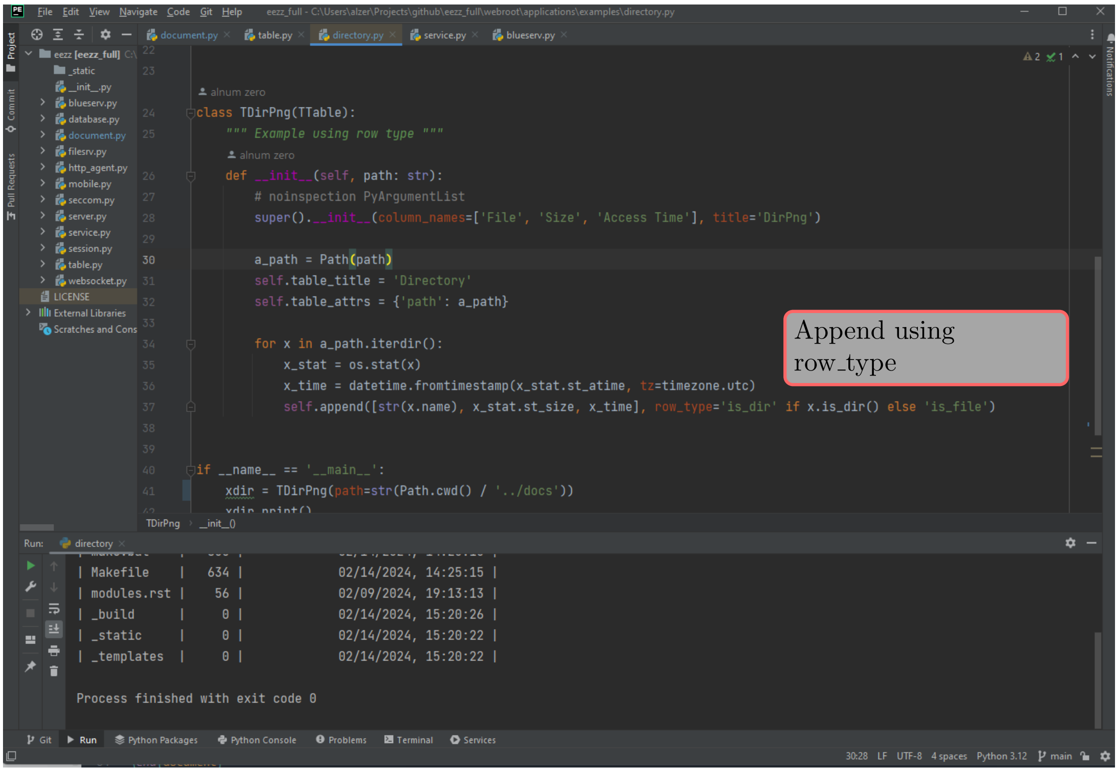


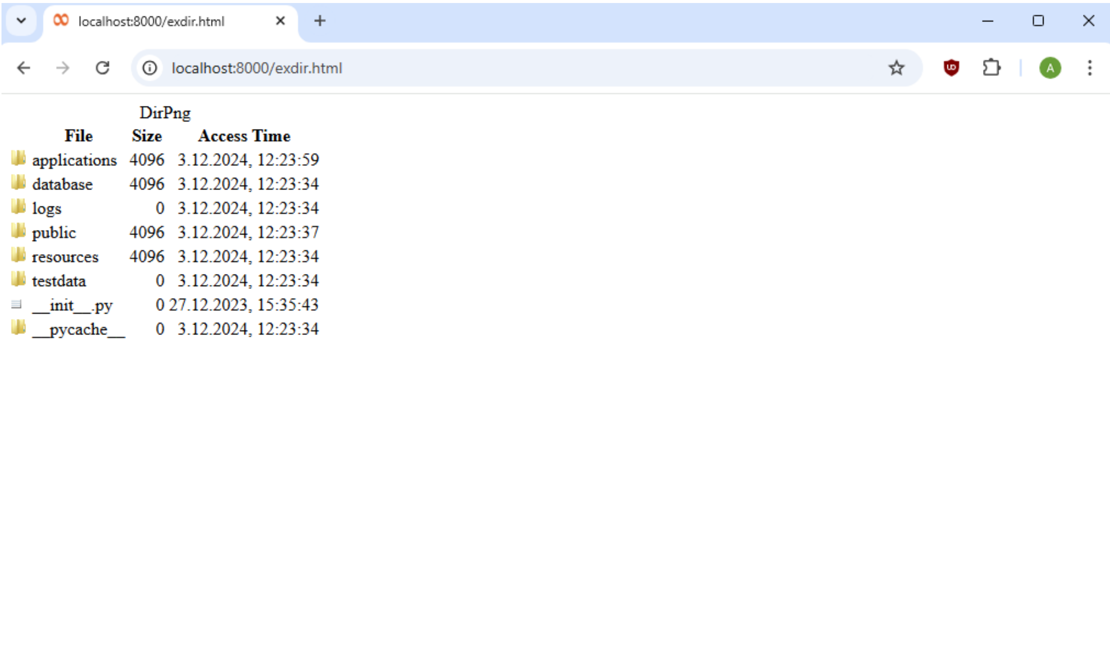
Figure 7: Example for Directory Listing

In the listing listing 10 you could see two templates with different match assignment. Each row type could define different actions for select and update.

```
        <tbody>
            <tr data-eezz="
                template: row,
                match:  is_file">

                <td><img src="file.png"></td>
                <td class='clzz_{cell.type}'
                    data-eezz="template: cell">{cell.value}</td></tr>

            <tr data-eezz="
                template: row,
                match:  is_dir">

                <td><img src="dir.png"></td>
                <td class='clzz_{cell.type}'
                    data-eezz="template: cell">{cell.value}</td></tr>
        </tbody>
```

Listing 10: simple-tree1.html

For the ASCII output nothing changes. The template nodes could be interpreted like a switch statement. For each row the render machine will lookup the corresponding template. In our example one of the differences is the first cell, which loads individual icons, according to the row type as shown in the output in Figure 8



;

Figure 8: Listing with different Row Types

5.3 Tree Views

In the previous examples I have chosen a directory listing. With the class *TTableTree* it is also possible to show the entire directory tree and open/close the branches dynamically.

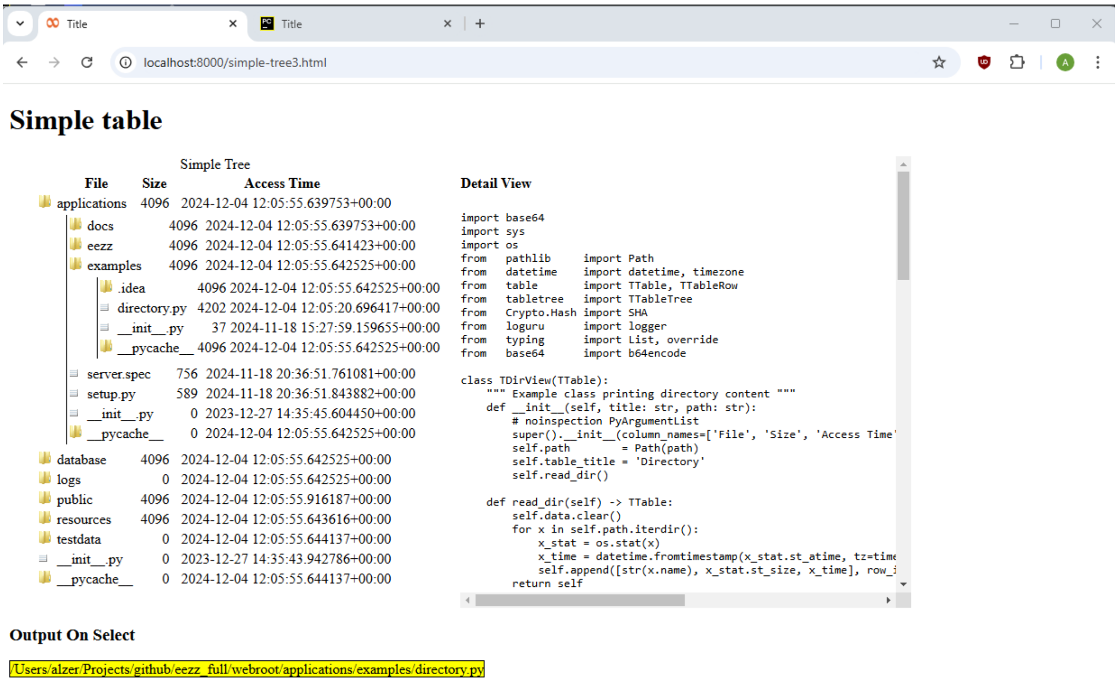
To do so, I defined the class *TDirTreeDetails(TTableTree)*, implemented the abstract method *TTableTree.open_dir* and adjusted the *HTML* page as shown in Listing 11.

```
64     <table id="Directory"
65         data-eezz='assign: examples.directory.TDirTreeDetails(title="Simple Tree", path="/
             ↪ home/user")'>
66         <thead>
67             <tr data-eezz="
68                 template: row,
69                 match:  header">
70 
71                 <th></th><th
72                     class="clzz_th"
73                     data-eezz="
74                         template: cell,
75                         event:  do_sort(column={cell.index}),
76                         update: this.tbody">{cell.value}</th></tr>
77         </thead>
78         <tbody>
79             <tr data-eezz="
80                 template: row,
81                 match:  is_file,
82                 event:  on_select(index={row.row_id}),
83                 update: path_label.innerHTML = {row.row_id},
84                         text_detail.innerHTML = read_file(path={row.row_id})">
85 
86                 <td><img src="file.png"></td>
87                 <td class='clzz_{cell.type}'
88                     data-eezz="template: cell">{cell.value}</td></tr>
89 
90             <tr data-eezz="
91                 template: row,
92                 match:  is_dir,
93                 event:  open_dir(path={row.row_id}),
94                 update: this.subtree = this.tbody,
95                         path_label.innerHTML = {row.row_id}">
96 
97                 <td><img src="dir.png"></td>
98                 <td class='clzz_{cell.type}'
99                     data-eezz="template: cell">{cell.value}</td></tr>
100        </tbody>
```

Listing 11: simple-tree3.html

Important for the tree view is the update sequence in line 94 in Listing 11. This statement puts the sub-tree to the right place. The *JavaScript* implementation toggles open and close. The application could choose to keep the data.

In the example above, the directory node sends an update to the yellow text element to show the selected path and the file node triggers reading and displaying the files content, as shown in fig. 9.



;

Figure 9: Example for Table and Details

Of cause, this would work with pictures as well.

5.4 Grid Views

The grid layout shows, that this pattern could be assigned to various types. We create a template tile, which is reproduced for each row of in the *TTable*. Essential for this to work is the class assignment to *class=clzz_grid*.

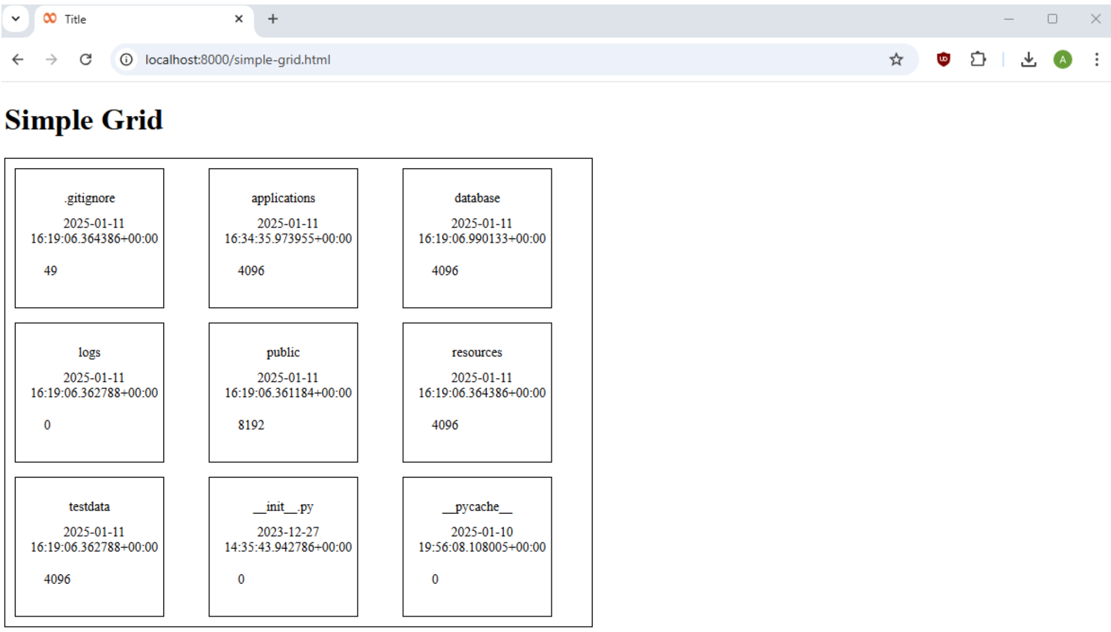
In the following the parent element has the style attribute *display:grid* and takes the role of the table structure in the previous examples and assigns a user method.

In the next level of the hierarchy you define an element, which defines the layout for each row in the table using the attribute *data-eezz="template:row"*. The third level of the hierarchy elements with attribute *data-eezz="template:cell(column-name)"* have access on cell values using the column name in brackets, which in our case is ("File", "Access Time", "Size"). In this example we place the cell values on relative positions within a tile. If the column name contains a space you need to specify quotes.

```
    <div class="clzz_grid" id="Directory" data-eezz='assign: examples.directory.TDirView( title="Simple Tree", path="/Users/alzer/Projects/github/eezz_full/webroot")'>

        <div class="grid-item" style = "position: relative;"
            data-eezz = "
                template: row,
                match:  body">

            <span style = "position: relative; vertical-align: center; top: 20px"
                data-eezz = "template: cell (File)">{cell.value}</span>

            <span style = "position: absolute; left: 10px; top: 50px"
                data-eezz = "template: cell ('Access Time')">{cell.value}</span>

            <span style = "position: absolute; left: 30px; top: 100px"
                data-eezz = "template: cell (Size)">{cell.value}</span>
        </div>
    </div>
```

Listing 12: Simple Grid Layout

The result is shown in the next picture fig. 10. Again it would be possible to define different tiles for different row types.



;

Figure 10: Example for Grid View

Now you could let your imagination run wild and give the grid tiles any outer appearance or add motion, connecting the transform properties of an elements with the back-end.

5.5 Form Input

Forms are implemented in this environment as a logical extension of the grid concept. I defined the class *TFormInput(TTable)* as shown in Listing 13 and set *TTable.visible_rows=1*. I want only one tile to become visible as input dialog.

One table row is needed to specify the data-types and depending on the user interface designer, using the given values as default or help. The result is shown in listing 14.

```python
class TFormInput(TTable):
    def __init__(self, title: str):
        super().__init__(column_names=['Title', 'Description', 'Medium', 'Technique', 'Price'], title=title)
        self.visible_items = 1
        self.append(['','','','', 0.0,''])

    def register(self, input_row: list) -> TTableRow:
        # todo: implement register activities here
        return super().append(input_row)


@dataclass
```

Listing 13: Python Script driving Form Input

In the *HTML* page, you specify the input tags and the submit button, as shown in listing 14. The elements of the form are collected using square brackets as selectors. The framework places all elements into the row in the correct order, allowing the back-end method to use the *TTable.append* method to store the incoming values.

```
    <div class="clzz_grid" id="FormInput"
        data-eezz='assign: examples.bookshelf.TFormInput(title="Simple Form")'>

        <div style="display: grid; grid-template-columns: auto auto; gap:5px; width:300px"
            data-eezz ="template: row" data-eezz-match="body">

            <span>Title</span>
            <input  type =      "text"
                    data-eezz = "template: cell (Title)"/>

            <span>Description</span>
            <textarea rows="4" cols="50"
                    data-eezz = "template: cell (Description)"></textarea>

            <span>Medium</span>
            <input  type =      "text"
                    data-eezz = "template: cell (Medium)"/>

            <span>Technique</span>
            <input  type =      "text"
                    data-eezz = "template: cell (Technique)"/>

            <span>Size</span>
            <input  type =      "text"
                    data-eezz = "template: cell (Size)"/>

            <span>Price</span>
            <input  type =      "text"
                    data-eezz = "template: cell (Price)"/>

            <input  type        =  "button"
                    id          =  "commit_button"
                    value       =  "submit"
                    data-eezz   =  "event: register(input_row = [template.cell])"/>

        </div>
```

Listing 14: Form Input HTML

As a result, you can define a form input with individual input tag elements. You can implement input checks in JavaScript within the browser or in the back-end. You can provide feedback about expected and submitted data, making this concept much more efficient and easier to implement than other approaches.

;

Figure 11: Form Input

6 Compound Documents

To elevate the previous chapter, the following examples demonstrate how to create a compound document based on the features described above. A compound document consists of a description as a Manifest and a set of embedded files. The Manifest contains attributes such as the author, creation date, and file names. With this, we can implement content management. In combination with *TTable* one single row correspond to a document and all rows together would define a bookshelf.

6.1 Creating a Bookshelf

The class *eezz.TDocument* encapsulates the file handling, which includes download and management of compressed archives. The chunk-based, non-blocking download feature allows a feedback via a progress bar, which is particularly useful for large files.

In the first step I want to create a document, so I need an input dialog. The class *TSimpleShelf(TTable,TDocument)* maps the document attributes to table column as shown in in Listing 15.

```python
@dataclass
class TSimpleShelf(TTable, TDocument):
    shelf_name:     str
    attributes:     List[str]    = None
    file_sources:   List[str]    = None
    column_names:   list         = None
    current_row:    TTableRow    = None
    id:             str          = ''

    def __post_init__(self):
        """ Initialize the hierarchy of inheritance """
        # Adjust attributes
        self.attributes   = ['Title', 'Header', 'Description', 'Medium', 'Technique', 'Price', 'Size', 'Status']
        self.file_sources = ['main', 'detail']
        self.id           = self.shelf_name
        TDocument.__post_init__(self)

        # Set the column names and create the table
        self.column_names = self.attributes
        TTable.__post_init__(self)

        self.visible_items = 1
        self.append(table_row=['','','','','', 0.0,'',''], row_type='input')
```

Listing 15: Bookshelf Example

The process of generating a document is separated in three phases. Phase one saves the attributes into the *TTable*, phase two is the download process of files and the last phase generates an archive with all the input.

The cascade is triggered with the statements in the commit button in Listing 16. The *data-eezz:event* is executed. For the update target there is a function call instead of an element attribute. An update target is always executed on the browser, so this is a call to a *JavaScript* function, with the argument compiled to "*read_files({files:{ [main, detail] }})* ".

This is an EEZZ function, included with *websocket.js* and it organizes the download. The method uses the statement shown in Listing 17 to evaluate the recipient of the data. It shows also how an update of a progress bar could be implemented. Important is the "sync" option in this case, so that the progress bar outcome is not random.

```html
    <input  type ="button" value = "submit"
            data-eezz =  "
                event:  prepare_document(values = [template.cell]),
                update: read_files(files = [main, detail]) "/>
```

Listing 16: Trigger Download

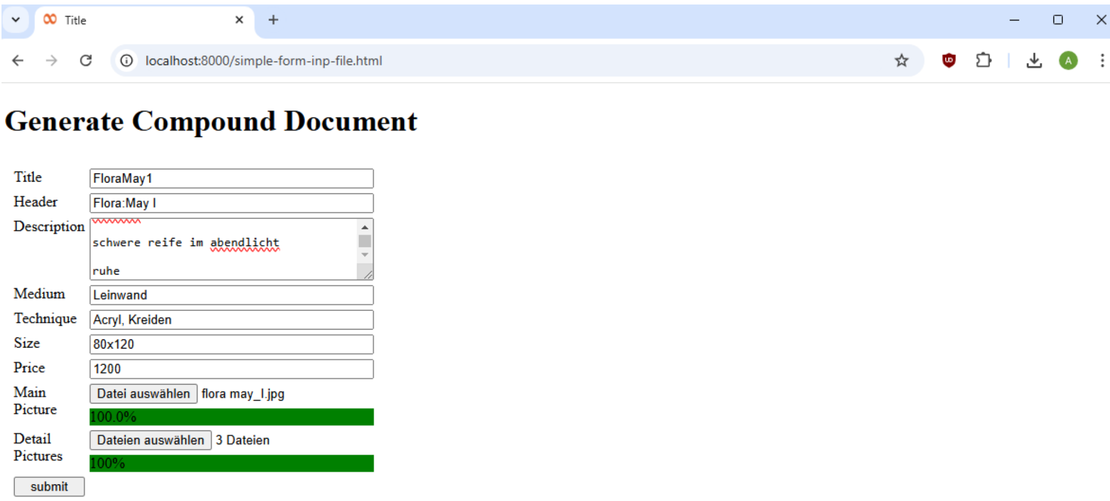
The referenced *HTML* tags listed in the call to *read_files* contain the download instruction, which is in this case is the method *TDocument.download_file*. The download has two arguments: a descriptor and a binary stream and the download object has two attributes *file and byte-stream*.

The method is executed on the back-end and returns a percentage value as *UTF-8* encoded byte-stream, to update the progress bar.

```
<div style="display: grid; grid-template-row: auto auto; row-gap: 4px;">
    <input
        type = "file"
        data-eezz="
            template: cell (detail), process: sync,
            update:
                progress_detail.style.width = download_file(file=this.file, stream=this.bytestream),
                progress_detail.innerHTML   = progress_detail.style.width"/>

    <div style="width:200px; background-color:red">
        <div id="progress_detail"
            style="width: 50%; background-color:green">50%</div></div>
    </div>
```

Listing 17: File Download Snipplet

The dialog would look as follows after the download was successful.



;

Figure 12: Form Input with File Download

The manifest and all files are zipped into a resulting compound document. For any part of an *HTML* page, you can now reference such a document to create content. This approach makes it much easier to make changes, add, or remove content.

6.2 Display a Bookshelf

To display the compound document, we need to load the data from local storage, which is achieved with the statement *data-eezz:oninit*. The method specified in this statement is executed before rendering.

```
    <div class="clzz_grid" id="bookshelf"
        data-eezz='
            assign:  examples.bookshelf.TSimpleShelf(shelf_name="Paintings"),
            oninit:  load_documents()'>
```

Listing 18: Assign with Initialization

The Manifest for compound documents, stored in the archives, are designed in a way, that restoring a document from data stream could be done in just a few lines.

```
    def load_documents(self) -> TTableRow:
        """ Load the bookshelf from scratch """
        self.clear()
        self.visible_items = 20

        x_path = self.path / self.shelf_name
        for x in x_path.glob('*.tar'):
            self.manifest.loads(self.read_file(x.stem, 'Manifest'))
            x_row_values      = self.manifest.document.values()
            self.selected_row = self.append(x_row_values, row_id=x_row_values[0])
        return self.selected_row
```

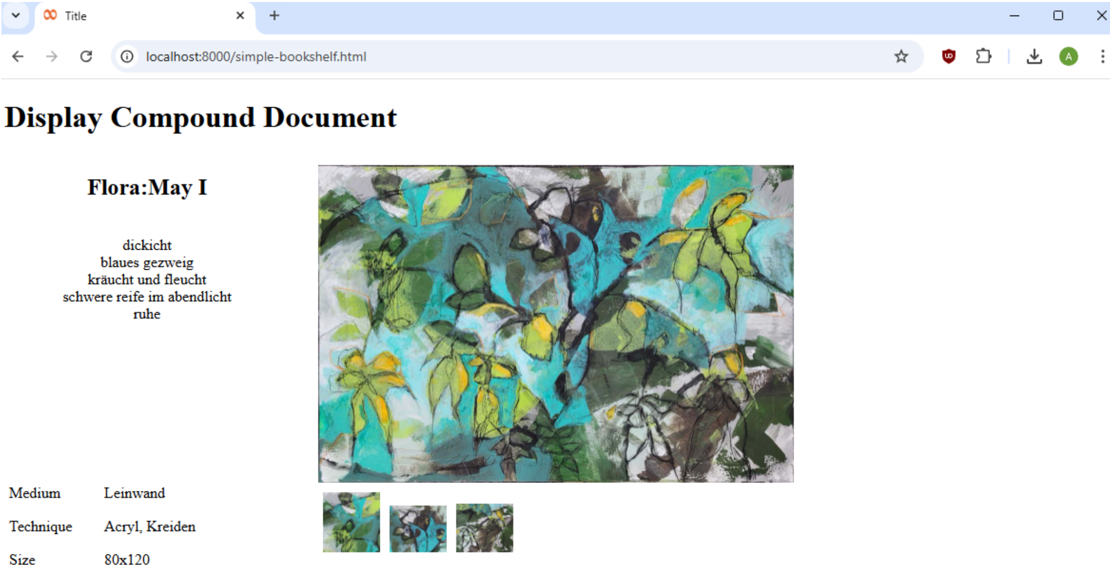
Listing 19: Load Bookshelf

Images could be loaded either using browser source path or using a back-end method defined by the *data-eezz:onload* statement.

```
            <div class="pictures_panel">
                <img style="max-width:500px; height:auto"
                    data-eezz = "
                        template: cell (main),
                        onload:
                            this.src = read_file(
                                document_title  = {cell.id},
                                file_name       = {cell.value})"/>
```

Listing 20: Loading Images from Archive

Looking at the result keep in mind, that this is a template for all documents in a bookshelf. You could restrict the number of tiles via *TTable.visible_items*, you could sort, filter and navigate. You could manage the content of the bookshelf by adding, removing or simply filter the entries in the content directory.



;

Figure 13: Bookshelf first Tile

I made use of a the format directives for the text above. The *data-eezz:format* statement gives the framework a hint to replace double newlines either with *”
” line-break or ”<p>” paragraph.*

Compare this with the project WordPress ZedlitzArt. I achieved the same results with my new approach, but with significant less overhead and less time for adding new pictures.

7 Summary and Perspective

My goal was to establish some user interface to my programs. It was important for me to stay platform independent and I wouldn't accept any major restrictions in functionality or performance. The EEZZ package is a real revolution in programming techniques for me.

With only some very few lines of coding and some minor effort in learning the EEZZ syntax, I was able to achieve quite remarkable results. Furthermore I targeted some of the main issues of my past projects. Beginning with incredible difficulties setting up automated tests for UI applications or facing issues with database handling and performance. Publishing my result I hope, that some projects might also benefits from this work.

There is still some work to do, to make the usage more robust and to investigate for proper error feedback and logging. Implementation and documentation of the current state could be found at github EEZZ.