aiogram-dialog

Tishka17

Aug 14, 2022

CONTENTS:

CHAPTER

ONE

OVERVIEW

1.1 Concept

Aiogram-dialog is a GUI framework for telegram bot. It is inspired by ideas of Android SDK and React.js

Main ideas are:

1. Split data retrieving and message rendering
2. Unite rendering buttons and processing clicks
3. Better states routing
4. Widgets

The main building block of your UI is **Window**. Each window represents a message sent to user and processing of a user reaction on it.

Each window consists of **Widgets** and callback functions. Widgets can represent message text and keyboard. Callbacks are used to retrieve required data or process user input.

You combine windows into **Dialog**. This allows you to switch between windows creating different scenarios of communication with user.

In more complex cases you can create more than one dialog. Then you can start new dialogs without closing previous one and automatically return back when it is closed. You can pass data between dialogs keeping they state isolated at the same time.

1.2 Quickstart

Install library:

```
pip install aiogram_dialog
```

Create states group for your dialog:

```
from aiogram.dispatcher.filters.state import StatesGroup, State

class MySG(StatesGroup):
    main = State()
```

Create at least one window with buttons or text:

```
from aiogram.dispatcher.filters.state import StatesGroup, State

from aiogram_dialog import Window
from aiogram_dialog.widgets.kbd import Button
from aiogram_dialog.widgets.text import Const


class MySG(StatesGroup):
    main = State()


main_window = Window(
    Const("Hello, unknown person"),  # just a constant text
    Button(Const("Useless button"), id="nothing"),  # button with text and id
    state=MySG.main,  # state is used to identify window between dialogs
)
```

Create dialog with your windows:

```
from aiogram_dialog import Dialog

dialog = Dialog(main_window)
```

Let's assume that you have created your aiogram bot with dispatcher and states storage as you normally do. It is important you have a storage because **aiogram_dialog** uses `FSMContext` internally to store it state:

```
from aiogram import Bot, Dispatcher, executor
from aiogram.contrib.fsm_storage.memory import MemoryStorage

storage = MemoryStorage()
bot = Bot(token='BOT TOKEN HERE')
dp = Dispatcher(bot, storage=storage)
```

To start using your dialog you need to register it. Also library needs some additional registrations for its internals. To do it we will create **DialogRegistry** and use it to register our dialog

```
from aiogram_dialog import DialogRegistry

registry = DialogRegistry(dp)  # this is required to use `aiogram_dialog`
registry.register(dialog)  # register a dialog
```

At this point we have configured everything. But dialog won't start itself. We will create simple command handler to deal with it. To start dialog we need **DialogManager** which is automatically injected by library. Also mind the `reset_stack` argument. The library can start multiple dialogs stacking one above other. Currently we do not want this feature, so we will reset stack on each start:

```
from aiogram.types import Message
from aiogram_dialog import DialogManager, StartMode

@dp.message_handler(commands=["start"])
async def start(m: Message, dialog_manager: DialogManager):
    # Important: always set `mode=StartMode.RESET_STACK` you don't want to stack dialogs
    await dialog_manager.start(MySG.main, mode=StartMode.RESET_STACK)
```

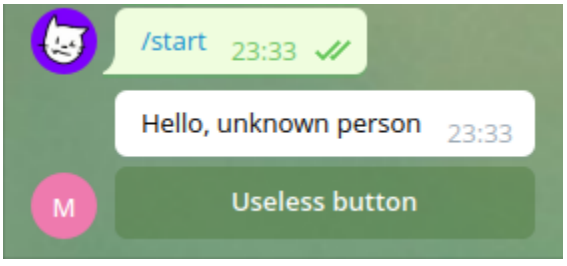
Last step, you need to start your bot as usual:

```
from aiogram import executor

if __name__ == '__main__':
    executor.start_polling(dp, skip_updates=True)
```

Summary:

```
from aiogram import Bot, Dispatcher, executor
from aiogram.contrib.fsm_storage.memory import MemoryStorage
from aiogram.dispatcher.filters.state import StatesGroup, State
from aiogram.types import Message

from aiogram_dialog import Window, Dialog, DialogRegistry, DialogManager, StartMode
from aiogram_dialog.widgets.kbd import Button
from aiogram_dialog.widgets.text import Const

storage = MemoryStorage()
bot = Bot(token='BOT TOKEN HERE')
dp = Dispatcher(bot, storage=storage)
registry = DialogRegistry(dp)


class MySG(StatesGroup):
    main = State()


main_window = Window(
    Const("Hello, unknown person"),
    Button(Const("Useless button"), id="nothing"),
    state=MySG.main,
)
dialog = Dialog(main_window)
registry.register(dialog)


@dp.message_handler(commands=["start"])
async def start(m: Message, dialog_manager: DialogManager):
    await dialog_manager.start(MySG.main, mode=StartMode.RESET_STACK)


if __name__ == '__main__':
    executor.start_polling(dp, skip_updates=True)
```

The result will look like:



CHAPTER

TWO

WIDGETS AND RENDERING

2.1 Passing data

Some widgets contain fixed text, others can show dynamic contents For example:

- `Const("Hello, {name}!")` will be rendered as `Hello, {name}!`
- `Format("Hello, {name}!")` will interpolate with window data and transformed to something like `Hello, Tishka17!`

So, widgets can use data. But data must be loaded from somewhere. To do it Windows and Dialogs have `getter` attribute. Getter can be either a function returning data or static dict or list of such objects.

So let's create a function and use it to enrich our window with data.

Note: In this and later examples we will skip common bot creation and dialog registration code unless it has notable differences with quickstart

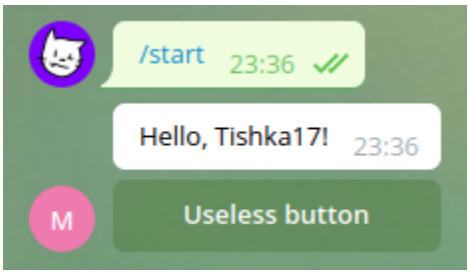
```
from aiogram.dispatcher.filters.state import StatesGroup, State

from aiogram_dialog import Window, Dialog
from aiogram_dialog.widgets.kbd import Button
from aiogram_dialog.widgets.text import Const, Format


class MySG(StatesGroup):
    main = State()


async def get_data(**kwargs):
    return {
        "name": "Tishka17",
    }


dialog = Dialog(
    Window(
        Format("Hello, {name}!"),
        Button(Const("Useless button"), id="nothing"),
        state=MySG.main,
        getter=get_data,  # here we set our data getter
```

(continues on next page)

(continued from previous page)

```
    )
)
```

It will look like:



Since version 1.6 you do not need getter to access some common objects:

- `dialog_data` -contents of corresponding field from current context. Normally it is used to store data between multiple calls and windows withing single dialog
- `start_data` - data passed during current dialog start. It is also accessible using `current_context`
- `middleware_data` - data passed from middlewares to handler. Same as `dialog_manager.data`
- `event` - current processing event which triggered window update. Be careful using it, because different types of events can cause refreshing same window.

2.2 Widget types

2.2.1 Base information

Currently there are 3 kinds of widgets: *texts*, *keyboards* and *media*.

- **Texts** used to render text anywhere in dialog. It can be message text, button title and so on.
- **Keyboards** represent parts of `InlineKeyboard`
- **Media** represent media attachment to message

Also there are 2 general types:

- `Whenable` can be hidden or shown depending on data or some conditions. Currently al widgets are whenable. See: *Hiding widgets*
- `Actionable` is any widget with action (currently only any type of keyboard). It has `id` and can be found by that id. It recommended for all stateful widgets (e.g Checkboxes) to have unique id within dialog. Buttons with different behavior also must have different ids.

Note: Widget id can contain only ascii letters, numbers, underscore and dot symbol.

- `123`, `com.mysite.id`, `my_item` - valid ids
- `hello world`, `my:item`, - invalid ids

2.2.2 Text widget types

Every time you need to render text use any of text widgets:

- `Const` - returns text with no midifications
- `Format` - formats text using `format` function. If used in window the data is retrived via `getter` funcion.
- *Multi* - multiple texts, joined with a separator
- *Case* - shows one of texts based on condition
- `Progress` - shows a progress bar
- *Jinja* - represents a HTML rendered using jinja2 template

2.2.3 Keyboard widget types

Each keyboard provides one or multiple inline buttons. Text on button is rendered using text widget

- *Button* - single inline button. User provided `on_click` method is called when it is clicked.
- *Url* - single inline button with url
- *Group* - any group of keyboards one above another or rearranging buttons.
- *ScrollingGroup* - the same as the `Group`, but with the ability to scroll through pages with buttons.
- `ListGroup` - group of widgets applied repeated multiple times for each item in list
- *Row* - simplified version of group. All buttons placed in single row.
- *Column* - another simplified version of group. All buttons placed in single column one per row.
- *Checkbox* - button with two states
- *Select* - dynamic group of buttons intended for selection use.
- *Radio* - switch between multiple items. Like select but stores chosen item and renders it differently.
- *Multiselect* - selection of multiple items. Like select/radio but stores all chosen items and renders them differently.
- *Calendar* - simulates a calendar in the form of a keyboard.
- `SwitchTo` - switches window within a dialog using provided state
- `Next/Back` - switches state forward or backward
- `Start` - starts a new dialog with no params
- `Cancel` - closes the current dialog with no result. An underlying dialog is shown

2.3 Combining texts

To combine multiple texts you can use `Multi` widget. You can use any texts inside it. Also you can provide a string separator

```
from aiogram_dialog.widgets.text import Multi, Const, Format


# let's assume this is our window data getter
async def get_data(**kwargs):
```

(continues on next page)

(continued from previous page)

```
    return {"name": "Tishka17"}


# This will produce text `Hello! And goodbye!`
text = Multi(
    Const("Hello!"),
    Const("And goodbye!"),
    sep=" ",
)

# This one will produce text `Hello, Tishka17, and goodbye {name}!`
text2 = Multi(
    Format("Hello, {name}"),
    Const("and goodbye {name}!"),
    sep=", ",
)

# This one will produce `01.02.2003T04:05:06`
text3 = Multi(
    Multi(Const("01"), Const("02"), Const("2003"), sep="."),
    Multi(Const("04"), Const("05"), Const("06"), sep=":"),
    sep="T"
)
```

To select one of the texts depending on some condition you should use `Case`. The condition can be either a data key or a function:

```
from typing import Dict

from aiogram_dialog import DialogManager
from aiogram_dialog.widgets.text import Multi, Const, Format, Case


# let's assume this is our window data getter
async def get_data(**kwargs):
    return {"color": "red", "number": 42}


# This will produce text `Square`
text = Case(
    {
        "red": Const("Square"),
        "green": Const("Unicorn"),
        "blue": Const("Moon"),
    },
    selector="color",
)


# This one will produce text `42 is even!`
def parity_selector(data: Dict, case: Case, manager: DialogManager):
    return data["number"] % 2
```

(continues on next page)

(continued from previous page)

```
text2 = Case(
    {
        0: Format("{number} is even!"),
        1: Const("It is Odd"),
    },
    selector=parity_selector,
)
```

2.4 Jinja HTML rendering

It is very easy to create safe HTML messages using Jinja2 templates. Documentation for template language is available at official jinja web page

To use it you need to create text using `Jinja` class instead of `Format` and set proper `parse_mode`. If you do not want to set default parse mode for whole bot you can set it per-window.

For example you can use environment substitution, cycles and filters:

```
from aiogram.dispatcher.filters.state import StatesGroup, State
from aiogram.types import ParseMode

from aiogram_dialog import Window
from aiogram_dialog.widgets.text import Jinja


class DialogSG(StatesGroup):
    ANIMALS = State()


# let's assume this is our window data getter
async def get_data(**kwargs):
    return {
        "title": "Animals list",
        "animals": ["cat", "dog", "my brother's tortoise"]
    }


html_text = Jinja("""
<b>{{title}}</b>
{% for animal in animals %}
* <a href="https://yandex.ru/search/?text={{ animal }}">{{ animal|capitalize }}</a>
{% endfor %}
""")

window = Window(
    html_text,
    parse_mode=ParseMode.HTML,  # do not forget to set parse mode
    state=DialogSG.ANIMALS,
```

(continues on next page)

(continued from previous page)

```
    getter=get_data
)
```

It will be rendered to this HTML:

```
<b>Animals list</b>
* <a href="https://yandex.ru/search/?text=cat">Cat</a>
* <a href="https://yandex.ru/search/?text=dog">Dog</a>
* <a href="https://yandex.ru/search/?text=my brother&#39;s tortoise">My brother&#39;s␣
↪tortoise</a>
```

If you want to add custom filters or do some configuration of jinja Environment you can setup it using `aiogram_dialog.widgets.text.setup_jinja` function

2.5 Keyboards

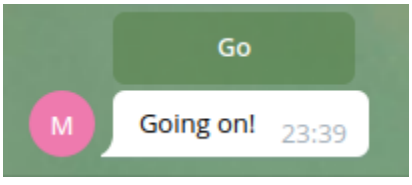
2.5.1 Button

In simple case you can use keyboard consisting of single button. Button consts of text, id, on-click callback and when condition.

Text can be any `Text` widget, that represents plain text. It will receive window data so you button will have dynamic caption

Callback is normal async function. It is called when user clicks a button Unlike normal handlers you should not call callback.answer(), as it is done automatically.

```
from aiogram.types import CallbackQuery

from aiogram_dialog import DialogManager
from aiogram_dialog.widgets.kbd import Button
from aiogram_dialog.widgets.text import Const


async def go_clicked(c: CallbackQuery, button: Button, manager: DialogManager):
    await c.message.answer("Going on!")


go_btn = Button(
    Const("Go"),
    id="go",  # id is used to detect which button is clicked
    on_click=go_clicked,
)
```



If it is unclear to you where to put button, check *Quickstart*

2.5.2 Url

Url represents a button with an url. It has no callbacks because telegram does not provide any notifications on click.

Url itself can be any text (including `Const` or `Format`)

```
from aiogram_dialog.widgets.kbd import Url
from aiogram_dialog.widgets.text import Const

go_btn = Url(
    Const("Github"),
    Const('https://github.com/Tishka17/aiogram_dialog/'),
)
```

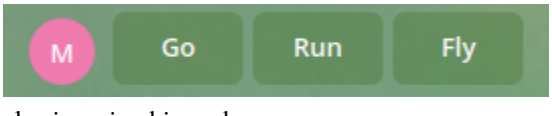
2.5.3 Grouping buttons

Normally you will have more than one button in your keyboard.

Simplest way to deal with it - unite multiple buttons in a `Row`, `Column` or other `Group`. All these widgets can be used anywhere you can place a button.

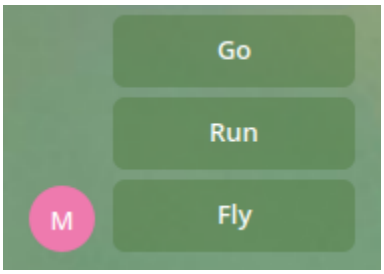
Row widget is used to place all buttons inside single row. You can place any keyboard widgets inside it (for example buttons or groups) and it will ignore any hierarchy and just place telegram buttons in a row.

```
from aiogram.types import CallbackQuery

from aiogram_dialog import DialogManager
from aiogram_dialog.widgets.kbd import Button, Row
from aiogram_dialog.widgets.text import Const


async def go_clicked(c: CallbackQuery, button: Button, manager: DialogManager):
    await c.message.answer("Going on!")


async def run_clicked(c: CallbackQuery, button: Button, manager: DialogManager):
    await c.message.answer("Running!")


row = Row(
    Button(Const("Go"), id="go", on_click=go_clicked),
    Button(Const("Run"), id="run", on_click=run_clicked),
    Button(Const("Fly"), id="fly"),
)
```



Column widget is like a row, but places everything in a column, also ignoring hierarchy.

```python
from aiogram_dialog.widgets.kbd import Button, Column
from aiogram_dialog.widgets.text import Const

column = Column(
    Button(Const("Go"), id="go"),
    Button(Const("Run"), id="run"),
    Button(Const("Fly"), id="fly"),
)
```

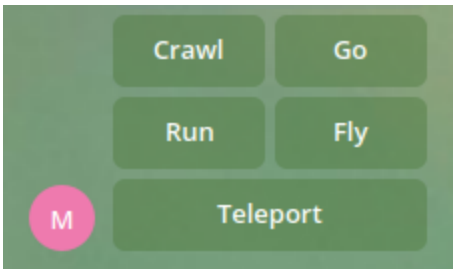


Group widget does more complex unions. By default, it places one keyboard below another. For example, you can stack multiple rows (or groups, or whatever)

```python
from aiogram_dialog.widgets.kbd import Button, Group, Row
from aiogram_dialog.widgets.text import Const

group = Group(
    Row(
        Button(Const("Go"), id="go"),
        Button(Const("Run"), id="run"),
    ),
    Button(Const("Fly"), id="fly"),
)
```

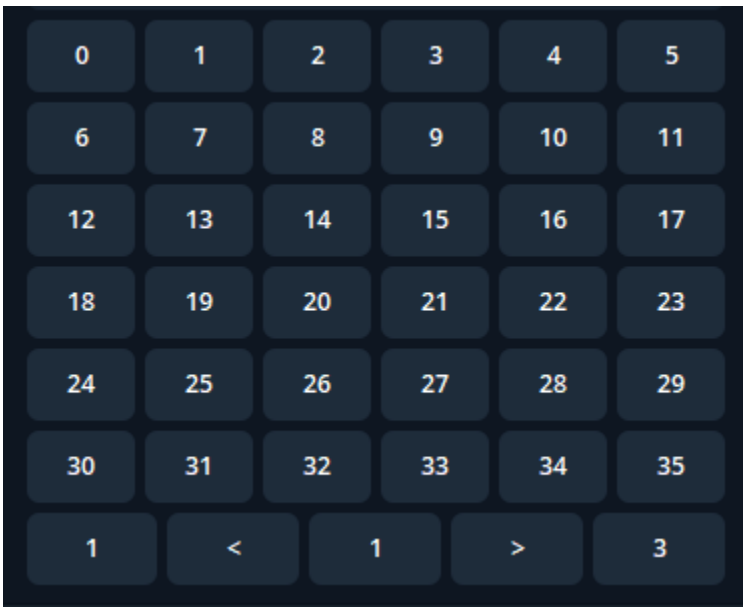
Also it can be used to produce rows of fixed width. To do it just set `width` to desired value. Honestly, `Row` and `Column` widgets are groups with predefined width.

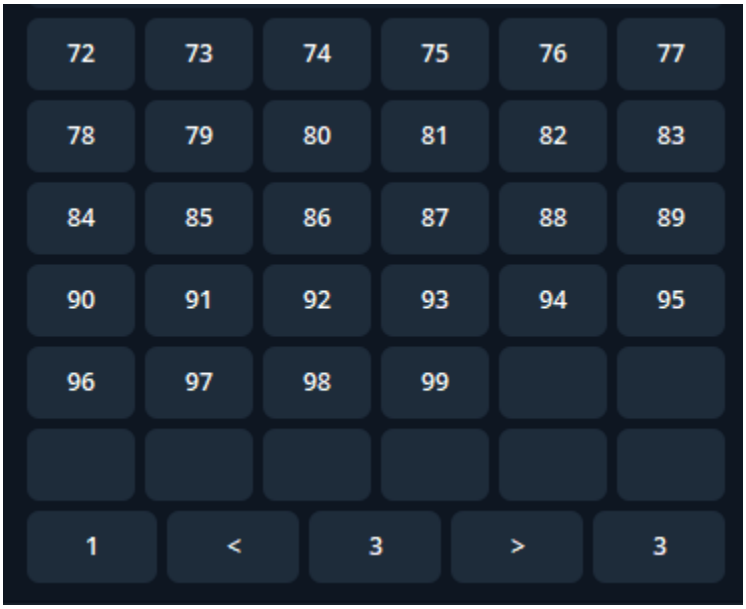
```python
from aiogram_dialog.widgets.kbd import Button, Group
from aiogram_dialog.widgets.text import Const

group = Group(
    Button(Const("Crawl"), id="crawl"),
    Button(Const("Go"), id="go"),
    Button(Const("Run"), id="run"),
    Button(Const("Fly"), id="fly"),
    Button(Const("Teleport"), id="tele"),
    width=2,
)
```



ScrollingGroup widget combines buttons into pages with the ability to scroll forward and backward and go to the last or first page with buttons. You can set the `height` and `width` of the keyboard. If there are not enough buttons for the last page, the keyboard will be filled with empty buttons keeping the specified height and width.

```
from aiogram_dialog.widgets.kbd import Button, ScrollingGroup
from aiogram_dialog.widgets.text import Const


def test_buttons_creator(btn_quantity):
    buttons = []
    for i in btn_quantity:
        i = str(i)
        buttons.append(Button(Const(i), id=i))
    return buttons


test_buttons = test_buttons_creator(range(0, 100))

scrolling_group = ScrollingGroup(
    *test_buttons,
    id="numbers",
    width=6,
    height=6,
)
```





2.5.4 Checkbox

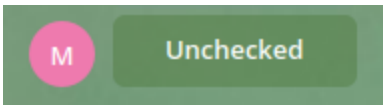
Some of the widgets are stateful. They have some state which is affected by on user clicks.

On of such widgets is **Checkbox**. It can be in checked and unchecked state represented by two texts. On each click it inverses its state.

If a dialog with checkbox is visible, you can check its state by calling `is_checked` method and change it calling `set_checked`

As button has `on_click` callback, checkbox has `on_state_changed` which is called each time state switched regardless the reason

```
from aiogram_dialog import DialogManager, ChatEvent
from aiogram_dialog.widgets.kbd import Checkbox, ManagedCheckboxAdapter
from aiogram_dialog.widgets.text import Const


async def check_changed(event: ChatEvent, checkbox: ManagedCheckboxAdapter, manager:␣
↪DialogManager):
    print("Check status changed:", checkbox.is_checked())


check = Checkbox(
    Const("✓  Checked"),
    Const("Unchecked"),
    id="check",
    default=True,  # so it will be checked by default,
    on_state_changed=check_changed,
)
```



Note: State of widget is stored separately for each separate opened dialog. But all windows in dialog share same storage. So, multiple widgets with same id will share state. But at the same time if you open several copies of same dialogs they will not mix their states

2.5.5 Select

Select acts like a group of buttons but data is provided dynamically. It is mainly intended to use for selection a item from a list.

Normally text of selection buttons is dynamic (e.g. `Format`). During rendering an item text, it is passed a dictionary with:

- `item` - current item itself
- `data` - original window data
- `pos` - position of item in current items list starting from 1
- `pos0` - position starting from 0

So the main required thing is items. Normally it is a string with key in your window data. The value by this key must be a collection of any objects. If you have a static list of items you can pass it directly to a select widget instead of providing data key.

Next important thing is ids. Besides a widget id you need a function which can return id (string or integer type) for any item.

```
from typing import Any
import operator

from aiogram.types import CallbackQuery

from aiogram_dialog import DialogManager
from aiogram_dialog.widgets.kbd import Select
from aiogram_dialog.widgets.text import Format


# let's assume this is our window data getter
async def get_data(**kwargs):
    fruits = [
        ("Apple", '1'),
        ("Pear", '2'),
        ("Orange", '3'),
        ("Banana", '4'),
    ]
    return {
        "fruits": fruits,
        "count": len(fruits),
    }
```

(continues on next page)

(continued from previous page)

```
async def on_fruit_selected(c: CallbackQuery, widget: Any, manager: DialogManager, item_
↪id: str):
    print("Fruit selected: ", item_id)


fruits_kbd = Select(
    Format("{item[0]} ({pos}/{data[count]})"),  # E.g `✓ Apple (1/4)`
    id="s_fruits",
    item_id_getter=operator.itemgetter(1),  # each item is a tuple with id on a first␣
↪position
    items="fruits",  # we will use items from window data at a key `fruits`
    on_click=on_fruit_selected,
)
```

Note: Select places everything in single row. If it is not suitable for your case - simply wrap it with *Group* or *Column*

2.5.6 Radio

Radio is staeful version of select widget. It marks each clicked item as checked deselecting others. It stores which item is selected so it can be accessed later

Unlike for the `Select` you need two texts. First one is used to render checked item, second one is for unchecked. Passed data is the same as for `Select`

Unlike in normal buttons and window they are used to render an item, but not the window data itself.

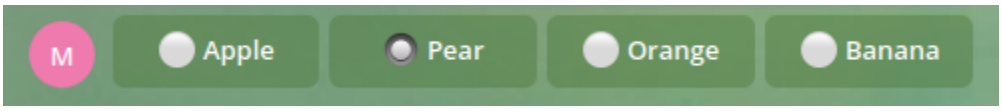
Also you can provide `on_state_changed` callback function. It will be called when selected item is changed.

```
import operator

from aiogram_dialog.widgets.kbd import Radio
from aiogram_dialog.widgets.text import Format


# let's assume this is our window data getter
async def get_data(**kwargs):
    fruits = [
        ("Apple", '1'),
        ("Pear", '2'),
        ("Orange", '3'),
        ("Banana", '4'),
    ]
    return {
        "fruits": fruits,
        "count": len(fruits),
    }
```

(continues on next page)

(continued from previous page)

```
fruits_kbd = Radio(
    Format(" {item[0]}"),  # E.g ` Apple`
    Format(" {item[0]}"),
    id="r_fruits",
    item_id_getter=operator.itemgetter(1),
    items="fruits",
)
```



Useful methods:

- `get_checked` - returns an id of selected items
- `is_checked` - returns if certain id is currently selected
- `set_checked` - sets the selected item by id

2.5.7 Multiselect

Multiselect is another kind of stateful selection widget. It very similar to `Radio` but remembers multiple selected items

Same as for `Radio` you should pass two texts (for checked and unchecked items). Passed data is the same as for `Select`

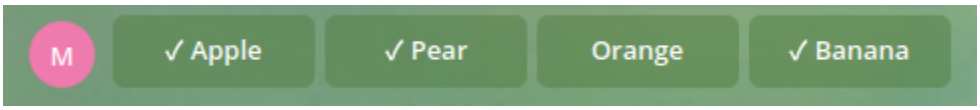
```
import operator

from aiogram_dialog.widgets.kbd import Multiselect
from aiogram_dialog.widgets.text import Format


# let's assume this is our window data getter
async def get_data(**kwargs):
    fruits = [
        ("Apple", '1'),
        ("Pear", '2'),
        ("Orange", '3'),
        ("Banana", '4'),
    ]
    return {
        "fruits": fruits,
        "count": len(fruits),
    }


fruits_kbd = Multiselect(
    Format("✓ {item[0]}"),  # E.g `✓ Apple`
    Format("{item[0]}"),
    id="m_fruits",
    item_id_getter=operator.itemgetter(1),
```

(continues on next page)

(continued from previous page)

```
    items="fruits",
)
```

After few clicks it will look like:



Other useful options are:

- `min_selected` - limits minimal number of selected items ignoring clicks if this restriction is violated. It does not affect initial state.
- `max_selected` - limits maximal number of selected items
- `on_state_changed` - callback function. Called when item changes selected state

To work with selection you can use this methods:

- `get_checked` - returns a list of ids of all selected items
- `is_checked` - returns if certain id is currently selected
- `set_checked` - changes selection state of provided id
- `reset_checked` - resets all checked items to unchecked state

Warning: `Multiselect` widgets stores state of all checked items even if they disappear from window data. It is very useful when you have pagination, but might be unexpected when data is really removed.

2.5.8 Calendar

Calendar widget allows you to display the keyboard in the form of a calendar, flip through the months and select the date. The initial state looks like the days of the current month. It is possible to switch to the state for choosing the month of the current year or in the state of choosing years.

```
from datetime import date

from aiogram.types import CallbackQuery

from aiogram_dialog import DialogManager
from aiogram_dialog.widgets.kbd import Calendar


async def on_date_selected(c: CallbackQuery, widget, manager: DialogManager, selected_
↪date: date):
    await c.answer(str(selected_date))

calendar = Calendar(id='calendar', on_click=on_date_selected)
```


June 2021
Mon
Tue
Wed
Thu
Fri
Sat
Sun
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
Prev month
Zoom out
Next month


year 2021
January
February
March
April
May
June
July
August
September
October
November
December


2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028

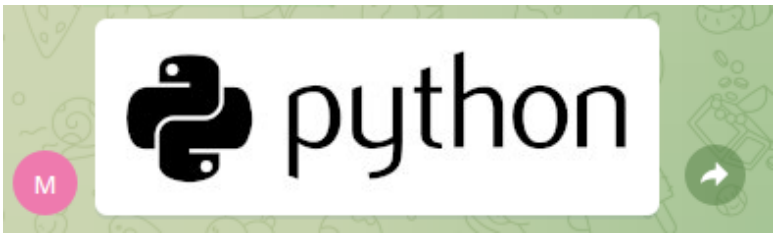
2.6 Media widget types

Currently `StaticMedia` is only existing out of the box media widget. You can use it providing `path` or `url` to the file, it's ContentType and additional parameters if required. Also you might need to change media type (`type=ContentType.Photo`) or provide any additional params supported by aiogram using `media_params`

Be careful using relative paths. Mind the working directory.

```
from aiogram_dialog.widgets.media import StaticMedia

windows = Window(
    StaticMedia(
        path="/home/tishka17/python_logo.png"),
        type=ContentType.PHOTO,
    ),
    state=DialogSG.greeting,
)
```

It will look like:



For more complex cases you can read source code of `StaticMedia` and create your own widget with any logic you need.

Note: Telegram allows to send files using `file_id` instead of uploading same file again. This make media sending much faster. `aiogram_dialog` uses this feature and caches sent file ids in memory

If you want to persistent `file_id` cache, implement `MediaIdStorageProtocol` and pass instance to your dialog registry

---

2.7 Hiding widgets

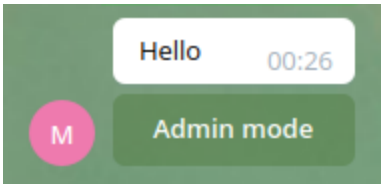
Actually every widget can be hidden including texts, buttons, groups and so on. It is managed by `when` attribute. It can be either a data key or a predicate function

```
from typing import Dict

from aiogram.dispatcher.filters.state import StatesGroup, State

from aiogram_dialog import Window, DialogManager
from aiogram_dialog.widgets.kbd import Button, Row, Group
from aiogram_dialog.widgets.text import Const, Format, Multi
from aiogram_dialog.widgets.when import Whenable
```

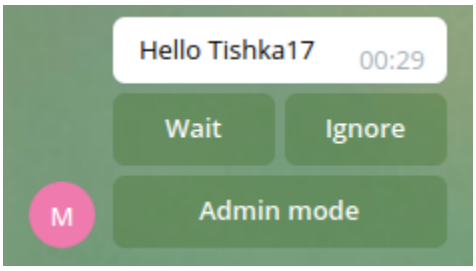
(continues on next page)

(continued from previous page)

```
class MySG(StatesGroup):
    main = State()


async def get_data(**kwargs):
    return {
        "name": "Tishka17",
        "extended": False,
    }


def is_tishka17(data: Dict, widget: Whenable, manager: DialogManager):
    return data.get("name") == "Tishka17"


window = Window(
    Multi(
        Const("Hello"),
        Format("{name}", when="extended"),
        sep=" "
    ),
    Group(
        Row(
            Button(Const("Wait"), id="wait"),
            Button(Const("Ignore"), id="ignore"),
            when="extended",
        ),
        Button(Const("Admin mode"), id="nothing", when=is_tishka17),
    ),
    state=MySG.main,
    getter=get_data,
)
```



If you only change data setting `"extended": True` the window will look differently



CHAPTER

THREE

TRANSITIONS

3.1 Types of transitions

Talking to user you will need to switch between different chat states. It can be done using four types of transitions:

- *State switch* inside dialog. Doing so you will just show another window.
- *Start* a dialog in same stack. In this case dialog will be added to task stack with empty dialog context and corresponding window will be shown instead on previously visible one
- *Start* a dialog in new stack. In this case dialog will be shown in a new message and behave independently from current one.
- *Close* dialog. Dialog will be removed from stack, its data erased. underlying dialog will be shown

3.2 Task stack

To deal with multiple opened dialogs **aiogram_dialog** has such thing as dialog stack. It allows dialogs to be opened one over another ("stacked") so only one of them is visible.

- Each time you start a dialog new task is added on top of a stack and new dialog context is created.
- Each time you close a dialog, task and dialog context are removed.

You can start same dialog multiple times, and multiple contexts (identified by `intent_id`) will be added to stack preserving the order. So you must be careful restarting you dialogs: do not forget to clear stack or it will eat all your memory

Starting with version 1.0 you can create new stacks but default one exists always.

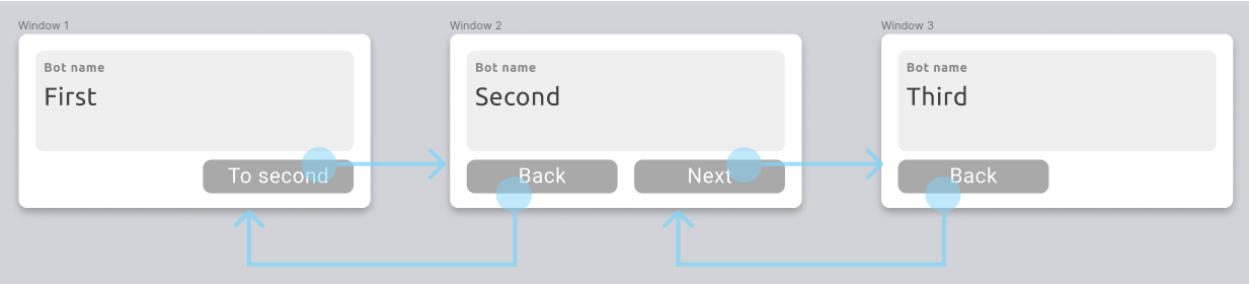
3.3 State switch

Simplest thing you can do to change UI layout is to switch dialog state. It does not affect task stack and just draws another window. Dialog context is kept the same, so all your data is still available.

There are several ways to do it:

- `dialog.switch_to` method. Pass another state and window will be switched
- `dialog.next` method. It will switch to the next window in the same order they were passed during dialog creation. Cannot be called when the last window is active
- `dialog.back` method. Switch to the opposite direction (to the previous one). Cannot be called when the first window is active

Let's create thee windows with buttons and these transitions:



The code may look like:

```
from aiogram.dispatcher.filters.state import StatesGroup, State
from aiogram.types import CallbackQuery

from aiogram_dialog import Dialog, DialogManager, Window
from aiogram_dialog.widgets.kbd import Button, Row
from aiogram_dialog.widgets.text import Const


class DialogSG(StatesGroup):
    first = State()
    second = State()
    third = State()


async def to_second(c: CallbackQuery, button: Button, manager: DialogManager):
    await manager.dialog().switch_to(DialogSG.second)


async def go_back(c: CallbackQuery, button: Button, manager: DialogManager):
    await manager.dialog().back()


async def go_next(c: CallbackQuery, button: Button, manager: DialogManager):
    await manager.dialog().next()


dialog = Dialog(
    Window(
        Const("First"),
        Button(Const("To second"), id="sec", on_click=to_second),
        state=DialogSG.first,
    ),
    Window(
        Const("Second"),
        Row(
            Button(Const("Back"), id="back2", on_click=go_back),
            Button(Const("Next"), id="next2", on_click=go_next),
        ),
        state=DialogSG.second,
    ),
```

(continues on next page)

(continued from previous page)

```
    Window(
        Const("Third"),
        Button(Const("Back"), id="back3", on_click=go_back),
        state=DialogSG.third,
    )
)
```

It is ok to use these methods in message handler or if you have additional logic. But for simple cases it looks too complex. To simplify it we have special types of buttons. Each one can contain custom text if needed:

- `SwitchTo` - calls `switch_to` when clicked. State is provided via constructor attribute
- `Next` - calls `next` when clicked
- `Back` - calls `back` when clicked

An example from above may be rewritten using these buttons:

```
from aiogram.dispatcher.filters.state import StatesGroup, State

from aiogram_dialog import Dialog, Window
from aiogram_dialog.widgets.kbd import Row, SwitchTo, Next, Back
from aiogram_dialog.widgets.text import Const


class DialogSG(StatesGroup):
    first = State()
    second = State()
    third = State()


dialog = Dialog(
    Window(
        Const("First"),
        SwitchTo(Const("To second"), id="sec", state=DialogSG.second),
        state=DialogSG.first,
    ),
    Window(
        Const("Second"),
        Row(
            Back(),
            Next(),
        ),
        state=DialogSG.second,
    ),
    Window(
        Const("Third"),
        Back(),
        state=DialogSG.third,
    )
)
```

Note: You can wonder, why we do not set an id to Back/Next buttons. Though it is normally recommended, these buttons do usually the same action so they have default `id`.

If you have multiple buttons of the same type in a window with `on_click` callback, you should explicitly set different ids.

CHAPTER

FOUR

MIGRATION FROM PREVIOUS VERSIONS

Incompatible changes with 0.11:

- `reset_stack` was replaced with `StartMode`. E.g. `reset_stack=true` is now `mode=StartMode.RESET_STACK`
- dialog no more changes current aiogram state
- **In manager `context` and `current_intent()` were replaced with `current_context()` call.**
  - `dialog_data` is a dict to hold user data
  - `widget_data` is a dict to hold data of widgets
  - `start_data` is a data provided whe dialog start
  - `state` is current dialog state
- When subdialog finishes parent is restored with previous state, not which it was started
- Changed signature of `on_process_result` callback. It now accepts start data used to start subdialog
- `Group.keep_rows` option removed. Set `width=None` (default value) if you want to keep rows.

CHAPTER

FIVE

HELPER TOOLS (EXPERIMENTAL)

5.1 State diagram

You can generate image with your states and transitions.

Firstly you need to install [graphviz](https://graphviz.org/download/) into your system. Check installation instructions on official site.

Install library with tools extras:

```
pip install aiogram_dialog[tools]
```

Import rendering method:

```
from aiogram_dialog.tools import render_transitions
```

Call it passing your registry or list of dialogs:

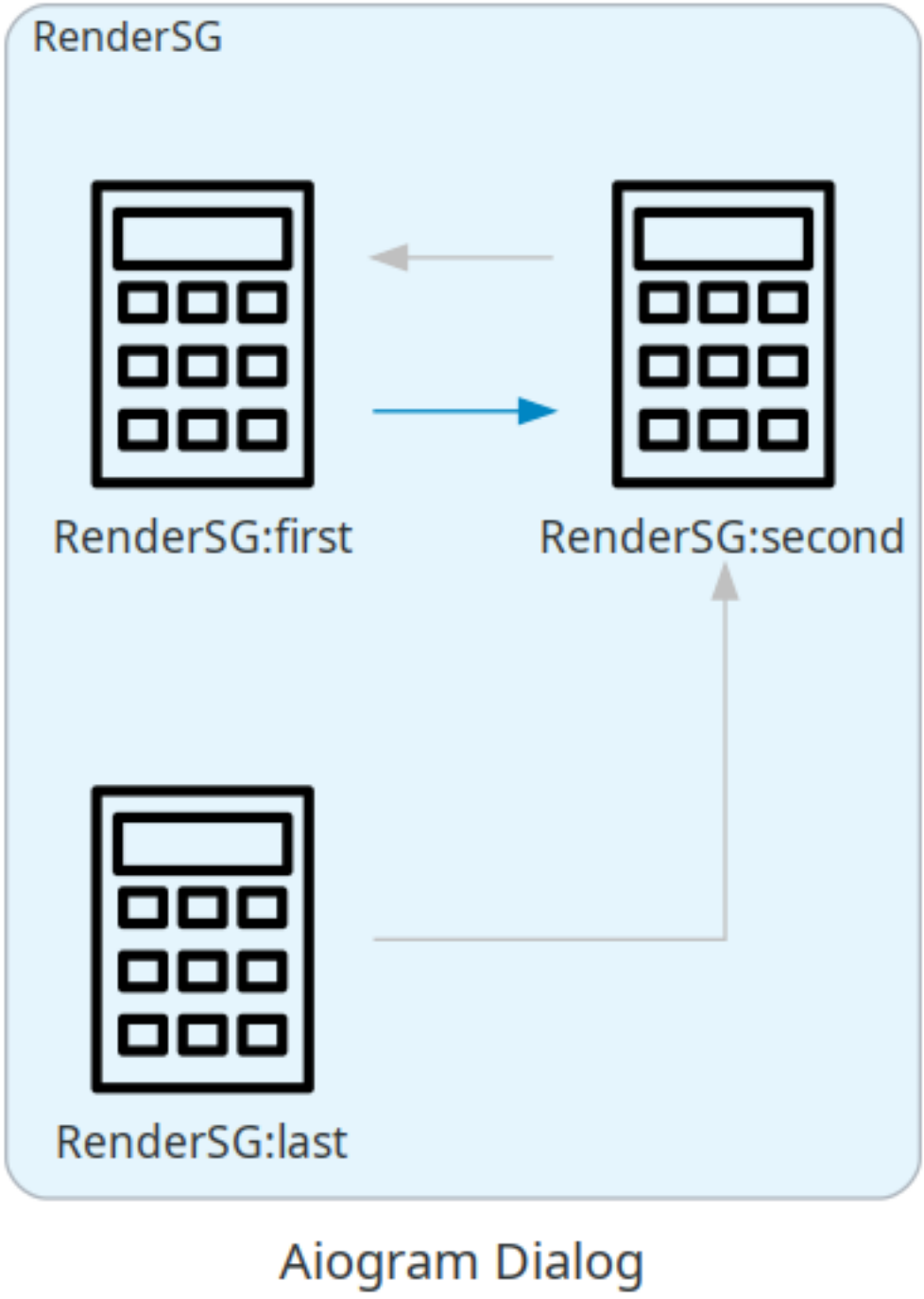
```
from aiogram.dispatcher.filters.state import StatesGroup, State
from aiogram.types import Message

from aiogram_dialog import Dialog, Window, DialogManager
from aiogram_dialog.tools import render_transitions
from aiogram_dialog.widgets.input import MessageInput
from aiogram_dialog.widgets.kbd import Next, Back
from aiogram_dialog.widgets.text import Const


class RenderSG(StatesGroup):
    first = State()
    second = State()
    last = State()


async def on_input(m: Message, dialog: Dialog, manager: DialogManager):
    manager.current_context().dialog_data["name"] = m.text
    await dialog.next()


dialog = Dialog(
    Window(
        Const("1. First"),
```

(continues on next page)

(continued from previous page)

```
        Next(),
        state=RenderSG.first,
    ),
    Window(
        Const("2. Second"),
        Back(),
        MessageInput(on_input),
        state=RenderSG.second,
    ),
    Window(
        Const("3. Last"),
        Back(),
        state=RenderSG.last,
    ),
)

# this is diagram rendering
render_transitions([dialog])
```

Run your code and you will get `aiogram_dialog.png` in working directory:


RenderSG
RenderSG:first
RenderSG:second
RenderSG:last
Aiogram Dialog

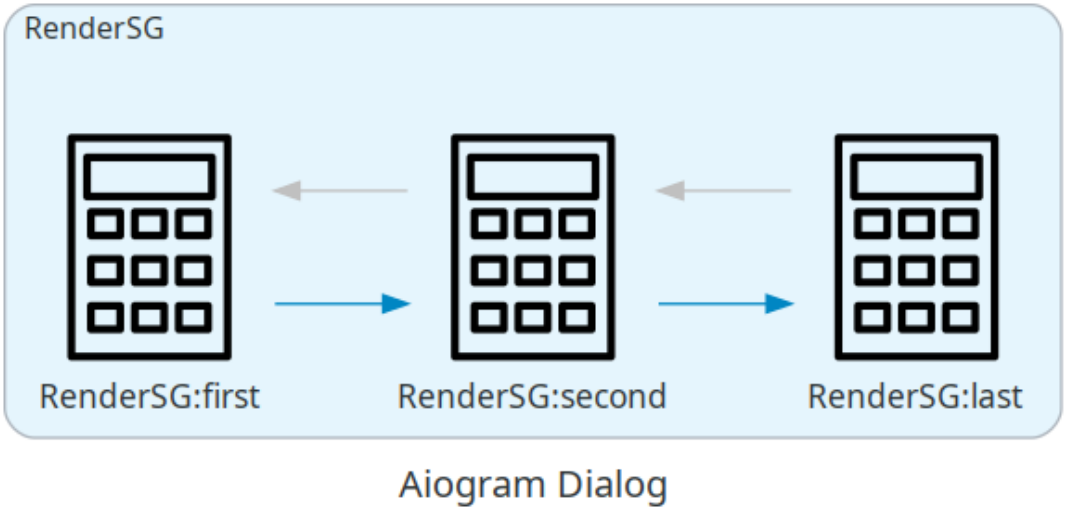
5.1.1 State transition hints

You may notice, that not all transitions are show on diagram. This is because library cannot analyze source code of you callbacks. Only transitions, done by special buttons are shown.

To fix this behavior you can set `preview_add_transitions` parameter of window:

```
from aiogram.dispatcher.filters.state import StatesGroup, State
from aiogram.types import Message

from aiogram_dialog import Dialog, Window, DialogManager
from aiogram_dialog.tools import render_transitions
from aiogram_dialog.widgets.input import MessageInput
from aiogram_dialog.widgets.kbd import Next, Back
from aiogram_dialog.widgets.text import Const


class RenderSG(StatesGroup):
    first = State()
    second = State()
    last = State()


async def on_input(m: Message, dialog: Dialog, manager: DialogManager):
    manager.current_context().dialog_data["name"] = m.text
    await dialog.next()  # rendering tool cannot detect this call


dialog = Dialog(
    Window(
        Const("1. First"),
        Next(),
        state=RenderSG.first,
    ),
    Window(
        Const("2. Second"),
        Back(),
        MessageInput(on_input),
        state=RenderSG.second,
        preview_add_transitions=[Next()],  # this is a hint for rendering tool
    ),
    Window(
        Const("3. Last"),
        Back(),
        state=RenderSG.last,
    ),
)

render_transitions([dialog])
```

Run the code and check updated rendering result:



5.2 Dialogs preview

Import rendering method:

```
from aiogram_dialog.tools import render_preview
```

Add some data to be shown on preview using `preview_data` parameter of window:

```
class RenderSG(StatesGroup):
    first = State()


dialog = Dialog(
    Window(
        Format("Hello, {name}"),
        Cancel(),
        state=RenderSG.first,
        preview_data={"name": "Tishka17"},
    ),
)
```

Call it passing your registry and filename somewhere inside your asyncio code:

```

await render_transitions(registry, "preview.html")
```

Together it will be something like this:

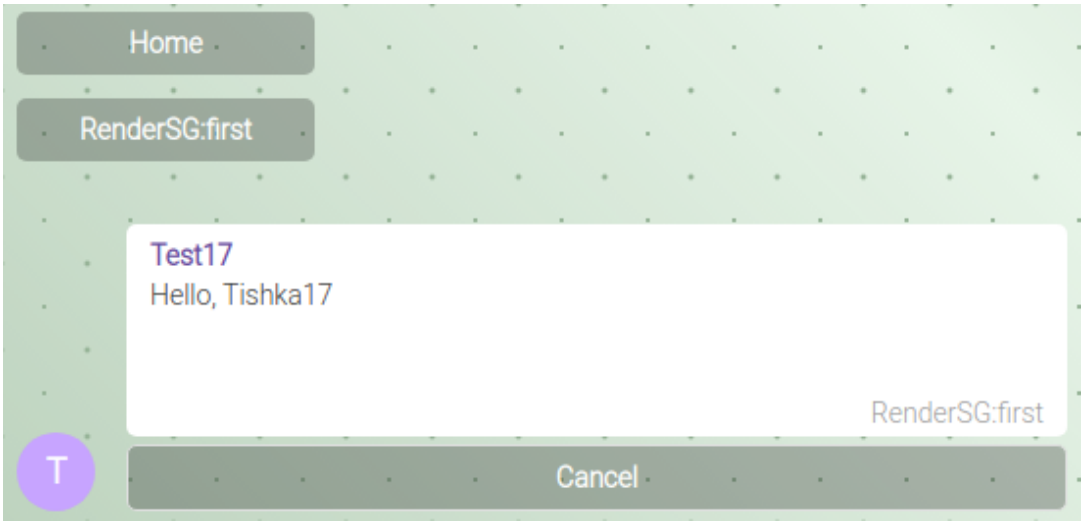
```
import asyncio
```

(continues on next page)

(continued from previous page)

```
from aiogram import Bot, Dispatcher
from aiogram.contrib.fsm_storage.memory import MemoryStorage
from aiogram.dispatcher.filters.state import StatesGroup, State

from aiogram_dialog import Dialog, Window, DialogRegistry
from aiogram_dialog.tools import render_preview
from aiogram_dialog.widgets.kbd import Cancel
from aiogram_dialog.widgets.text import Format


class RenderSG(StatesGroup):
    first = State()


dialog = Dialog(
    Window(
        Format("Hello, {name}"),
        Cancel(),
        state=RenderSG.first,
        preview_data={"name": "Tishka17"},
    ),
)

storage = MemoryStorage()
bot = Bot(token='BOT TOKEN HERE')
dp = Dispatcher(bot, storage=storage)
registry = DialogRegistry(dp)
registry.register(dialog)


async def main():
    await render_preview(registry, "preview.html")


if __name__ == '__main__':
    asyncio.run(main())
```

As a result you will see a html file in working directory, that can be opened in browser to preview how all dialogs will look like.


Home
RenderSG:first
Test17
Hello, Tishka17
RenderSG:first
T
Cancel

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search