Back to HTML

Albert Zedlitz

November 2023

Abstract

In today's fast-evolving technological landscape, bridging the gap between command-line applications and web-based user interfaces is crucial for enhancing both functionality and accessibility. This project introduces a new and innovative way to attach command-line applications to *HTML* output, utilizing the power of *WebSocket* technology.

Traditionally, command-line tools operate in a terminal environment, limiting user interaction to text-based input and output. By leveraging *WebSockets*, a bi-directional communication protocol for real-time data transfer, this project enables seamless integration of command-line applications into modern web interfaces. This opens up a wide range of possibilities, including interactive dashboards, real-time monitoring, and dynamic front-end user experiences, all while retaining the power and flexibility of traditional command-line utilities.

Our approach in the EEZZ project focuses on creating a simple, efficient, and scalable solution for developers and system administrators, providing them with the ability to run, manage, and interact with CLI tools directly through a web browser. This not only makes the applications more accessible but also enhances collaboration and ease of use for non-technical users.

By connecting the backend power of CLI applications to the rich interactivity of *HTML*, this project aims to redefine how command-line utilities are utilized in web-based environments, bringing real-time, responsive interaction to the forefront of system administration, development workflows, and beyond.

It's back to *HTML* in a sense, that the main tasks in this concept could be handled without deep diving into *JavaScript* or any of the available wrapping libraries. Loops are driven by templates and data sets, not by explicit commands, which reduces possible programming errors to a minimum. The framework also restricts the amount of data to display to a visible frame, so that the runtime is predictable under all circumstances even for high data volumes.

1 Introduction

As developers strive to build increasingly dynamic applications, the integration of interfaces presents both opportunities and challenges. This project introduces a new approach for attaching command-line applications to *HTML* outputs, using *WebSocket* technology to enable real-time, interactive user interfaces.

While graphical user interfaces opens up a new world of possibilities for developers, it also addresses several key issues that many face, particularly those without specialized training in user interface (UI) design. Developers who excel in back-end programming or systems-level development often struggle when tasked with creating intuitive, visually appealing UIs. This can result in poor user experiences, which negatively affect the usability of even the most powerful command-line applications.

In the process of integrating command-line applications with user interface front ends, many developers turn to third-party libraries like *Qt* or *Microsoft Visual C++ (MSVC)* for handling the graphical interface and cross-platform support. However, incorporating these libraries into a project introduces several critical issues:

- **Platform Coverage:** Third-party libraries often have varying levels of support for different operating systems. *Qt*, for example, is renowned for its cross-platform capabilities, but it can add significant overhead to projects where the primary goal is to focus on the core functionality of command-line tools. 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 must 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 a 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 scale 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.

Bridging the gap between a terminal-based interface and a browser-friendly *HTML* output not only simplifies user interaction but also reduces the complexity of UI design for developers. It would decouple both worlds and would enable specialists to work on the user interface without touching the application sources. It would allow to add format specification, localization rules and translation of texts, all things which the core developer didn't spend any time on. And it solves and overcome all the disadvantages mentioned above.

In the following we would like to show how software engineers could stick console in- and output, making life easier for test and maintenance, and still make it possible to attach a graphical user interface with all its flexibility any time later on, without interfering with the given sources.

This approach has many facets and is not restricted to any back-end programming language. The first implementation idea in this direction was actually in 1995, enabling the transition of *DOS* programs written in *Modula-2*, which were implemented using semi-graphic ASCII terminal interface, to *Windows NT*. The main requirement was leaving the sources unchanged. This approach was very successful and so we stepped into new projects of such kind in 2015, generating an interface for a low level C++ profiler and 2017 enabling the usage of *ABAP* running in an HTTP browser. The latter setup was surprisingly flexible and portable and so we where able to reuse this work in a *Python* environment, on which we will focus in the following.

Form the experience with ABAP we created also convenient wrapper for database access, which shows how to handle SQL access in such an environment, restricting output volume and allowing navigation in a huge result sets.

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*, *eezz.database.TDatabaseTable*)
- 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 could be replaced by a module for Apache or as extension in NGInx. The main task would be the implementation of a callback within the websocket implementation.

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.

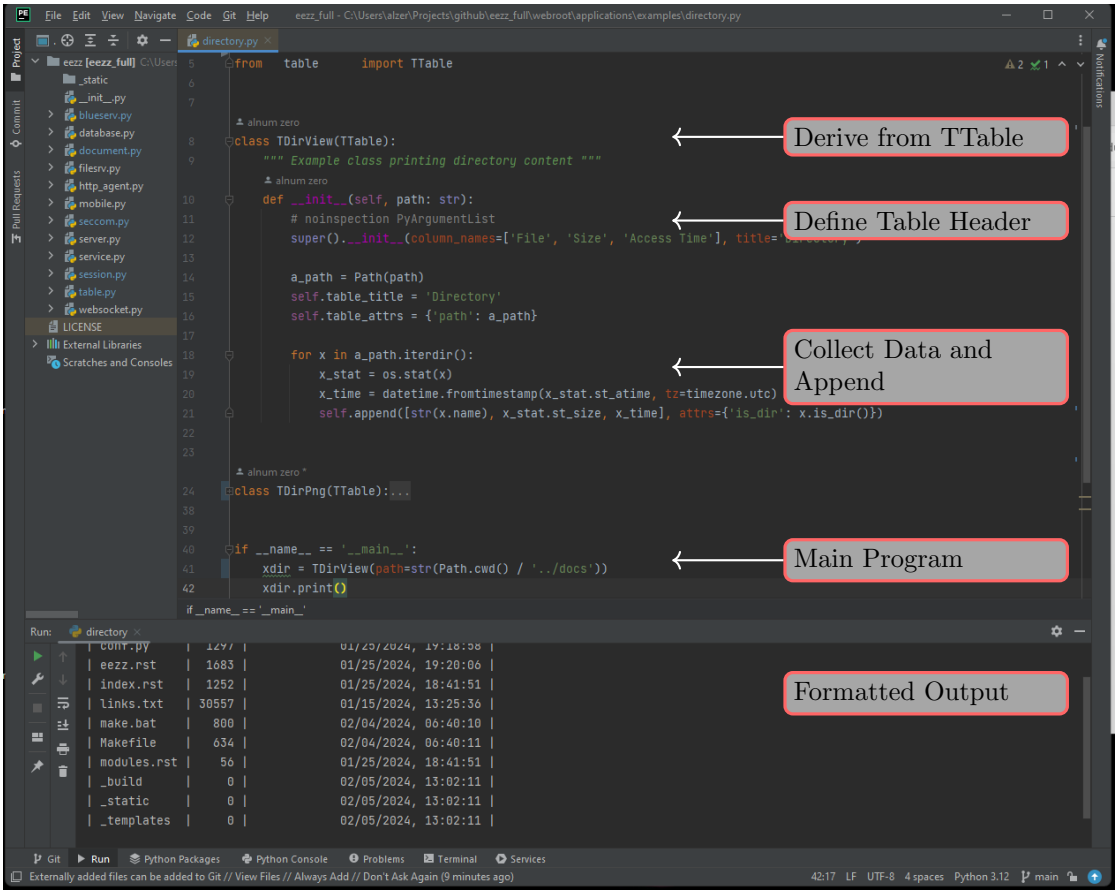

Figure 1: Example for Directory Listing

First thing you should notice is, that there is no hint 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* is a build-in navigation in the data set. You specify the number of rows to return for each access and you could navigate using enumeration class *TNavigation*. This reduces the time for rendering a page significantly.

The extension *TDatabaseTable* is another central element of this concept. It encapsulates database access transparently, with the same properties and interfaces as the base class. I

will mention it here, because database usage is one of the most critical points in your project, where things might get out of control. In quite a few cases I saw SQL statements scattered throughout the project sources, which makes them and very hard to maintain, to adjust or to optimize. Access to the database with *TDatabaseTable* will add the options *limit* and *offset* in most cases, to reduce the data flow in the entire application stack to the absolute minimum and even unrestricted selects will not burst your applications memory. All selected data are stored into a local *TTable* buffer, so that further access could circumvent the most common mistakes using an SQL database. Data, which you append into the table become persistent, if you call the method *TDatabaseTable.commit*. You decide, whether to use cache or database access, by setting property *TDatabaseTable.is_synchron*, before using methods *TDatabaseTable.do_select* or *TDatabaseTable.get_visible_rows*. The database table is created in background for you, using the properties *TTable.title*, *TTable.column_names* and *TDatabaseTable.column_descr*.

There are a huge number of methods and here I will focus on a very special one in this context. The method *TTable.do_select* takes a dictionary with column names as keys. The values could be regular expressions for cache access and for database access using the *sqlite3 'like'* syntax. In general the access to the database is in most cases transparent and there should be no need to worry about SQL any time in this environment.

Another interesting extension is *TTableTree*, which allows the implementation of tree views in this concept. In implements the methods *TTableTree.open_dir* and *TTableTree.read_file*. The former creates a new *TTableTree* object and inserts it at the specified row. A call to an already opened structure will close it The latter reads a file and returns a byte stream.

The attribute *TTable.row.row_id* is a unique key. Trying to insert the same key would issue an *TTableInsertException*. The *TDatabaseTable.row_id* is calculated using the primary keys, so that there will be no double entries. For a *TTableTree* is id has to be unique for the entire tree, which is achieved using a path like structure.

TTable and its derivatives are designed to minimize the data flow from application to user interface, keeping in mind, that rendering data, which will not become visible, would be a vast of time and resources.

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.

```
1 <!DOCTYPE html>
2 <head><title>Title</title>
3   <script data-eezz="template:websocket"></script></head>
4 <html><body>
5   <h1>Simple Table</h1>
6   <table data-eezz="assign:examples.directory.TDirView({query.path})"></table>
7 </body>
8 </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 "*assign*" statement connects our example *TTable* to the *HTML*. The ASCII output in our example above is rendered to fit into this place. That's all you need and the minimal setup for for the first go.

With the curly brackets you have access to the following:

- URL query section: {*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.

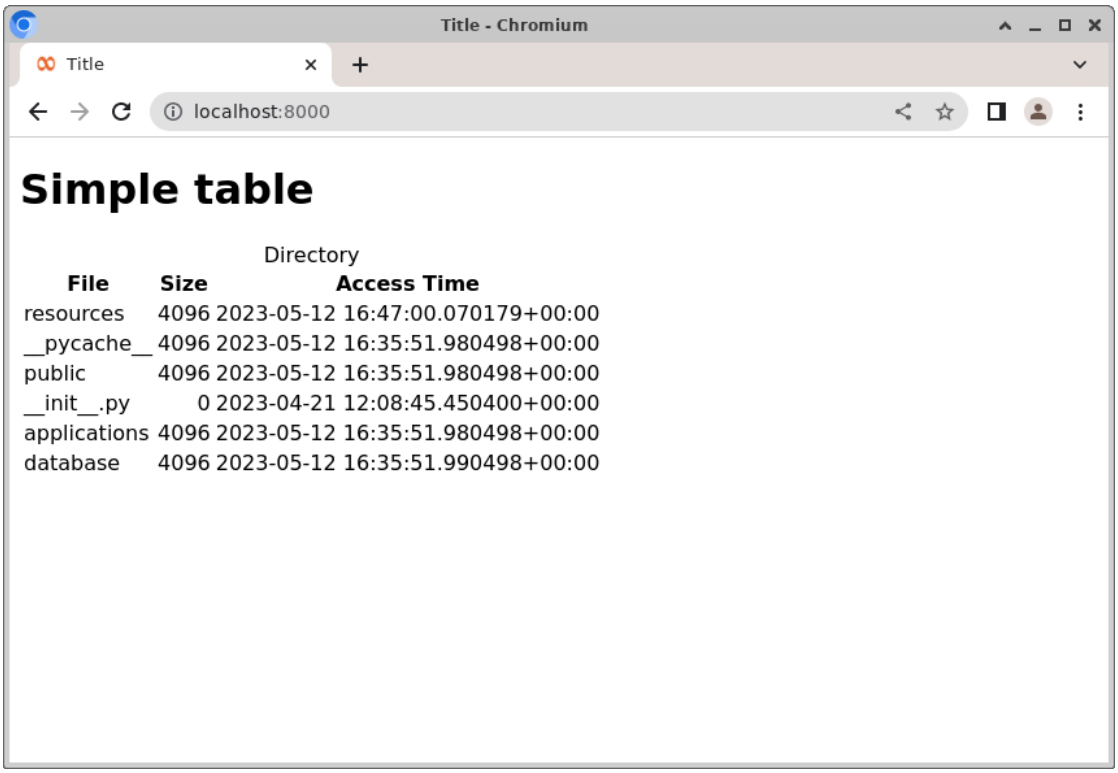

Figure 2: Browser Output: http://localhost:8000

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. Such a small HTML File should be created by the software engineer to hand over to the design expert, so that he knows the associated class and function arguments. Now he could start working with HTML to design the page layout.

In the following sections I will show, that the environment is not restricted on HTML tables, but could also generate trees or grid container. Latter could be used as a starting point for an individual layout. This feature and underlying technology of WebSocket would enable you to design any kind of application interface, even for multi user games, because data could be pushed to an existing page and you application is triggered by activities in the browser. The incoming stream is converted to the right data type and the methods are called by name, so you are not bothered with input validation or output formatting.

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 core developer, you could concentrate on programming in Python. No SQL, GUI-packages, JavaScript nor any of its wrappers or frameworks to interoperate with. We have a clear cut between UI related staff and the rest. The *TTable* is optimized for the usage in this framework, as it could reduce the amount of data to transport in the entire stack, without artificial cuts. And TDatabaseTable offers a transparent way to work with an SQL database, using buffers to reuse selected data and automatically restrict the data volume.

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 JavaScript extension.

The EEZZ extension uses only the user defined property *data-eezz* on dedicated *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 static 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.

```
1 ?start            : list_statements
2
3 ?list_statements  : [ statement  ("," statement  )* ]
4 ?statement        : "event"     ":" function_call               ->
      funct_assignment
5                   | "assign"    ":" function_call               ->
                        table_assignment
6                   | "update"    ":" list_updates                -> update_section
```

```
7                   | "onload"    ":" list_updates                -> onload_section
8                   | "post_init" ":" function_call               -> post_init
9                   | "download"  ":" document "," files          -> download
10                  | "template"  ":" string ( "(" value_string ")" )?  ->
                        template_section
11                  | string      ":" string                      ->
                        parameter_section
12                  | qualified_string ":" value_string           -> setenv
13
14  document        : "document"  "(" list_arguments ")"
15  files           : "files"     "(" list_arguments ")"
16
17  list_updates    : [ update_item  ("," update_item  )* ]
18
19  update_item     : qualified_string "=" function_call        -> update_function
20                  | qualified_string "=" update_string
```

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

As you can see, the grammar description is quit small. The target *template_section* has some hidden aspects. Only some verbs are translated to "*data-eezz-**", as for example "*template*" or "*match*".

3.2 Templates

A template is a node, which is used for rendering and are used to generate table rows and cells respectively. Template child nodes, which are not marked as template are taken as constant and will show up for each entry. This way you could add constant rows and cells.

Missing table parts in our first example Listing 1 are injected using the template definition in Listing 3. Here you find the usage of curly brackets and template definition for the complete table node.

```
<body>
    <table id="{table.id}"
        data-eezz="assign: {table.module}, name: {table.name}, columns: [{table.columns}]">
        <caption
            data-eezz="template: table">{table.title}</caption>
        <thead>
            <tr data-eezz="
                template: row,
                match:  header">

                <th class="clzz_th"
                    data-eezz="
                        template: cell,
                        event: do_sort(index={cell.index})">{cell.value}
                </th>
            </tr>
        </thead>
        <tbody>
            <tr data-eezz="
                template: row,
                match:  body,
                event:  do_select(index={row.index})">

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

Listing 3: Extract of template.html

The *data-eezz:assign* statement connects the *HTML* tag with the python object, which has to inherit from *TTable*. All *data-eezz:event* and *data-eezz:assign* methods have to return a *TTableRow* object for further processing.

The *data-eezz:update* statement may also reference *TTable* methods. These methods on the other hand have to return python bytestream objects. In the update section you specify, which elements on the page needs to be updated after a given event.

The table values are placed into *HTML* elements with attribute *data-eezz:template*. There are row and cell templates. Row templates are used to to design a different layout based on the value of *TTable.row_type* string attribute, which is referenced in the page on *data-eezz:match*. Default supported values are *header* and *body*. In the example of a directory view, you are able to add different icons in front of a row, based on the file type. A row template has access to all *TTableRow* attributes as *row*, using curly brackets.

The cell template has access to all *TTableCell* attributes as *cell*, especially of cause to the value as *cell.value*. It's possible to mix in non template cells into a row. Using the generic cell template this is either at the begin or at the end of a row. For more control you could also assign a cell to a specific column and in this cases you could mix in non template entries on all positions.

3.3 Events

The following standard events are generated by the framework:

- *TTable.do_sort(column = index)* for each column in the thead section
- *TTable.do_select(filters = row_id)* for each row in the tbody section

On a table node, the update is organized implicit. You could also define any other parts of the page to react on events using the *data-eezz:update* key. In this syntax you specify "*update: target = source*", where target and source are HTML node attributes addressed by the node-id: "*<node_id>.<attribute>*".

If you want to display a detail view for our example, you would need to overwrite the select statement and define a new table attribute, say *TDirView.details*, which takes the details to display. In a second step, you create an HTML text node with a unique id, for example *id_detail*, which should display the details in the browser. Now you could create the update statement within the *data-eezz* block: "*update: id_detail.innerHTML=this.details*". You could use the reserved id "*this*", if the source is the current node. Now any click on a row entry will update the given node with the values you extract from the selected row and assigned to the given table attribute. This concept will work not only for text, but also for pictures or tables. Further more, you could choose any source-id in the current HTML document, to exchange data to any target.

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 row. Here you could put your python TTable project.

Arrows are not always interrupted going from very left to right or vice versa, but there is a clear hierarchy. A call is only transferred between neighboring nodes in one step, which is abbreviated in the graph.

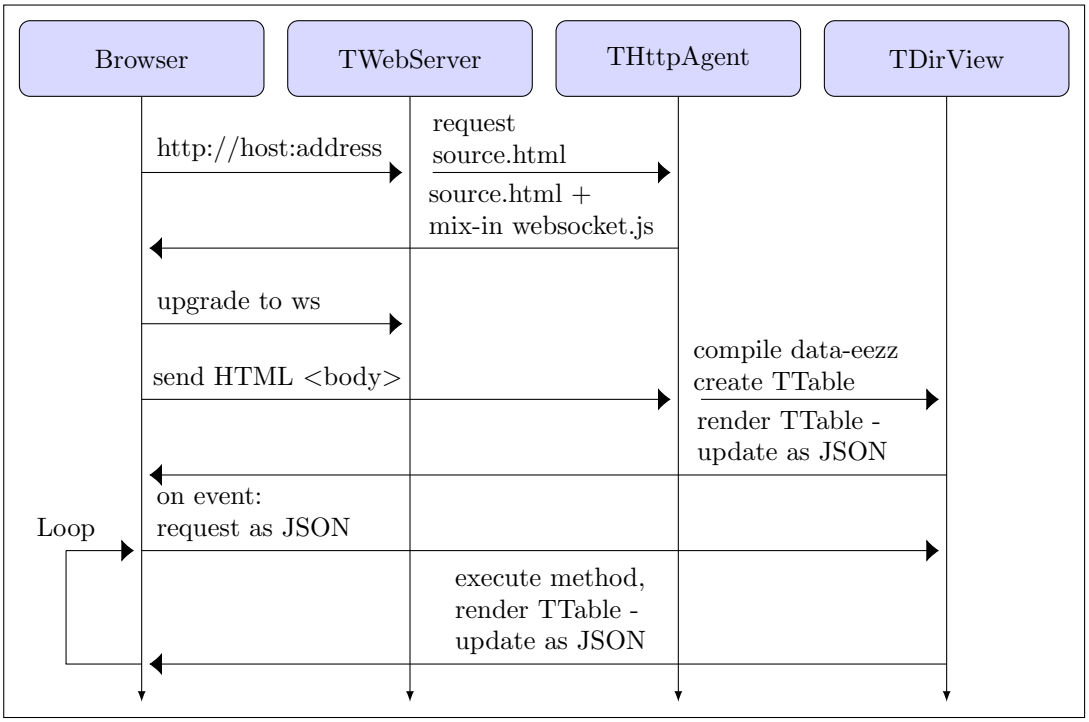

Figure 3: Time lapsed diagram

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. As you could see, the interference with the source code and page layout are minimal and even the grammar is quite compact.

3.5 Summary

The EEZZ extension allows you to connect *HTML DOM* element attributes with the programming logic of a back-end system. It does not introduce another wrapper to *JavaScript* nor introduce another programming language. It does not allow any function variables definition or any programming logic.

It allows a galvanic separation of back-end programming and front-end design. The concept introduces a user interface without the usual security flaws. The technology stack is flat and easy to maintain.

4 Improve the Output

4.1 Using Meta Data

The Listing 4 shows how to access *data-eezz:update* elements. You assign and implement the 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. In listing 4 shows how to manipulate the all new elements before output. Regarding the time format, this is possible, because the cell attributes for a *Python datetime* object is rendered to the *timestamp* attribute. From this you could derive the local time in any format. This example shows, that it's possible to add meta data to each cell.

```
20      <script type="text/javascript">
21          eezz.on_update = (a_element) => {
22              x_element  = document.getElementById(a_element);
23              x_list     = x_element.querySelectorAll("td[timestamp]");
24
25              for (var inx in x_list) {
26                  var x_time       = x_list[inx];
27                  var x_timestamp  = x_time.getAttribute('timestamp');
28                  var date         = new Date(x_timestamp * 1000);
29                  x_time.innerHTML = date.toLocaleTimeString();
30              }
31          }
32
33      </script>
```

Listing 4: 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. Another detail is the yellow element, which is updated with selecting a row. This is done with the update command, described in the next chapter.

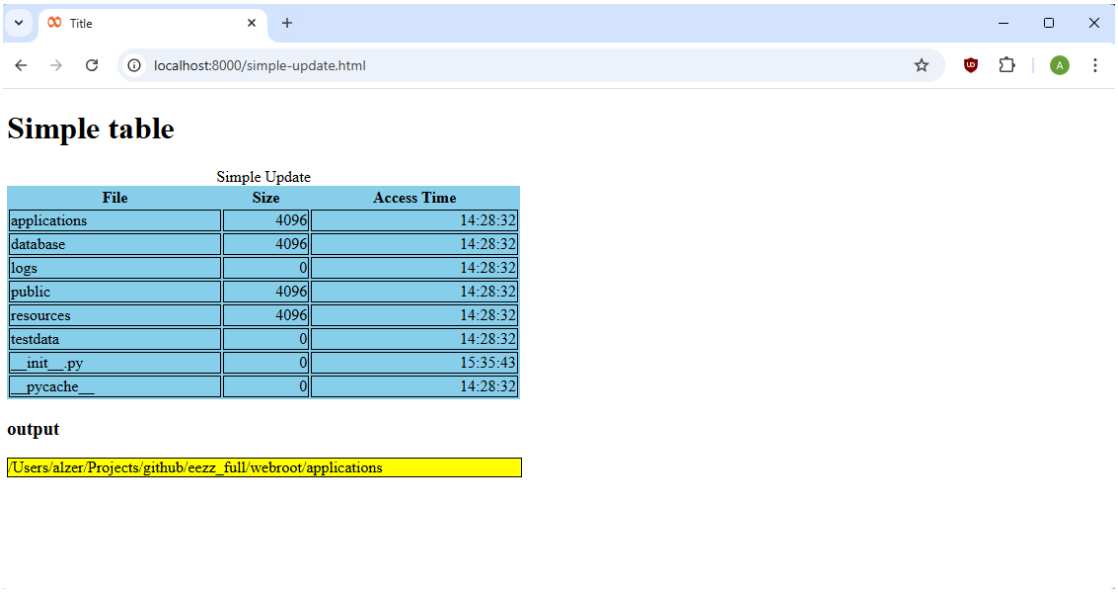

Figure 4: Browser Output: http://localhost:8000/simple-update.html

4.2 Matching Templates

The Listing 5 shows the update request within the *data-eezz* attribute. In this statement the attribute *innerHTML* of an element with the id *path* will be updated after processing the event. The value is compiled as python format string and in this case returns the row_id.

```
42      <table id="Directory"
43          data-eezz="
44              assign: examples.directory.TDirView(title='Simple Update', path='/home/user/')">
45          <tbody>
46              <tr data-eezz="
47                  template: row,
48                  match:  body,
49                  event:  on_select(index={row.row_id}),
50                  update: path.innerHTML={row.row_id}">
51
52                  <td class='clzz_{cell.type}'
53                      data-eezz="template: cell">{cell.value}</td></tr>
54          </tbody>
```

Listing 5: update statement in simple-update.html

The next fragment in listing 6 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, a file icon if set to *is_file*.

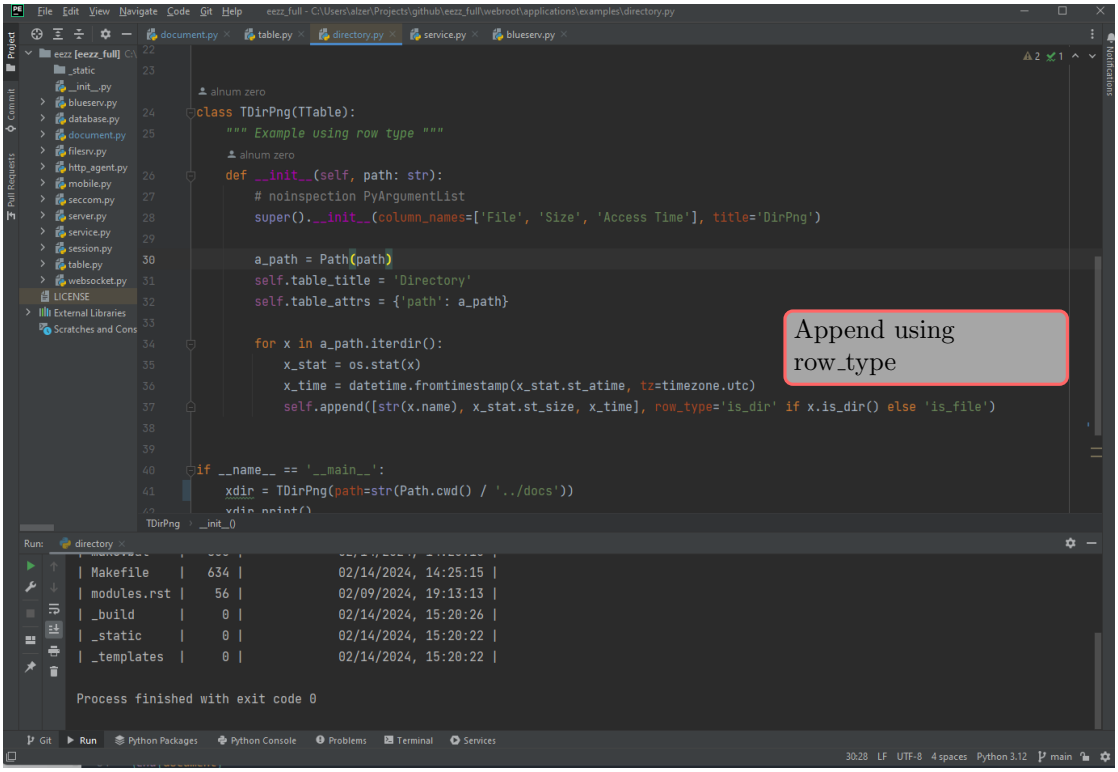

Figure 5: Example for Directory Listing

In the listing listing 6 you could see two templates with different match assignment. Each row type could define different actions for select and update.

```
43          <tbody>
44              <tr data-eezz="
45                  template: row,
46                  match:  is_file">
47
48                  <td><img src="file.png"></td>
49                  <td class='clzz_{cell.type}'
50                      data-eezz="template: cell">{cell.value}</td></tr>
51
52              <tr data-eezz="
53                  template: row,
54                  match:  is_dir">
55
56                  <td><img src="dir.png"></td>
57                  <td class='clzz_{cell.type}'
58                      data-eezz="template: cell">{cell.value}</td></tr>
59          </tbody>
```

Listing 6: simple-tree1.html

For the ASCII output nothing changes. Only for the browser, you could now match the row

types. The template nodes are interpreted like a switch or case statement. For each row the render machine will lookup the corresponding, matching template. In our example the difference is the first cell, which loads individual icons, according to the row type.

In addition you see, that each template has a different prefix element to display different icons for each row type. The result is shown in fig. 6

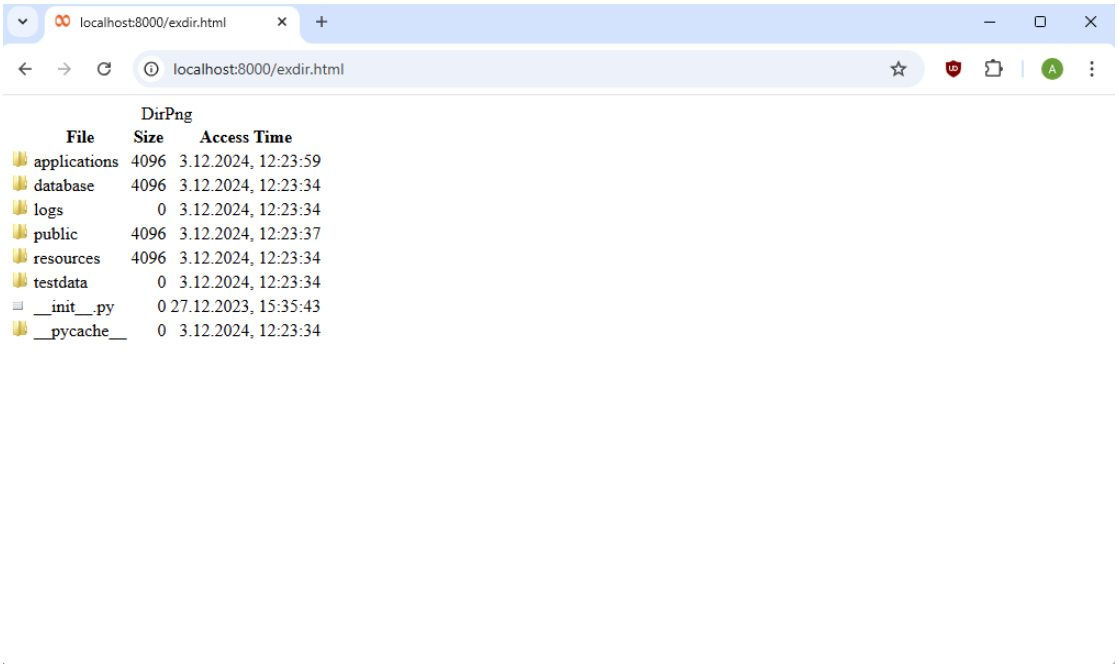

Figure 6: Listing with different Row Types

5 Enhanced User Interaction

5.1 Tree Views

At this step it would be nice, if we could just open the directory node as a tree víew. So let me introduce *TTableTree*, which implements some maintenance for organizing trees. It provides the methods *TTableTree.open_dir* to create a tree node and *TTableTree.read_file* to read any file content. If we define a derivative of this class and activate the methods on the appropriate file types the tree view would be ready to start.

```
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 7: simple-tree3.html

Important for the tree view is the update sequence "*this.subtree=this.tbody*". This puts the *tbody* part of the table in place. With the option "*this.subtree=this.tbody.tfoot*" the *tfoot* is included as well. It's possible to redirect the subtree into another element replacing *this* with the elements id.

The update sequence in the *template:row* with *match:is_file* updates the attribute *innerHTML* of the element *id=text_detail* with the return value of the function *read_file* The update of the element *id=path* with the yellow background is kept from the previous example. The result is shown in fig. 7.

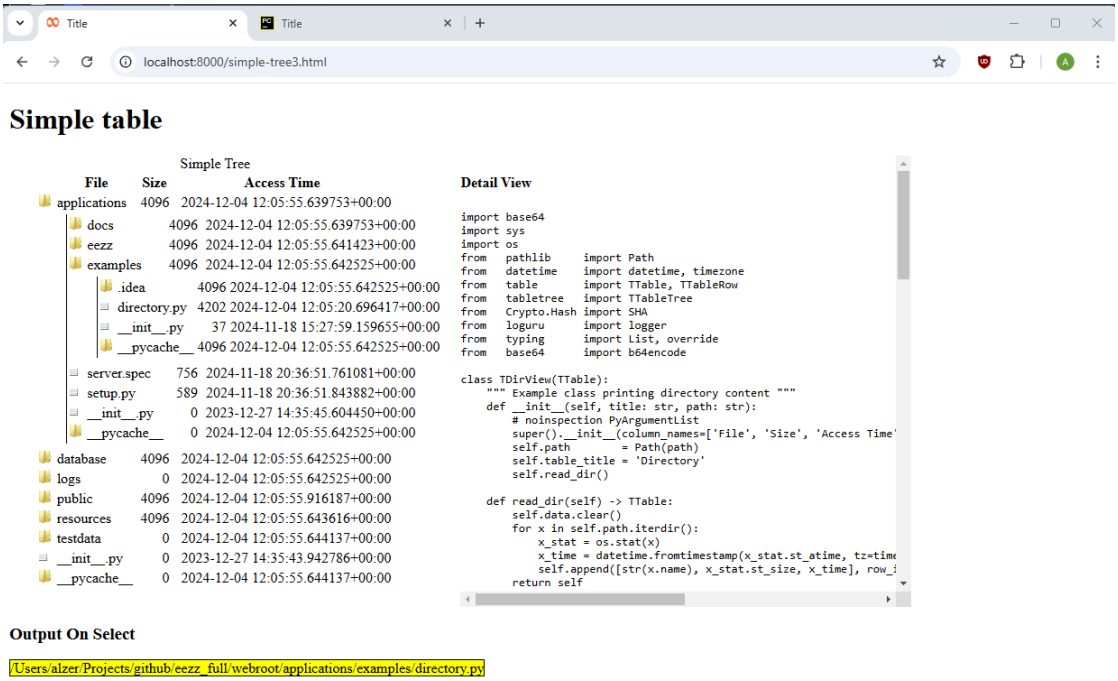

Figure 7: Example for Table and Details

Of cause, this would work with pictures as well as shown in fig. 8, after navigating to *testdata* and selecting *bird.jpg*. In this case we insert an assignment to an `img.src` on this page. For a proper tree view, it would be necessary to specify an appropriate template for each data type.

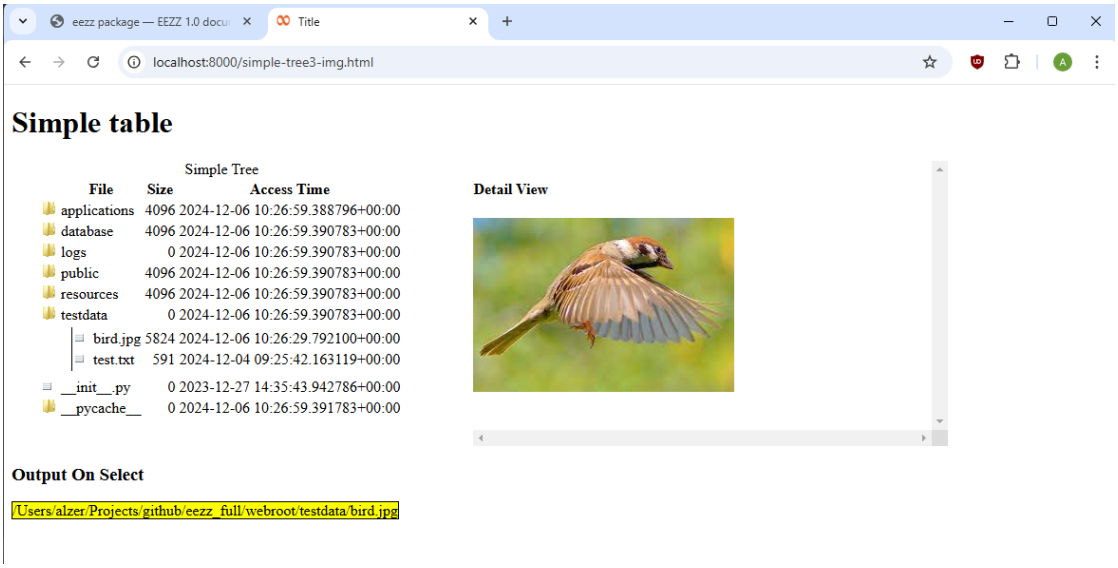

Figure 8: Example for Table and Image Details

Now we got all pieces together and we could start with styling. Distribute elements on the page, choose nice colors and run the application.

5.2 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.

```
56      <div class="clzz_grid" id="Directory" data-eezz='assign: examples.directory.TDirView(
            ↪ title="Simple Tree", path="/Users/alzer/Projects/github/eezz_full/webroot")'>
57
58          <div class="grid-item" style = "position: relative;"
59              data-eezz = "
60                  template: row,
61                  match:  body">
62
63              <span style = "position: relative; vertical-align: center; top: 20px"
64                  data-eezz = "template: cell (File)">{cell.value}</span>
65
66              <span style = "position: absolute; left: 10px; top: 50px"
67                  data-eezz = "template: cell ('Access Time')">{cell.value}</span>
68
69              <span style = "position: absolute; left: 30px; top: 100px"
70                  data-eezz = "template: cell (Size)">{cell.value}</span>
71          </div>
72      </div>
```

Listing 8: Simple Grid Layout

The result is shown in the next picture fig. 9. Again it would be possible to define different tiles for different row types.

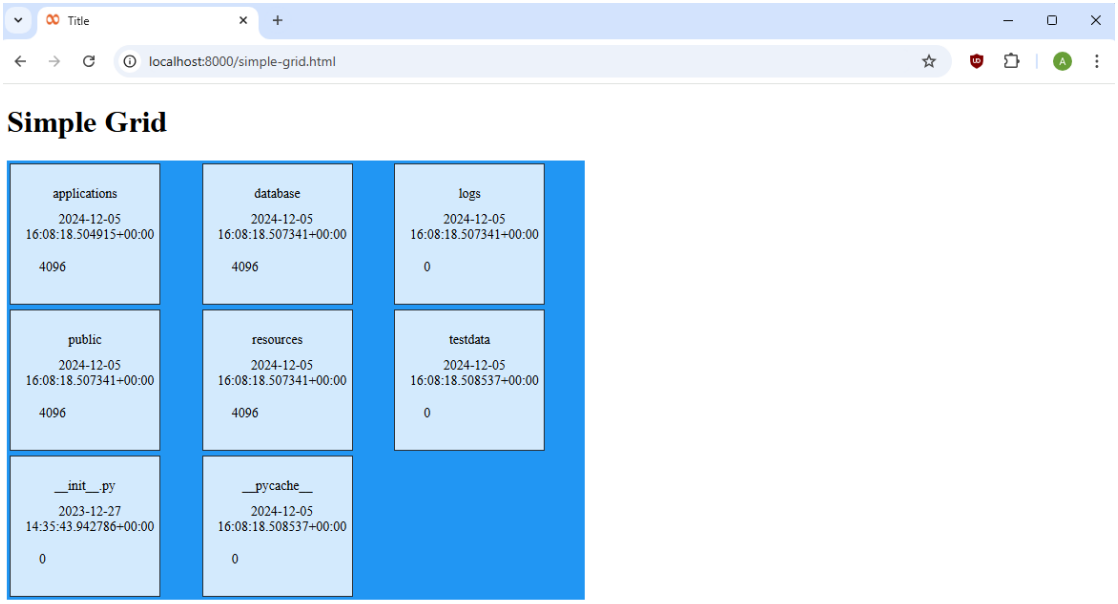

Figure 9: Example for Grid View

In the HTML we reference such cells with the details attribute

5.3 Cell Arrays

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 the grid on the other hand it allows you to create a set of elements of the

same kind. Take for example the calculator app in the following listing Listing 9. It defines two columns, one for the numbers and one for the operators. The *TCalc* class defines two methods to update a display, both returning a byte-stream

```
4  class TCalc(TTable):
5      def __init__(self):
6          self.number_input: int  = 0
7          self.stack: list        = list()
8          self.op:    str         = ''
9
10         super().__init__(column_names=['numpad', 'op'], visible_items=1)
11         self.append([[1,2,3,4,5,6,7,8,9,0], ['+','-','*','=']])
12
13     def key_pad_input(self, key) -> bytes:
14         self.number_input *= 10
15         self.number_input += int(key)
16         return f'{self.number_input}'.encode('utf8')
17
18     def key_op_input(self, key) -> bytes:
19         result: float = 1.0 * self.number_input
20
21         self.stack.append(self.number_input)
22         self.number_input = 0
23
24         if len(self.stack) == 1:
25             self.stack.append(key)
26         if len(self.stack) == 3:
27             if self.stack[1] == '+':
28                 result = self.stack[0] + self.stack[2]
29             elif self.stack[1] == '-':
30                 result = self.stack[0] - self.stack[2]
31             elif self.stack[1] == '*':
32                 result = self.stack[0] * self.stack[2]
33             self.stack.clear()
34
35             if key != '=':
36                 self.stack.append(result)
37         return f'{result}'.encode('utf8')
```

Listing 9: 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.

```
27      <div style="display: grid; grid-template-columns: repeat(3, 30px); gap:5px;">
28        <input  type  ='button'
29                value ="{detail.value}"
30                data-eezz ="
31                    template: cell(numpad.detail),
32                    event :   get_header_row(),
33                    update:   display.innerHTML = key_pad_input(key={detail.value})"/>
34      </div>
35      <div style="display: grid; grid-template-rows: repeat(5, 30px); gap:5px;">
36        <input  type  ='button'
37                value ="{detail.value}"
38                data-eezz ="
39                    template: cell(op.detail),
40                    event:    get_header_row(),
41                    update:   display.innerHTML = key_op_input(key={detail.value})"/>
```

Listing 10: Calculator Layout

As a result we have an calculator as a working application with low efforts. Such approach enables some fast prototyping, which is always a good starting point for further development.

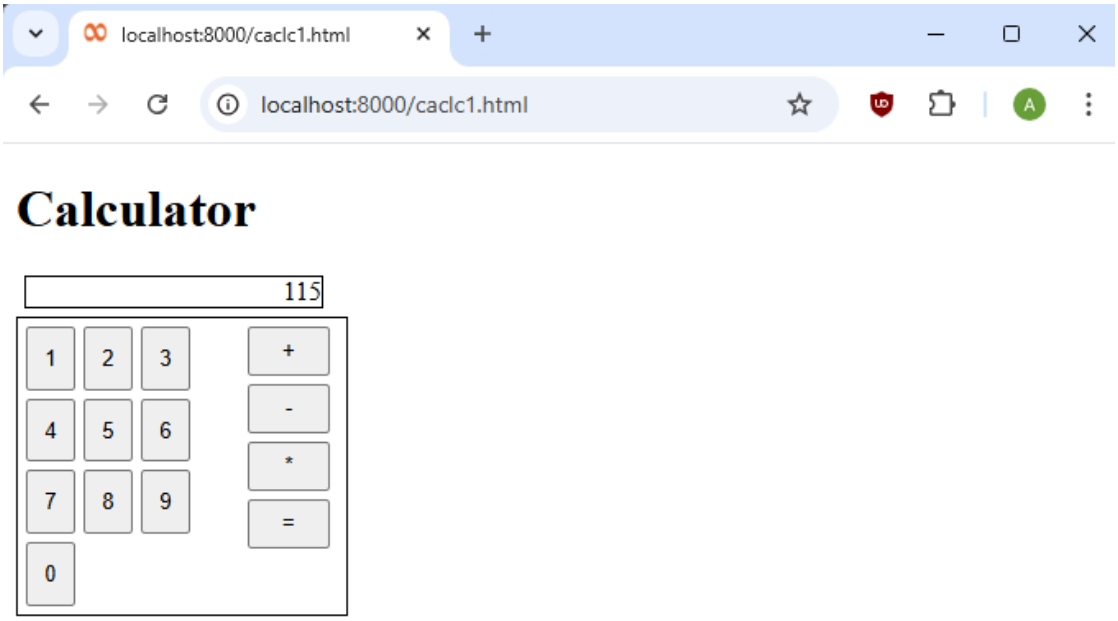

Figure 10: Calculator with Cell-Array

5.4 Form Input

Forms are implemented in the environment as a logical extension of the grid concept. We define a `TTable` and set *TTable.visible_rows=1*, where we specify the default or help values and the data types of each cell, as shown in listing 11.

```
111 class TFormInput(TTable):
112     def __init__(self, title: str):
113         super().__init__(column_names=['Name', 'FirstName', 'City'], title=title)
114         self.visible_items = 1
115         self.append(['mame', 'first name', 'city'])
116 
117     def register(self, table_row: TTableRow) -> None:
118         print(f"Register...{table_row}")
119         super().append(table_row)
```

Listing 11: Python Script driving Form Input

In the *HTML* page, you specify the input tags and the submit button, as shown in listing 12. 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.

```
59         <div class="grid-item" style="position:relative;"
60             data-eezz="
61                 template: row,
62                 match:  body">
63 
64             <input type = "text" style = "position: absolute; top: 10px; left: 10px"
65                 data-eezz = "template: cell (Name)"/>
66 
67             <input type = "text" style = "position: absolute; top: 40px; left: 10px;"
68                 data-eezz = "template: cell (FirstName)"/>
69 
70             <input type = "text" style = "position: absolute; top: 70px; left: 10px;"
71                 data-eezz = "template: cell (City)"
72                 style = "position: absolute; top: 70px; left: 10px;"/>
73 
74             <input type = "button" style = "position: absolute; top: 100px; left: 10px;"
75                 id = "commit_button" value = "submit"
76                 data-eezz ="event: register(table_row = [template.cell])"/>
77         </div>
78     </div>
```

Listing 12: 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. There is no need to distribute and match values in your back-end application.

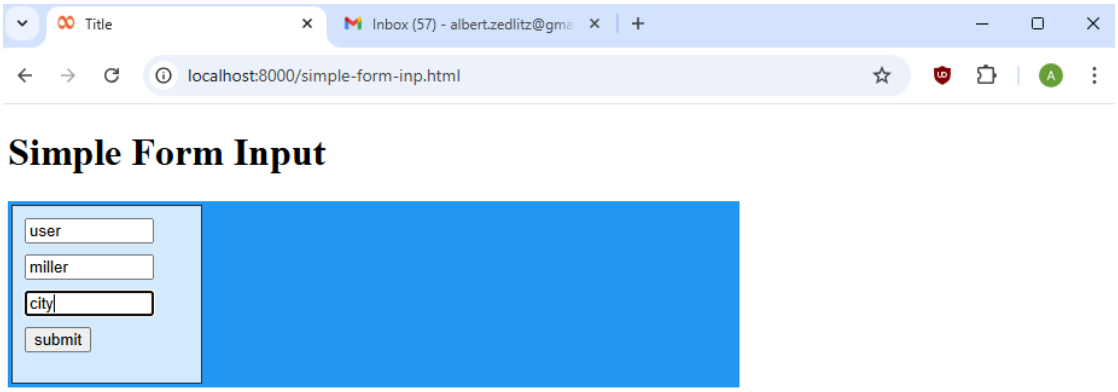

Figure 11: Form Input

You could derive your own *TTableCell* to provide additional help on error or on hoover.

5.5 Creating 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 for each document to fill a tile on your HTML page.

The file management is bundled in the class *TDocument*. In combination with the class *TFile*, a chunk-based, non-blocking download allows for possible feedback via a progress bar, which is particularly useful for large files.

To make this work in the browser, we need to create a new class that inherits from both *TDocument* and *TTable*, resulting in a bookshelf where each entry in the table represents a document. We associate the document attributes with the table columns, enabling us to create an input form for data and file selection.

This approach allows for efficient content management and easy updates. By encapsulating the document attributes and file management within these classes, we can ensure a smooth and organized workflow. Additionally, the non-blocking download mechanism enhances the user experience by providing real-time feedback during file transfers.

Overall, this method simplifies the process of managing and displaying compound documents on your HTML page, making it easier to maintain and update content as needed.

```
19 class TSimpleShelf(TTable, TDocument):
20     shelf_name:     str
21     attributes:     List[str]    = None
22     file_sources:   List[str]    = None
23     column_names:   list         = None
24
25     def __post_init__(self):
26         """ Initialize the hierarchy of inheritance """
27         # Adjust attributes
28         self.attributes   = ['title', 'descr', 'price', 'valid']
29         self.file_sources = ['main', 'detail']
30         TDocument.__post_init__(self)
31
32         # Set the column names and create the table
33         self.column_names = self.attributes
34         TTable.__post_init__(self)
35
36         self.append(table_row=['' for x in self.column_names])
```

Listing 13: Bookshelf Example

To explain the process, Let's start step by step starting with the triggering of document creation and looking at the eezz commands for the commit button in Listing 14.

The button event targets the method *TDocument.prepare_document*, which collects and stores the input values as document attributes. The update command shown in the example is a JavaScript method. You can think of the eezz update notation as the left hand for JavaScript and the right hand for the server. If the left hand is a method, it will be executed in the browser runtime as JavaScript.

With this the server sets a trigger in the browser, after initialization of the document, to start the download using the list of names correlated as templates: *data-eezz=template:cell(name)*. In our example this translates to a JavaScript call with a JSON attribute: *read_files({'files':['main', 'detail']* }). Of cause it would be possible to write own JavaScript extensions.

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

Listing 14: 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 UTF8 encoded byte-stream, to update the progress bar.

```
112 <div style="display: grid; grid-template-row: auto auto; row-gap: 4px;">
113     <input
114         type = "file"
115         data-eezz="
116             template: cell (detail),
117             update:
118                 progress_detail.style.width = download_file(file=this.file, stream=this.
                        ↪ bytestream),
119                 progress_detail.innerHTML   = progress_detail.style.width"/>
120
121     <div style="width:200px; background-color:red">
122         <div id="progress_detail"
123             style="width: 50%; background-color:green">50%</div></div>
124     </div>
```

Listing 15: File Download Snipplet

For the download the class *TDocument* uses *filesrv.TFile*, which handles fragmented data input for the download stream. The dialog would look as follows after the download was successful.

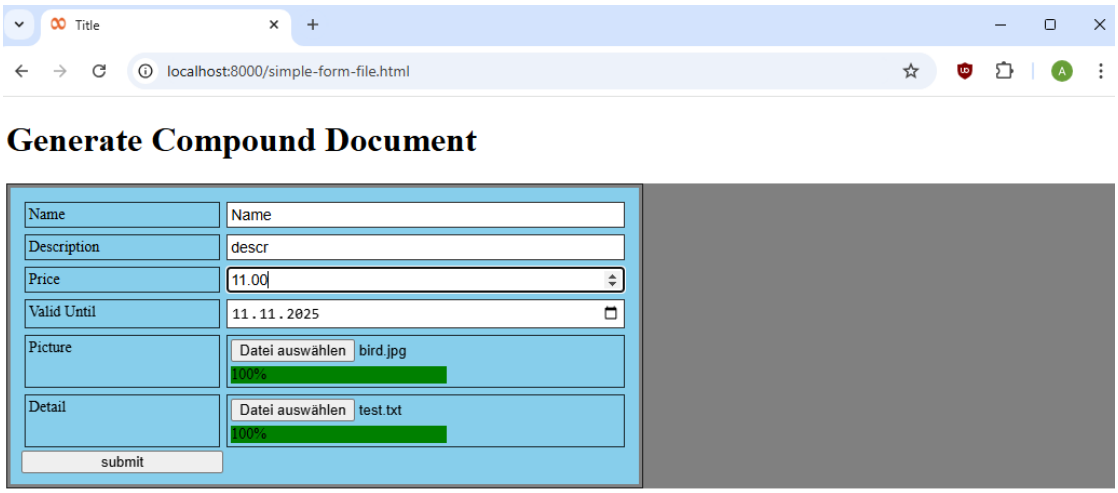

Figure 12: Form Input with File Download

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.

To accomplish this, you need to store the location and header content in a database. We achieve this with some minor steps:

- Derive from *TDatabase* instead of *TTable* and set the parameter *database_name*.
- Adopt the naming convention of the SQL column and table names. The table name is set to the value of the variable *TDatabase.title*.
- Call the method *TDatabase.commit* after successfully creating a document. For lines, which should not be visible in database, just use *TTable.append(self,...)*

```
53 class TDatabaseShelf(TDatabaseTable, TDocument):
54     shelf_name:     str
55     attributes:     List[str]   = None
56     file_sources:   List[str]   = None
57     column_names:   list        = None
58
59     def __post_init__(self):
60         """ Initialize the hierarchy of inheritance """
61         self.attributes   = ['title', 'descr', 'price', 'valid']
62         self.file_sources = ['main', 'detail']
63         TDocument.__post_init__(self)
64
65         # Set the column names and create the table
66         # Hide the first line from database commit using TTable append
67         self.database_name = f'{shelf_name}.db'
68         self.column_names  = [f'C{x.capitalize()}' for x in self.attributes]
69
70         TDatabaseTable.__post_init__(self)
71         TTable.append(self, table_row=['' for x in self.column_names])
```

Listing 16: Bookshelf with Database access

The data are loaded from the database by calling the built-in method *TDatabase.get_visible_rows*. You can use the filter method to apply a wide range of selection criteria. This process is fairly simple and demonstrates one of the key features of the entire concept. The database itself is stored at the location specified with environment variable *TService.database_path*.

5.6 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:

```
38     <div class='clzz_grid'
39         id="WebCamClass"
40         data-eezz='
41             assign: examples.webcam.TCamera(),
42             update: webcamimg.src = read_frame()'>
43     </div>
44
45     <img id='webcamimg'/>
```

Listing 17: WebCam as Push Service

In our example in Listing 17 the element with the id *webcamimg* is changed whenever the function *read_frame* returns a picture. The implementation is straight forward as shown in Listing 18.

```
8  class TCamera(TTable):
9      def __init__(self):
10         self.cam = cv2.VideoCapture(0)
11         super().__init__(column_names=['camera'])
12         logger.debug(self.column_names)
13
14     def read_frame(self) -> bytes:
15         time.sleep(5)
16         ret, frame  = self.cam.read()
17         ret, buffer = cv2.imencode('.png', frame)
18         jpg = BytesIO(buffer)
19         return jpg.getvalue()
```

Listing 18: WebCam Implementation

And that's it, as shown in Figure 13. 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.

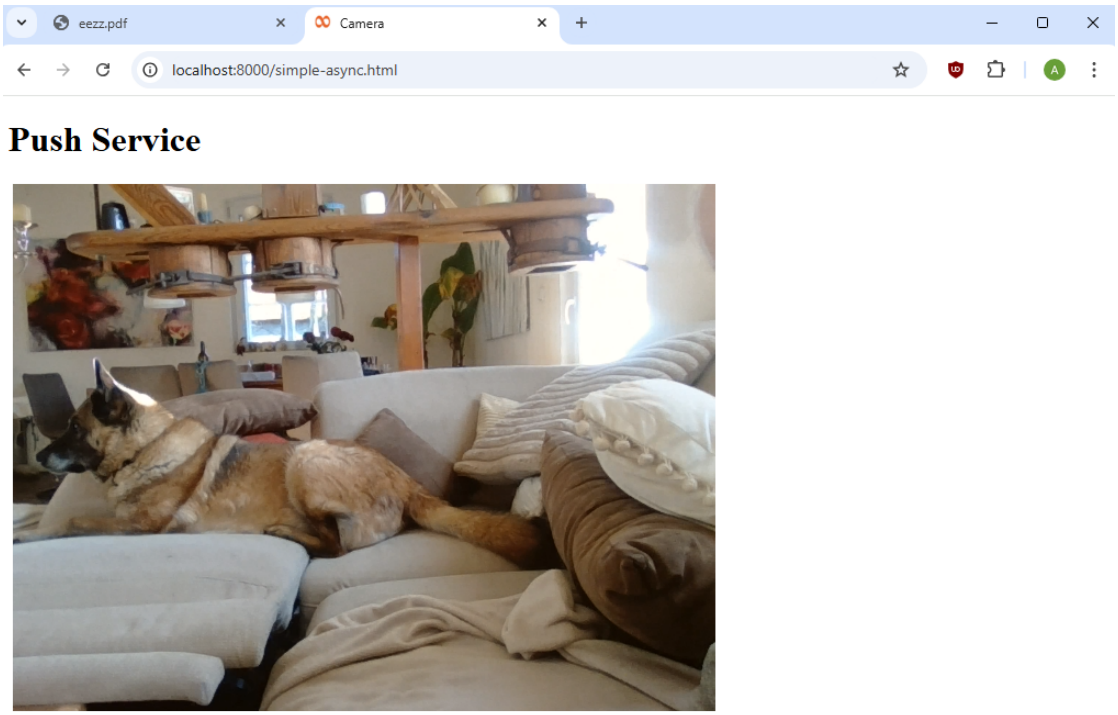

Figure 13: WebCam as Push Service

5.7 Summary

This chapter provided a brief overview of the potential interactions between HTML and TTable. We demonstrated how easily Python class attributes, such as `cell.type` or `cell.value`, can be accessed. Additionally, by utilizing row types, we can design unique rows (e.g., different icons) for each user-defined row type. Finally, by using `JavaScript.eezz.on_update`, we can access and manipulate incoming data in the browser before it is output.

6 Conclusion

We can summarize the benefits of this approach as follows:

- Start the project with low effort as a command line app and add a graphical user interface later on.
- Define graphic layout and translation without interfering with the application code.
- Start the graphical user interface with on-board tools and migrate to cloud features later on, if necessary.
- Reduce the number of programming languages and frameworks to an absolute minimum.
- A test framework is very easy to set up, as you can run the application and user interface independently.

Summarizing the benefits of this approach, you can start a command line-driven project and add a user interface later on. The user interface can be designed without touching the application code. There are no text segments or formatting that are out of the control of the UI designer, ensuring a smooth design. Strings can be translated consistently, and there is no risk of sending unexpected HTTP fragments.

In the user interface, you specify the methods you want to call, including the input parameters, thus avoiding complex HTTP forms for the browser and mapping procedures in the application. The update requests allow you to place the return value of any request into the user interface, providing a direct response for any event.

The application developer can test their program without driving a user interface, which is a significant benefit and makes it easier to establish automatic tests.

In this setup, the browser is designed as an application interface for a single user. The Python HTTP server used is not meant to go online and has special features to access the user ID and coupled Bluetooth devices. For public access, a more sophisticated server should be used, and some features for accessing local data have to be dropped. Thus, the current implementation can be enhanced for cloud applications, as done earlier in an ABAP project with some restrictions.