Python AdminUI

Jun 25, 2022

Contents:

Write professional web interface with Python.

If you need a simple web interface and you don't want to mess around with HTML, CSS, React, Angular, Webpack or other fancy Javascript frontend stuff, this project is for you. Now you can write web pages, forms, charts and dashboards with only Python.

This library is good for: data projects, tools and scripts, small IT systems and management systems, Hacker or Hackathon projects. Basically if you need an interface for your system and you don't care much about customizing the style or performance for large traffic, consider this package.

This project is based on Flask and Ant Design Pro.

Features:

- Forms and detail pages
- Line and Bar Chart
- Create decent looking menus
- Data tables with pagination
- Adaptive to small screens and mobile devices
- No HTML, CSS, JS needed

CHAPTER 1

Installation and Quick Start

Install the package with pip:

```
pip install adminui
```

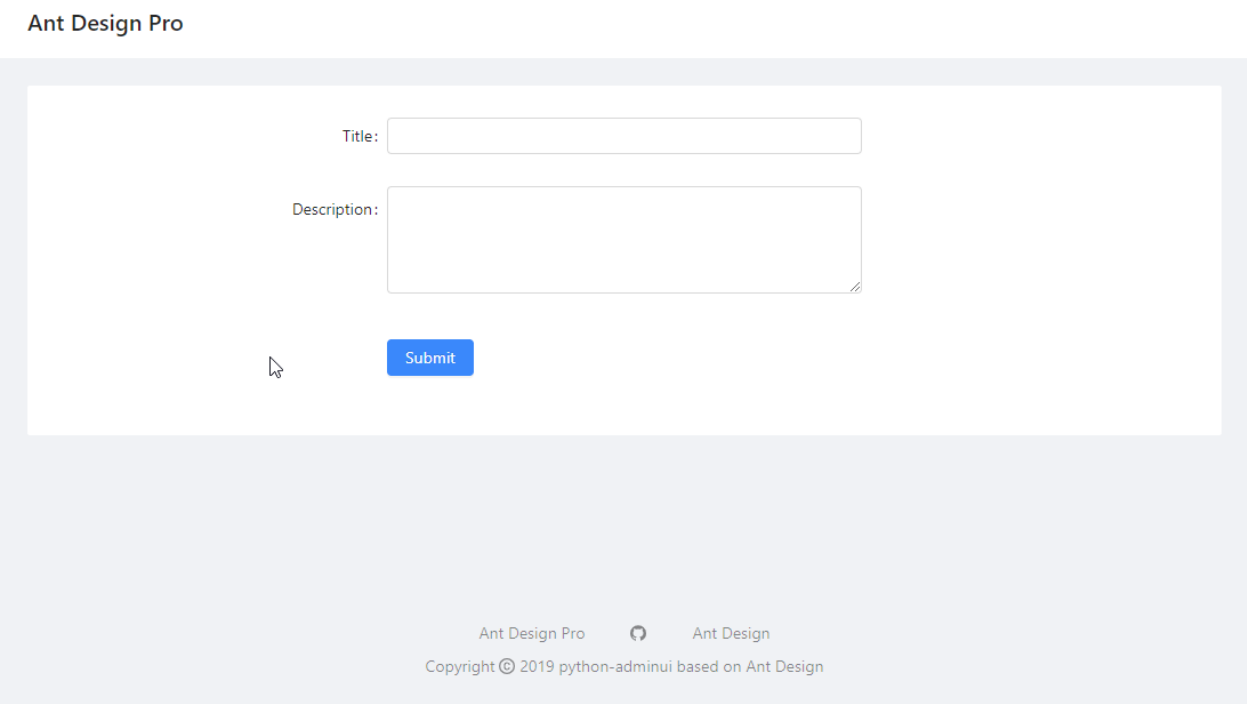
Now create a python file for your project, say example.py:

```
from adminui import *

app = AdminApp()

def on_submit(form_data):
    print(form_data)

@app.page('/', 'Form')
def form_page():
    return [
        Form(on_submit = on_submit, content = [
            TextField('Title', required_message="The title is required!"),
            TextArea('Description'),
            FormActions(content = [
                SubmitButton('Submit')
            ])
        ])
    ]

if __name__ == '__main__':
    app.run()
```

Run the python file from terminal:

```
python example_form.py
```

Now visit http://127.0.0.1:5000/ to see the index page. It should look like this:



1.1 Basic Concepts and Example Breakdown

First of all, you need to create a AdminApp object:

```
app = AdminApp()
```

Then you add pages to your app. Use the `@app.page` decorator, followed by a custom function (its name doesn't matter) that returns an array of the page elements shown when the user visit a certain url:

```
@app.page('/', 'Form')
def form_page():
    return [ ...put page contents here... ]
```

The `@app.page` decorator receives two argument. One for the url ('/' in this case), and the other for the title of the page("Form").

By decorating a function, you'll be able to return different page elements or data each time the user visit the page.

All page elements are Python objects. You may refer to other chapters of this documentation to know how to use them. We can see that the example above created a form, a text field, a text area and a submit button.

Finally, run the app using:

```
app.run()
```

1.2 Use FastAPI instead of Flask

Set `use_fastapi=True` when creating the `app`; and `prepare()` instead of `run` to expose the app to uvicorn.

The basic example will be:

```
from adminui import *

app = AdminApp(use_fastapi=True)

def on_submit(form_data):
    print(form_data)

@app.page('/', 'Form')
def form_page():
    return [
        Form(on_submit = on_submit, content = [
            TextField('Title', required_message="The title is required!"),
            TextArea('Description'),
            FormActions(content = [
                SubmitButton('Submit')
            ])
        ])
    ]

fastapi_app = app.prepare()
```

Then run from command line:

```
uvicorn example_fastapi:fastapi_app
```

CHAPTER 2

Creating a Form

To create a form, insert a `Form` object inside the list describing the page:

```
@app.page('/', 'Form')
def form_page():
    return [
        Form(on_submit = my_function,
               content = [ ... content of the form ... ])
    ]
```

Here, you may want to provide a function that will be called when the user submits the form, so you can process the data. The function may have any name, here `my_function` is used as an example. This function needs to be defined with an argument, which will be the form values as a dictionary:

```
def my_function(values):
    ... do things with values ...
```

In the content section of the form, you may add `TextField` and other form controls. See the next section for details

Finally, don't forget to add a submit button. Insert a `FormActions` object with `Submit` at the end of the form:

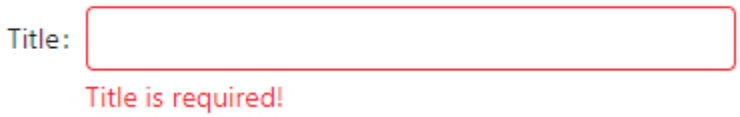
```
FormActions(content = [
    SubmitButton('Submit')
])
```

Now your form is created.

2.1 List of Form Controls

Here are a list of controls you may use in your form:

2.1.1 TextField




```
TextField('Title', required_message='Title is required!')
```

Set `password=True` to setup a password field:

```
TextField('Enter Password', password=True)
```

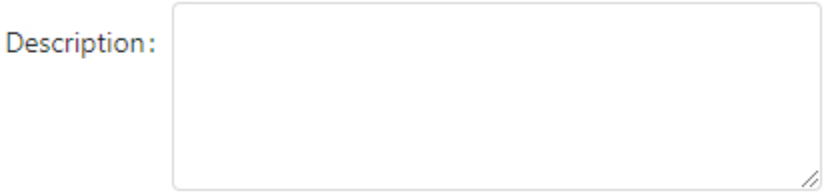
class adminui.**TextField**(*title*, *name=None*, *required_message=None*, *password=False*, *disabled=False*, *value=None*, *placeholder=None*, *on_change=None*, *id=None*)

Create a text field in the form.

Parameters

- **title** – the title of the field
- **name** – the key of the dictionary data passed to the callback when the form is submitted
- **required_message** – if other than None, the field will be required and an message will be shown when the field is not filled at form submission.
- **placeholder** – the text message shown when the field is not filled.
- **password** – set to True if it's a password box
- **disabled** – set to True to make the control disabled

2.1.2 TextArea




```
TextArea('Description')
```

class adminui.**TextArea**(*title*, *name=None*, *required_message=None*, *value=None*, *placeholder=None*, *disabled=False*, *on_change=None*, *id=None*)

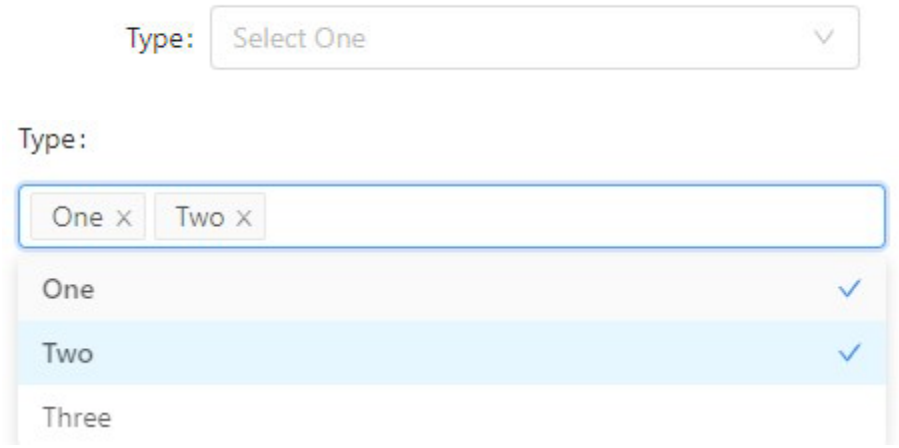
Create a text area object

Parameters

- **title** – the title of the field
- **name** – the key of the dictionary data passed to the callback when the form is submitted
- **required_message** – if other than None, the field will be required and an message will be shown when the field is not filled at form submission.
- **placeholder** – the text message shown when the field is not filled.

- **disabled** – set to True to make the control disabled

2.1.3 Select




```
SelectBox('Type', data=['One', 'Two', 'Three'], placeholder="Select One")
# use multiple=True to allow multiple selecting:
SelectBox('Type', data=['One', 'Two', 'Three'], placeholder="Select Many",␣
↪multiple=True)
# use tags=True to input tags - users can input arbitrary text as a tag:
SelectBox('Type', data=['One', 'Two', 'Three'], placeholder="Select Many", tags=True)
```

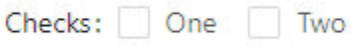
class adminui.**SelectBox**(*title*, *name=None*, *value=None*, *data=[]*, *placeholder=None*, *on_change=None*, *required_message=None*, *multiple=False*, *disabled=False*, *tags=False*, *id=None*)

Create a dropdown box for selecting in a list

Parameters

- **title** – the title of the field
- **name** – the key of the dictionary data passed to the callback when the form is submitted
- **required_message** – if other than None, the field will be required and an message will be shown when the field is not filled at form submission.
- **data** – the options in the select box. in format of a list of str or a list of [title, value] list e.g. [‘one’, ‘two’, ‘three’] or [[‘one’, 1], [‘two’, 2]], both accepted
- **placeholder** – the text message shown when the field is not filled.
- **disabled** – set to True to make the control disabled

2.1.4 Checkboxes




```
CheckboxGroup('Checks', data=['One', 'Two'])
```

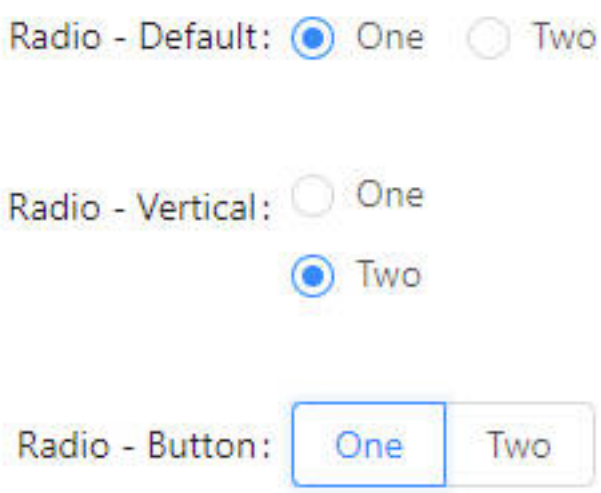
class adminui.**CheckboxGroup**(*title*, *name=None*, *data=[]*, *value=None*, *disabled=False*, *id=None*, *on_change=None*)

Create a group of checkbox for multi-selecting

Parameters

- **title** – the title of the field
- **name** – the key of the dictionary data passed to the callback when the form is submitted
- **data** – the title and value of individual checkboxes. in format of a list of str or a list of [title, value] e.g. ['one', 'two', 'three'] or [['one', 1], ['two', 2]], both accepted disabled: set to True to make the control disabled

2.1.5 Radios




```
RadioGroup('Radio - Default', data=['One', 'Two'], on_change=on_change),
RadioGroup('Radio - Vertical', data=[['One', 1], ['Two', 2]], on_change=on_change,␣
↪format='vertical'),
RadioGroup('Radio - Button', data=[['One', 1], ['Two', 2]], on_change=on_change,␣
↪format='button'),
```

class adminui.**RadioGroup**(*title*, *name=None*, *data=[]*, *value=None*, *format='default'*, *disabled=False*, *on_change=None*, *id=None*)

Create a group of radio buttons

Parameters

- **title** – the title of the field
- **name** – the key of the dictionary data passed to the callback when the form is submitted
- **value** – the default value of the selected radio
- **data** – the title and value of individual checkboxes. in format of a list of str or a list of [title, value] e.g. ['one', 'two', 'three'] or [['one', 1], ['two', 2]], both accepted
- **format** – default | button; how the radio box is displayed and arranged
- **disabled** – set to True to make the control disabled

2.1.6 DatePicker


```
DatePicker('Date')
```


```
DatePicker('Range', pick='range')
```

class adminui.**DatePicker**(*title*, *name=None*, *value=None*, *pick='date'*, *on_change=None*, *id=None*)

Create a date picker to pick a date or date range

Parameters

- **title** – the title of the field
- **name** – the key of the dictionary data passed to the callback when the form is submitted
- **pick** – 'date' | 'month' | 'week' | 'range'.

2.2 Callback when form item changes

If you want to do something, say update a part of the page when user select an item in the SelectBox or input text on the TextFields, you may add on_change handlers in your Python code:

```
TextField('Title', required_message='Title is required!', on_change=on_change),

# for the handler:
def on_change(value, values):
    print(value)
    print(values)
```

See chapter *Page Actions*. for details on what can handlers do.

The handler could be a function taking one or two parameters: the first will be the new value of the form item; the second one will be a dictionary of all the values in the form where the form item lives. This will be useful for example when your data shown in the page is filtered by a list of criterions.

2.3 Slider, Switch and other controls

These are not form controls. But they respond to on_change handlers.

2.3.1 Slider


```
Slider(on_change=on_change)
# this will render a slider, with 0~50 range
# and user can pick up a range, with an initial value 20~30
Slider(0, 50, range=True, value=[20,30])
```

2.3.2 Switch

```
Switch(on_change=on_change)
```

2.3.3 Checkbox

```
Checkbox(on_change=on_change)
```

CHAPTER 3

Page Actions

If you want to redirect the user to another page after form submission, you may return page actions in the `on_submit` callback of the form:

```
def on_detail():
    return NavigateTo('/detail')
```

Now AdminUI supports `NavigateTo` and `Notification` as page actions:

class adminui.**NavigateTo**(*url='/'*)
Page Action: navigate the user to another page

Parameters **url** – the url of the new page

class adminui.**Notification**(*title=''*, *text=None*, *type='default'*)
Send right-top corner notification to the user

Parameters

- **title** – the title of the notification
- **text** – the text body of the notification
- **type** – default | success | info | warning | error: type of the notification box

If you want to combine two actions together, return a list. Say you want to notify the user twice:

```
return [
    Notification('A Notification', 'the content of the notification'),
    Notification('A Notification', 'the content of the notification'),
]
```

Aside from form submission, you can also use page action on other elements like a button.

class adminui.**Button**(*title='Go'*, *style=None*, *icon=None*, *on_click=None*, *link_to=None*, *id=None*)
Create a general button on the page

Parameters

- **title** – the title shown on the button

- **style** – use 'primary' for the "primary button" (a big blue button)
- **icon** – the icon in front of the button title. For string values see https://ant.design/components/icon/
- **on_click** – a custom function will be called when the button is clicked

You may use page actions inside the on_click callback of the button.

3.1 Update page content in Page Actions

If you wish to change a part of the page in Page Actions, first identify the element with `id` attribute:

```
@app.page('/', 'Control Page')
def control_page():
    return [
        Card(content=[
            Button('Change Content', on_click=on_change_content),
            Button('Change Element', on_click=on_change_self),
        ]),
        Card(id='detail_card'),
        Card('Paragraph Card', [
            Paragraph('This is the original content', id='paragraph')
        ])
    ]
```

Then during a page action, you may return a `ReplaceElement` to replace an element with another:

```
def on_change_self():
    return ReplaceElement('paragraph', Paragraph('This element has been changed'))
```

Thus when users click the button "Change Element", a new paragraph will replace the old one.

You may also choose to update some attributes of an element. The following code changes the `content` value of "detail_card" element when the users click the "Change Content" button.:

```
return UpdateElement('detail_card', content=[
        DetailGroup('Refund Request', content=[
            DetailItem('Ordre No.', 1100000),
            DetailItem('Status', "Fetched"),
            DetailItem('Shipping No.', 1234567),
            DetailItem('Sub Order', 1135456)
        ]),
    ])
```

3.2 Use a timer

Use timer, to let your Python function run every some seconds. You can also return Page Actions in the timer's `on_fire` function:

```
@app.page('/', 'Detail Page')
def detail_page():
    return [
        Timer(on_fire=on_timer_fire, data='hello timer'),
```

(continues on next page)

(continued from previous page)

```
        ...
    ]
```

Timers can be set at any position of the page. To remove a timer, replace it with `ReplaceElement` page action.

See example_page_actions.py for the complete example. https://github.com/bigeyex/python-adminui/blob/master/python/example_page_actions.py

CHAPTER 4

Upload Files

File uploader can be a individual component or be a part of a form:

```
@app.page('/', 'form')
def form_page():
    return [
        Card('Upload Form', [
            Upload(on_data=on_upload)  # use Upload individually
        ]),

        Form(on_submit = on_submit, content = [
            Upload(name='upload', on_data=on_upload),     # embed uploads in a form
            FormActions(content = [
                SubmitButton('Submit')
            ])
        ])
    ]
```

The `on_data` handler will be called if a file is uploaded:

```
def on_upload(file):
    print("=== file name will be: ===")
    print(file['file_name'])
    print('=== the file is stored in: ===')
    print(app.uploaded_file_location(file))
```

Use `app.uploaded_file_location(file)` to get the absolute path of the uploaded file.

When you gave a `name` to the Upload Component, you can also retrive the file when the form is submitted:

```
def on_submit(form_data):
    print(app.uploaded_file_location(form_data['upload'])) # e.g. the upload component
↪'s name is 'upload'
```

By default, the file will be stored in /upload of the folder containing the starter Python file, but you can change it by passing an `upload_folder` when creating the AdminUI App:

```
app = AdminApp(upload_folder='Your custom folder')
```

CHAPTER 5

Creating a Menu

To create a menu for your project, use the `set_menu` function of the `AdminApp` object:

```
app.set_menu([ ... a list of menu items ...])
```

the `set_menu` method takes an array of `MenuItem` objects:

class adminui.**MenuItem**(*name*, *url=''*, *icon=None*, *auth_needed=None*, *children=[]*)

Represents a menu item

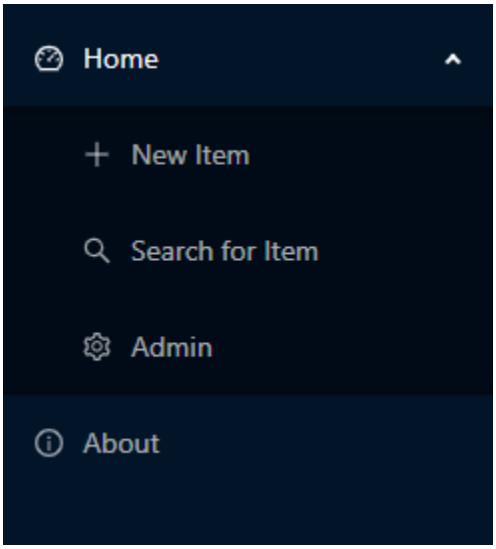
Parameters

- **name** (*str*) – the title of the menu
- **url** (*str, optional*) – the url the menu will navigate to. Defaults to ''.
- **icon** (*str, optional*) – the icon of the menu. See https://ant.design/components/icon/. Defaults to None.
- **auth_needed** (*str, optional*) – the permission needed for user to access this page. e.g. 'user' or 'admin'
- **children** (*list, optional*) – set this if the menu has a sub-menu. Defaults to [].

You may nest `MenuItem` to create sub-menus. Here's a complete example:

```
app.set_menu(
    [
        MenuItem('Home', '/', icon="dashboard", children=[
            MenuItem('New Item', '/new', icon="plus"),
            MenuItem('Search for Item', '/search', icon="search"),
            MenuItem('Admin', '/admin', icon="setting")
        ]),
        MenuItem('About', '/about', icon="info-circle")
    ]
)
```

It looks like this:


Home
New Item
Search for Item
Admin
About

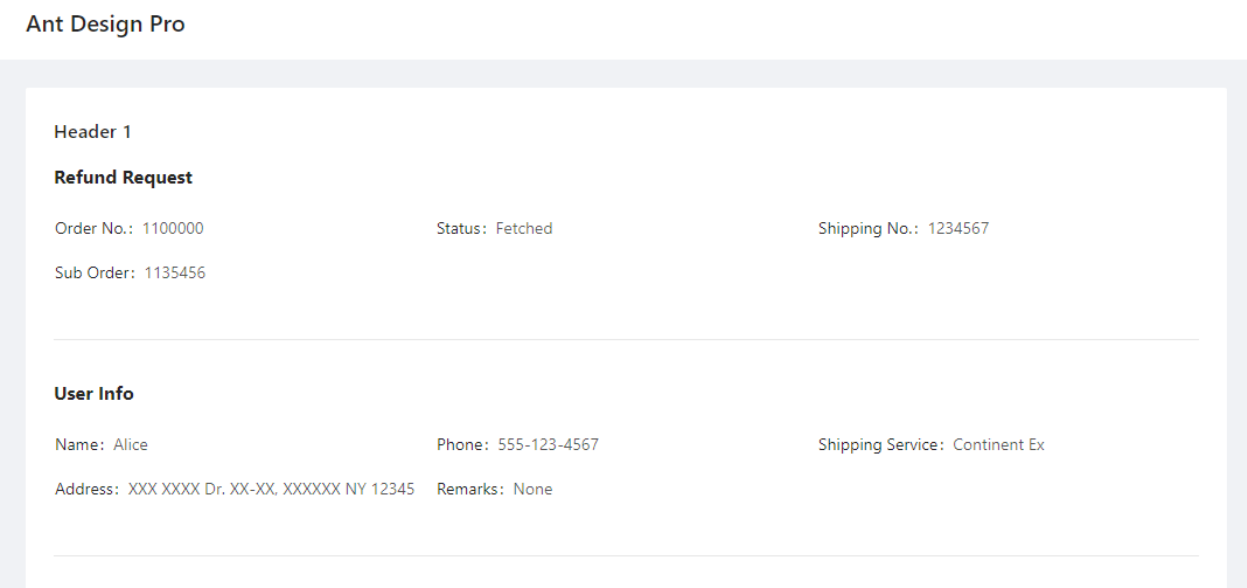
CHAPTER 6

Layout and Creating a Detail Page

When you want to show complex information on the page, you may use `Card`, `Header`, `DetailGroup`, `DetailItem` and `Divider` to help you format the page:

```
@app.page('/detail', 'Detail Page')
def detail_page():
    return [
        Card(content=[
            Header('Header of the record', 1),
            DetailGroup('Refund Request', content=[
                DetailItem('Order No.', 1100000),
                DetailItem('Status', "Fetched"),
                DetailItem('Shipping No.', 1234567),
                DetailItem('Sub Order', 1135456)
            ]),
            Divider(),
            DetailGroup('User Info', content=[
                DetailItem('Name', "Alice"),
                DetailItem('Phone', "555-123-4567"),
                DetailItem('Shipping Service', 'Continent Ex'),
                DetailItem('Address', 'XXX XXXX Dr. XX-XX, XXXXXX NY 12345'),
                DetailItem('Remarks', "None")
            ]),
        ])
    ]
```

It looks like this:



class adminui.**DetailGroup**(*title=''*, *content=None*, *bordered=False*, *column=3*, *size=False*, *layout='horizontal'*, *id=None*)

The container for DetailItem, used to display a record with multiple fields

Parameters

- **title** – the title of the detail group
- **content** – a list of DetailItem
- **bordered** – show a bordered description list, see https://3x.ant.design/components/descriptions/
- **size** – default | middle | small - the size of the table when in bordered mode
- **column** – number of columns shown in the list
- **layout** – horizontal | vertical s

class adminui.**DetailItem**(*title=''*, *value=''*, *id=None*)

A little piece of text with a title and a value, used to display a field in a record

Parameters

- **title** – the title of the field
- **value** – the value of the field

If you are working with a dashboard, `Row`, `Column`, `Statistic` may come in handy. `ChartCard` can make a neat little block to hold your numbers and charts. See example:

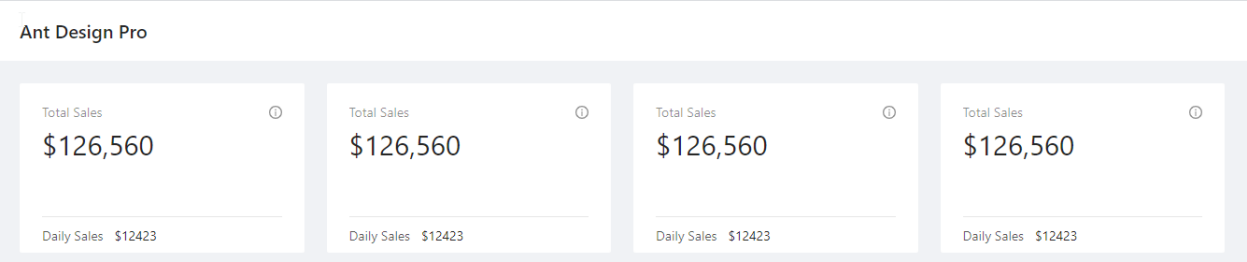
```
@app.page('/dashboard', 'Dashboard')
def dashboard_page():
return [
    Row([
        Column([
            ChartCard('Total Sales', '$126,560', 'The total sales number of xxx',␣
↪height=50,
                  footer=[Statistic('Daily Sales', '$12423', inline=True)])
        ]),
```

(continues on next page)

(continued from previous page)

```
        Column([
            ChartCard('Total Sales', '$126,560', 'The total sales number of xxx',␣
↪height=50,
                footer=[Statistic('Daily Sales', '$12423', inline=True)])
        ]),
        Column([
            ChartCard('Total Sales', '$126,560', 'The total sales number of xxx',␣
↪height=50,
                footer=[Statistic('Daily Sales', '$12423', inline=True)])
        ]),
        Column([
            ChartCard('Total Sales', '$126,560', 'The total sales number of xxx',␣
↪height=50,
                footer=[Statistic('Daily Sales', '$12423', inline=True)])
        ]),
    ])
]
```

It creates a page like this:



If you just want to display some text, use `Paragraph`. You may set the color of the Paragraph:

```
Paragraph("The text of Paragraph", color="red")
```

If you wish to format rich text or other complex content, you may use RawHTML Element. Beware this is dangerous because if you pass unfiltered user text (e.g. from a piece of user inputted text stored in the database), this user text may contain dangerous code that may run on the client's computer:

```
RawHTML('a raw <font color="red">HTML</font>')
```

Here's the list of layout-related classes:

class adminui.**Card**(*title=None*, *content=None*, *id=None*)

A white-boxed container to hold content in sections

Parameters

- **title** – the title of the card container
- **content** – list of page elements inside the container

class adminui.**Header**(*text=''*, *level=4*, *id=None*)

Display a header text on the screen

Parameters

- **text** – text body of the header
- **level** – the header level. level=1 means a first level header(h1)

class adminui.**Paragraph**(*text=''*, *id=None*, *color=None*)
Display a paragraph of text

class adminui.**Row**(*content=None*, *id=None*)
Display a row with multiple Columns for layout purpose

the width of the row will be automatically calculated by len(content). e.g. a row with four columns will make a 4-column layout. It is also adaptive in small screens and mobile devices.

Parameters **content** – a list of Column objects

class adminui.**Column**(*content=None*, *size=1*, *id=None*)
Column in the Row for multi-column layout

Parameters

- **content** – a list of page elements
- **size** – the "weight" of the column width. for example, 2 columns with both size=1 will have the same width; but a column with size=2 and one with size=1 will make a 2:1 layout.

class adminui.**ChartCard**(*title=None*, *value=None*, *tooltip=None*, *footer=None*, *content=None*, *height=46*, *id=None*)
A card container with values, tooltips and a footer. Mostly used in dashboards

Parameters

- **title** – the container title
- **value** – (str), the big text shown on the card
- **tooltip** – the text shown when the user hover on the tooltip icon
- **footer** – list of page elements shown on the footer of the card
- **content** – list of page elements shown as the content of the card
- **height** – the height of the card, to make it looks consistant across columns

class adminui.**Statistic**(*title=''*, *value=0*, *show_trend=False*, *inline=False*, *id=None*)
A piece of text for showing a statistic number, may include a little trend arrow (up or down)

Parameters

- **title** – the title of the statistic number
- **value** – the number itself
- **show_trend** – if set True, a upper arrow will appear for positive numbers, and a down for negative numbers
- **inline** – if set True, the title and the value will be in the same line and the font size will be smaller

CHAPTER 7

Page with a Parameter

Sometimes, you need to make the page respond to url parameters. For example, you wish:

```
http://yourdomain.com/article/3
```

shows the third article in the database. In this case, you register the page as such:

```
@app.page('/article', 'Article')
def form_page(param):
    return [ ... the content of the page ... ]
```

Then, when users come with url like `/article/<param>`, the `param` part will be passed as the first parameter of the handling function.

If you the page has multiple parameters in the url like `/page_name?param_a=value_a&param_b=value_b`, you can get them with the second parameter in your page handler:

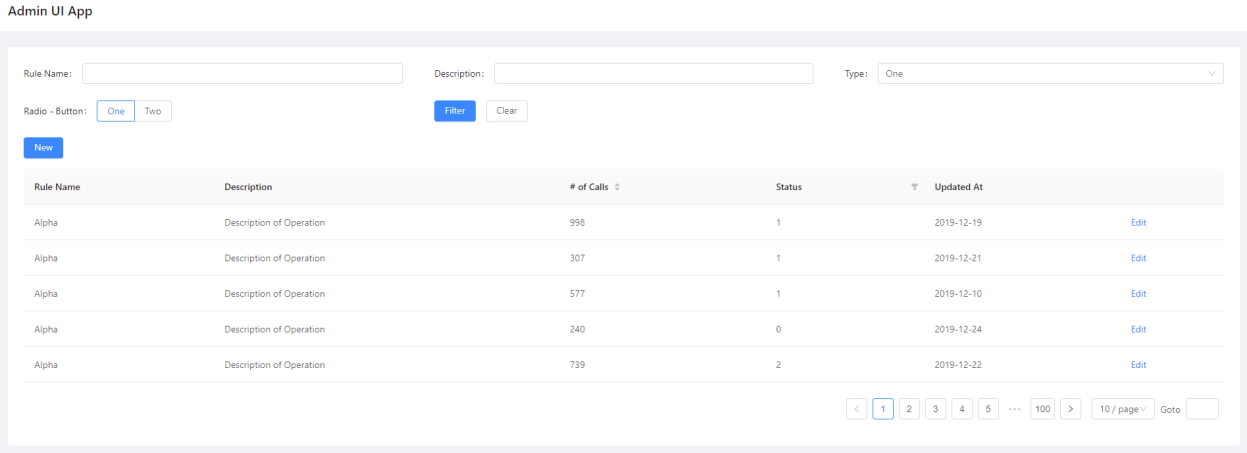
```
@app.page('/detail', 'Detail Page')
def detail_page(arg, all_args):
    ... # all_args will be a dictionary of all the params
```

CHAPTER 8

Use Data Tables

A table page looks like this:

```
@app.page('/', 'Table')
def table_page():
    return [
        Card(content = [
            DataTable("Example Table", columns=table_columns,
                data=TableResult(table_data))])
    ]
```



We use DataTable class to construct a table with data:

class adminui.**DataTable**(*title=''*, *columns=[]*, *data=[]*, *row_actions=[]*, *table_actions=[]*, *filter_form=None*, *on_data=None*, *size='default'*, *scroll_x=None*, *scroll_y=None*, *id=None*)

Insert a data table to the page

Parameters

- **title** – the title of the table

- **columns** – a list-of-dictionaries as column definition. e.g.: [{'title': 'Rule Name', 'dataIndex': 'name'}, ... other columns] [{'title': 'Rule Name', 'dataIndex': 'name', 'sorter': True, 'filterOptions': ['abc', 'def']}, ... other columns] for each column,

  title: the column title dataIndex: its key for the TableResult data dictionary (optional) sorter: (True/False) the column is sortable (optional) filterOptions: ([strings]) a list of options shown as filters (optional) linkTo: display the data as a link to another field of the dataIndex specified (optional) status: dataIndex of another column, shown as the status badge fo the data. The value of the other column must be one from success | processing | default | error | warning

- **data** – a TableResult object for the initial data of the table
- **row_actions** – a list of TableRowAction objects, which means actions shown on each row. Leave it blank if you don't need any action
- **table_actions** – a list of page elements shown on top of the table. Controls such as "New Item" buttons could be listed here.
- **on_data** – a callback function that returns a TableResult object if the user turns a page. an argument will be passed as {'current_page':..., 'page_size':..., 'sorter': {sorted_column_data_index}_{ascend|decsend}} Leave it None if you're sure there is only one page of data.
- **size** – size of the table (default | middle | small)

8.1 Prepare data for a Table

DataTable needs a column definition and a data parameter. the data required in the columns field looks like this:

```
table_columns = [
    {'title': 'Rule Name', 'dataIndex': 'name'},
    {'title': 'Description', 'dataIndex': 'desc'},
    {'title': '# of Calls', 'dataIndex': 'callNo'},
    {'title': 'Status', 'dataIndex': 'status'},
    {'title': 'Updated At', 'dataIndex': 'updatedAt'}
]
```

each column is a dictionary, with `title` and `dataIndex`. `dataIndex` will be used as the key of the provided table rows data:

```
table_data = [
                "callNo": 76,
                "desc": "Description of Operation",
                "id": 0,
                "name": "Alpha",
                "status": 3,
                "updatedAt": "2019-12-13"
            },
            ... other rows
]
```

`table_data` need to be passed with a `TableResult` object:

```
DataTable("Example Table", columns=table_columns,
                data=TableResult(table_data))
```

`TableResult` will also be used in case of pagination.

8.2 Render a column as a link

If you want a column in the table shown as a link, set the column definition as:

```
{'title': 'Rule Name', 'dataIndex': 'name', 'linkTo': 'link'},
```

Then the column's `title` will be shown as a link, linking to data field with key `link`

8.3 Render a column as a badge

To render badges on columns, define the column as:

```
{'title': 'Rule Name', 'dataIndex': 'name', 'status': 'badgeStatus'},
```

Whereas `badgeStatus` in the example is the dataIndex of another column, whose value takes from `success | processing | default | error | warning`, which will be the appearance of the badge.

8.4 Pagination

In case you have multiple pages of data for the table and you can only display some in a page (for example, records read from a database), you'll need pagination.

You need to do two things:

First, fill `TableResult` with information such as total number of records, current page and page size. So the table knows how many page buttons it should display to the user:

```
DataTable("Example Table", columns=table_columns,
            data=TableResult(table_data, 1000), on_data=on_page)
```

class adminui.**TableResult**(*data=[]*, *total=None*, *current_page=1*, *page_size=10*)

Table data used in the "data" column of DataTable, or returned when table pages are requested

Parameters

- **data** – a list of dictionary serves as table data. e.g.: [{id: 1, name: 'Alice', '_actions': ['view', 'edit'], ... more fields} ...more rows of data] id is required as a data record. '_actions' fields dictates which action is available for this row. If omitted, all actions will be available; an empty list means no actions.

- **total** – the total number of records available, may be more than len(data), at which time a pagination bar will be shown.
- **current_page** – the current page of the record, so the frontend will know which page to highlight
- **page_size** – how many records are there in a page.

Second, provide a `on_data` callback function to `DataTable`, so AdminUI knows what data to load when the user turns a page:

```
def on_page(args):
    records = (... load records somewhere from the database,
        with args['current_page'] and args['page_size'])
    return TableResult(mock_table_data(5), 1000, args['current_page'])
```

Now you have a table serving multi-paged data.

8.5 Action Links for each Row

You may also add an action link to each row. This is useful when the user can do something to the records. For example, for a table of articles, the user may wish to edit a single one.

In this case, fill the `row_actions` argument with a list of `TableRowAction`:

```
DataTable("Example Table", columns=table_columns,
                data=TableResult(table_data)
                row_actions=[
                    TableRowAction('edit', 'Edit', on_click=on_edit),
                    TableRowAction('edit', icon='edit', on_click=on_edit), # use icons
                ])
```

class adminui.**TableRowAction**(*id*, *title=''*, *on_click=None*, *icon=None*)

Represent an action link shown in each row of the table

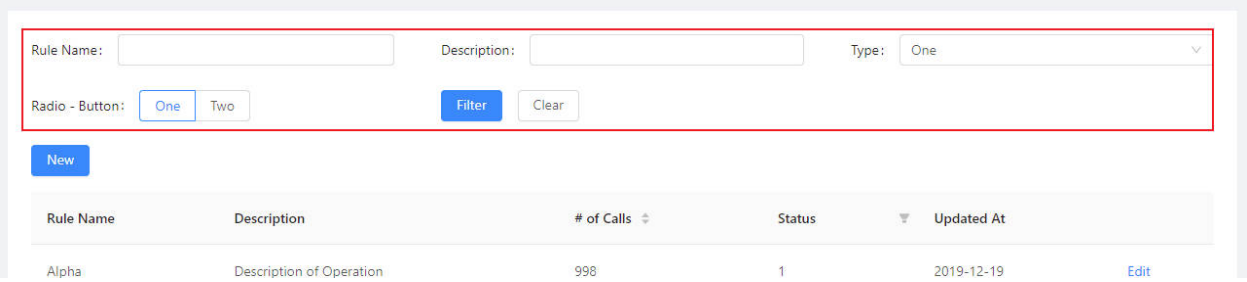
Parameters

- **id** – the id of the action, used in the '_actions' field of TableResult data.
- **title** – the title of the action link
- **on_click** – the callback function called when the user clicked the link. the data row will be passed as the argument of the function

In this case, an "Edit" link will be shown on the right side of each row of the table. If the user clicks one of them, the passed function `on_edit` will be called:

```
def on_edit(record):
    ...do something with the table record
```

8.6 Use Filter Forms



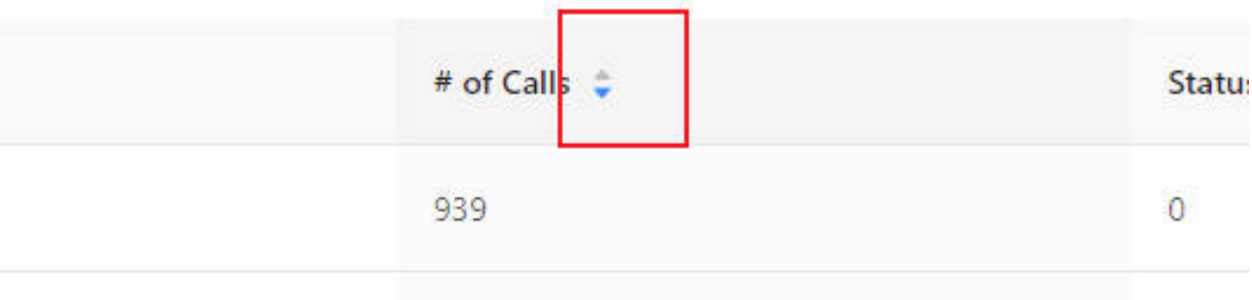
You may add a filter form, to let users search in your table. Each time the user submits the form or switch pages, values in the form will be passed along to the `on_data` callback.

To add filter form, set `filter_form` field in DataTable element:

```
DataTable("Example Table", columns=...,  data=..., on_data=on_page,
            filter_form=FilterForm([
                TextField('Rule Name'),
                TextField('Description'),
                SelectBox('Type', data=['One', 'Two', 'Three'], placeholder="Select␣
↪One"),
                RadioGroup('Radio - Button', data=[['One', 1], ['Two', 2]], format=
↪'button'),
            ], submit_text='Filter', reset_text='Clear'),
            row_actions=[...],
            table_actions=[...])
```

Note that you can change the text on submit button and reset button, using `submit_text` and `reset_text` field.

8.7 Sortable and Filterable Columns



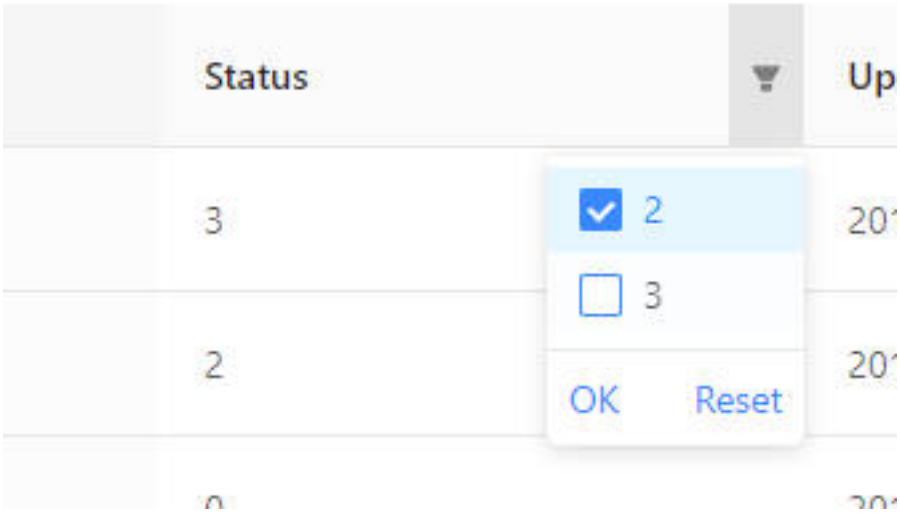
To make table column sortable, set `sorter=True` to the column definition:

```
table_columns = [
    {'title': '# of Calls', 'dataIndex': 'callNo', 'sorter': True},
    ...
]
```

Then, when the user click on the header of the column, `on_data` callback will receive an additional `sorter` argument like:

```
sorter: "callNo_descend"
```

Separated by underscore, the first part is the `dataIndex` field, the second part is descend or ascend according the current sorting status.



To make a table column filterable, set filters on the column definition like:

```
{'title': 'Status', 'dataIndex': 'status', 'filters': [{'text': 2, 'value': 2}, {'text
↪': 3, 'value': 3}]},
```

Then the column will be filterable. When the user filters some columns, `on_data` will receive arguments like:

```
filters: {status: "3,2"}
```

where the key will be the filtered column's data index, and the value will be the filtered values, separated by commas.

8.8 Change the height of rows

pass `size` to DataTable will allow you to change the row height. You may choose from `'default' | 'middle' | 'small'`:

```
DataTable(..., size='small')
```

8.9 Scroll X for the table

setting the `scroll_x` and `scroll_y` attributes for `DataTable`, will let scroll bar show up when the table is too long. Useful for tables with many columns or rows:

```
DataTable(..., scroll_x=1000)
```

A complete example of table is listed here https://github.com/bigeyex/python-adminui/blob/master/python/example_table.py

CHAPTER 9

Use Charts

9.1 Line Chart

`LineChart` takes a list of of the data (numbers, will be the x axis), and a list of the lables:

```
chart_labels = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
chart_data = [1.5, 2.3, 4.3, 2.2, 5.1, 6.5, 2.3, 2.3, 2.2, 1.1]
LineChart(chart_data, chart_labels)
```

You may put more than 1 series on the line chart:

```
LineChart({'series1': chart_data, 'series2': chart_data2}, chart_labels)
```

set `show_area=True` to fill the area of the chart, and set `smooth=False` to stop smoothing the line.

You may set the `height` in every type of chart.

class adminui.**LineChart**(*data=[], labels=None, show_axis=True, show_line=True, show_area=False, smooth=True, height=300, line_color=None, area_color=None, columns=['x', 'y'], id=None*)

Create a line chart

Parameters

- **data** – the data shown on the chart. It can be: a) an array like [2, 3, 4, 5] b) a dict of series, like { 'series1': [2, 3, 4, 5], 'series2': [6, 7, 8, 9] }
- **labels** – labels corresponding to the data. Must be to the same length of the data e.g. ['a', 'b', 'c', 'd']
- **show_axis** – True for displaying the x and y axis, labels and markers
- **show_line** – False for hiding the line, if you wish to make a pure-area chart
- **show_area** – True then the area below the line will be filled
- **smooth** – True then the line will be smoothed out

- **height** – the height of the chart
- **line_color** – line color as a string if data is a list else an array of colors
- **area_color** – area color as a string if data is a list else an array of colors

9.2 Bar Chart

Basic usage:

```
BarChart(chart_data, chart_labels)
```

Multiple bars will be put side-by-side. Stack them with `stack=True`:

```
BarChart({'series1': chart_data, 'series2': chart_data2}, chart_labels, stack=True),
```

class adminui.**BarChart**(*data=[], labels=None, show_axis=True, height=300, color=None, columns=['x', 'y'], stack=False, id=None*)

Create a bar chart

Parameters

- **data** – the data shown on the chart. It can be: a) an array like [2, 3, 4, 5] b) a dict of series, like { 'series1': [2, 3, 4, 5], 'series2': [6, 7, 8, 9] }
- **labels** – labels corresponding to the data. Must be to the same length of the data e.g. ['a', 'b', 'c', 'd']
- **show_axis** – True for displaying the x and y axis, labels and markers
- **height** – the height of the chart
- **color** – color as a string if data is a list else an array of colors

9.3 Pie Chart

Basic usage:

```
PieChart(chart_data, chart_labels)
```

class adminui.**PieChart**(*data=[], labels=None, height=300, color=None, title=None, id=None*)

Create a bar chart

Parameters

- **data** – the data shown on the chart. e.g. an array like [2, 3, 4, 5]
- **labels** – labels corresponding to the data. Must be to the same length of the data e.g. ['a', 'b', 'c', 'd']
- **height** – the height of the chart
- **title** – the title of the chart

9.4 Scatter Plot

Scatter Plot takes `x`, `y` series, and optionally `size` and `color` (both can be a constant or an array of data):

```
ScatterPlot(list_of_x, list_of_y),
ScatterPlot(list_of_x, list_of_y, color=list_of_color_data, size=list_of_size_data),
```

class adminui.**ScatterPlot**(*x=[]*, *y=[]*, *labels={'color': 'color', 'size': 'size', 'x': 'x', 'y': 'y'}*, *height=300*, *color=None*, *size=3*, *opacity=0.65*, *id=None*)

Create a bar chart

Parameters

- **x** – data in the x axis e.g. an array like [2, 3, 4, 5]
- **y** – data in the y axis
- **color** –

  a) color of the points; b) a series of data shown as the color e.g. [1, 2, 1, 2]
- **size** –

  a) (int) size of data points; b) a series of data as the size e.g. [1, 2, 3, 1, 2, 3]
- **labels** – labels of the x, y axis
- **height** – the height of the chart
- **opacity** – the opacity of the data points

An example with chart is listed here: https://github.com/bigeyex/python-adminui/blob/master/python/example_chart.py https://github.com/bigeyex/python-adminui/blob/master/python/example_dash.py

CHAPTER 10

Require users to Log in

AdminUI comes with a login page. To enable it, specify a function to handle login:

```
@app.login()
def on_login(username, password):
    if username=='alice' and password=='123456':
        return LoggedInUser("Alice")
    else:
        return LoginFailed()
```

The function will receive the username and password users inputted; you need to return LoggedInUser or LoginFailed depending on the result. Instead of a simple if statement, typically you need to check the credential against a database or something.

If you want to redirect logged in user to a different page, set the redirect_to argument in LoggedInUser, like:

```
return LoggedInUser("Alice", redirect_to='/detail')
```

The returned LoggedInUser and LoginFailed may contain more information:

class adminui.**LoggedInUser**(*display_name=''*, *auth=['user']*, *avatar='https://gw.alipayobjects.com/zos/antfincdn/XAosXuN*
user_info=None, *redirect_to=None*)

Returned by login handler, represent a successfully logged in user.

Parameters

- **display_name** – the display name of the user
- **auth** – a list of permission string the user have. Will be checked against in pages or menus
- **avatar** – the avatar of the user
- **user_info** – info for future use, accessible by app.current_user()['user_info']

class adminui.**LoginFailed**(*title='Login Failed'*, *message='Username or password is incorrect'*)

Returned by login handler, represent a failed login attempt

Parameters

- **title** – the title shown in the error message. default: 'Login Failed'
- **message** – the error message content. default: 'Username or password is incorrect'

10.1 Pages requires authorization

By default, any user can visit the page you described. If you want to show the page to users only with certain permission, add an auth_needed attribute to your page:

```
@app.page('/', 'Logged In', auth_needed='user')
def form_page():
    return [
        Card('Logged In', [
            Header('You are logged in')
        ])
    ]
```

In this way, only logged in users can visit this page. Other users will be redirected to the login page.

You may also use *auth_needed='admin'*, then a user logged in with:

```
LoggedInUser("Alice", auth=['user', 'admin'])
```

May access this page, since the user Alice has 'admin' authority.

10.2 Menus for different user roles

Menus can also be protected, by attaching *auth_needed* attribute. For example:

```
MenuItem('User Home', '/user_home', icon="dashboard", auth_needed='user')
```

10.3 Forget password link and register link

You may change the forget password link, or register link on the login page, like:

```
app.register_link={'Sign Up': '/signup'}
app.forget_password_link={'Forget Password': '/forget'}
```

If they are set to `None`, they won't show up on the screen.

CHAPTER 11

Customization

You can change the title of your app:

```
app.app_title = "AdminUI APP"
```

And you can change the logo. It accepts an URL:

```
app.app_logo = "http://...."
```

And you can change the copyright (in the footer) text like:

```
app.copyright_text = 'App with Login by AdminUI'
```

(Leave a blank string if you don't need it)

Or change the footer links:

```
app.footer_links = {'Github': 'https://github.com/bigeyex/python-adminui', 'Ant Design
↪': 'https://ant.design'}
```

Change menu style with:

```
app.app_styles = {'nav_theme': 'light', 'layout': 'topmenu'}
```

where nav_theme takes 'dark'(default) or 'light'; layout takes 'sidemenu'(default) or 'topmenu'

11.1 Favicons

You may set another favicon other than the default one:

```
app.app_favicon = os.path.join(Path(__file__).parent.absolute(), 'new-favicon.png')
```

CHAPTER 12

Serve Static Files

If you have static files, like images, to serve, or you want to let the users access the upload folder, you can add more static path with:

```
app.static_files = {'/upload': os.path.join(Path(__file__).parent.absolute(), 'upload
↪')}
```

then the user can access the files in the upload folder (of the starting python file's path), with urls like `/upload/...`. `app.static_files` takes a dictionary, with the key as the path in the URL, and the value as the path in your file system.

CHAPTER 13

Other Components

13.1 Icons

You can add one of the Ant Design icons using `Icon`:

```
Icon('sync')
```

A full list can be seen at: https://3x.ant.design/components/icon-cn/

only outline style icons are supported.

You may set the color and size of the icon.

class adminui.**Icon**(*name=''*, *color='rgba(0, 0, 0, 0.8)'*, *size=16*, *rotate=False*, *spin=False*, *id=None*)

Display an Ant Design Icon

Parameters

- **name** – name of the icon, see https://ant.design/components/icon/ for a detailed list
- **color** – color of the icon
- **size** – (font) size of the icon
- **rotate** – (bool)rotation angle of the icon
- **spin** – (bool)whether the icon is spinning

13.2 Progress

You can draw a progress bar with:

```
Progress(30)
Progress(30, format='circle')
```

13.3 Image

To add an image:

```
Image('https://url-of-the-image', 'alt-text', width=300)
```

13.4 Group (HTML Div)

Use Group to group content together, so you may change them with `UpdateElement` page action

(in fact, it just creates a element in html):

```
Group(id='id of the content', content=[...content in the group])
```

13.5 Tabs

Use tabs to group content together:

```
Tabs([
    Group(name='title of tab 1', content=[
        ... content of tab 1
    ]),
    Group(name='title of tab 2', content=[
        ... content of tab 2
    ]),
]),
```

You may set the tab appearance with `position`, `format`, and `size`, see

class adminui.**Tabs**(*content=[]*, *position='top'*, *format='line'*, *size='default'*, *id=None*)
Display a group of tabs. Each content item is a tab

Parameters

- **position** – top | left | bottom | right, where the tab is (decide if it's horizontal or vertical)
- **format** – line | card - the style of the tabs
- **size** – default | large | small

13.6 Spin

Creates a spinning wheel:

```
Spin()
```

You can also setup a region masked under a spinning wheel. When the loading is done, remove it with Page Actions:

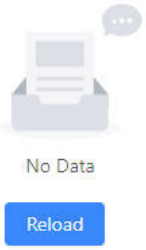
```
Spin('loading', content=[
    ... content under the mask...
]),
```

class adminui.**Spin**(*title=''*, *content=[]*, *size='default'*, *id=None*)
A spinning icon, indicating something is loading

Parameters

- **size** – default | small | large: the size of the spinning wheel
- **content** – (optional), when creating a container with a "loading" mask, put its content here
- **title** – the text shown below the spinning wheel

13.7 Empty Status



Display the empty status:

```
Empty()
```

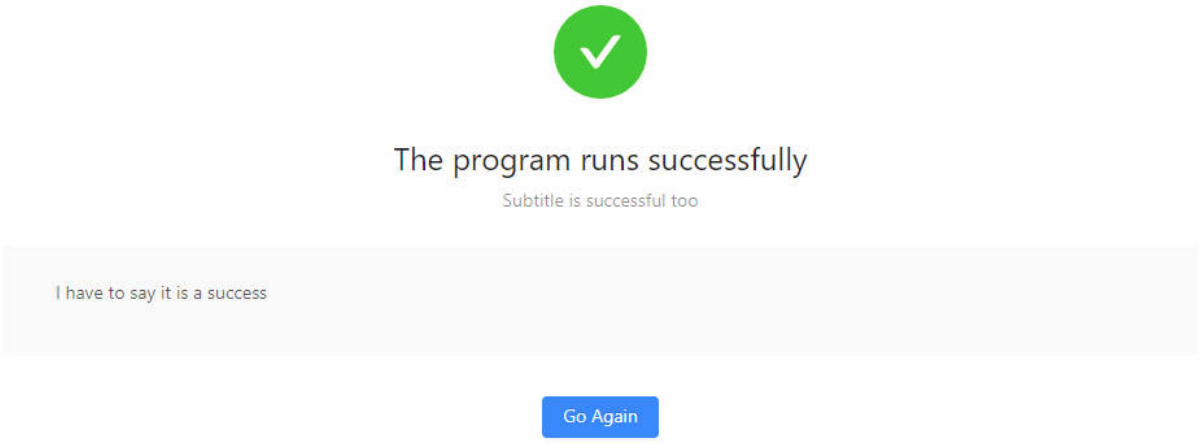
class adminui.**Empty**(*title=None*, *content=[]*, *simple_style=False*, *id=None*)

Display an Empty status

Parameters

- **title** – the description text shown on the empty indicator
- **content** – (optional), put additional buttons or elements below the icon
- **simple_style** – set True to deploy a simple style of empty box

13.8 Result



Display the result of an operation:

```
Result('The program runs successfully')
```

class adminui.**Result**(*title=None*, *status='success'*, *sub_title=None*, *content=[]*, *extra=[]*, *id=None*)

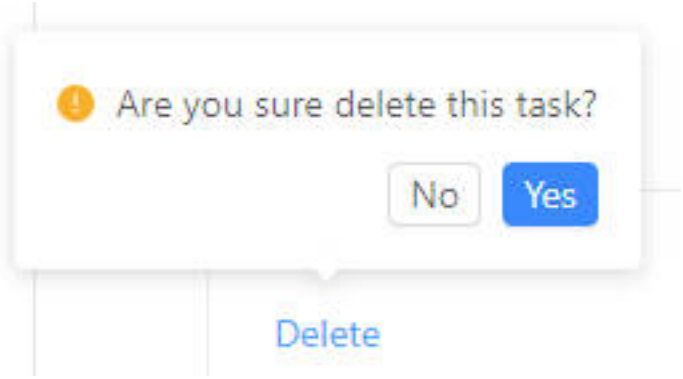
Display the result (success, failure, 403, 404...) of an action

Parameters

- **title** – the title of the result feedback

- **sub_title** – the sub-title of the result section
- **status** – 'success' | 'error' | 'info' | 'warning'| '404' | '403' | '500'
- **content** – put additional buttons or elements inside a result box
- **extra** – extra action buttons on the result section

13.9 Popconfirm



Let the user confirm before performing an action:

```
Popconfirm('Are you sure to do something?', on_submit=on_submit, content=[
    ... put adminui elements here, which will trigger the confirm box when clicked ...
], data=CUSTOM_DATA)
```

note that `CUSTOM_DATA` may be passed to the `on_submit` callback, when the user choose YES in the confirm box.

You may customize the OK and Cancel button of the pop confirm box by setting `ok_text` and `cancel_text` of the element

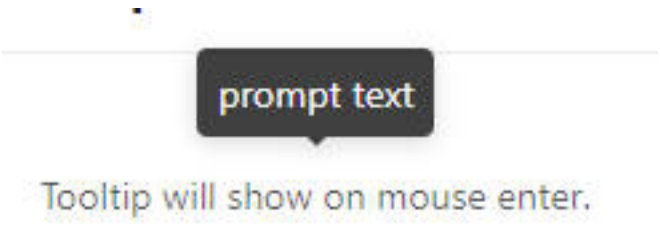
class adminui.**Popconfirm**(*title=None*, *content=[]*, *on_submit=None*, *ok_text='Yes'*, *cancel_text='No'*, *data=None*, *id=None*)

Before the user perform an action, ask him/her to confirm twice.

Parameters

- **title** – the text shown on the pop confirm box
- **content** – the enclosed content, which shows the Popconfirm when clicked
- **on_submit** – (func) callback after user clicked 'ok'
- **ok_text** – text shown on the OK button
- **cancel_text** – text shown on the Cancel button
- **data** – data which will passed as the parameter to the callback

13.10 Tooltip



Display a tooltip when the mouse is hovering some elements:

```
Tooltip('Text on the tooltip', [
    ... content which will trigger the tooltip when hovered ...
])
```

class adminui.**Tooltip**(*title=None*, *content=[]*, *placement='top'*, *id=None*)

Show a tooltip when the user moves the mouse upon its content

Parameters

- **title** – the text shown on the tooltip
- **content** – the content, which will show the tooltip when the mouse is on it
- **placement** – one of top | left | right | bottom | topLeft | topRight | bottomLeft | bottomRight | leftTop | leftBottom | rightTop | rightBottom

CHAPTER 14

Organizing your App

When your app grows bigger, you may need to split it to multiple files.

While there are many ways to do this in Python, adminui provides a simple way for simple apps.

For (a minimal simple) example, you want to make an app with a home page and a detail page, so the structure is like:

```
home.py
detail.py
```

in `home.py`, you layout the home page like:

```
from adminui import *

app = AdminApp()
@app.page('/', 'home')
def home_page():
    ... layout the home page ...

app.set_as_shared_app() # set the app as the shared app, so it can be accessed␣
↪globally
import detail           # import all the other files in your project (you can import␣
↪files recursively
                        # meaning if you have admin_pages.py, you can import all the␣
↪admin related pages there
                        # and in home.py just import admin_pages)

if __name__ == '__main__':
    app.run()
```

note the `app.set_as_shared_app()` makes the app exposed in the whole project, and then in the `home.py` , `detail.py` is imported.

in `detail.py`, use `AdminApp.shared_app()` to access the app add add more pages:

```
# content in detail.py
from adminui import *

app = AdminApp.shared_app() # now you have the app ready to add more pages

@app.page('/detail', 'Detail Page')
def detail_page():
    ... layout the detail page ...
```

CHAPTER 15

Indices and tables

- genindex
- search

Index