yaji - Manual

Release 0.1.463

Josef Hahn

Aug 16, 2020

CONTENTS

CHAPTER

ONE

LICENSE

yaji is written by Josef Hahn under the terms of the AGPLv3.

Please read the *LICENSE* file from the package and the *Dependencies* section for included third-party stuff.

CHAPTER

TWO

ABOUT

Yaji is a Python package allowing to implement graphical user interfaces that are accessible by just web browsers. This includes usual desktop or mobile web browsers, even on remote machines, but also embedded desktop applications.

CHAPTER

THREE

UP-TO-DATE?

Are you currently reading from another source than the homepage? Are you in doubt if that place is up-to-date? If yes, you should visit https://pseudopolis.eu/wiki/pino/projs/yaji and check that. You are currently reading the manual for version 0.1.463.

CHAPTER

FOUR

MATURITY

yaji is in production-stable state.

CHAPTER

FIVE

DEPENDENCIES

There are external parts that are used by yaji. Many thanks to the projects and all participants.

 *PyQt5 incl. WebEngine*, optional : otherwise you will need a usual web browser

 *banner image*, included : _meta/background.png; license CC BY 2.0; copied from here.

CHAPTER

SIX

OVERVIEW

6.1 Hello World

```
class MyApplication(yaji.Application):

    def __init__(self):
        super().__init__()
        self.setbodyleft(yaji.View(label="Hallo Welt."))
```

Yaji is a library for graphical user interfaces that are accessible by web browsers. However, it is not a framework for websites. It would lack some features for that on the one hand, while it has some features that would not make sense for that on the other hand. Instead, the entire web server comes up when an end user starts a Yaji-enabled application. That application steers one single user interface, as a usual program with a local user interface would do. The web server is bound to that particular process with all that user interface state, which does not make it a good choice for a website.

Compared to a local user interface, it is a bit more tricky to implement, but it allows to access the user interface easily via the network. It might be not worth taking the additional effort, if this network capability is not a requirement for you.

Writing browser side code is possible, but not needed. Yaji allows implementing rich user interfaces by just Python code. It also allows direct access on browser side for advanced cases, e.g. by additional JavaScript code, but a good practice is to use that in a minimal way (usually not at all).

6.2 How applications are shown

By default, a Yaji user interface tries to open a window on your local desktop for your user interface on start. This follows the assumption that local access is the preferred way.

Internally, it uses the WebEngine in PyQt for showing a window that mostly is a tiny browser, just without all the controls around the actual website. If that fails, it tries to open itself in the local browsers.

Furthermore, if it succeeds or not, there is a tiny local web server, which usual web browsers can connect to. Of course, depending on network structure and firewall rules, this can also be a web browser on another machine. It is possible to connect to the user interface with multiple browsers in parallel, even though this is only useful in a few scenarios. Depending on how it is set up by the application around it, it may or may not be possible to disconnect from it completely with all browsers, and to reconnect back much later without loss of state data.

Note: Setting the environment variable `YAJI_CALLCMD` to a custom command line causes a Yaji application to open new browser views by executing that instead of its default routine. This command line should contain `%u` for taking

the URL to open.

6.3 Views run decoupled from application code

There is one challenge with this approach by nature that leads to the additional trickyness in comparison to a classic local user interface. In the local case, there is one "UI" thread with a main loop, which all user interface code gets piped through in a mostly first-in-first-out way. This makes it easy to keep things in a consistent state: Whenever any interaction by the user occurs, program code can directly compute a new user interface state, like showing/hiding or enabling/disabling some parts of the user interface depending on some conditions. That typically happens in no time the user could realize. The user is also not able to intervent meanwhile nevertheless, even if computation takes longer, since all further user interactions just take a place in the queue of the main loop until the computation finishes. For example, there is no way to click on a button while a computation is going on that eventually disables it (unless e.g. some multi-threaded application code is explicitly doing it this way).

A Yaji application has to implement such situations differently, because that single main loop does not exist. The application (Python) code and the presentation of the user interface run in a much less encoupled way. The user interface will typically not block while application code is running. Furthermore, application code that modifies the user interface will potentially finish before the view actually reflects that (which is quite understandable because there can be from no to *n* views, e.g. by opening it in multiple browser tabs). This makes it harder to keep the user interface in a consistent and 'correct' state (whatever that means for a particular user interface logic). For example, a user could interact in some way that triggers the application code to disable a particular button, but then click on that button before it actually becomes disables. In fact, there are two asynchronous transitions in this case: The user interaction triggering the application code and the application code triggering the button state change. There are some ways to work around that, like curtains, or implementing some pieces of JavaScript code, which will be mentioned in detail later. All of them come with some additional efforts to spend.

6.4 Sample applications

For many aspects of Yaji there are some small sample applications. The following text will explicitly refer to some of them, but it is a good general advice to study them for better understanding. However, there are some remarks one should keep in mind while reading them:

- They tend to keep things simple and are generally not production ready,
- they typically do completely useless stuff, just for showing a particular feature or aspect of Yaji, and
- they sometimes solve a problem in a 'creative' way, while one would solve it in completely different ways for real code,
- they sometimes play around with different ways of doing the same thing (e.g. writing a data structure in different ways) if that is worth showing.
- However, many of them show only a subset of the available functionality.

The sample applications are located in `./_meta/sampleapps`. Each `.py` file is for one sample application (but `____sampleapp.py` which has a special internal meaning). They are directly executable from command line. Some of them come with additional files, e.g. a browser side JavaScript, which are important parts of the application. For a `foo.py` this would be `__foo.js`. They are automatically loaded by Yaji without an explicit reference to them somewhere in the `.py` code.

6.5 A typical Yaji application

A typical Yaji application potentially does the same things as local graphical applications do. It implements a user interface based on Yaji typically in the following way (just trying to give an idea about many different things; it is *not* required to understand each point in detail now):

- It implements a subclass of *yaji.app.Application*. This can either be a real application, or just another abstract enhancement for implementing by deeper subclasses. The following assumes a full, non-abstract implementation.
  - Inside the constructor, it configures the user interface by assembling complete forms from basic parts like buttons, labels and layouts, and assigning that to the main view.
    - Those views are *connected* to the application code by data bindings and event bindings. The former keep data on view side in sync with some data on application code side, while the latter trigger some code execution when some event occurs in the view (e.g. the user triggered a button).
    - Application code only sets completely assembled views and cannot change only some properties of some subelements or insert/remove/replace subelements of a view; with some important remarks:
      - Data bindings are an exception to that rule, since keeping a property of a widget in sync with application data is exactly their primary duty. Furthermore, since `visibility` and `enabled` are also just usual properties of each widget, many kinds of dynamic changes in the user interface structure can be realized by data bindings to them.
      - The usual main view is split into a left and a right column, so there are at least two separate views one can set independently from the other. There are also some more situations that involve building a view, like dialogs.
      - Setting the main views to new ones at some times during the application execution is possible, so structural changes of the user interface during application execution are possible by completely rebuilding a view each time.
  - It provides request handlers for processing all kinds of requests by the view side. For now, those are the handlers used in the event bindings (see above), but they can have other uses. So they react on some kinds of user interactions like triggered buttons or menu items (while *data* bindings are only internally related to that).
    - A request handler is a method of your subclass, usually named in a particular way, and annotated in a particular way. It contains the application specific handler code for that event.
  - It optionally overrides some methods in order to customize some low-level behavior, e.g. as one piece of the integration into something bigger.
- It optionally provides some browser side resources for advanced cases, e.g. a `.css` style file or a `.js` JavaScript. If there is a file like `__foo.js` for an application `foo.py`, Yaji will automatically load it without any code needed.
  - Browser side code can implement more complex and more tailor-made widget classes that can be used by the application code instead of simpler widgets like buttons and labels. They can react to the user in a more direct way than application code can, so this is needed in some cases.
  - It can do everything user interface related that application code can do, and much more, in a more direct way, meaning: next to the user, but far away from the application code side instead.
  - It has access to data sources, but as mentioned earlier, there is will be a time gap between updates on application code side and browser side.
  - Communication with the application code side is based on Ajax calls and/or higher-level mechanisms based on that.

The following chapters will go more into detail.

6.6 Application starting and stopping

At some time typically during startup, the application code creates an instance of that subclass and calls *yaji.app.Application.start()* on it. Eventually, there should be some code somewhere that triggers to stop the application again. A later chapter will go more into details, but the following code is one example of doing both:

```
from yaji import *  # do not use * in your code


class MyApplication(Application):

    def __init__(self):
        super().__init__()
        self.setbodyleft(View('Button', label="Stop",
                              OnClicked=self.bindevent(BackendFunction('stop'))))

    @RequestHandler.for_urls(auto=True)
    def _do_stop(self):
        self.stop()


if __name__ == '__main__':
    MyApplication().start()
```

6.7 Correctly importing it

There is nothing one can particularly do wrong with importing `yaji`. However, there are different ways, of doing it, and mixing them can confuse at first. This documentation does mix it occassionally for different reasons, so you should be aware of it.

The `yaji` package is divided into some submodules, each of them having some direct members, like class or function definitions. The main application base class is `Application` inside the `app` submodule. So, after `import yaji.app` its qualified name is `yaji.app.Application`. The root package itself also includes all members that are typically used by outside code. This is for convenience as it reduces the amount of typing and/or thinking about the correct imports to make, but it can be confusing when seeing Yaji class or function names. After just `import yaji` the same class has the qualified name `yaji.Application`. Due to technical reasons it has only this name and not the longer one at all (of course it is possible to make both import statements if that is needed e.g. during some code refactorings).

So, whenever a short name is mentioned somewhere, there is also a longer name (which also includes the submodule name) that is pointing to the same thing. The API reference here lists all module members with their long names, so one way to find out a long name is to search for the very last segment of the name in the API reference.

For keeping lines shorter, most examples in this text assume `from yaji import ...` and would just refer to e.g. `Application`.

CHAPTER

SEVEN

BASIC USER INTERFACES

7.1 setbodyleft and setbodyright

Application code sets up the user interface inside the constructor and optionally any time later, by building complex user interface structures from basic ones like buttons or labels, and layout elements, which align widgets (or other layout elements) inside it in a particular way. It applies that by calling *yaji.app.Application.setbodyleft()* and/or *yaji.app.Application.setbodyright()*, like the following:

```
def __init__(self):
    ...
    self.setbodyleft(View('Label', label="Foo"))
```

This shows a widget of class `Label` with some properties, precisely a label showing the text "Foo". As the method name suggests, there is a distinction between left and right. This does not mean anything as long as either only `setbodyleft` or `setbodyright` is used. When both are set in parallel, the view splits into a left and a right side. There is a splitter dividing both sides, which the user can move.

7.2 View specifications

The parameters passed to *yaji.app.Application.setbodyleft()* and some other functions are instances of *yaji.gui.View*. They completely specify a user interface view with simple elements like buttons, and stack them together in some potentially complex ways with *layouts*. Each subpart within this structure, i.e. each simple elements as well as each layout is a `View`, up to the final complete specification. Each `View` instance has a *class*, which is a string like `'Button'`, `'Label'` or `'HorizontalStack'` and some property assignments. Those assignments can be just any values of types that make sense (e.g. strings for label texts). They can also be data bindings, and for events, they are usually event bindings. The following example shows another view configuration, still without layouts, and kind of nonsensical:

```
View('Button', label=self.binddata('buttonlabeldatasource'), enabled=False,
     OnClicked=self.bindevent(BackendFunction('dosomething')))
```

Data bindings and event bindings are explained in detail later.

There is a shortcut for label, which you could find used in other applications like the samples. Whenever a `label` property is given but no class, `"Label"` class is used. So, the following lines each specify the same thing:

```
View('Label', label="Foo")
View(label="Foo")
```

7.2.1 Layouts

Building complex views out of simple pieces stacked together uses the same mechanism, with e.g. a `'HorizontalStack'` class widget with one of its properties used for assigning children widgets. However, for builtin layouts, there is a slightly different, shorter notation, since there are special classes available for them. The following example stacks some pieces together:

```
self.setbodyleft(
    VerticalStack(
        View('Label', label="Foo"),
        HorizontalStack(
            View('Label', label="Bar"),
            View('Label', label="Baz")
        )
    )
)
```

Next to *yaji.gui.HorizontalStack* and *yaji.gui.VerticalStack* there are also some more layout. The *yaji.gui.Grid* layout is a quite flexible and powerful one:

```
Grid(
    View('Label', label="Foo", row=0, col=0),
    View('Label', label="Bar", row=1, col=0),
    View('Label', label="Baz", row=1, col=1)
)
```

7.2.2 Scroll views

Another kind of container, similar to a layout, is *yaji.gui.ScrollView*. It is used whenever a subelement inside the user interface might be larger than the space available for it. It does so by scroll logic and can be used like this:

```
ScrollView(
    View('SomethingLarge')
)
```

Note: Having more than one widget inside a scroll view requires to place a layout around them before.

There are also some other, more complex containers like tab views or carousels. A later chapter helps to find out details about them. However, there are no shortcuts available for them, so using them requires a complete `View` specification with a class name and some property assignments.

7.2.3 Sizing

Due to the size the view window has and due to the user interface specification assembled from hierarchies of layouts and actual widgets, sizing for the inner leaf widgets is predetermined to some degree. However, the view window typically does not have exactly the perfect size for presenting the specified user interface. It might be too small in one or both dimensions. That would break the usability of the interface and could be considered a defect. *Scroll views* should be used in order to avoid that. Whenever the view is larger than necessary, the additional space is distributed to the widgets. This happens while considering a stretch factor on each widget, which allows to influence how the distribution takes place. Specify the optional `hstretch` and/or `vstretch` properties for that, like this:

```
HorizontalStack(
    View('Label', label="Foo", hstretch=2),
    View('Label', label="Bar", hstretch=5),
    # we can also override the auto sizing (but still stretching)
    View('Label', label="Baz", hstretch=1, width='30pt')
)
```

Since each layout behaves like a widget (in fact is a widget), stretch factors can also be assigned to them. This can be crucial for getting the desired result.

In some pieces of code one might find the usage of those stretch factors by a different name: `horizontalStretchAffinity` and `verticalStretchAffinity` do exactly the same with a more verbose name.

Layouts do not only stretch widgets, but they might also align them on the other axis by giving them the same height or width. The properties `strictHorizontalSizing` and `strictVerticalSizing` can be set to `True` for making a widget ultimately refuse any sizes larger that its native one even for alignment purposes.

7.3 Configuring the main view itself

After the introduction about how to specify a user interface body, this chapter shows some ways to configure the outer frame of that.

7.3.1 Header text

The application view has an optional first and second header text. Both are shown in the header of the main view. Set them like this:

```
class MyApplication(Application):

    def __init__(self):
        super().__init__()
        self.sethead1("My Application")
        self.sethead2("Foobar Baz")
```

7.3.2 Icons

The application icon can be specified like this:

```
class MyApplication(Application):

    def __init__(self):
        super().__init__()
        self.add_staticfile_location("/my/static/files/directory")
        self.icon = "someimage.png"
```

7.4 Sidebar and head control

In a similar way as described in *setbodyleft and setbodyright*, application code can also specify a sidebar and/or a small widget inside the header. See `yaji.app.Application.setsidebar()` and `yaji.app.Application.setheadcontrol()` for that.

7.5 Finding more information in the API references

This documentation text introduces each larger block of functionality in some depth, but it might not provide all details and all possibilities for each one; in fact it might not even mention all detail aspects of the common functionality, like listing all available widget classes. The same is also true for the sample applications. This kind of information are provided by the API references instead.

Later in this text, there is an API reference for the application code (i.e. backend) side, which lists the Python side of the Yaji API.

7.5.1 Clove

The foundation and implementation of widgets and user interfaces on the browser side is the *Clove* library. It implements what a widget is, how they play together, and what widget classes exist. There is a dedicated Clove documentation, which lists all that, including descriptions of all those widget classes and what properties they have.

It is located in `README.clove.pdf`.

Note: There is a common naming scheme in all APIs related to Yaji regarding non-public members. Unless there is a better way, they at least have a name starting with "_". Some of them are listed in the API references for special purposes, although they are non-public. In most cases it is okay to ignore them.

CHAPTER

EIGHT

DATA SOURCES AND DATA BINDINGS

Data sources are a high-level mechanism for data exchange (potentially) between application code side and browser side. It is the primary mechanism for showing non-static data in a user interface.

A data source is a special object (i.e. an instance of some particular classes) that keeps any kind of data and that can be used for a data binding somewhere inside a *view specification*. The data binding will lead to a connection between the data source and some property in that view specification, which keeps them in sync. This can be used in both directions: It updates the bound widget property with new data when the data source content changes, and it updates the data source content when the widget property changes (e.g. when the user typed some text in a field).

The following shows a simple example for a data source usage:

```
class MyApplication(Application):

    def __init__(self):
        super().__init__()
        ds = self.create_datastore()
        self.setbodyleft(View('EditBox', text=self.binddata(ds)))
```

See also *yaji.app.Application.create_datastore()* and *yaji.app.Application.binddata()*. The variable `ds` is a *yaji.datastore.DataStore* object. It can be used to set or get the text content of that `EditBox` in this case.

Note: It is possible to use one data source in more than one data binding.

8.1 Server and local data sources

There are two different types of data sources. A *yaji.datastore.DataStore* as created in the example above comes with an actual data storage on application code side. Data bindings will keep this storage in sync with the user interface state, so application code can read from it and change it. Another type is *yaji.guibase.BrowserSideDatasource*. Those data sources can be created by *yaji.app.Application.create_clientlocal_datastore()* instead, and used in the same way as in the example above.

The latter type of data sources (also called a *local data source*, contrary to a *server data source*) are not connected to the application code side. They will not exchange any data between multiple browser windows presenting the same application either. Each view (i.e. each browser window showing the application) has an own, isolated storage for it. Due to that, a single binding like in the example above has limited usage, but having more than one binding for the same local data source might be usable.

Note: Local data sources are a tradeoff. Their obvious disadvantage is the decoupled way in which they keep their data, outside of the range of the application code. Their first duty is to help keeping the user interface in a consistent state. Server data sources need to push their data to the server and to let the server push it to all the other bindings.

As mentioned earlier, all data exchange mechanisms between application code side and browser side are inherently indirect and deferred, i.e. a change on one side will eventually be reflected on the other side at a somewhat later time. A local data source reflects the other bindings in a much more direct way, which is very helpful in some situations.

Although application code is not able to make changes to local data sources, it can initialize the data source, like in the following:

```
with self.create_clientlocal_datastore() as ds:
    ds.setvalue("Foo")
self.setbodyleft(View('EditBox', text=self.binddata(ds)))
```

8.2 Data bindings

The examples above show the usage of *yaji.app.Application.binddata()* for binding an existing datasource to some widget properties. There is also a slightly different notation, which also can be found in some of the sample applications. It uses *yaji.app.Application.bindserver()* and *yaji.app.Application.bindlocal()* instead, typically with a string argument. They do a similar thing, but with a data source name. If there already is a data source with that name, it uses that, otherwise it creates a new one (a server or local one). Note that also *yaji.app.Application.create_datastore()* and *yaji.app.Application.create_clientlocal_datastore()* allow to assign a name to the new data source.

There are some configuration switches that control the behavior of a data binding. See the parameters of *yaji.app.Application.binddata()*.

8.2.1 Direction

The data direction of a binding specifies if data updates happen either only from the datasource to the widget property, or only the other way around, or in both directions. See the `data_direction` parameter of *yaji.app.Application.binddata()*, the `bindings_text.py` sample application, and the following example:

```
self.setbodyleft(View('EditBox', text=self.binddata(ds, yaji.BindDirection.ToWidget)))
```

Note: The default is bidirectional flow as this is the only useful thing for typical cases.

8.2.2 Converters

A data binding can be equipped with a converter that translates between the data representation inside the datasource and some view-specific representation. This is an advanced feature that also involves including browser side code, which is the subject of a much later chapter. See the `convertername` parameter of *yaji.app.Application.binddata()* and the `bindings_converters.py` sample application.

8.3 Data sources beyond bindings

There are also other usages of data sources that do not involve data bindings at all. Most of them use data sources in a much broader manner, storing more complex data structures instead of just single values in them.

A data source allows to have that structures by the following framework:

- A data source *node* can store an arbitrary value, and
  - potentially has a two dimensional grid of child nodes.
- Each data source has a root node.

A data binding only makes use of a small part of that flexibility since it only works with the root cell and no structure of child nodes. But in a broader sense, data sources can also keep data in a list, table or tree structure by that. This is used e.g. in `"TreeView"` widgets by directly assigning a data source to the *datasource* property (see the `treeview.py` and `tableview.py` sample applications) and for keeping menus (see the `menus.py` sample application).

CHAPTER

NINE

EVENT BINDINGS

Data bindings allow to transfer data, including a way to let application code listen for changes e.g. made by the user. But they do not help for some other kinds of user interaction. A button for example, which can be clicked by the user, is not helpful with a data binding. Instead, they provide some events that application code can bind to.

9.1 bindevent

The available events depend on the widget class, so they can be found in the *Clove API reference* for builtin ones. For a button, an event binding could be like this:

```
self.setbodyleft(View('Button', OnClicked=self.bindevent(BackendFunction('something
↪'))))
```

Implementing the handler function involves adding a new method to the *yaji.app.Application* implementation and annotating it in a particular way. Later chapters show more details, but for the example above, a handler could be implemented like this:

```
class MyApplication(Application):
    ...

    @RequestHandler.for_urls(auto=True)
    def something(self):
        spectacularly_succeed(fail_on_mondays=True)
```

9.2 BackendFunction

A *yaji.guibase.BackendFunction* as used in the event binding above is a reference to a function/method on the application code side.

Later chapters will also introduce other ways to use them.

They also allow to pass some additional arguments to the function.

9.3 BrowserFunction

Similar to *yaji.guibase.BackendFunction* there is also *yaji.guibase.BrowserFunction*. They bind to a function on browser side. This is an advanced feature that typically also involves including browser side code, which is the subject of a much later chapter.

See the `eventbindings.py` sample application for both kinds of function references and ignore the `_do_` prepending the method name for the moment.

9.4 Event data

Whenever a browser side event occurs, it comes with some event data. Its content depends on the event, and for many events there are no useful event data at all. See *yaji.request.Request.uieventdata* and the `eventbindings.py` sample application for (a particularly useless example of) how to get this data.

CHAPTER

TEN

REQUEST HANDLERS

The *bindevent example above* makes use of a *yaji.guibase.BackendFunction* and defines a handler method for it, that is annotated to be a *request handler*. Request handlers can be called from browser side, for executing event handlers, but also for other things explained in later chapters.

See also *yaji.reqhandler.RequestHandler* for more details.

Note: Request handler calls do not get enqueued in a main loop, but all of them execute in parallel, in separate threads.

10.1 The *_do_* prefix

The easiest way to define a handler method is to just give it the same name as referred to in the *yaji.guibase.BackendFunction*. There is another naming scheme however, which is recommended to use instead, that means prepending `_do_` to the method name (leaving the name in the event binding untouched). The method name would then be `_do_something` instead.

10.2 URL mapping

URL mapping is the lookup process that finds the right request handler for the URL of a request from browser side. For the *bindevent example above* it would find the `something` (or `_do_something`) method for the request URL `/something`.

Instead of this default naming scheme, it is also possible to associate a request handler with other URL patterns. See the parameters of *yaji.reqhandler.RequestHandler.for_urls()* for more information. This is optional and mostly for naming aesthetics.

10.3 Downstream execution

By default, request handlers are executed while leaving the response stream open for the final response. Later chapters show how to work with such responses. For event handlers, however, responses do not have any meaning.

Whenever a request handler potentially runs for a longer time, this can be a problem. There is no precise threshold that defines 'a longer time', as it depends on browsers and network infrastructure, but it is safe to consider anything larger than a few seconds as long. In such cases, request handlers should be annotated as *downstream* ones. They directly return an empty acknowledgment to the browser and execute afterwards. Those handlers are not able to do

some things, e.g. to return a response to the browser. The latter is not a restriction for event handlers anyway (in any other case, a custom indirect way to return the response has to be established).

Annotating a request handler by *yaji.reqhandler.RequestHandler.run_downstream()* makes a request handler a downstream one, like in the following example:

```
@RequestHandler.for_urls(auto=True)
@RequestHandler.run_downstream()
def something(self):
    go_on_summer_vacation()
```

CHAPTER

ELEVEN

MENUS

A Yaji menu is a list of actions and submenus, grouped by separators, very similar to menus in desktop user interfaces. An easy way to provide a menu to a user is to have a main view menu, like in the following example:

```
class MyApplication(Application):
    ...

    def __init__(self):
        super().__init__()
        self.setactions([Action("Do Foo", BackendFunction("foo")),
                         Action("Do Bar", BrowserFunction("bar"))])
```

This example shows a menu with two items, both with a label text and a reference to some function to execute (like for *bindevent*). Instead of a list, it is also possible to use a data source, which allows dynamic additions, removals and changes. The `menus.py` sample application shows a larger example.

11.1 Actions

A menu action is an item in a menu that can be executed. The *menu example above* specifies some of them. See *yaji.guibase.Action* for more details.

11.2 Submenus

A submenu is another list of menu items that the user can pop up on demand. See *yaji.guibase.Submenu* for more details.

11.3 Separators

A separator is a visible highlight like a line for visually grouping items. See *yaji.guibase.Separator* for more details.

CHAPTER

TWELVE

USER FEEDBACK

The user feedback subsystem provides simple ways for simple dialog based interactions with the user. Those dialogs can show a message text to the user and have some buttons or ask the user to enter a text, and some other things. See also *yaji.userfeedback.UserFeedbackController*, `yaji.Application.userfeedback`, the `eventbindings.py` sample application and the following example:

```
class MyApplication(Application):
    ...

    @RequestHandler.for_urls(auto=True)
    @RequestHandler.run_downstream()
    def _do_something(self):
        self.userfeedback.messagedialog("Hallo Wereld.")
```

Note: User feedback cannot be used in request handlers that are not *downstream* ones, because that would inevitably violate the rule of avoiding long execution times with them.

CHAPTER

THIRTEEN

DIALOGS

Dialogs are a way of interacting with the user in pseudo popups. Compared to *user feedback*, dialogs are more powerful and flexible. Dialogs should only be used if the additional flexibility is needed, since it implies more coding.

Showing a dialog needs a *yaji.gui.View* specification *as above*. At first, *yaji.app.Application.create_dialog()* creates a new *yaji.gui.Dialog* for a view specification, then yaji.Dialog.show() shows it to the user.

See the `dialog.py` sample application.

CHAPTER

FOURTEEN

INTERNATIONALIZATION

There are many aspects of internationalization, many of them handled automatically by the language infrastructures, if program code is straight-forward enough.

14.1 Translations

One of the other aspects is translating user interface texts, so the user interface can be shown in the native language of the user.

Application code can register text translations during initialization by an internal key and a translation in some languages for each text, like in the following:

```
class MyApplication(Application):

    def __init__(self):
        super().__init__()
        self.add_translations('Welcome', en="Welcome!", it="Benvenuto!", nl="Welkom!")
```

Other application code can refer to those texts like in the following example:

```
self.setbodyleft(View(label=TrStr('Welcome')))
```

If the user language is not available, the English text is shown.

CHAPTER

FIFTEEN

APPLICATION STOPPING

The application shutdown procedure is a bit complicated internally, since there must be a stable and defined interaction between application code side and browser side in different situations. The application code can stop the application, the browser side can trigger it, and the user sometimes can just close all browser windows.

See *yaji.app.Application.stop()* for details.

CHAPTER

SIXTEEN

FORCEFULLY KEEPING A BROWSER VIEW OPEN

The default behavior of Yaji is to try keeping one view open. When the user closes all browser windows, it will open a new one, so the user is forced to correctly stop the application. The `stop_implicitly_when_browser_closed` parameter of *yaji.app.Application* allows to control this behavior.

It is possible to temporarily override that choice for the execution of a code block with the *yaji.app.Application.do_stop_implicitly_when_browser_closed()* and *yaji.app.Application.dont_stop_implicitly_when_browser_closed()* context managers.

CHAPTER

SEVENTEEN

CLOSED NOTIFICATION

If the default behavior of *forcefully keeping a browser view open* is enabled, a re-opened view will show a notification. This notification explains why the browser opened again (and allows to close the application in some situations). The `show_browser_closed_notification` parameter of *`yaji.app.Application`* allows to control this behavior.

CHAPTER

EIGHTEEN

SHUTDOWN DIALOG

In some situations, the application stops without closing all opened views. This usually happens whenever a view is opened in a usual web browser, since application shutdown will not close those browsers or tabs inside them. The default behavior is to show a shutdown dialog in those views, which disables the complete user interface and shows an generic information text. The `skip_shutdown_dialog` parameter of `yaji.Application` allows to control this behavior.

CHAPTER

NINETEEN

BROWSER SIDE

A Yaji application is allowed to include browser side resources like JavaScripts or style sheets for more flexibility in user interface implementations and for more direct interactions with the user, avoiding expensive data transfer with the application code side.

This leaves the world of Python and the application code context. It is an advanced feature, which should be avoided if a similar implementation without browser side code is possible.

19.1 Adding browser side resources

The simplest way to include browser side resources is to locate a JavaScript in `__foo.js` and/or a style sheet in `__foo.css` in the same directory as the main application `foo.py`.

More resources can be add with *yaji.app.Application.add_clientscript()* and *yaji.app.Application.add_clientstyle()*.

Directories for additional static second-level resources (e.g. images used in style sheets) can be added with *yaji.app.Application.add_staticfile_location()*.

19.2 Custom widgets

One good reason for including a JavaScript is to implement a custom widget class, i.e. a class that directly or indirectly extends `clove.Widget`. Details about this are beyond the scope of this text. See the `customwidget_simple.py` sample application and the *Clove API reference*.

19.3 Yaji browser side API

There is a rich Yaji API on browser side, which provides mechanisms for data exchange with the application code side, user interface features, and more. Some of its basic features are introduced in the next chapters. There is a dedicated API reference for it below, where you can look up details about the functions mentioned in the following.

19.3.1 yaji.ready

An included JavaScript gets executed at an early step of initialization. The infrastructure is not ready for many things at this moment. yaji.ready calls a function once the initialization is finished. See the `yaji_ready.py` sample application.

19.3.2 Ajax

Ajax requests are a low-level way to communicate with the application code side. They can be done by the `XMLHttpRequest` web API or any other ones. There is also yaji.ajax(), which can be used for that.

On application code side, *request handlers* will answer Ajax requests. Non *downstream* request handlers can return an arbitrary serializable data structure that is passed back as response to browser side. Any Ajax API allows to retrieve and process those responses.

Request parameters

Ajax requests can carry parameters in the usual ways, either encoded in the url or in a POST body, which is usually abstracted by the Ajax API. On application code side, a request handler will receive those parameters by the arguments of the method call. The following example makes an Ajax request in browser code:

```
yaji.ajax({url: 'something', data: {x: 13, y: 37}});
```

The request handler can be defined like in the following example for receiving the parameters:

```
@RequestHandler.for_urls(auto=True)
def _do_something(self, x, y):
    do_something_spectacular_in(x, y)
```

Note: The request handler is required to take exactly the arguments it gets. Violating that is potentially a critical error. Since request handlers are just Python functions, their signatures can specify default values for arguments and variable arguments like `**kwargs` in order to solve this problem.

Request parameter types

By default, request arguments are passed to the request handlers as strings. Type annotations lead to automatic conversions, so the following example is working code:

```
@RequestHandler.for_urls(auto=True)
def _do_something(self, x: int, y: int):
    return 100 * x + y
```

See the `requesthandler.py` sample application. See *`yaji.app.Application.add_requestparam_type_converter()`* for custom types.

Request objects

Inside a request handler it is possible to access some extended aspects of the request, like the request url, and to participate on building the response in low-level ways, by a request object. See *yaji.request.Request*, *yaji.app.Application.current_request()* and the `request.py` sample application.

19.3.3 Client events

Client events are a low-level mechanism that allows the application code side to broadcast events to the open view(s). They should not to be confused with *event bindings*, which are a high-level mechanism working in the other direction.

A client event has a name that specifies the kind of event, and some arbitrary event data. Application code can trigger a client event like in the following example:

```
self.triggerevent('somethinghappened', id=1337)
```

See also *yaji.app.Application.triggerevent()*.

Browser side code can register a handler during initialization like in the following example:

```
yaji.addEventHandler('somethinghappened', (data) => {
    doSomethingWith(data.id);
});
```

19.3.4 Appconfig

The appconfig mechanism has some similarities to *data sources* but is specialized for global values. It stores some arbitrary global application state data by a key name, backed on application code side, and provides an easy browser side API for consuming them. Application code can set a value like in the following example:

```
self.appconfig.setconfig('headertext', "Schildpad")
```

See also *yaji.appconfig.AppConfig* and *yaji.app.Application.appconfig*.

Browser side code can consume those values by registering a handler during initialization like in the following example:

```
yaji.appconfig.addHandler('headertext', (value) => {
    setMyFunnyHeader("Een " + value + "je");
});
```

Note: There is no risk in registering an appconfig handler after it was set by application code. When a handler is added, if there already is a value, the handler will be executed once with that value.

19.3.5 Curtains

Curtains are a way to work around the problems of the *decoupling between application code side and browser side*. In some defined situation, curtains allow to disable the user interface or to shield it in some other ways, until all transitions between application code side and browser sides are finished. This stops the user from interventing with the user interface while it is not guaranteed to be in a consistent state.

Curtains are implemented on browser side. There is a default yaji.Curtain implementation, which can also be subclassed for blocking the user interface in different ways (e.g disabling parts of it).

See the `curtain.py` sample application.

19.4 JSON serialization

JSON serialization is used heavily for data transfer from application code side to browser side (it is also used for some transfers from browser side) and in this direction can also be extended by serialization routines for custom types. Those extensions have an application side code (serialization) part and a browser side (deserialization) code part.

The former is realized by adding a *_to_simple_repr_()* method to the custom type, like in the following example:

```
class MyObject:
    ...

    def _to_simple_repr_(self):
        return 'MyObject', {'foo': self.__foo}
```

This method returns a deserializer name and a dictionary of arbitrary data that are serializable. The deserializer name can be equal to the class name on application code side, but it actually specifies the class on browser code side that handles the deserialization. Such a class can be like in the following example:

```
class MyObject {
    ...

    static _from_simple_repr_(data) {
        return new MyObject(data.foo, 42);
    }

}
```

CHAPTER

TWENTY

MIDDLEWARE

Middleware are a plugin mechanism for affecting internal request handling in a pluggable and potentially custom way. See *yaji.core.Middleware*, *yaji.app.Application.add_middleware()* and the `middleware.py` sample application.

There are also some methods of *yaji.app.Application*, often beginning with `on`, that can be overridden for controlling other aspects of low-level behavior.

CHAPTER

TWENTYONE

APPLICATION TAKEOVER

Application takeover allows multiple Yaji applications to share a browser view by taking it over from a parent application on startup and releasing it when stopping.

This is a feature for exotic scenarios. It can avoid new windows or browser tabs for each new application instance when many applications run in an interleaved way.

See the `takeover.py` sample application.

CHAPTER

TWENTYTWO

STORING APPLICATIONS STATICALLY

For exotic scenarios, and only if the application implementation is compatible to it, *yaji.app.Application.storeasstaticapplication()* can be used for storing the entire application to a directory of static files. This can again be opened with a usual web browser, but also without any application backend running.

CHAPTER

TWENTYTHREE

LOGGING

There is a Python `logging` logger used by Yaji for writing messages for diagnostics. It can also be used by application code. See *yaji.core.Log* and *yaji.core.log*.

Note: Setting the environment variable `YAJI_LOGDEBUG` to `1` causes a Yaji application to write verbose debug log messages to the terminal.

TODO colon js colon func refs

CHAPTER

TWENTYFOUR

BACKEND SIDE API REFERENCE

24.1 yaji package

24.1.1 Submodules

24.1.2 yaji.app module

class yaji.app.**Application**(*parentid=None*, *returntoparent=True*, ***, *show_browser_closed_notification=False*, *stop_implicitly_when_browser_closed=False*, *skip_shutdown_dialog=False*, *browser_hook_heartbeat_threshold=120*, *head1=None*, *head2=None*, *icon=None*, *mainview_icon=None*)

Bases: object

Base class for a Yaji web application.

Parameters

- **parentid** (*Optional[str]*) – Optional id of a parent Application, which e.g. can be redirected back to after exit.
- **returntoparent** (*bool*) – If to return back to the *parentid* application after exit in the user's browser.
- **show_browser_closed_notification** (*bool*) – If to show a useful notification (including a way to close the application) after the browser was closed by the user and restarted.
- **stop_implicitly_when_browser_closed** (*bool*) – If to consider the application as intendedly stopped when the user has closed the browser instead of opening a new one.
- **skip_shutdown_dialog** (*bool*) – If not to show a ui-blocking 'application stopped' dialog on ui shutdown.
- **browser_hook_heartbeat_threshold** (*int*) – The time window within the backend excepts a heartbeat from the browser side before it tries to open a new browser window.
- **head1** (*Optional[str]*) – The 1st level header text.
- **head2** (*Optional[str]*) – The 2nd level header text.
- **icon** (*Optional[str]*) – The window icon.
- **mainview_icon** (*Optional[yaji.guibase.Icon]*) – The icon of the mainview header.

_Application__ComputeContentMiddleware
alias of `Application.__ComputeContentMiddleware`

_Application__PostParamsMiddleware
alias of `Application.__PostParamsMiddleware`

_Application__SetStopImplicitlyWhenBrowserClosed
alias of `Application.__SetStopImplicitlyWhenBrowserClosed`

_Application__YajiHTTPRequestHandler
alias of `Application.__YajiHTTPRequestHandler`

_Application__YajiTCPServer
alias of `Application.__YajiTCPServer`

_Application__create_datastore_helper(*name*, *createfct*)
Creates and adds a new (serverside or browserside) datastore.

Parameters

- **name** (*Optional[str]*) – The datastore name (or `None`).
- **createfct** (*Callable[[str], Any]*) – Callback function for actually creating a new datastore.

Return type Any

static _Application__get_content(*content*, *frompath*)
Returns a bytes content with an additional ending newline.

Parameters

- **content** (*Optional[AnyStr]*) – The content to return.
- **frompath** (*Optional[str]*) – The file path to fetch the content from.

Return type bytes

_Application__get_datastore_cell(*datastorename*, *id*)
Helper for datastore accesses.

Parameters

- **datastorename** (*str*) –
- **id** (*Optional[int]*) –

_Application__get_datastore_helper(*name*, *create_if_not_exist*, *creatfct*)
Returns a (serverside or browserside) datastore by name if one exists, otherwise see parameters.

Parameters

- **name** (*str*) – The datastore name.
- **create_if_not_exist** (*bool*) – If to create a fresh one (otherwise returns `None`) if no one exists with that name.
- **creatfct** (*Callable[[], Any]*) – Callback function for actually creating a new datastore.

Return type Optional[Any]

_Application__get_requesthandler_for_url(*urlpath*, *app*)
Returns the registered request handler function for a request url path.

Parameters

- **urlpath** (*str*) – The path segment of the request url.
- **app** (yaji.app.Application) –

Return type Tuple[Optional[Callable], Dict[str, str]]

_browser_was_reopened()
Notifies that the browser needed to be reopened. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Return type None

_callhandler(*handler*, *request*)
Calls a request handler for a request and returns its result, but includes some error handling. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Parameters

- **handler** (*Callable*) – The request handler.
- **request** (yaji.request.Request) – The request.

Return type Tuple[Optional[Any], Optional[Exception], Optional[str]]

_do__yj_answer_userfeedback(**_)

_do__yj_application_stop()

_do__yj_application_tryclosebrowser(*checks=True*)

Parameters **checks** (*bool*) –

_do__yj_clientscript()

_do__yj_datastore_info(*name*)

_do__yj_datastore_pull(*name*, *id=None*)

Parameters **id** (*Optional[int]*) –

_do__yj_datastore_push(*name*, *value*, *id=None*)

Parameters **id** (*Optional[int]*) –

_do__yj_dialogs_close(*dialogid*)

Parameters **dialogid** (*int*) –

_do__yj_dialogs_list()

_do__yj_initscript()

_do__yj_lasteventid()

_do__yj_listfs(*path*, *only_dirs*, *show_hidden*)

_do__yj_pullevent(*lastid*)

Parameters **lastid** (*int*) –

_do__yj_returntakeover()

_do__yj_setappconfigvalue(*key*, *value*)

_do__yj_takeover(*url*)

_do__yj_unhandled_client_error(*error=''*, *src=''*, *line=''*)

_findhandler(*request*)
Returns the handler function for a request. *Override this method in custom subclasses or leave the default implementation.*

Parameters **request** (yaji.request.Request) – The request to serve.

Return type Optional[Callable]

_get_requestparam_type_converter(*paramtype*)
Returns a type converter for a request param. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Parameters **paramtype** – The python type of the parameter to convert.

_get_rootpagecontent()
Returns the main page content. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Return type bytes

_idcnt = 0

_json_make_serializable(*o*)
Makes some special objects json serializable. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly. Override this method in custom subclasses or leave the default implementation.*

Parameters **o** (*Any*) – The object to serialize.

Return type Any

property _middlewares
Returns the sorted list of middlewares. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Return type List[‘Middleware’]

_openbrowser(*url*)
Opens a browser view for ui rendering. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Parameters **url** (*str*) – The url to open.

Return type None

_openbrowser_pyhtmlviewargs()
Returns additional arguments for pyhtmlview. *Override this method in custom subclasses or leave the default implementation.*

Return type Dict[str, Any]

_staticfile_path_to_abspath(*relpath*)
Searches a file in the pool of static files locations by relative path and returns the absolute path if such a file exists. See *add_staticfile_location()*.

Parameters **relpath** (*str*) – The relative static file path.

Return type str

_tryclosebrowser(*withchecks=True*)
Tries to close the browser window.

Note: This will not work in all situations, e.g. typically not if the app is opened in the system browser. Your application should handle that situation gracefully (e.g. by the default app shutdown notification). *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Parameters **withchecks** (*bool*) – If set, it makes some checks before, e.g. if implicit stopping is enabled.

Return type bool

add_clientscript(*content=None*, ***, *frompath=None*)
Adds application client javascript.

Call this from your constructor.

Parameters

- **content** (*Optional[AnyStr]*) – The content to return.
- **frompath** (*Optional[str]*) – The file path to fetch the content from.

Return type None

add_clientstyle(*content=None*, ***, *frompath=None*)
Adds additional css application styles.

Call this from your constructor.

Parameters

- **content** (*Optional[AnyStr]*) – The content to return.
- **frompath** (*Optional[str]*) – The file path to fetch the content from.

Return type None

add_middleware(*middleware*)
Adds a `Middleware`.

Parameters **middleware** (`yaji.core.Middleware`) – The middleware to add.

Return type None

add_requestparam_type_converter(*paramtype*, *converter*)
Adds a custom type converter for request params.

Parameters

- **paramtype** (*type*) – The python type.
- **converter** (*Callable[[str], Any]*) – A function that converts a string coming from the web client to a `paramtype`.

Return type None

add_staticfile_location(*path*)
Adds a local file path as an additional location for reading static files from. Whenever static files are requested (e.g. by the browser requesting `/static/...`), Yaji tries to find it in one of those locations (in insertion order).

Parameters **path** (*str*) – The root path to add as static files source.

Return type None

add_translations(*stringname*, ***texts*)
Adds translations in some languages for a string.

Parameters

- **stringname** (*str*) – The text stringname, as used in `TrStr`.
- **texts** (*str*) – Different translations by two-letter language code.

Return type None

property appconfig
The application AppConfig.

Return type *AppConfig*

binddata(*datasource*, *data_direction=None*, *, *convertername=None*)
Declares binding a property of a `View` to an application code side or local browser side data source. This is similar to *`bindserver()`* and *`bindlocal()`*, but detects automatically if this is a server or local binding. See also *`create_datastore()`*, *`create_clientlocal_datastore()`*.

Parameters

- **datasource** (*Union[str, yaji.datastore.DataStore, yaji.guibase.BrowserSideDatasource]*) – The data source name or DataStore/BrowserSideDatasource. This data source must already exist.
- **data_direction** (*Optional[str]*) – The data flow direction. See `BindDirection`.
- **convertername** (*Optional[str]*) – The converter name.

bindevent(*function*, *, *curtain=None*)
Declares binding an event property of a `View` to a application code side or local browser side action. This triggers the execution of that action whenever that event is triggered.

Parameters

- **function** (`yaji.guibase.AbstractFunction`) – The function to bind to this event.
- **curtain** (*Optional[Union[bool, str]]*) – A curtain expression (pointing to a yaji.Curtain in browser side javascript), or `True` for the default curtain (which might not be a good solution).

bindlocal(*datasource*, *data_direction=None*, *, *convertername=None*)
Declares binding a property of a `View` to a local browser side data source.

Note: You should use *`binddata()`* instead, unless you want to use it with a data source name that might not exist yet.

Parameters

- **datasource** (*Union[str, yaji.guibase.BrowserSideDatasource]*) – The data source name or BrowserSideDatasource.
- **data_direction** (*Optional[str]*) – The data flow direction. See `BindDirection`.
- **convertername** (*Optional[str]*) – The converter name.

bindserver(*datasource*, *data_direction=None*, *, *convertername=None*)
Declares binding a property of a `View` to a application code side data source.

Note: You typically should use *`binddata()`* instead.

Parameters

- **datasource** (*Union[str, yaji.datastore.DataStore]*) – The data source name or DataStore.
- **data_direction** (*Optional[str]*) – The data flow direction. See BindDirection.
- **convertername** (*Optional[str]*) – The converter name.

property bodyleftviewactionlabel
The menu action label for the left main body widget.

Return type Optional[TrStrOrStrTyping]

This property is also settable.

property bodyrightviewactionlabel
The menu action label for the right main body widget.

Return type Optional[TrStrOrStrTyping]

This property is also settable.

property browser_hook_heartbeat_threshold
The time window within the backend excepts a heartbeat from the browser side before it tries to open a new browser window.

Return type int

create_clientlocal_datastore(*name=None*)
Creates a new browserside data source and returns it. See also *get_clientlocal_datastore()*, *bindlocal()*.

Parameters **name** (*Optional[str]*) – The datastore name (e.g. used in yaji.getClientlocalDatastore() on browser side). Default is a random name.

Return type *yaji.guibase.BrowserSideDatasource*

create_datastore(*name=None*, *, *curtain=None*)
Creates a new (real serverside) DataStore and returns it. See also *get_datastore()*, *bindserver()*.

Parameters

- **name** (*Optional[str]*) – The datastore name (e.g. used in yaji.getDatastore() on browser side). Default is a random name.
- **curtain** (*Optional[Union[bool, str]]*) – A curtain expression (pointing to a yaji.Curtain in browser side javascript), or True for the default curtain (which might not be a good solution).

Return type *yaji.datastore.DataStore*

create_dialog(*cfg*, ***showcfg*)
Creates a dialog.

Parameters

- **cfg** (*yaji.gui.View*) – The view specification.
- **showcfg** (*Any*) – Additional configuration for dialog presentation.

Return type *yaji.gui.Dialog*

property current_request
Returns the Request associated to the current request (in a request handler).

Return type *Request*

do_stop_implicitly_when_browser_closed(*value=True*)
Use such instances via `with` for temporarily enabling implicit application stop when browser closed.

Parameters **value** (*bool*) – If `False`, does the reverse this, same as *dont_stop_implicitly_when_browser_closed()*.

Return type AbstractContextManager

dont_stop_implicitly_when_browser_closed()
Use such instances via `with` for temporarily disabling implicit application stop when browser closed.

Return type AbstractContextManager

enable_authentication(*, *authfct*, *realm=None*)
Enables authentication by a custom authenticator function.

Parameters

- **authfct** (*Callable[[str, str], bool]*) – The authenticator function: f(username, password) -> `True` iff authenticated.
- **realm** (*Optional[str]*) – The http auth realm name.

Return type None

get_clientlocal_datastore(*name*, *create_if_not_exist=True*)
Returns a browserside data source by name. If one exists, it returns that, otherwise see parameters. See also *create_clientlocal_datastore()*, *bindlocal()*.

Parameters

- **name** (*str*) – The datastore name.
- **create_if_not_exist** (*bool*) – If to create a new one if no one exists with that name yet (otherwise returns `None`).

Return type Optional[*yaji.guibase.BrowserSideDatasource*]

get_datastore(*name*, *create_if_not_exist=True*)
Returns a (real serverside) DataStore by name. If one exists, it returns that, otherwise see parameters. See also *create_datastore()*, *bindserver()*.

Parameters

- **name** (*str*) – The datastore name.
- **create_if_not_exist** (*bool*) – If to create a new one if no one exists with that name yet (otherwise returns `None`).

Return type Optional[*yaji.datastore.DataStore*]

get_translations(*stringname*, ***texts*)
Returns translations in all available languages for a string. See also *add_translations()*.

Parameters

- **stringname** (*str*) – The text stringname.
- **texts** (*Any*) –

Return type Dict[str, str]

get_translations_stringnames()
Returns all available stringnames for translations. See also *add_translations()*.

Return type List[str]

property head1
The 1st level header text.

Return type Optional[TrStrOrStrTyping]

This property is also settable.

property head2
The 2nd level header text.

Return type Optional[TrStrOrStrTyping]

This property is also settable.

property icon
The application icon.

You may set it to a path to a png file (relative to `/static` as the web browser sees it). See *add_staticfile_location()*.

Return type Optional[*Icon*]

This property is also settable.

property id
The application id.

Return type str

property isrunning
Checks if the application is currently running (started and no stop triggered yet).

Return type bool

property isstaticapplication
Returns if this application is currently running as static one. See *storeasstaticapplication()*.

Return type bool

property mainview_icon
The mainview header icon.

The application icon is used if no mainview icon is set.

You may set it to a path to a png file (relative to `/static` as the web browser sees it). See *add_staticfile_location()*.

Return type Optional[*Icon*]

This property is also settable.

onbrowserreopened()
Reacts on the fact that the browser was opened again. *Override this method in custom subclasses or leave the default implementation.*

Return type None

oninitialize()
Initializes the application. *Override this method in custom subclasses or leave the default implementation.*

This is called during application startup, so later than `__init__` (or never if the app does not start).

Note: You should just override `__init__` instead if possible!

Return type None

onopenbrowsererror()
Reacts on errors when opening the browser. *Override this method in custom subclasses or leave the default implementation.*

Return type None

onopenbrowserinformationoutput(*kind*, *message*)
Reacts on information output (e.g. by printing it to stdout) while opening the browser. *Override this method in custom subclasses or leave the default implementation.*

Parameters

- **kind** (*int*) – The internal code of the message.
- **message** (*str*) – The message text.

Return type None

onprocessrequesterror(*request*, *error*)
Reacts on process request errors, e.g. by logging. *Override this method in custom subclasses or leave the default implementation.*

Parameters

- **request** (*str*) – The path part of the request url.
- **error** (*Exception*) – The Exception.

Return type None

onunhandledclienterror(*error*, *src*, *line*)
Reacts on unhandled client errors. Returning `True` makes the application shut down. *Override this method in custom subclasses or leave the default implementation.*

Parameters

- **error** (*str*) – An error description (might be empty).
- **src** (*str*) – The source file where the error occurred (might be empty).
- **line** (*str*) – The line in the source file where the error occurred (might be empty).

Return type bool

property parentid
The id of the parent application (if any).

Return type str

property returntoparent
If to return to the parent application after exit.

Return type bool

setactions(*actions*)
Sets the main menu actions.

Parameters **actions** (*List[yaji.guibase.AbstractAction]*) – List or datasource of actions.

Return type None

setbodyleft(*cfg*, *switchto=True*)
Sets the left main body widget.

Parameters

- **cfg** (*Optional[yaji.gui.View]*) – A view specification.
- **switchto** (*bool*) – If to switch to this view (if in single-side mode).

Return type None

setbodyright(*cfg*, *switchto=True*)
Sets the right main body widget.

Parameters

- **cfg** (*Optional[yaji.gui.View]*) – A view specification.
- **switchto** (*bool*) – If to switch to this view (if in single-side mode).

Return type None

setheadcontrol(*cfg*)
Sets the head control widget.

Parameters **cfg** (*Optional[yaji.gui.View]*) – A view specification.

Return type None

setsidebar(*cfg*)
Sets the sidebar widget.

Parameters **cfg** (*Optional[yaji.gui.View]*) – A view specification.

Return type None

setsplitterposition(*v*)
Sets the main body widgets splitter position.

Parameters **v** (*float*) – The splitter position (from left `0.0` to right `1.0`).

Return type None

property show_browser_closed_notification
If to show a useful notification (including a way to close the application) after the browser was closed by the user and restarted.

Return type bool

This property is also settable.

property showonly
Switches to show only the left or right main body widget. Is `0` for left only, `1` for right only, `None` for both.

Return type Optional[int]

This property is also settable.

property skip_shutdown_dialog
If not to show a ui-blocking 'application stopped' dialog on ui shutdown.

Return type bool

This property is also settable.

start()
Starts the application.

Note: If your instance was already started and stopped, you have to create a new instance for another run.

Return type None

stop()

Stops the application.

- It requests the parent application's returntakeover procedure in some situations (see later),
- then invokes stopping the user interface (triggers *_yj_stopui*),
  - which on browser side leads to 'uiShutdown' and back redirection to parent application in some situations
- then shuts down the http server.
  - so the _yj_stopui event might never actually arrive, depending on timings (not an actual problem).

A uiShutdown disables the user interface by triggering the *OnUiShutdown* event (but only one time; further calls are ignored). Application code can register custom handlers to this event, but it does at least this:

- visually disables the user interface and shows a 'shut down' text if *skip_shutdown_dialog==False* (the default)
- if it happened due to the user tried to close a PyHtmlView and *show_browser_closed_notification==True* (not the default): makes a '_yj_application_tryclosebrowser' request, so PyHtmlView support can close the window or parent applications can take control back

The browser side also runs *uiShutdown* e.g. when

- *_yj_pullevent* requests fail (assumption: the application code side has stopped), or
- after a *yaji.stop()* was called (see later).

A uiShutdown does not stop stuff like event polling, and will leave *yaji.isrunning==True*! The browser side *yaji.stop()* does the following:

- requests '_yj_application_stop', which calls *Application.stop()* on application code side
  - when backend answered: running uiShutdown
  - with disabled user interface during the request if
    - *skip_shutdown_dialog==False* (the default) or
    - it was explicitly called this way

Browser side calls *yaji.stop()* e.g. in those situations:

- custom application code on browser side triggered it for application exit
- a '_yj_unhandled_client_error' occurred and the backend decided for shutdown (which is not the default)
- by PyHtmlView support when the user tries to close the window

There is a flag *yaji.isrunning* that is *True* if the backend application is not known to be stopped so far, but might even be *True* after a uiShutdown. It will become *False* when

- *_yj_pullevent* requests fail (assumption: the application code side has stopped), or
- the *_yj_stopui* (see above) event was received.

Return type None

property stop_implicitly_when_browser_closed
If to consider the application as intendedly stopped when the user has closed the browser instead of opening a new one.

Return type bool

This property is also settable.

storeasstaticapplication(*targetdir*)
Stores the application as static files.

Note: The application has to support that, since there are some pitfalls.

Parameters **targetdir** (*str*) – The target directory path.

Return type None

switch_to_bodyleft()
Switches to the left body view (if in single-side mode).

Return type None

switch_to_bodyright()
Switches to the right body view (if in single-side mode).

Return type None

property takenover
If the current application is taken over by another one and paused at the moment.

Return type bool

triggerevent(*name*, ***eventdata*)
Triggers an event on browser side.

Parameters

- **name** (*str*) – The event name (e.g. used in yaji.addEventHandler() on browser side).
- **eventdata** (*Any*) – Additional event data to pass.

Return type None

try_addclientresources_from_default_location(*fpath*)
Tries to add a style file and script file from a file naming convention, so you do not have to call *add_clientstyle()* and *add_clientscript()*.

If you pass `"/some/path/foo.py"`, it tries to load `"/some/path/__foo.css"` (and `.js`).

Parameters **fpath** (*str*) – Full path of your module file. You could use `__file__` for that.

Return type None

property url
The application root url for browser access.

Return type str

property urlmap
The application's url to request handler map.

Return type List[Tuple['Pattern', str, Callable]]

property userfeedback
User feedback handler (for message boxes, ...).

Note: You must not use it in usual request handlers, but only in downstream ones! See `RequestHandler.run_downstream()`.

Return type *UserFeedbackController*

waituntilstopped()
Blocks execution while the application is running.

Return type None

property wasreopened
If the browser needed to be opened again.

Return type bool

24.1.3 yaji.appconfig module

class yaji.appconfig.**AppConfig**(*app*)
Bases: object

Application configuration storage.

This can store arbitrary configuration items, which then become available on browser side as well.

Parameters **app** (Application) –

_AppConfig__markinitialized()
Marks the configuration as initialized (in order to handle requests a bit differently). *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Return type None

getconfig(*key*, *default=None*)
Returns a configuration value by key.

Parameters

- **key** (*str*) – The configuration key.
- **default** (*Optional[Any]*) – The default value (if no such key).

Return type Any

getconfigs(*_markinitialized=False*)
Returns the complete configuration. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Parameters **_markinitialized** (*bool*) – If to consider the configuration as initialized by this call.

Return type Dict[str, Any]

setconfig(*key*, *value*)
Sets a configuration value by key.

Parameters

- **key** (*str*) – The configuration key.
- **value** (*Any*) – The new value.

Return type None

24.1.4 yaji.auth module

class yaji.auth.**AuthMiddleware**(*, *realm*)
Bases: *yaji.core.Middleware*

Middleware for authentication.

Parameters **realm** (*str*) – The http auth realm name.

_authenticate_by_password(*username*, *password*)
This method authenticates a user by a password. *Override this method in custom subclasses.*

Returns *True* for successful authentication.

Parameters

- **username** (*str*) –
- **password** (*str*) –

Return type bool

_checkauth(*request*)

24.1.5 yaji.browser module

class yaji.browser.**BrowserHook**(*url*, *app*)
Bases: object

Helper for opening browsers. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Parameters

- **url** (*str*) – The url to open.
- **app** (Application) –

heartbeat()
Gives a browser heartbeat (so the application code side knows that the browser is still alive).

Return type None

start()
Starts the hook.

Return type None

stop()
Stops the hook.

Return type None

24.1.6 yaji.core module

class yaji.core.**Directories**
Bases: object

Some important paths.

base = '/tmp/anise.29031.7/yaji/yaji'

staticfiles = '/tmp/anise.29031.7/yaji/yaji/static'

class yaji.core.**Log**
Bases: object

Logging.

debug(*s*, **a*)
Logs a debug message.

Parameters **s** (*str*) –

Return type None

error(*s*, **a*)
Logs an error message.

Parameters **s** (*str*) –

Return type None

info(*s*, **a*)
Logs an info message.

Parameters **s** (*str*) –

Return type None

warning(*s*, **a*)
Logs a warning message.

Parameters **s** (*str*) –

Return type None

class yaji.core.**Middleware**
Bases: object

A middleware controls some internal stuff like request processing in a custom way.

See Application.add_middleware().

static request_processor(*index=0*)
Marks a function as a request processor. It takes a Request parameter and reads and/or modifies parts of it.

Parameters **index** (*int*) – The execution order index (low value: early execution).

yaji.core.**log: *yaji.core.Log* = <yaji.core.Log object>**
The logger.

24.1.7 yaji.datastore module

class yaji.datastore.**DataStore**(*name*, *app*, ***, *curtain=None*, *_editor_for=None*)
Bases: object

A clove-like datastore which then becomes available on browser side as well.

This is part of a particular piece of internal infrastructure and is typically not required to be used directly.

Parameters

- **name** (*str*) –
- **app** (Application) –
- **curtain** (*Optional[Union[bool, str]]*) –

class Node(*datastore*)
Bases: object

A location inside a datastore, having a value and potentially a 2-dimensional grid of child nodes.

This is part of a particular piece of internal infrastructure and is typically not required to be used directly.

Parameters **datastore** (DataStore) –

_to_simple_repr_()

appendcolumn(*values=None*)
Appends a column (like inserting to the end).
Parameters **values** (*Optional[List[Optional[Any]]]*) – An optional list of values to put into the new nodes.
Return type Optional[*yaji.datastore.DataStore.Node*]

appendrow(*values=None*)
Appends a row (like inserting to the end).
Parameters **values** (*Optional[List[Optional[Any]]]*) – An optional list of values to put into the new nodes.
Return type Optional[*yaji.datastore.DataStore.Node*]

getvalue()
Returns the value of this node.
Return type Optional[Any]

insertcolumn(*i*, *values=None*)
Inserts a column.
Parameters
- **i** (*int*) – The position to insert the column to.
- **values** (*Optional[List[Optional[Any]]]*) – An optional list of values to put into the new nodes.

Return type Optional[*yaji.datastore.DataStore.Node*]

insertrow(*i*, *values=None*)
Inserts a row.
Parameters
- **i** (*int*) – The position to insert the row to.
- **values** (*Optional[List[Optional[Any]]]*) – An optional list of values to put into the new nodes.

Return type Optional[*yaji.datastore.DataStore.Node*]

removecolumn(*i*)
Removes a column.

Parameters **i** (*int*) – The position to remove the column from.
Return type None

removerow(*i*)
Removes a row.
Parameters **i** (*int*) – The position to remove the row from.
Return type None

setvalue(*value=None*)
Sets the value of this node.
Parameters **value** (*Optional[Any]*) – The new value.
Return type None

property valuepointer
Returns a value pointer for this node.
Return type *DataStore.ValuePointer*

class ValuePointer(*irow*, *icol*, *node*)
Bases: object

A value pointer is kind of an address or reference to a node (i.e. a location) inside a datastore.

Note: The root node is usually referenced by None.

This is part of a particular piece of internal infrastructure and is typically not required to be used directly.

Parameters

- **irow** (*int*) –
- **icol** (*int*) –
- **node** (DataStore.Node) –

_DataStore__trigger_update_event(*node*)
Triggers an update event.

Parameters **node** (yaji.datastore.DataStore.Node) – The node that was updated.

Return type None

_lock = <unlocked _thread.lock object>

_nextid = 0

_nodedict = {}

_to_simple_repr_()

add_onchanged_handler(*handler*)
Add a custom handler for changes in this datastore.

Parameters **handler** (*Callable[[DataStore.Node], None]*) – A handler function.

Return type None

appendcolumn(*ptr=None*, *values=None*)
Appends a column (like inserting to the end).

Parameters

- **ptr** (*Optional[DataStore.ValuePointer]*) – The parent node (by ValuePointer).

- **values** (*Optional[List[Optional[Any]]]*) – An optional list of values to put into the new nodes.

Return type Optional[*Node*]

appendrow(*ptr=None*, *values=None*)
Appends a row (like inserting to the end).

Parameters

- **ptr** (*Optional[DataStore.ValuePointer]*) – The parent node (by ValuePointer).
- **values** (*Optional[List[Optional[Any]]]*) – An optional list of values to put into the new nodes.

Return type Optional[*Node*]

columncount(*ptr=None*)
Returns the column count of a node (by ValuePointer).

Parameters **ptr** (*Optional[DataStore.ValuePointer]*) – The value pointer of the node to query.

Return type int

property curtain
The datastore curtain expression (pointing to a yaji.Curtain in browser side javascript).

Return type Union[bool, Optional[str]]

static getnodebyid(*ajaxid*)
Returns the node for an ajax id.

Parameters **ajaxid**(*int*) –

Return type Optional[*yaji.datastore.DataStore.Node*]

getvalue(*ptr=None*)
Returns the value of a node (by ValuePointer).

Parameters **ptr** (*Optional[DataStore.ValuePointer]*) – The value pointer of the node to query.

Return type Optional[Any]

insertcolumn(*i*, *ptr=None*, *values=None*)
Inserts a column.

Parameters

- **i** (*int*) – The position to insert the column to.
- **ptr** (*Optional[DataStore.ValuePointer]*) – The parent node (by ValuePointer).
- **values** (*Optional[List[Optional[Any]]]*) – An optional list of values to put into the new nodes.

Return type Optional[*Node*]

insertrow(*i*, *ptr=None*, *values=None*)
Inserts a row.

Parameters

- **i** (*int*) – The position to insert the row to.

- **ptr** (*Optional[DataStore.ValuePointer]*) – The parent node (by ValuePointer).
- **values** (*Optional[List[Optional[Any]]]*) – An optional list of values to put into the new nodes.

Return type Optional[*Node*]

property name
The datastore name.

Return type str

node_to_idpath(*node*)
Returns a path of node ids from a given node up the hierarchy to root (as list).

Parameters **node** (yaji.datastore.DataStore.Node) – The node to query.

Return type List[int]

node_to_ptr(*node*)
Returns the ValuePointer for a Node.

Parameters **node** (yaji.datastore.DataStore.Node) – The node to query.

Return type Optional[*yaji.datastore.DataStore.ValuePointer*]

parent(*ptr=None*)
Returns the parent ValuePointer for a given ValuePointer.

Parameters **ptr** (*Optional[DataStore.ValuePointer]*) – The value pointer of the node to query.

Return type *yaji.datastore.DataStore.ValuePointer*

ptr_to_node(*ptr*)
Returns the Node for a ValuePointer.

Parameters **ptr** (*Optional[DataStore.ValuePointer]*) – The value pointer of the node to query.

Return type *yaji.datastore.DataStore.Node*

removecolumn(*i*, *ptr=None*)
Removes a column.

Parameters

- **i** (*int*) – The position to remove the column from.
- **ptr** (*Optional[DataStore.ValuePointer]*) – The parent node (by ValuePointer).

Return type None

removerow(*i*, *ptr=None*)
Removes a row.

Parameters

- **i** (*int*) – The position to remove the row from.
- **ptr** (*Optional[DataStore.ValuePointer]*) – The parent node (by ValuePointer).

Return type None

property rootnode
The root node.

Return type *DataStore.Node*

rowcount(*ptr=None*)
Returns the row count of a node (by ValuePointer).

Parameters **ptr** (*Optional[DataStore.ValuePointer]*) – The value pointer of the node to query.

Return type int

setvalue(*value=None*, *, *ptr=None*)
Sets the value of a node (by ValuePointer).

Parameters

- **value** (*Optional[Any]*) – The new value.
- **ptr** (*Optional[DataStore.ValuePointer]*) – The value pointer of the node to modify.

Return type None

valuepointer(*irow*, *icol*, *parent*)
Returns a ValuePointer by a parent ValuePointer and a row and a column index.

Parameters

- **irow** (*int*) – The row index.
- **icol** (*int*) – The column index.
- **parent** (*Optional[DataStore.ValuePointer]*) – The value pointer to the parent node.

Return type *yaji.datastore.DataStore.ValuePointer*

24.1.8 yaji.dialogcontroller module

class yaji.dialogcontroller.**DialogController**(*app*)
Bases: object

A controller for dialogs.

Parameters **app** (Application) –

close_dialog(*dialog*)
Closes a dialog.

Parameters **dialog** (Dialog) – The dialog to close.

get_dialog_by_id(*dialogid*)
Returns a dialog by id.

Parameters **dialogid** (*int*) – The dialog id.

Return type Optional[*Dialog*]

get_dialogs()
Returns a list of all open dialogs.

Return type List[*Dialog*]

show_dialog(*dialog*)
Shows a dialog.

Parameters **dialog** (Dialog) – The dialog to show.

24.1.9 yaji.gui module

class yaji.gui.**Dialog**(*ctl*, *cfg*, ***showcfg*)
Bases: object

A dialog.

This is part of a particular piece of internal infrastructure and is typically not required to be used directly.

Parameters

- **ctl** (DialogController) – The dialog controller.
- **cfg** (yaji.gui.View) – The dialog view configuration.
- **showcfg** (*Any*) – Additional configuration for dialog presentation.

_exec_closed_handlers()
Executes the closed handlers.

Return type None

_set_dialogid(*dialogid*)
Sets the dialog id.

Parameters **dialogid** (*Optional[int]*) –

Return type None

_to_simple_repr_()

add_closed_handler(*fct*)
Adds a closed handler. It gets executed when the dialog closes.

Parameters **fct** (*Callable[[], None]*) –

Return type None

close()
Closes the dialog.

Return type None

property dialogid
The dialog id. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Return type int

show()
Shows the dialog.

Return type None

class yaji.gui.**Grid**(**children*, ***b*)
Bases: *yaji.gui._AbstractContainer*

A grid view.

Parameters **children** (yaji.gui.View) – The child views.

class yaji.gui.**HorizontalStack**(**children*, ***b*)
Bases: *yaji.gui._AbstractContainer*

A horizontal stack view.

Parameters **children** (yaji.gui.View) – The child views.

class yaji.gui.**ScrollView**(*body*, ***b*)
Bases: *yaji.gui._AbstractContainer*

A scroll view.

Parameters **body** (*Optional[yaji.gui.View]*) – The child view.

class yaji.gui.**Spacer**(***b*)
Bases: *yaji.gui.View*

A spacer.

Parameters

- **clsname** – The clove widget class name. See clove documentation.
- **enabled** – If the widget is enabled (instead of being locked for user interaction).
- **visibility** – The visibility, as one of guibase.Visibility.
- **hstretch** – The horizontal stretch affinity factor.
- **strictHorizontalSizing** – If to not overfulfill horizontal sizing requirements even for alignment purposes.
- **vstretch** – The vertical stretch affinity factor.
- **strictVerticalSizing** – If to not overfulfill vertical sizing requirements even for alignment purposes.
- **styleClass** – Additional CSS classes. See also Application.add_clientstyle().
- **width** – A fixed width (specified as in css).
- **height** – A fixed height (specified as in css).
- **b** – Additional configuration. Depends on the widget class.

class yaji.gui.**VerticalStack**(**children*, ***b*)
Bases: *yaji.gui._AbstractContainer*

A vertical stack view.

Parameters **children** (yaji.gui.View) – The child views.

class yaji.gui.**View**(*clsname=None*, *enabled=None*, *visibility=None*, *hstretch=None*, *strictHorizontalSizing=None*, *vstretch=None*, *strictVerticalSizing=None*, *styleClass=None*, *width=None*, *height=None*, ***b*)
Bases: dict

A widget build configuration for user interface definition. Used e.g. as parameters in some Application methods. See also the sample applications.

Parameters

- **clsname** (*Optional[str]*) – The clove widget class name. See clove documentation.
- **enabled** (*Optional[bool]*) – If the widget is enabled (instead of being locked for user interaction).

- **visibility** (*Optional[str]*) – The visibility, as one of guibase.Visibility.
- **hstretch** (*Optional[float]*) – The horizontal stretch affinity factor.
- **strictHorizontalSizing** (*Optional[bool]*) – If to not overfulfill horizontal sizing requirements even for alignment purposes.
- **vstretch** (*Optional[float]*) – The vertical stretch affinity factor.
- **strictVerticalSizing** (*Optional[bool]*) – If to not overfulfill vertical sizing requirements even for alignment purposes.
- **styleClass** (*Optional[str]*) – Additional CSS classes. See also Application.add_clientstyle().
- **width** (*Optional[str]*) – A fixed width (specified as in css).
- **height** (*Optional[str]*) – A fixed height (specified as in css).
- **b** (*Any*) – Additional configuration. Depends on the widget class.

class _DataBinding(*datasource*, *data_direction*, *convertername*)

Bases: object

A data binding.

Parameters

- **datasource** (*Any*) – The internal data source definition.
- **data_direction** (*Optional[str]*) – The data direction. See BindDirection.
- **convertername** (*Optional[str]*) – Name of the converter class.

_to_simple_repr_()

class _EventBinding(*function*, *curtain*)

Bases: object

An event binding.

Parameters

- **function** (AbstractFunction) – The function to bind to an event.
- **curtain** (*Optional[Union[bool, str]]*) – The curtain expression.

_to_simple_repr_()

class yaji.gui.**Wrap**(**children*, ***b*)

Bases: *yaji.gui._AbstractContainer*

A wrap view.

Parameters **children** (yaji.gui.View) – The child views.

class yaji.gui.**_AbstractContainer**(*clsname*, *paramname*, *singlechild*, *default*, **children*, ***b*)

Bases: *yaji.gui.View*

Abstract container view.

Parameters

- **clsname** (*str*) – The clove classname.
- **paramname** (*str*) – The clove property name.
- **singlechild** (*bool*) – If this widget is the container for only a single child widget.

- **default** (*Any*) – The default value.
- **children** (yaji.gui.View) – Child views.

24.1.10 yaji.guibase module

class yaji.guibase.**AbstractAction**(*, *label*, *icon*, *checkable*, *checked*, *disabled*, *invisible*)
Bases: object

Abstract menu action (can be an actual action, submenus, separators, . . .).

Parameters

- **label** (*str*) – The label of this action.
- **icon** (*Optional[Icon]*) – The action icon.
- **checkable** (*bool*) – If this action is checkable (i.e. has a checkbox in menu).
- **checked** (*bool*) – If this action is checked.
- **disabled** (*bool*) – If this action is disabled (cannot be triggered).
- **invisible** (*bool*) – If this action is invisible.

_to_simple_repr_()

class yaji.guibase.**AbstractFunction**
Bases: object

An abstract callable function.

class yaji.guibase.**Action**(*label*, *function*, *, *icon=None*, *checkable=False*, *checked=False*, *disabled=False*, *invisible=False*)
Bases: *yaji.guibase.AbstractAction*

A menu action.

Parameters

- **label** (*str*) – The label of this action.
- **function** (yaji.guibase.AbstractFunction) – The function to call for this action.
- **icon** (*Optional[Icon]*) – The action icon.
- **checkable** (*bool*) – If this action is checkable (i.e. has a checkbox in menu).
- **checked** (*bool*) – If this action is checked.
- **disabled** (*bool*) – If this action is disabled (cannot be triggered).
- **invisible** (*bool*) – If this action is invisible.

_to_simple_repr_()

class yaji.guibase.**BackendFunction**(*url*, *funcargs=None*)
Bases: *yaji.guibase.AbstractFunction*

A callable application code side function.

Parameters

- **url** (*str*) – The url to trigger for executing this action.
- **funcargs** (*Optional[Dict[str, Any]]*) – Function arguments.

_to_simple_repr_()

class yaji.guibase.**BindDirection**
Bases: object

Data flow directions of data bindings.

Bidirectional = 'bidirectional'
Bidirectional data transfer.

ToDatasource = 'todatasource'
Data transfer only from widget to datasource.

ToWidget = 'towidget'
Data transfer only from datasource to widget.

class yaji.guibase.**BrowserFunction**(*funcname*, *funcargs=None*)
Bases: *yaji.guibase.AbstractFunction*

A callable browser side function.

Parameters

- **funcname** (*str*) – The name of the function to trigger on browser side.
- **funcargs** (*Optional[List[Any]]*) – Function arguments.

_to_simple_repr_()

class yaji.guibase.**BrowserSideDatasource**(*name*)
Bases: object

A browser side datasource. You can only edit it before pushing it to browser side, and only inside a `with` block.

Parameters **name** (*str*) –

_to_simple_repr_()

property name
The data source name.

Return type str

class yaji.guibase.**Icon**(*by_url=None*, *by_symbol=None*)
Bases: object

Icon.

Parameters

- **by_url** (*Optional[str]*) – A icon by image source url.
- **by_symbol** (*Optional[str]*) – A icon by symbol character.

_to_simple_repr_()

property src
The icon source.

Return type str

property srcfunc
The icon source function.

Return type str

class yaji.guibase.**Separator**
Bases: *yaji.guibase.AbstractAction*

A menu separator.

Parameters

- **label** – The label of this action.
- **icon** – The action icon.
- **checkable** – If this action is checkable (i.e. has a checkbox in menu).
- **checked** – If this action is checked.
- **disabled** – If this action is disabled (cannot be triggered).
- **invisible** – If this action is invisible.

_to_simple_repr_()

class yaji.guibase.**Submenu**(*label*, *, *icon=None*, *disabled=False*, *invisible=False*)
Bases: *yaji.guibase.AbstractAction*

A submenu of a menu.

Parameters

- **label** (*str*) – The label of this action.
- **icon** (*Optional[Icon]*) – The action icon.
- **disabled** (*bool*) – If this action is disabled (cannot be triggered).
- **invisible** (*bool*) – If this action is invisible.

_to_simple_repr_()

class yaji.guibase.**Visibility**
Bases: object

The visibility of a widget.

Invisible = 'clove_invisible'
Invisible but taking space.

InvisibleCollapsed = 'clove_invisiblecollapsed'
Invisible and collapsed.

Visible = 'clove_visible'
Visible.

24.1.11 yaji.i18n module

class yaji.i18n.**TrStr**(*stringname*, **args*)
Bases: object

An i18n-aware string for displaying on browser side. When returned in a json structure, the browser side transparently gets a translated version.

Parameters

- **stringname** (*str*) – The string name (as used in clove.i18n.addString() on browser side).
- **args** (*Any*) – String pieces for replacing "*foo*" patterns in the translation with.

_to_simple_repr_()

property args
String pieces for replacing "*foo*" patterns in the translation with.

Return type Dict[str, Any]

property stringname
The string name (as used for referencing the translations for one text).

Return type str

yaji.i18n.**TrStrOrStrTyping**
Type annotation for something that can be either a `str` or a *TrStr*.

alias of Union[str, yaji.i18n.TrStr]

24.1.12 yaji.pyhtmlview module

class yaji.pyhtmlview.**PyHtmlView**
Bases: object

This optional functionality allows to present the user interface of the application in an own bare window instead of the usual browser. It is used by default if not configured otherwise and if the system is compatible to it.

It requires PyQt to be available.

static is_system_compatible()
Tests if your system is compatible with PyHtmlView.

Return type bool

static openview(*, *show_url*, *icon*, *wndclassname*, *scriptedclosing*, *window_width=None*, *window_height=None*, *window_maximized=False*)
Opens a new view with some content.

Parameters

- **show_url** (*str*) – The url to show content from.
- **icon** (*str*) – The window icon.
- **wndclassname** (*str*) – The window class name.
- **scriptedclosing** (*Optional[str]*) – A javascript expression to execute for closing.
- **window_width** (*Optional[int]*) – Width of the window in pixels.
- **window_height** (*Optional[int]*) – Height of the window in pixels.
- **window_maximized** (*bool*) – If to maximize the window.

24.1.13 yaji.reqhandler module

class yaji.reqhandler.**RequestHandler**
Bases: object

Decorators for request handler functions, i.e. methods of an Application that handle http requests.

classmethod _RequestHandler__compile_re_for_urlpattern(*urlpattern*)
Returns a compiled `re` regexp for a url pattern string.

Parameters **urlpattern** (*str*) – The url pattern string to convert.

Return type Pattern

classmethod for_urls(**urlpatterns*, *auto=False*)
Makes a method a request handler for a particular url pattern. A url pattern is a string like `"foos/<fooid>/bars/"`, with `"<fooid>"` binding a parameter that you have to add to your handler function signature.

Parameters

- **urlpatterns** (*Union[str, re.Pattern]*) – The url patterns to bind the request handler to (strings or compiled `re` regexps).
- **auto** (*bool*) – If to construct the url pattern from the method name.

classmethod run_downstream()
Defines this request handler to run downstream. In this mode, the browser instantly gets an empty response and does not wait longer. This is recommended (and sometimes required; e.g. for user feedback) for operations that can take longer.

Note: This forbids to return any data from the handler, but it does not forbid to modify data sources, to set appconfig, and some other ways to transfer results.

---

classmethod with_param_type(*paramname*, *paramtype*)
Defines a parameter of a request handler to be of a particular data type. For custom types, either see `Application.add_requestparam_type_converter()` or make your type constructor accept a `(str)` call.

You usually should use type annotations on request handlers instead of this function!

Parameters

- **paramname** (*str*) – The parameter name. This is the name in the `...&name=...&...` part of the url and of the argument of the request handler.
- **paramtype** (*type*) – The python type of this parameter.

24.1.14 yaji.request module

class yaji.request.**Request**(*urlpath*, *header*, *application*)
Bases: object

A request from the client, including ways to give the response.

This is part of a particular piece of internal infrastructure and is typically not required to be used directly.

Parameters

- **urlpath** (*str*) – The complete request url path, including query string.
- **header** (*Dict[str, str]*) – The http request headers.
- **application** (Application) –

_Request__InRequest
alias of `Request.__InRequest`

_current_clientrequest_ctxvar_ = <ContextVar name='_current_clientrequest_ctxvar_' def

_in_request()
Returns a context manager for setting the current request. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Return type AbstractContextManager

_set_lparam(*key*, *value*)
Sets a request parameter value. Only infrastructure and middleware would use that. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Parameters

- **key** (*str*) – The parameter key.
- **value** (*List[str]*) – The parameter values (as list).

Return type None

_set_runs_downstream()
Sets this request to be handled downstream. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Return type None

property application
The application that got this request.

Return type *Application*

static current()
Returns the Request associated to the current request (in a request handler).

Return type *yaji.request.Request*

get_param(*key*, *defaultval=None*)
Returns the value of a request parameter.

Note: This drops values beyond the first one, so do not use it if you have lists.

Parameters

- **key** (*str*) – The parameter key.
- **defaultval** (*Optional[str]*) – The default value, if no such parameter exists.

Return type Optional[str]

get_param_as_list(*key*)
Returns the value of a request parameter as list (for having no or more than one value by parameter key).

Parameters **key** (*str*) – The parameter key.

Return type List[str]

get_response_header(*key*)
Returns the http response header entry as stored before with *set_response_header()*.

Parameters **key** (*str*) – The header key.

Return type str

get_response_header_keys()

Return type List[str]

property header
The request headers.

Return type Dict[str, str]

property lparams
The request parameters as list (for having no or more than one value by parameter key).

Return type Dict[str, List[str]]

property params
The request parameters in a flat structure (not as list, contrary to *lparams()*).

Note: This drops values beyond the first one, so do not use it if you have lists.

Return type Dict[str, str]

property response_body
The response body.

Return type Optional[bytes]

This property is also settable.

property response_errortext
The error text (if an error occurred).

Return type Optional[str]

This property is also settable.

property response_httpcode
The http response code.

Return type Optional[int]

This property is also settable.

property runs_downstream
If this request is handled downstream. See `RequestHandler.run_downstream()`.

Return type bool

property scratchpad
A dictionary for storing some custom things, typically by middleware.

Return type Dict[str, Any]

set_response_header(*key*, *value*)
Sets an http response header entry.

Parameters

- **key** (*str*) – The header key.
- **value** (*str*) – The header value.

Return type None

property skip_processing
If to skip further request processing. Used by middleware. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Return type bool

This property is also settable.

property uieventdata
The event data dictionary that contains details about an ui event coming from the browser.

This only makes sense in a request handler for an event binding (see `Application.bindevent()`) and is `None` otherwise.

Return type Dict[str, Optional[Any]]

property urlbarepath
The request url's path without query string, like `"/foo/bar"`.

Return type str

property urlpath
The request url's complete path, like `"/foo/bar?id=13&p=a"`.

Return type str

property urlpathquery
The request url's querystring, like `"id=13&p=a"`.

Return type str

24.1.15 yaji.userfeedback module

class yaji.userfeedback.**UserFeedbackController**(*app*)
Bases: object

A controller for user feedback operations.

Use methods inside it for communicating with the user during some operations. Such an object is automatically available during execution, there is no need to create new ones.

This is part of a particular piece of internal infrastructure and is typically not required to be used directly.

Parameters **app** (Application) –

_interpret_response(*request*)
Processes a userfeedback response from browser side.

Parameters **request** (yaji.request.Request) – A client request that contains a user-feedback answer.

Return type None

choicedialog(*question*, *choices*)
Shows a choice dialog to the user and returns the selected item (or `None` when cancelled).

Parameters

- **question** (*Union[str, yaji.i18n.TrStr]*) – The question text to show.
- **choices** (*List[Union[str, yaji.i18n.TrStr]]*) – The list of choices the user has to select from.

Return type Optional[int]

filesystemdialog(*fstype='file'*, *question=''*, *startpath='/'*)
Shows a file/directory selection dialog to the user and returns the path to the selected item (or `None`).

Parameters

- **fstype** (*str*) – The type of filesystem items to select.
- **question** (*Union[str, yaji.i18n.TrStr]*) – The question text to show.

- **startpath** (*str*) – The directory path to start in.

Return type Optional[str]

inputdialog(*question*, *defaultanswer=''*)
Shows an input dialog to the user and returns the entered text (or None when cancelled).

Parameters

- **question** (*Union[str, yaji.i18n.TrStr]*) – The question text to show.
- **defaultanswer** (*Union[str, yaji.i18n.TrStr]*) – The default answer text that is written to the text field when showing.

Return type Optional[str]

make_raw_request(*kind*, ***params*)
Makes a low-level userfeedback request.

Parameters

- **kind** (*str*) – The raw userfeedback kind name as known on browser side. You can register custom kind handlers on browser side with yaji.userfeedback.registerHandler().
- **params** (*Any*) – Some kind-specific parameters.

Return type Any

messagedialog(*message*, *buttons=None*)
Shows a message dialog to the user and returns the index of the selected button.

Parameters

- **message** (*Union[str, yaji.i18n.TrStr]*) – The message text to show.
- **buttons** (*Optional[List[Union[str, yaji.i18n.TrStr]]]*) – The buttons to offer.

Return type int

multilineinputdialog(*question*, *defaultanswer=''*)
Like *inputdialog()* but multi-line capable.

Parameters

- **question** (*Union[str, yaji.i18n.TrStr]*) – The question text to show.
- **defaultanswer** (*Union[str, yaji.i18n.TrStr]*) – The default answer text that is written to the text field when showing.

Return type Optional[str]

passworddialog(*question*, *defaultanswer=''*)
Shows a password dialog to the user and returns the password (or None).

Parameters

- **question** (*Union[str, yaji.i18n.TrStr]*) – The question text to show.
- **defaultanswer** (*Union[str, yaji.i18n.TrStr]*) – The default answer text that is written to the password field when showing.

Return type Optional[str]

24.1.16 Module contents

Web based user interfaces running in a local browser on top of clove.

Read about *yaji.app.Application* and its methods for more details.

The user interface is implemented on browser side by the clove library. See *Clove* for a documentation about available widgets and more details.

Also take a look at the sample apps in `_meta/sampleapps`. Each `.py` file there that does not begin with _ is one application. Some might also have a javascript and/or stylesheet file (named like `__foo.js`).

CHAPTER

TWENTYFIVE

BROWSER SIDE API REFERENCE

class TrStr(*stringname*, *args*)
An i18n-aware (translatable) string.

class YajiPopupMenuButton()
A special clove::PopupMenuButton connected to application's action execution framework. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

class YajiDataStore(*dscfg*)
A special clove::AjaxAsyncDatasource backed by a datasource on application code side. See also app.Application.create_datastore() and app.Application.bindserver().

class YajiConfiguration(*app*)
The application configuration. See also app.Application.appconfig().

YajiConfiguration.**addHandler**(*key*, *fct*, *runAlsoOnNeutralAssignments*)

Adds a configuration handler for a config key. It gets called whenever the value for this key
changes, and also directly if there already is something stored for that key at execution time.

param key The configuration key. param fct The handler function. param runAlsoOnNeutralAssignments If to run it also for assignments that have not changed the value.

YajiConfiguration.**getConfig**(*key*)
Returns a configuration value by key. param key The configuration key.

YajiConfiguration.**setConfig**(*key*, *value*)
Requests to set a configuration value for a key. Note: This happens asynchronously by a request to the backend. param key The configuration key. param value The new configuration value.

class YajiClientEvents(*app*)
Internal handling of client events. See YajiClient.addEventHandler(). *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

class AbstractFilesystemDialog(*config*)
Abstract filesystem picker dialog builder. See subclasses.

AbstractFilesystemDialog.**showDialog**()
Shows the dialog and calls either *onsuccess* or *oncancel*.

class OpenFileDialog(*config*)
Open file dialog builder.

class OpenDirectoryDialog(*config*)
Open directory dialog builder.

class Dialog(*dlgdata*)
A dialog (either modal or non-modal). *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

class YajiDialogs(*app*)
Manager for backend controlled dialogs.

class YajiUserFeedback(*app*)
User feedback manager. This is used for simple dialogs like message boxes, input boxes, and so on.

YajiUserFeedback.**registerHandler**(*kind*, *fct*)
Registers a handler for a custom kind of user feedback dialog. param kind The custom dialog kind name. param fct The custom handler function for this dialog kind.

YajiUserFeedback.**show**(*feedbackcfg*, *onanswered*)
Shows a feedback dialog by feedback configuration and calls a callback for the answer. param feedbackcfg Feedback configuration, incl. *kind* and some kind-dependent parameters. param onanswered The callback for the dialog answer.

class PyHtmlViewSupport(*app*)
Browser side support for PyHtmlView.

This is part of a particular piece of internal infrastructure and is typically not required to be used directly.

PyHtmlViewSupport.**_onwindowclose**()
Used by PyHtmlView when the user tries to close the window.

class YajiAjaxError(*url*, *httpcode*, *body*)
Backend errors in answering ajax requests.

class YajiAjaxRequest(*app*)
Handle object for an ajax request (triggered by YajiClient.ajax()).

class Curtain()
User interface curtains are used for disabling parts of the user interface during some backend operations.

The default implementation disables the entire user interface with a modal busy animation for that, so in many cases you have to subclass an own implementation.

Curtain.**close**()
Implements closing the curtain, so blocking some parts of the user interface in some ways. *Override this method in custom subclasses or leave the default implementation.* If you override it, also override open().

Curtain.**execute**(*fct*)
Executes a function inside the curtain, so the curtain gets closed before, and gets opened when processing is idle again.

Curtain.**open**()
Implements opening the curtain, so unblocking the user interface again. *Override this method in custom subclasses or leave the default implementation.* If you override it, also override close().

class YajiClient()
The application main object. Usually you can access it by just *yaji*.

YajiClient.**OnUiShutdown**
A clove::Event that is triggered on user interface shutdown. See also YajiClient.uiShutdown().

YajiClient.**addEventHandler**(*key*, *fct*)
Adds a handler function for a backend event. See app.Application.triggerevent(). param key The event key (or event 'name'). param fct The handler function. Receives the event data as first argument.

YajiClient.**ajax**(*config*)
Makes an ajax request.

It returns a Promise-like object, so it can be `await`ed.

config may contain the same as for *$.ajax*, and also: - *finished*: Callback that gets executed after either success or error. - *alsoAfterShutdown*: If to make that request even if the ui is already stopped. param config The request configuration.

YajiClient.**appconfig**
Instance of YajiConfiguration.

YajiClient.**getClientlocalDatastore**(*name*)
Returns a browser side datasource by name. See also app.Application.create_clientlocal_datastore() and app.Application.binddata(). param name The datasource name.

YajiClient.**getDatastore**(*name*)
Returns a application code side datasource by name. See also app.Application.create_datastore() and app.Application.binddata(). param name The datasource name.

YajiClient.**hasExternalCloseControls**
If this client view has dedicated external close controls (e.g. via PyHtmlView).

YajiClient.**isStaticApplication**
If the application is static. See app.Application.storeasstaticapplication().

YajiClient.**isrunning**
If the application is currently running.

YajiClient.**mainview**
Instance of clove::MainView.

YajiClient.**pyhtmlview**
Instance of PyHtmlViewSupport.

YajiClient.**ready**(*fct*)
Executes a function on startup once the yaji app is initialized. param fct The function to execute.

YajiClient.**stop**(*forceBlockUiDuringRequest*)
Completely stops the application, including application code side. See app.Application.stop() for explanation about stop procedure, and also YajiClient.uiShutdown(). param forceBlockUiDuringRequest If to force freezing the user interface by a modal panel (which automatically

happens only in some situations) during stop request.

YajiClient.**tryJsonStringify**(*o*)
Returns a json representation for an object, just containing primitive values, arrays and dicts. param o The object to serialize.

YajiClient.**uiShutdown**()
Shuts down the user interface (without killing the backend). See also YajiClient.stop().

YajiClient.**userfeedback**
Instance of YajiUserFeedback.

PYTHON MODULE INDEX

y

INDEX

Symbols

A

B

C

D

E

F

G

H

I

T

U

V

W

Y