Introduction

What is Flet?

Flet is a framework that allows building web, desktop and mobile applications in Python without prior experience in frontend development.

You can build a UI for your program with Flet controls which are based on Flutter by Google. Flet goes beyond merely wrapping Flutter widgets. It adds its own touch by combining smaller widgets, simplifying complexities, implementing UI best practices, and applying sensible defaults. This ensures that your applications look stylish and polished without requiring additional design efforts on your part.

Flet app example

Create a sample "Counter" app:

counter.py

```
import flet as ft

def main(page: ft.Page):
    page.title = "Flet counter example"
    page.vertical_alignment = ft.MainAxisAlignment.CENTER

    txt_number = ft.TextField(value="0", text_align=ft.TextAlign.RIGHT,
width=100)

    def minus_click(e):
        txt_number.value = str(int(txt_number.value) - 1)
        page.update()

    def plus_click(e):
        txt_number.value = str(int(txt_number.value) + 1)
        page.update()

    page.add(
        ft.Row(
```

```
            [
                ft.IconButton(ft.icons.REMOVE, on_click=minus_click),
                txt_number,
                ft.IconButton(ft.icons.ADD, on_click=plus_click),
            ],
            alignment=ft.MainAxisAlignment.CENTER,
        )
    )

ft.app(main)
```

To run the app install `flet` module (create a new Flet environment):

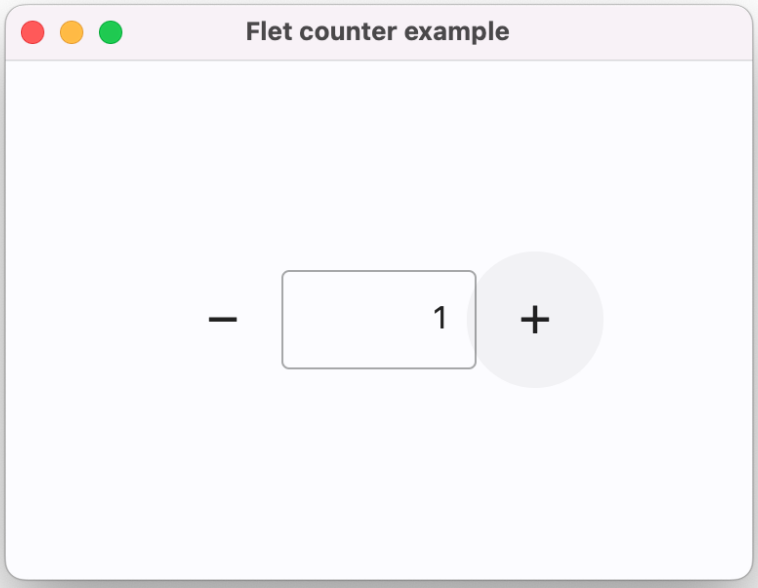
```
pip install flet
```

and run the program:

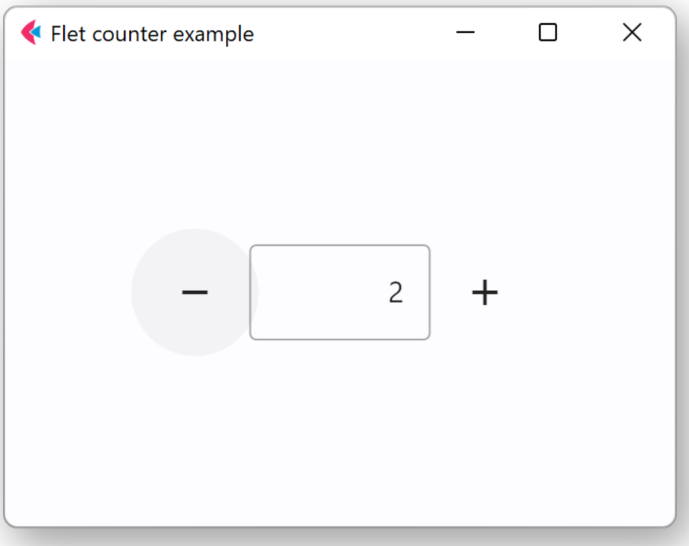
```
flet run counter.py
```

The app will be started in a native OS window - what a nice alternative to Electron!

macOS



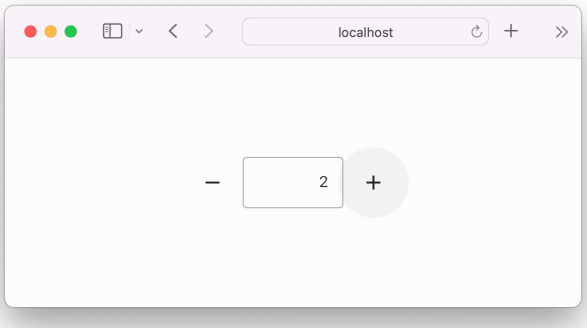
Windows



Now, run your app as a web app:

```
flet run --web counter.py
```

A new browser window or tab will be opened:



Edit this page

Getting started

Before you can create your first Flet app you need to setup your development environment which requires Python 3.8 or above and `flet` package.

We recommend installing Flet in a virtual environment which can be done in a number of different ways.

Prerequisites

Linux

If installing Flet on Linux, there are additional prerequisites.

WSL

Flet apps can be run on WSL 2 (Windows Subsystem for Linux 2). If you are getting `cannot open display` error following this guide for troubleshooting.

Python venv module

You can create virtual environment by running the following commands in your terminal:

macOS | Linux | Windows

```
mkdir first-flet-app
cd first-flet-app
python3 -m venv .venv
source .venv/bin/activate
```

Once you activated virtual environment, you'll see that your prompt now shows `(.venv)` prefix.

Now you can install the latest version of Flet in `.venv` virtual environment:

```
pip install flet
```

To check what version of Flet was installed:

```
flet --version
```

You can read more about Python `venv` module here.

Now you are ready to create your first Flet app.

Poetry

Another way to setup a virtual environment for your Flet project is using Poetry.

Once you have Poetry installed, run the following command in your terminal:

```
poetry new first-flet-app
```

This command will create a new directory called first-flet-app with the following structure:

```
first-flet-app/
├── pyproject.toml
├── README.md
├── first-flet-app/
│   └── __init__.py
└── tests/
    └── __init__.py
```

Now you can add Flet dependency to your project:

```
cd first-flet-app
poetry add flet
```

To check what version of Flet was installed:

```
poetry run flet --version
```

Now you are ready to create your first Flet app.

> **NOTE**
>
> When creating and running Flet app using Poetry, you'll need to use `poetry run` before each command!

Edit this page

Create a new Flet app

To create a new "minimal" Flet app run the following command:

```
flet create <project-name>
```

for example:

```
flet create my_flet_app
```

`<project-name>` will be used as a name of output directory.

Flet will create `<project-name>` directory with the following `main.py`:

```
import flet as ft

def main(page: ft.Page):
    page.add(ft.SafeArea(ft.Text("Hello, Flet!")))

ft.app(main)
```

> **NOTE**
>
> To create your Flet app in current directory, run the following command:
>
> ```
> flet create .
> ```

Flet program has `main()` function where you would add UI elements (controls) to a page or a window. The application ends with a blocking `ft.app()` function which initializes Flet app and runs `main()`.

To create a new Flet app from "counter" template run the following command:

```
flet create --template counter <project-name>
```

Or, to create Flet app from counter template in your current directory, run this command:

```
flet create --template counter .
```

You can find more information about `flet create` command here.

Now let's see Flet in action by running the app!

Edit this page

Running Flet app

Flet app can be run as a desktop or web app using a single `flet run` command.

Run as a desktop app

To run Flet app as a desktop app, use the following command:

```
flet run
```

This command will run `main.py` located in the current directory.

If you need to provide a different path to the file, use the following command:

```
flet run [script]
```

To run `main.py` located in a different directory, provide an absolute or a relative path to the directory where it is located, for example:

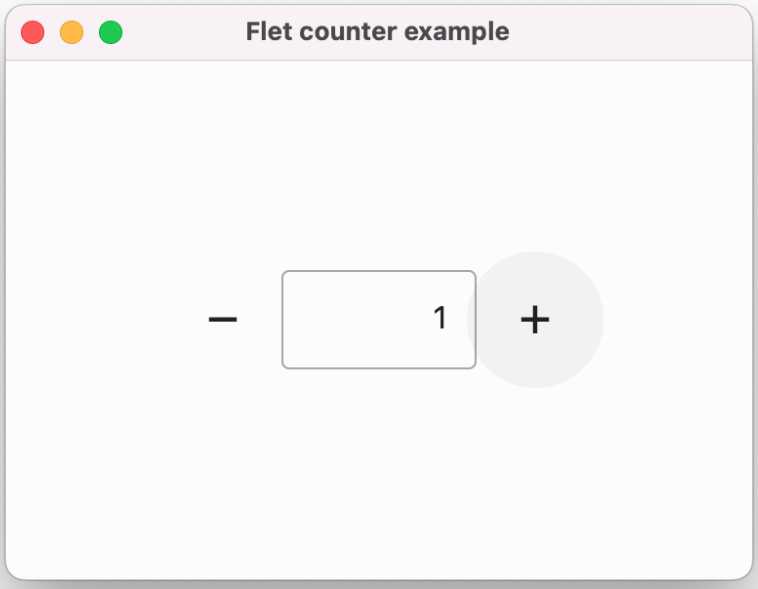
```
flet run /Users/JohnSmith/Documents/projects/flet-app
```

To run script with a name other than `main.py`, provide an absolute or a relative path to the file, for example:

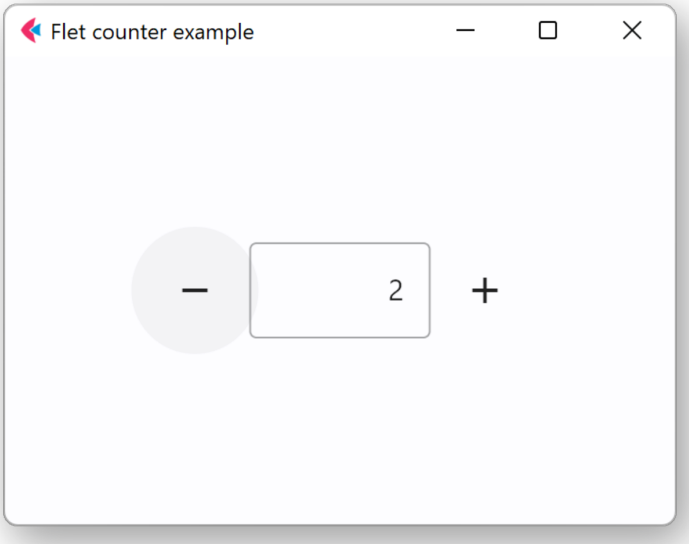
```
flet run counter.py
```

The app will be started in a native OS window:

macOS



Windows

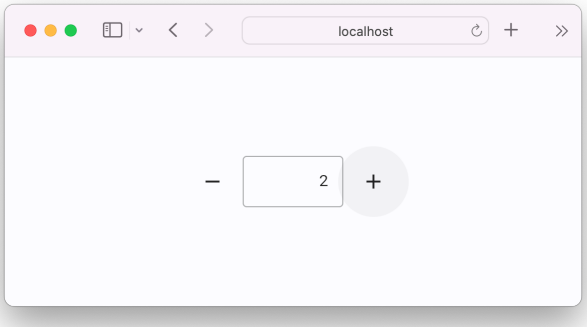


Run as a web app

To run Flet app as a web app, use the following command:

```
flet run --web [script]
```

A new browser window/tab will be opened and the app will be using a random TCP port:



To run on a fixed port use `--port` (`-p`) option, for example:

```
flet run --web --port 8000 app.py
```

Hot reload

By default, Flet will watch the script file that was run and reload the app whenever the file is changed and saved, but will not watch for changes in other files.

To watch all the files in the same directory, use the following command:

```
poetry run flet run -d [script]
```

To watch script directory and all sub-directories recursively, use the following command:

```
poetry run flet run -d -r [script]
```

You can find more information about `flet run` command here.

Edit this page

Flet controls

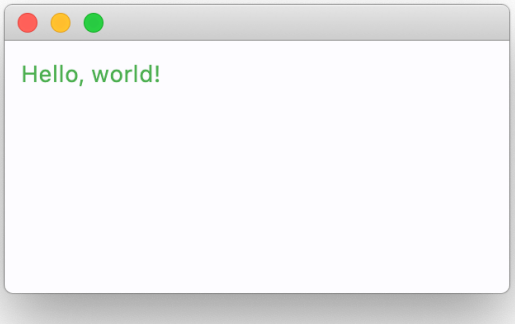
User interface is made of **Controls** (aka widgets). To make controls visible to a user they must be added to a `Page` or inside other controls. Page is the top-most control. Nesting controls into each other could be represented as a tree with Page as a root.

Controls are just regular Python classes. Create control instances via constructors with parameters matching their properties, for example:

```
t = ft.Text(value="Hello, world!", color="green")
```

To display control on a page add it to `controls` list of a Page and call `page.update()` to send page changes to a browser or desktop client:

```
import flet as ft

def main(page: ft.Page):
    t = ft.Text(value="Hello, world!", color="green")
    page.controls.append(t)
    page.update()

ft.app(main)
```



> **NOTE**
>
> In the following examples we will be showing just the contents of `main` function.

You can modify control properties and the UI will be updated on the next `page.update()`:

```
t = ft.Text()
page.add(t) # it's a shortcut for page.controls.append(t) and then
page.update()

for i in range(10):
    t.value = f"Step {i}"
    page.update()
    time.sleep(1)
```

Some controls are "container" controls (like Page) which could contain other controls. For example, `Row` control allows arranging other controls in a row one-by-one:

```
page.add(
    ft.Row(controls=[
        ft.Text("A"),
        ft.Text("B"),
        ft.Text("C")
    ])
)
```

or `TextField` and `ElevatedButton` next to it:

```
page.add(
    ft.Row(controls=[
        ft.TextField(label="Your name"),
        ft.ElevatedButton(text="Say my name!")
    ])
)
```

`page.update()` is smart enough to send only the changes made since its last call, so you can add a couple of new controls to the page, remove some of them, change other controls' properties and then call `page.update()` to do a batched update, for example:

```
for i in range(10):
    page.controls.append(ft.Text(f"Line {i}"))
    if i > 4:
        page.controls.pop(0)
```

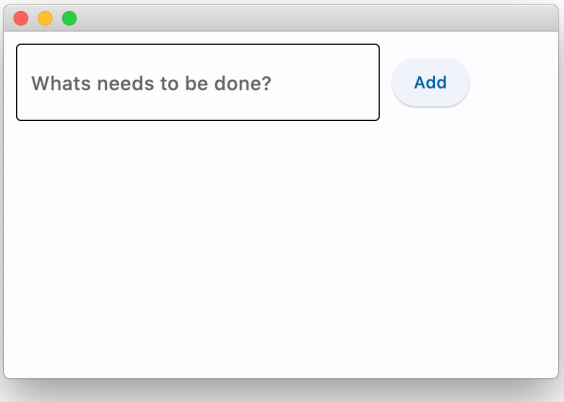
```
    page.update()
    time.sleep(0.3)
```

Some controls, like buttons, could have event handlers reacting on a user input, for example `ElevatedButton.on_click`:

```
def button_clicked(e):
    page.add(ft.Text("Clicked!"))

page.add(ft.ElevatedButton(text="Click me", on_click=button_clicked))
```

and more advanced example for a simple To-Do:

```
import flet as ft

def main(page):
    def add_clicked(e):
        page.add(ft.Checkbox(label=new_task.value))
        new_task.value = ""
        new_task.focus()
        new_task.update()

    new_task = ft.TextField(hint_text="What's needs to be done?",
width=300)
    page.add(ft.Row([new_task, ft.ElevatedButton("Add",
on_click=add_clicked)]))

ft.app(main)
```



INFO

> Flet implements *imperative* UI model where you "manually" build application UI with stateful controls and then mutate it by updating control properties. Flutter implements *declarative* model where UI is automatically re-built on application data changes. Managing application state in modern frontend applications is inherently complex task and Flet's "old-school" approach could be more attractive to programmers without frontend experience.

`visible` property

Every control has `visible` property which is `true` by default - control is rendered on the page. Setting `visible` to `false` completely prevents control (and all its children if any) from rendering on a page canvas. Hidden controls cannot be focused or selected with a keyboard or mouse and they do not emit any events.

`disabled` property

Every control has `disabled` property which is `false` by default - control and all its children are enabled. `disabled` property is mostly used with data entry controls like `TextField`, `Dropdown`, `Checkbox`, buttons. However, `disabled` could be set to a parent control and its value will be propagated down to all children recursively.

For example, if you have a form with multiple entry control you can set `disabled` property for each control individually:

```
first_name = ft.TextField()
last_name = ft.TextField()
first_name.disabled = True
last_name.disabled = True
page.add(first_name, last_name)
```

or you can put form controls into container, e.g. `Column` and then set `disabled` for the column:

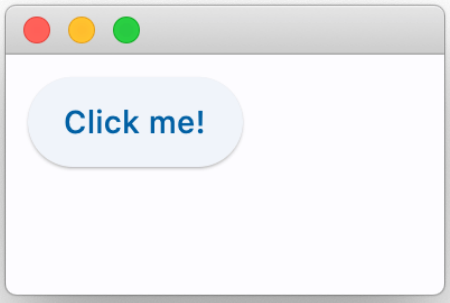
```
first_name = ft.TextField()
last_name = ft.TextField()
c = ft.Column(controls=[
    first_name,
    last_name
])
```

```
c.disabled = True
page.add(c)
```

Buttons

`Button` is the most essential input control which generates `click` event when pressed:

```
btn = ft.ElevatedButton("Click me!")
page.add(btn)
```



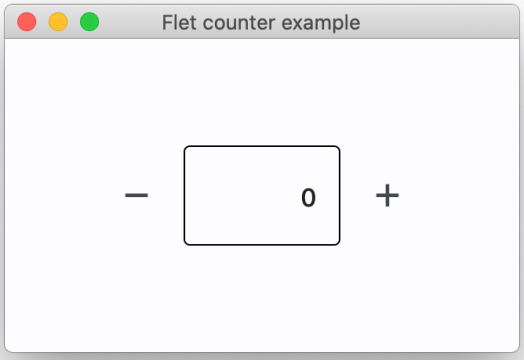
All events generated by controls on a web page are continuously sent back to your script, so how do you respond to a button click?

Event handlers

Buttons with events in "Counter" app:

```
import flet as ft

def main(page: ft.Page):
    page.title = "Flet counter example"
    page.vertical_alignment = ft.MainAxisAlignment.CENTER

    txt_number = ft.TextField(value="0", text_align="right", width=100)

    def minus_click(e):
        txt_number.value = str(int(txt_number.value) - 1)
```

```
        page.update()

    def plus_click(e):
        txt_number.value = str(int(txt_number.value) + 1)
        page.update()

    page.add(
        ft.Row(
            [
                ft.IconButton(ft.icons.REMOVE, on_click=minus_click),
                txt_number,
                ft.IconButton(ft.icons.ADD, on_click=plus_click),
            ],
            alignment=ft.MainAxisAlignment.CENTER,
        )
    )

ft.app(main)
```



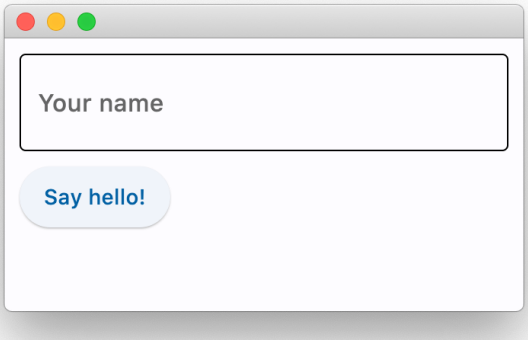
Textbox

Flet provides a number of controls for building forms: TextField, Checkbox, Dropdown, ElevatedButton.

Let's ask a user for a name:

```
import flet as ft

def main(page):
    def btn_click(e):
        if not txt_name.value:
```

```
            txt_name.error_text = "Please enter your name"
            page.update()
        else:
            name = txt_name.value
            page.clean()
            page.add(ft.Text(f"Hello, {name}!"))

    txt_name = ft.TextField(label="Your name")

    page.add(txt_name, ft.ElevatedButton("Say hello!",
on_click=btn_click))

ft.app(main)
```



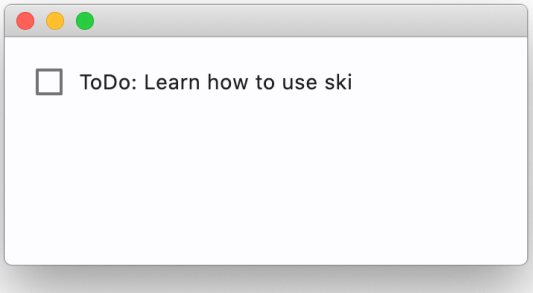
Checkbox

The Checkbox control provides you with various properties and events emmiters for ease of use.

Let's create a one checkbox ToDo:

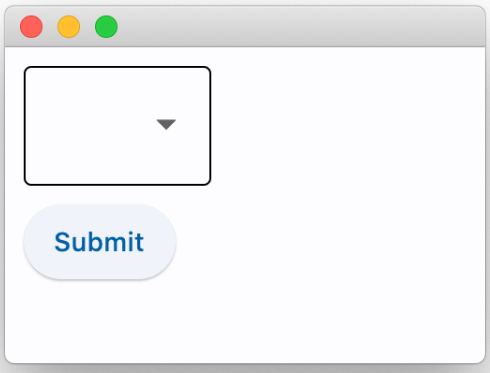
```
import flet as ft


def main(page):
    def checkbox_changed(e):
        output_text.value = (
            f"You have learned how to ski :  {todo_check.value}."
        )
        page.update()

    output_text = ft.Text()
```

```
    todo_check = ft.Checkbox(label="ToDo: Learn how to use ski",
value=False, on_change=checkbox_changed)
    page.add(todo_check, output_text)

ft.app(main)
```



Dropdown

```
import flet as ft


def main(page: ft.Page):
    def button_clicked(e):
        output_text.value = f"Dropdown value is:
{color_dropdown.value}"
        page.update()

    output_text = ft.Text()
    submit_btn = ft.ElevatedButton(text="Submit",
on_click=button_clicked)
    color_dropdown = ft.Dropdown(
        width=100,
        options=[
            ft.dropdown.Option("Red"),
            ft.dropdown.Option("Green"),
            ft.dropdown.Option("Blue"),
        ],
    )
    page.add(color_dropdown, submit_btn, output_text)

ft.app(main)
```



Edit this page

Custom controls

While Flet provides 100+ built-in controls that can be used on their own, the real beauty of programming with Flet is that all those controls can be utilized for creating your own reusable UI components using Python object-oriented programming concepts.

You can create custom controls in Python by styling and/or combining existing Flet controls.

Styled controls

The most simple custom control you can create is a styled control, for example, a button of a certain color and behaviour that will be used multiple times throughout your app.

To create a styled control, you need to create a new class in Python that inherits from the Flet control you are going to customize, `ElevatedButton` in this case:

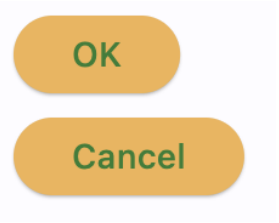
```
class MyButton(ft.ElevatedButton)
    def __init__(self, text):
        super().__init__()
        self.bgcolor = ft.colors.ORANGE_300
        self.color = ft.colors.GREEN_800
        self.text = text
```

Your control has a constructor to customize properties and events and pass custom data. Note that you must call `super().__init__()` in your own constructor to have access to the properties and methods of the Flet control from which you inherit.

Now you can use your brand-new control in your app:

```
import flet as ft

def main(page: ft.Page):

    page.add(MyButton(text="OK"), MyButton(text="Cancel"))
```

```
ft.app(main)
```



See example of using styled controls in Calculator App tutotial.

Handling events

Similar to properties, you can pass event handlers as parameters into your custom control class constructor:

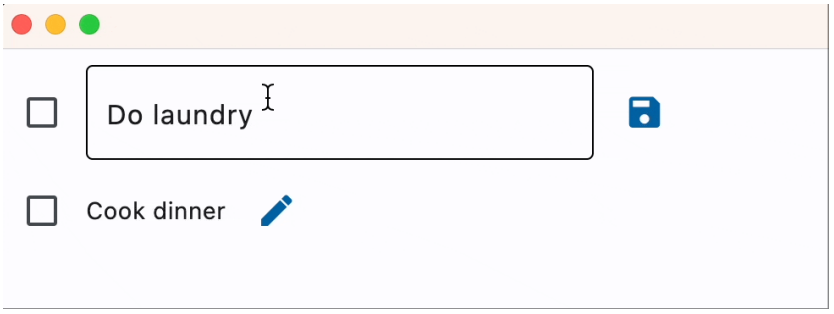
```
import flet as ft

class MyButton(ft.ElevatedButton):
    def __init__(self, text, on_click):
        super().__init__()
        self.bgcolor = ft.colors.ORANGE_300
        self.color = ft.colors.GREEN_800
        self.text = text
        self.on_click = on_click

def main(page: ft.Page):

    def ok_clicked(e):
        print("OK clicked")

    def cancel_clicked(e):
        print("Cancel clicked")

    page.add(
        MyButton(text="OK", on_click=ok_clicked),
        MyButton(text="Cancel", on_click=cancel_clicked),
    )

ft.app(main)
```

Composite controls

Composite custom controls inherit from container controls such as `Column`, `Row`, `Stack` or even `View` to combine multiple Flet controls. The example below is a `Task` control that can be used in a To-Do app:

```
import flet as ft
class Task(ft.Row):
    def __init__(self, text):
        super().__init__()
        self.text_view = ft.Text(text)
        self.text_edit = ft.TextField(text, visible=False)
        self.edit_button = ft.IconButton(icon=ft.icons.EDIT,
on_click=self.edit)
        self.save_button = ft.IconButton(
            visible=False, icon=ft.icons.SAVE, on_click=self.save
        )
        self.controls = [
            ft.Checkbox(),
            self.text_view,
            self.text_edit,
            self.edit_button,
            self.save_button,
        ]

    def edit(self, e):
        self.edit_button.visible = False
        self.save_button.visible = True
        self.text_view.visible = False
        self.text_edit.visible = True
        self.update()

    def save(self, e):
        self.edit_button.visible = True
        self.save_button.visible = False
        self.text_view.visible = True
        self.text_edit.visible = False
        self.text_view.value = self.text_edit.value
        self.update()

def main(page: ft.Page):

    page.add(
        Task(text="Do laundry"),
        Task(text="Cook dinner"),
    )
```

```
ft.app(main)
```



You can find more examples of composite custom controls in community examples and flet-contrib repos.

Life-cycle methods

Custom controls provide life-cycle "hook" methods that you may need to use for different use cases in your app.

`build()`

`build()` method is called when the control is being created and assigned its `self.page`.

Override `build()` method if you need to implement logic that cannot be executed in control's constructor because it requires access to the `self.page`. For example, choose the right icon depending on `self.page.platform` for your adaptive app.

`did_mount()`

`did_mount()` method is called after the control is added to the page and assigned transient `uid`.

Override `did_mount()` method if you need to implement logic that needs to be executed after the control was added to the page, for example Weather widget which calls Open Weather API every minute to update itself with the new weather conditions.

`will_unmount()`

`will_unmount()` method is called before the control is removed from the page.

Override `will_unmount()` method to execute clean-up code.

`before_update()`

`before_update()` method is called every time when the control is being updated.

Make sure not to call `update()` method within `before_update()`.

Isolated controls

Custom control has `is_isolated` property which defaults to `False`.

If you set `is_isolated` to `True`, your control will be isolated from outside layout, i.e. when `update()` method is called for the parent control, the control itself will be updated but any changes to the controls' children are not included into the update digest. Isolated controls should call `self.update()` to push its changes to a Flet page.

As a best practice, any custom control that calls `self.update()` inside its class methods should be isolated.

In the above examples, simple styled `MyButton` doesn't need to be isolated, but the `Task` should be:

```
class Task(ft.Row):
    def __init__(self, text):
        super().__init__()
        self.isolated = True
```

Adaptive apps

Flet framework allows you to develop adaptive apps which means having a single codebase that will deliver different look depending on the device's platform.

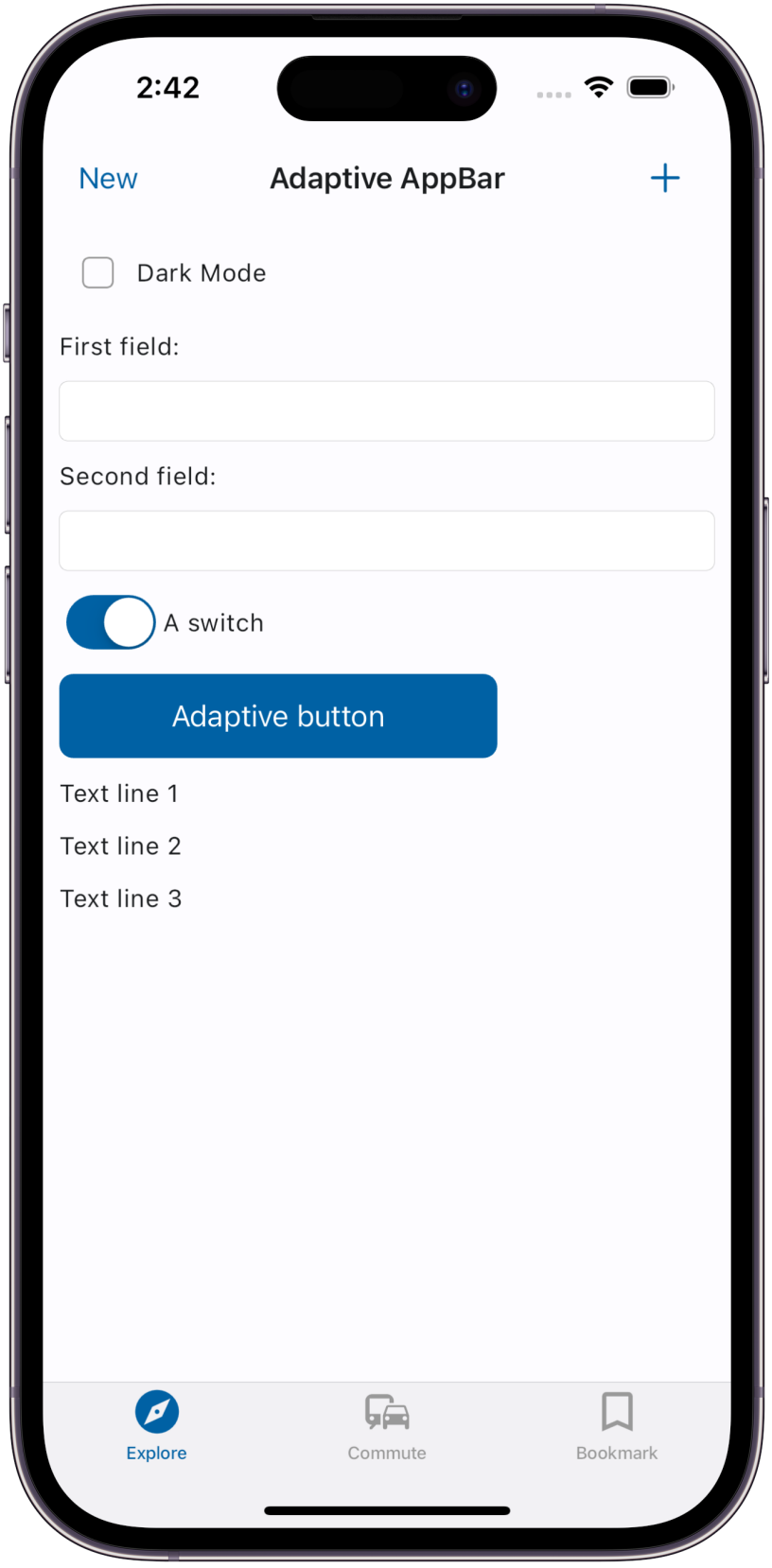
Below is the example of a very simple app that has a different look on iOS and Android platforms:

```
import flet as ft


def main(page):

    page.adaptive = True

    page.appbar = ft.AppBar(
        leading=ft.TextButton("New", style=ft.ButtonStyle(padding=0)),
        title=ft.Text("Adaptive AppBar"),
        actions=[
            ft.IconButton(ft.cupertino_icons.ADD,
style=ft.ButtonStyle(padding=0))
        ],
        bgcolor=ft.colors.with_opacity(0.04,
ft.cupertino_colors.SYSTEM_BACKGROUND),
    )

    page.navigation_bar = ft.NavigationBar(
        destinations=[
            ft.NavigationBarDestination(icon=ft.icons.EXPLORE,
label="Explore"),
            ft.NavigationBarDestination(icon=ft.icons.COMMUTE,
label="Commute"),
            ft.NavigationBarDestination(
                icon=ft.icons.BOOKMARK_BORDER,
                selected_icon=ft.icons.BOOKMARK,
                label="Bookmark",
            ),
        ],
        border=ft.Border(
            top=ft.BorderSide(color=ft.cupertino_colors.SYSTEM_GREY2,
width=0)
```

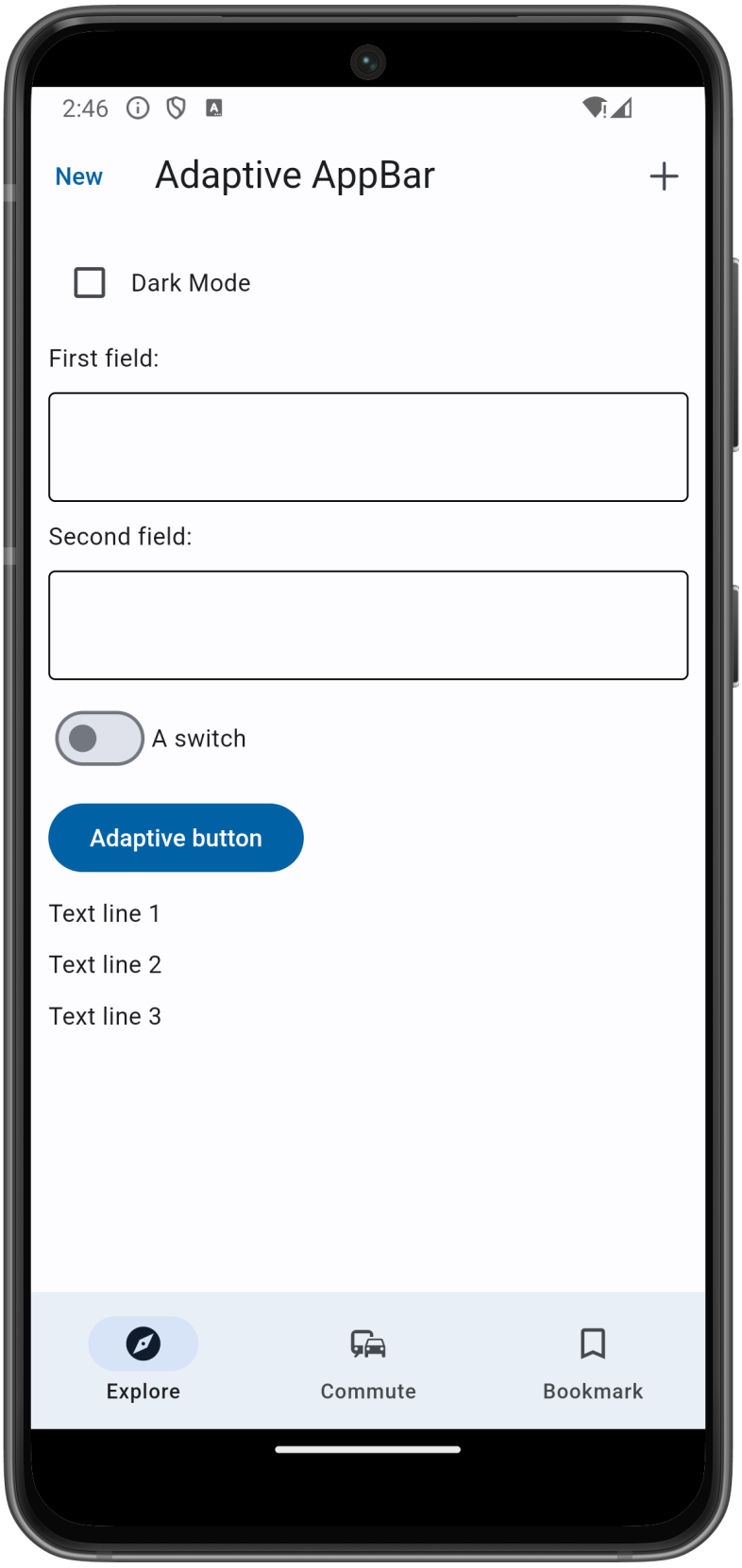
```
        ),
    )

    page.add(
        ft.SafeArea(
            ft.Column(
                [
                    ft.Checkbox(value=False, label="Dark Mode"),
                    ft.Text("First field:"),
                    ft.TextField(keyboard_type=ft.KeyboardType.TEXT),
                    ft.Text("Second field:"),
                    ft.TextField(keyboard_type=ft.KeyboardType.TEXT),
                    ft.Switch(label="A switch"),
                    ft.FilledButton(content=ft.Text("Adaptive
button")),
                    ft.Text("Text line 1"),
                    ft.Text("Text line 2"),
                    ft.Text("Text line 3"),
                ]
            )
        )
    )


ft.app(main)
```

By setting just `page.adaptive = True` you can make you app looking awesome on both iOS and Android devices:

iPhone


2:42
New
Adaptive AppBar
Dark Mode
First field:
Second field:
A switch
Adaptive button
Text line 1
Text line 2
Text line 3
Explore
Commute
Bookmark

Android



Material and Cupertino controls

Most of Flet controls are based on Material design.

There is also a number of iOS-style controls in Flet that are called Cupertino controls.

Cupertino controls usually have a matching Material control that has `adaptive` property which defaults to `False`. When using a Material control with `adaptive` property set to `True`, a different control will be created depending on the platform, for example:

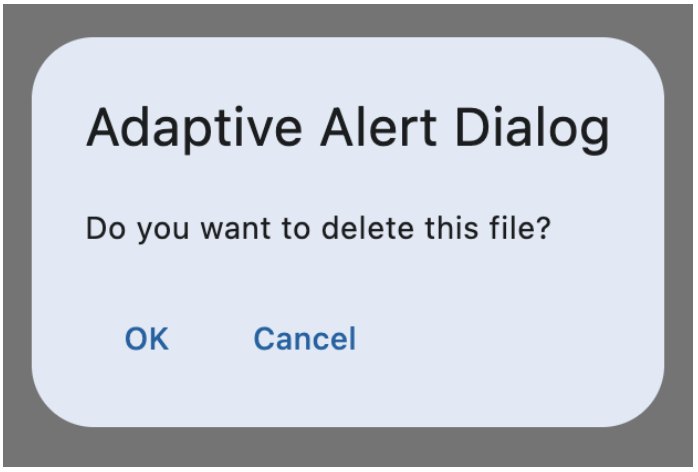
```
ft.Checkbox(adaptive=True, value=True, label="Adaptive Checkbox")
```

Flet checks the value of `page.platform` property and if it is `ft.PagePlatform.IOS` or `ft.PagePlatform.MACOS`, Cupertino control will be created; in all other cases Material control will be created.

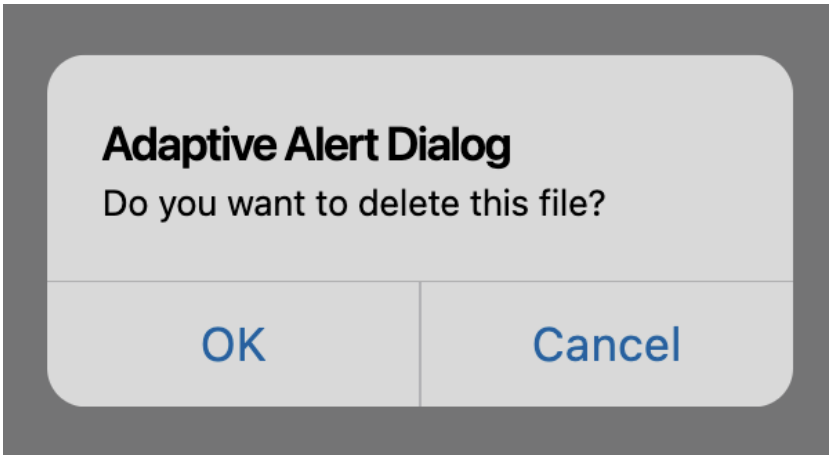
> **NOTE**
>
> `adaptive` property can be set for an individual control or a container control such as `Row`, `Column` or any other control that has `content` or `controls` property. If container control is adaptive, all its child controls will be adaptive, unless `adaptive` property is explicitly set to `False` for a child control.

Below is the list of adaptive Material controls and their matching Cupertino controls:

AlertDialog



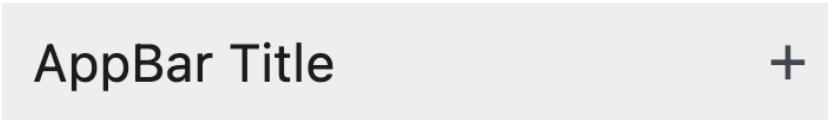
CupertinoAlertDialog



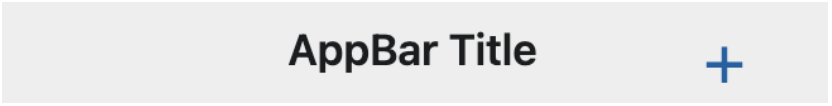
Any button in Dialog actions

CupertinoDialogAction

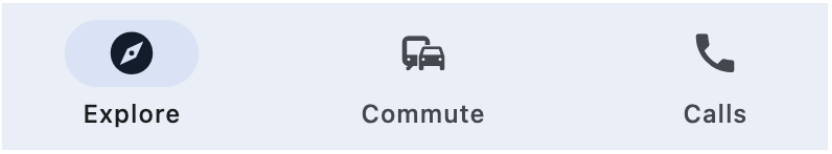
AppBar



CupertinoAppBar

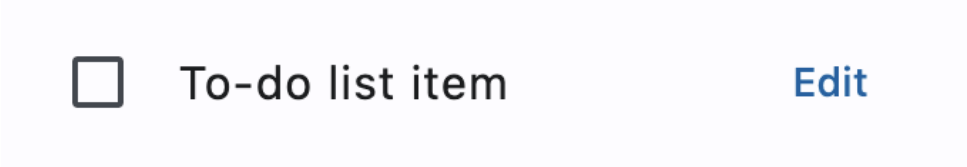


NavigationBar

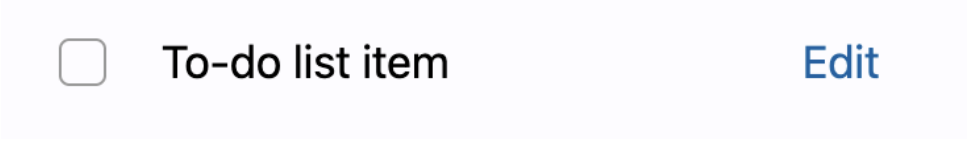


CupertinoNavigationBar

ListTile



CupertinoListTile



TextField

CupertinoTextField

Checkbox

CupertinoCheckbox

Slider

CupertinoSlider

Switch

CupertinoSwitch

Radio

CupertinoRadio

FilledButton

OK

CupertinoFilledButton

OK

FilledTonalButton

OK

CupertinoButton

OK

IconButton

CupertinoButton

ElevatedButton

OK

CupertinoButton

OK

OutlinedButton

OK

TextButton

OK

Custom adaptive controls

While Flet offers a number of controls that will be adapted to a platform automatically using their `adaptive` property, there will be cases when you need more specific adaptive UI presentation, for example, using different icon, background color, padding etc. depending on the platform.

With Flet, you can create your own reusable custom controls in Python that will inherit from a Flet control and implement specific properties you need. In the example below, we are creating a new `AdaptiveNavigationBarDestination` control that will be displaying different icon on iOS and Android:

```
class AdaptiveNavigationBarDestination(ft.NavigationBarDestination):
    def __init__(self, ios_icon, android_icon, label):
        super().__init__()
        self._ios_icon = ios_icon
        self._android_icon = android_icon
        self.label = label

    def build(self):
        # we can check for platform in build method because self.page
is known
        self.icon = (
            self._ios_icon
            if self.page.platform == ft.PagePlatform.IOS
            or self.page.platform == ft.PagePlatform.MACOS
            else self._android_icon
        )
```

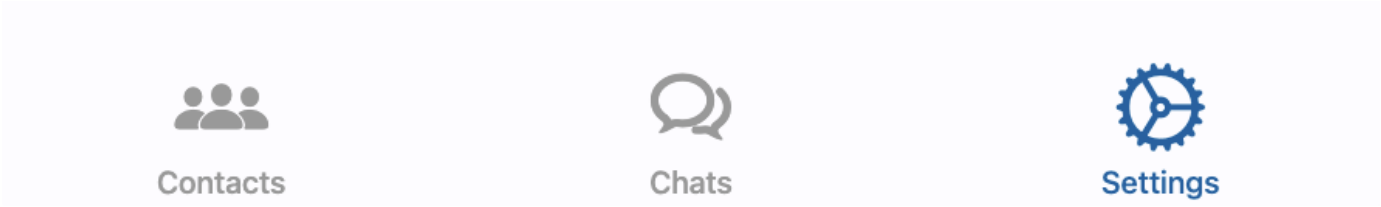
We will use `AdaptiveNavigationBarDestination` in `NavigationBar`:

```
import flet as ft
from adaptive_navigation_destination import
AdaptiveNavigationBarDestination

def main(page):

    page.adaptive = True

    page.navigation_bar = ft.NavigationBar(
        selected_index=2,
        destinations=[
            AdaptiveNavigationBarDestination(
```

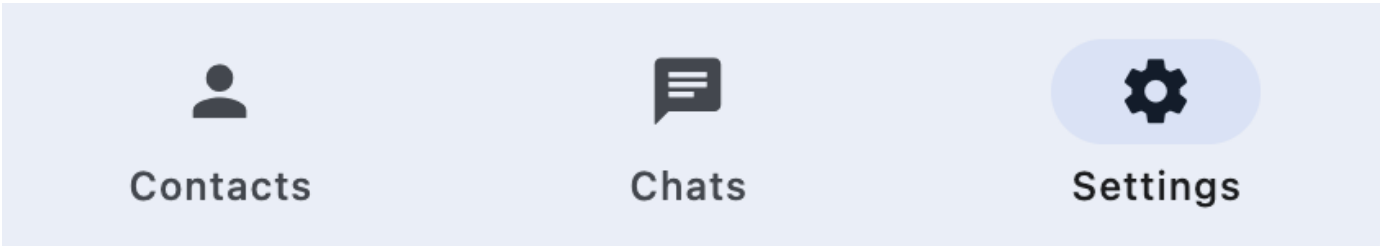
```
                ios_icon=ft.cupertino_icons.PERSON_3_FILL,
                android_icon=ft.icons.PERSON,
                label="Contacts",
            ),
            AdaptiveNavigationBarDestination(
                ios_icon=ft.cupertino_icons.CHAT_BUBBLE_2,
                android_icon=ft.icons.CHAT,
                label="Chats",
            ),
            AdaptiveNavigationBarDestination(
                ios_icon=ft.cupertino_icons.SETTINGS,
                android_icon=ft.icons.SETTINGS,
                label="Settings",
            ),
        ],
    )

    page.update()


ft.app(main)
```

Now the NavigationBar and icons within it will look like different on Android and iOS:

iOS



Android



> **NOTE**
>
> You may utilise reusable controls approach to adapt your app not only depending on the `platform` but also use `page.web` property to have different UI depending on

> wether the app is running in a browser or not, or even combine `platform` and `web` properties to have specific UI for your MACOS or Windows desktop apps.

Edit this page

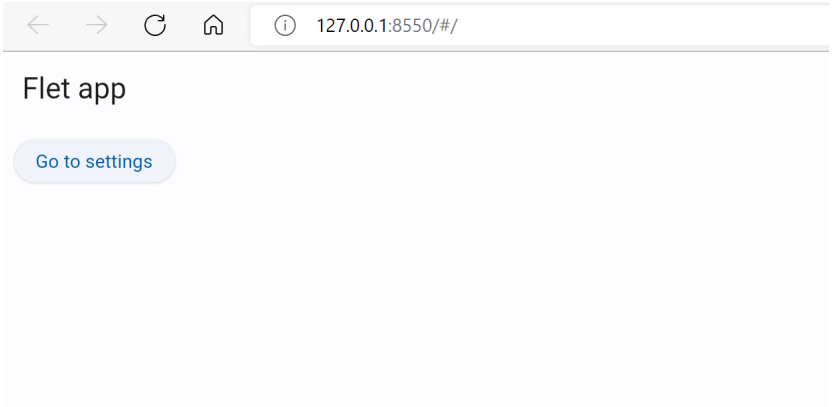
Navigation and routing

Navigation and routing is an essential feature of Single Page Applications (SPA) which allows organizing application user interface into virtual pages (views) and "navigate" between them while application URL reflects the current state of the app.

For mobile apps navigation and routing serves as a deep linking to specific application parts.

Well, it took more efforts than expected to add navigation and routing into Flet as the implementation is based on Navigator 2.0 Flutter API and required to replace Flet's "Page" abstraction with "Page and Views". Flutter's newer navigation and routing API has substantial improvements such as:

1. Programmatic control over history stack.
2. An easy way to intercept a call to "Back" button in AppBar.
3. Robust synchronization with browser history.



Explore source code of the example above.

Page route

Page route is a portion of application URL after `#` symbol:

Default application route, if not set in application URL by the user, is `/`. All routes start with `/`, for example `/store`, `/authors/1/books/2`.

Application route can be obtained by reading `page.route` property, for example:

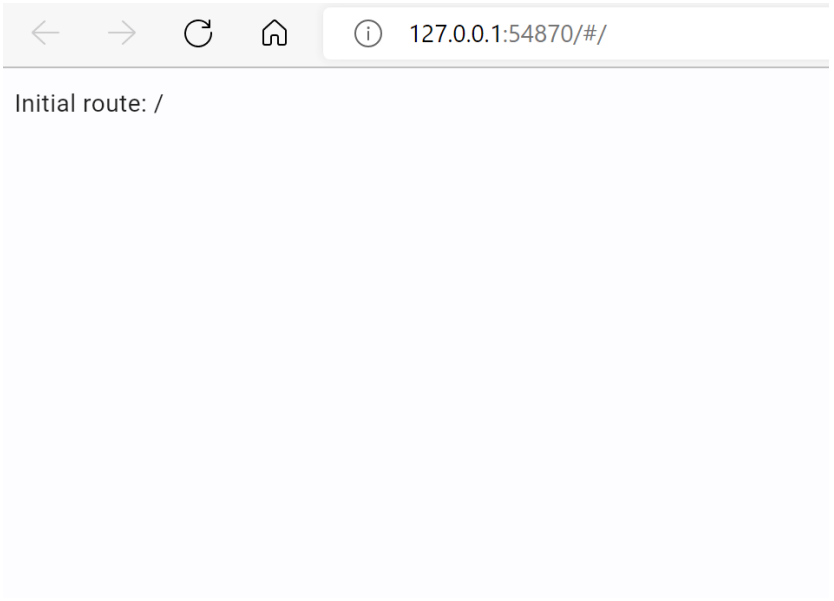
```
import flet as ft

def main(page: ft.Page):
    page.add(ft.Text(f"Initial route: {page.route}"))

ft.app(main, view=ft.AppView.WEB_BROWSER)
```

Grab application URL, open a new browser tab, paste the URL, modify its part after `#` to `/test` and hit enter. You should see "Initial route: /test".

Every time the route in the URL is changed (by editing the URL or navigating browser history with Back/Forward buttons) Flet calls `page.on_route_change` event handler:

```
import flet as ft

def main(page: ft.Page):
    page.add(ft.Text(f"Initial route: {page.route}"))

    def route_change(e: ft.RouteChangeEvent):
        page.add(ft.Text(f"New route: {e.route}"))

    page.on_route_change = route_change
    page.update()

ft.app(main, view=ft.AppView.WEB_BROWSER)
```

Now try updating URL hash a few times and then use Back/Forward buttons! You should see a new message added to a page each time the route changes:



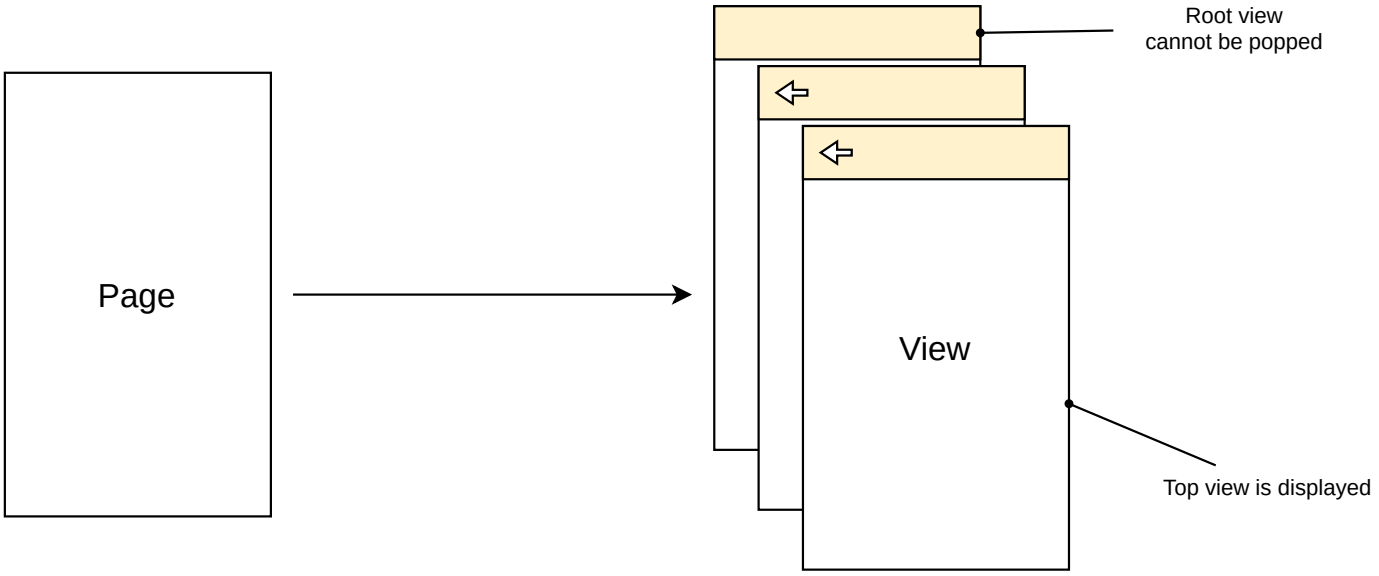
Route can be changed programmatically, by updating `page.route` property:

```
import flet as ft

def main(page: ft.Page):
    page.add(ft.Text(f"Initial route: {page.route}"))

    def route_change(e: ft.RouteChangeEvent):
        page.add(ft.Text(f"New route: {e.route}"))

    def go_store(e):
        page.route = "/store"
        page.update()

    page.on_route_change = route_change
    page.add(ft.ElevatedButton("Go to Store", on_click=go_store))

ft.app(main, view=ft.AppView.WEB_BROWSER)
```

Click "Go to Store" button and you'll see application URL is changed and a new item is pushed in a browser history. You can use browser "Back" button to navigate to a previous route.

Page views

Flet's Page now is not just a single page, but a container for View layered on top of each other like a sandwich:



A collection of views represents navigator history. Page has `page.views` property to access views collection.

The last view in the list is the one currently displayed on a page. Views list must have at least one element (root view).

To simulate a transition between pages change `page.route` and add a new `View` in the end of `page.view` list.

Pop the last view from the collection and change route to a "previous" one in `page.on_view_pop` event handler to go back.

Building views on route change

To build a reliable navigation there must be a single place in the program which builds a list of views depending on the current route. Other words, navigation history stack (represented by the list of views) must be a function of a route.

This place is `page.on_route_change` event handler.

Let's put everything together into a complete example which allows navigating between two pages:

```
import flet as ft

def main(page: ft.Page):
    page.title = "Routes Example"

    def route_change(route):
```

```
        page.views.clear()
        page.views.append(
            ft.View(
                "/",
                [
                    ft.AppBar(title=ft.Text("Flet app"),
bgcolor=ft.colors.SURFACE_VARIANT),
                    ft.ElevatedButton("Visit Store", on_click=lambda _:
page.go("/store")),
                ],
            )
        )
        if page.route == "/store":
            page.views.append(
                ft.View(
                    "/store",
                    [
                        ft.AppBar(title=ft.Text("Store"),
bgcolor=ft.colors.SURFACE_VARIANT),
                        ft.ElevatedButton("Go Home", on_click=lambda _:
page.go("/")),
                    ],
                )
            )
        page.update()

    def view_pop(view):
        page.views.pop()
        top_view = page.views[-1]
        page.go(top_view.route)

    page.on_route_change = route_change
    page.on_view_pop = view_pop
    page.go(page.route)


ft.app(main, view=ft.AppView.WEB_BROWSER)
```

Try navigating between pages using "Visit Store" and "Go Home" buttons, Back/Forward browser buttons, manually changing route in the URL - it works no matter what! :)

> **NOTE**
>
> To "navigate" between pages we used `page.go(route)` - a helper method that updates `page.route`, calls `page.on_route_change` event handler to update views

> and finally calls `page.update()`.

Notice the usage of `page.on_view_pop` event handler. It fires when the user clicks automatic "Back" button in `AppBar` control. In the handler we remove the last element from views collection and navigate to view's root "under" it.

Route templates

Flet offers `TemplateRoute` - an utility class based on repath library which allows matching ExpressJS-like routes and parsing their parameters, for example `/account/:account_id/orders/:order_id`.

`TemplateRoute` plays great with route change event:

```
troute = TemplateRoute(page.route)

if troute.match("/books/:id"):
    print("Book view ID:", troute.id)
elif troute.match("/account/:account_id/orders/:order_id"):
    print("Account:", troute.account_id, "Order:", troute.order_id)
else:
    print("Unknown route")
```

You can read more about template syntax supported by `repath` library here.

URL strategy for web

Flet web apps support two ways of configuring URL-based routing:

- **Path** (default) - paths are read and written without a hash. For example, `fletapp.dev/path/to/view`.
- **Hash** - paths are read and written to the hash fragment. For example, `fletapp.dev/#/path/to/view`.

To change URL strategy use `route_url_strategy` parameter of `flet.app()` method, for example:

```
ft.app(main, route_url_strategy="hash")
```

URL strategy for Flet Server can be configured with `FLET_ROUTE_URL_STRATEGY` environment variable which could be set to either `path` (default) or `hash`.

Edit this page

Testing Flet app on iOS

Start building awesome mobile apps in Python using just your computer and mobile phone!

Install Flet app to your iOS device. You will be using this app to see how your Flet project is working on iPhone or iPad.

To get started on your computer you need Python 3.8 or greater installed.

IMPORTANT

Your iOS device and computer must be connected to the same Wi-Fi or local network.

It's recommended to start with the creation of a new virtual environment:

macOS Linux Windows

```
python3 -m venv .venv
source .venv/bin/activate
```

Next, install the latest `flet` package:

```
pip install flet --upgrade
```

Ensure that Flet has successfully installed and Flet CLI is available in `PATH` by running:

```
flet --version
```

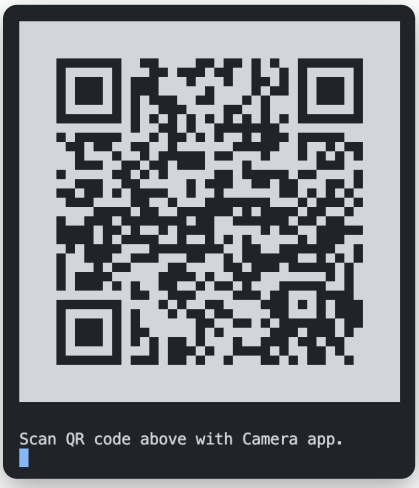
Create a new Flet project:

```
flet create my-app
cd my-app
```

Run the following command to start Flet development server with your app:

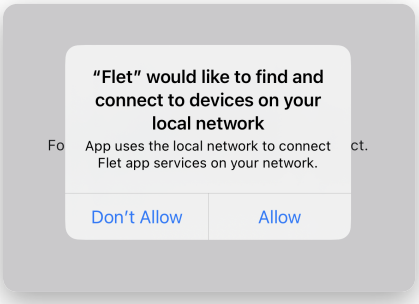
```
flet run --ios
```

A QR code with encoded project URL will be displayed in the terminal:



Open **Camera** app on your iOS device, point to a QR code and click **Open in Flet** link.

A dialog asking for permissions to access your local network will popup:



Click **Allow** and you should see your Flet app running.

Try updating `main.py` (for example, replace a greeting of `Text` control) - the app will be instantly refreshed on your iOS device.

You can try more complex Flet example from Introduction.

To return to "Home" tab either:

- Long-press anywhere on the screen with 3 fingers or
- Shake your iOS device.

You can also "manually" add a new project by clicking **"+"** button and typing its URL.

QUICK TEST

There is "Counter" Flet project hosted on the internet that you can add to Flet app to make sure everything works:

```
https://flet-counter-test-ios.fly.dev
```

Check "Gallery" tab for some great examples of what kind of projects could be done with Flet.

Explore Flet examples for even more examples.

Edit this page

Testing Flet app on Android

Start building awesome mobile apps in Python using just your computer and mobile phone!

Install Flet app to your Android device. You will be using this app to see how your Flet project is working on Android device.

To get started on your computer you need Python 3.8 or greater installed.

> **IMPORTANT**
>
> Your Android device and computer must be connected to the same Wi-Fi or local network.

It's recommended to start with the creation of a new virtual environment:

macOS | Linux | Windows

```
python3 -m venv .venv
source .venv/bin/activate
```

Next, install the latest `flet` package:

```
pip install flet --upgrade
```

Ensure that Flet has successfully installed and Flet CLI is available in `PATH` by running:

```
flet --version
```

Create a new Flet project:

```
flet create my-app
cd my-app
```

Run the following command to start Flet development server with your app:

```
flet run --android
```

A QR code with encoded project URL will be displayed in the terminal:

Open **Camera** app on your Android device, point to a QR code and click URL to open it in Flet app.

Try updating `main.py` (for example, replace a greeting of `Text` control) - the app will be instantly refreshed on your Android device.

You can try more complex Flet example from Introduction.

To return to "Home" tab either:

- Long-press anywhere on the screen with 3 fingers or
- Shake your Android device.

You can also "manually" add a new project by clicking **"+"** button and typing its URL.

> **QUICK TEST**
>
> There is "Counter" Flet project hosted on the internet that you can add to Flet app to make sure everything works:

```
https://flet-counter-test-ios.fly.dev
```

Check "Gallery" tab for some great examples of what kind of projects could be done with Flet.

Explore Flet examples for even more examples.

Edit this page

Async apps

Flet app can be written as an async app and use `asyncio` and other Python async libraries. Calling coroutines is naturally supported in Flet, so you don't need to wrap them to run synchronously.

By default, Flet executes control event handlers in separate threads, but sometimes that could be an ineffective usage of CPU or it does nothing while waiting for a HTTP response or executing `sleep()`.

Asyncio, on the other hand, allows implementing concurrency in a single thread by switching execution context between "coroutines". This is especially important for apps that are going to be published as static websites using Pyodide. Pyodide is a Python runtime built as a WebAssembly (WASM) and running in the browser. At the time of writing it doesn't support threading yet.

Getting started with async

You could mark `main()` method of Flet app as `async` and then use any asyncio API inside it:

```
import flet as ft

async def main(page: ft.Page):
    await asyncio.sleep(1)
    page.add(ft.Text("Hello, async world!"))

ft.app(main)
```

You can use `await ft.app_async(main)` if Flet app is part of a larger app and called from `async` code.

Control event handlers

Control event handlers could be both sync and `async`.

If a handler does not call any async methods it could be a regular sync method:

```
def page_resize(e):
    print("New page size:", page.window.width, page.window.height)

page.on_resize = page_resize
```

However, if a handler calls async logic it must be async too:

```
async def main(page: ft.Page):

    async def button_click(e):
        await some_async_method()
        page.add(ft.Text("Hello!"))

    page.add(ft.ElevatedButton("Say hello!", on_click=button_click))

ft.app(main)
```

Async lambdas

There are no async lambdas in Python. It's perfectly fine to have a lambda event handler in async app for simple things:

```
page.on_error = lambda e: print("Page error:", e.data)
```

but you can't have an async lambda, so an async event handler must be used.

Sleeping

To delay code execution in async Flet app you should use `asyncio.sleep()` instead of `time.sleep()`, for example:

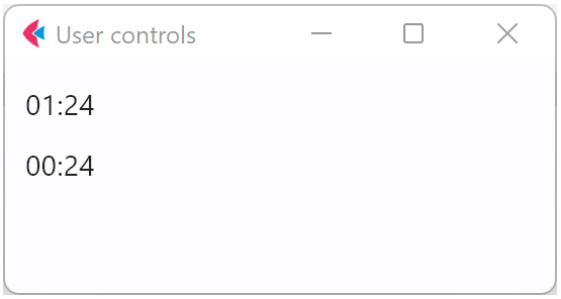
```
import asyncio
import flet as ft

def main(page: ft.Page):
    async def button_click(e):
        await asyncio.sleep(1)
```

```
        page.add(ft.Text("Hello!"))

    page.add(
        ft.ElevatedButton("Say hello with delay!",
on_click=button_click)
    )

ft.app(main)
```

Threading

To run something in the background use `page.run_task()`. For example, "Countdown" custom control which is self-updating on background could be implemented as following:

```
import asyncio
import flet as ft

class Countdown(ft.Text):
    def __init__(self, seconds):
        super().__init__()
        self.seconds = seconds

    def did_mount(self):
        self.running = True
        self.page.run_task(self.update_timer)

    def will_unmount(self):
        self.running = False

    async def update_timer(self):
        while self.seconds and self.running:
            mins, secs = divmod(self.seconds, 60)
            self.value = "{:02d}:{:02d}".format(mins, secs)
            self.update()
            await asyncio.sleep(1)
            self.seconds -= 1

def main(page: ft.Page):
    page.add(Countdown(120), Countdown(60))

ft.app(main)
```



Edit this page

Publishing Flet app to multiple platforms

Introduction

Flet CLI provides `flet build` command that allows packaging Flet app into a standalone executable or install package for distribution.

Platform matrix

The following matrix shows which OS you should run `flet build` command on in order to build a package for specific platform:

Run on / `flet build`	`apk/aab`	`ipa`	`macos`	`linux`	`windows`	`web`
macOS	✅	✅	✅			✅
Windows	✅			✅ (WSL)	✅	✅
Linux	✅			✅		✅

Prerequisites

Flutter SDK

Flutter SDK 3.16 or above must be installed and the path to both `flutter` and `dart` commands must be added to `PATH` environment variable.

On macOS we recommend installing Flutter SDK with "Download and install" approach.

On Linux we recommend installing Flutter SDK with Method 2: Manual installation (do not install Flutter with `snap`).

Pay attention to Flutter's own requirements for every platform, such as XCode and Cocopods on macOS, Visual Studio 2022 on Windows or additional tools and libraries on Linux.

Project structure

`flet build` command assumes the following Flet project structure.

```
/assets/
    icon.png
main.py
requirements.txt
pyproject.toml
```

`main.py` is the entry point of your Flet application with `ft.app(main)` at the end. A different entry point could be specified with `--module-name` argument.

`assets` is an optional directory that contains application assets (images, sound, text and other files required by your app) as well as images used for package icons and splash screens.

If only `icon.png` (or other supported format such as `.bmp`, `.jpg`, `.webp`) is provided it will be used as a source image to generate all icons and splash screens for all platforms. See section below for more information about icons and splashes.

`requirements.txt` is a standard pip file that contains the list of Python requirements for your Flet app. If this file is not provided only `flet` dependency will be installed during packaging.

> **NO PIP FREEZE**
>
> Do not use `pip freeze > requirements.txt` command to create `requirements.txt` for the app that will be running on mobile. As you run `pip freeze` command on a desktop `requirements.txt` will have dependencies that are not intended to work on a mobile device, such as `watchdog`.
>
> Hand-pick `requirements.txt` to have only direct dependencies required by your app, plus `flet`.

`pyproject.toml` can also be used by `flet build` command to get the list project dependencies. However, if both `requirements.txt` and `pyproject.toml` exist then `pyproject.toml` will be ignored.

The easiest way to start with that structure is to use `flet create` command:

```
flet create myapp
```

where `myapp` is a target directory.

> **PYPROJECT.TOML**
>
> Reading dependencies from `pyproject.toml` is not yet supported (issue), please use `requirements.txt` instead.

How it works

`flet build <target_platform>` command could be run from the root of Flet app directory:

```
<flet_app_directory> % flet build <target_platform>
```

where `<target_platform>` could be one of the following: `apk`, `aab`, `ipa`, `web`, `macos`, `windows`, `linux`.

When running from a different directory you can provide the path to a directory with Flet app:

```
flet build <target_platform> <path_to_python_app>
```

Build results are copied to `<python_app_directory>/build/<target_platform>`. You can specify a custom output directory with `--output` option:

```
flet build <target_platform> --output <path-to-output-dir>
```

`flet build` uses Flutter SDK and the number of Flutter packages to build a distribution package from your Flet app.

When you run `flet build <target_platform>` command it:

- Creates a new Flutter project in a temp directory from https://github.com/flet-dev/flet-build-template template. Flutter app will contain a packaged Python app in the assets and use `flet` and `serious_python` packages to run Python app and render its UI respectively. The project is ephemeral and deleted upon completion.
- Copies custom icons and splash images (see below) from `assets` directory into a Flutter project.
- Generates icons for all platforms using `flutter_launcher_icons` package.
- Generates splash screens for web, iOS and Android targets using `flutter_native_splash` package.
- Packages Python app using `package` command of `serious_python` package. All python files in the current directory and sub-directories recursively will be compiled to `.pyc` files. All files, except `build` directory will be added to a package asset.
- Runs `flutter build <target_platform>` command to produce an executable or an install package.
- Copies build results to `build/<target_platform>` directory.

Including optional controls

If your app uses the following controls their packages must be added to a build command:

- `Audio` control implemented in `flet_audio` package.
- `AudioRecorder` control implemented in `flet_audio_recorder` package.
- `Lottie` control implemented in `flet_lottie` package.
- `Rive` control implemented in `flet_rive` package.
- `Video` control implemented in `flet_video` package.
- `WebView` control implemented in `flet_webview` package.

Use `--include-packages <package_1> <package_2> ...` option to add Flutter packages with optional Flet controls.

For example, to build your Flet app with `Video` and `Audio` controls add `--include-packages flet_video flet_audio` to your `flet build` command:

```
flet build apk --include-packages flet_video flet_audio
```

Icons

You can customize app icons for all platforms (excluding Linux) with images in `assets` directory of your Flet app.

If only `icon.png` (or other supported format such as `.bmp`, `.jpg`, `.webp`) is provided it will be used as a source image to generate all icons.

- **iOS** - `icon_ios.png` (or any supported image format). Recommended minimum image size is 1024 px. Image should not be transparent (have alpha channel). Defaults to `icon.png` with alpha-channel automatically removed.
- **Android** - `icon_android.png` (or any supported image format). Recommended minimum image size is 192 px. Defaults to `icon.png`.
- **Web** - `icon_web.png` (or any supported image format). Recommended minimum image size is 512 px. Defaults to `icon.png`.
- **Windows** - `icon_windows.png` (or any supported image format). ICO will be produced of 256 px size. Defaults to `icon.png`. If `icon_windows.ico` file is provided it will be just copied to `windows/runner/resources/app_icon.ico` unmodified.
- **macOS** - `icon_macos.png` (or any supported image format). Recommended minimum image size is 1024 px. Defaults to `icon.png`.

Splash screen

You can customize splash screens for iOS, Android and web applications with images in `assets` directory of your Flet app.

If only `splash.png` or `icon.png` (or other supported format such as `.bmp`, `.jpg`, `.webp`) is provided it will be used as a source image to generate all splash screen.

- **iOS (light)** - `splash_ios.png` (or any supported image format). Defaults to `splash.png` and then `icon.png`.
- **iOS (dark)** - `splash_dark_ios.png` (or any supported image format). Defaults to light iOS splash, then to `splash_dark.png`, then to `splash.png` and then `icon.png`.

- **Android (light)** - `splash_android.png` (or any supported image format). Defaults to `splash.png` and then `icon.png`.
- **Android (dark)** - `splash_dark_android.png` (or any supported image format). Defaults to light Android splash, then to `splash_dark.png`, then to `splash.png` and then `icon.png`.
- **Web (light)** - `splash_web.png` (or any supported image format). Defaults to `splash.png` and then `icon.png`.
- **Web (dark)** - `splash_dark_web.png` (or any supported image format). Defaults to light web splash, then `splash_dark.png`, then to `splash.png` and then `icon.png`.

`--splash-color` option specifies background color for a splash screen in light mode. Defaults to `#ffffff`.

`--splash-dark-color` option specifies background color for a splash screen in dark mode. Defaults to `#333333`.

Flet app entry point

By default, `flet build` command assumes `main.py` as the entry point of your Flet application, i.e. the file with `ft.app(main)` at the end. A different entry point could be specified with `--module-name` argument.

Versioning

You can provide a version information for built executable or package with `--build-number` and `--build-version` arguments. This is the information that is used to distinguish one build/release from another in App Store and Google Play and is shown to the user in about dialogs.

`--build-number` - an integer number (defaults to `1`), an identifier used as an internal version number. Each build must have a unique identifier to differentiate it from previous builds. It is used to determine whether one build is more recent than another, with higher numbers indicating more recent build.

`--build-version` - a "x.y.z" string (defaults to `1.0.0`) used as the version number shown to users. For each new version of your app, you will provide a version number to differentiate it from previous versions.

Customizing packaging template

To create a temporary Flutter project `flet build` uses cookiecutter template stored in https://github.com/flet-dev/flet-build-template repository.

You can customize that template to suit your specific needs and then use it with `flet build`.

`--template` option can be used to provide the URL to the repository or path to a directory with your own template. Use `gh:` prefix for GitHub repos, e.g. `gh:{my-org}/{my-repo}` or provide a full path to a Git repository, e.g. `https://github.com/{my-org}/{my-repo}.git`.

For Git repositories you can checkout specific branch/tag/commit with `--template-ref` option.

`--template-dir` option specifies a relative path to a cookiecutter template in a repository given by `--template` option. When `--template` option is not used, this option specifies path relative to the `<user-directory>/.cookiecutters/flet-build-template`.

Extra args to `flutter build` command

`--flutter-build-args` option allows passing extra arguments to `flutter build` command called during the build process. The option can be used multiple times.

For example, if you want to add `--no-tree-shake-icons` option:

```
flet build macos --flutter-build-args=--no-tree-shake-icons
```

To pass an option with a value:

```
flet build ipa --flutter-build-args=--export-method --flutter-build-
args=development
```

Verbose logging

`--verbose` or `-vv` option allows to see the output of all commands during `flet build` run. We might ask for a detailed log if you need support.

Logging

All Flet apps output to `stdout` and `stderr` (e.g. all `print()` statements or `sys.stdout.write()` calls, Python `logging` library) is now redirected to `out.log` file. Writes to that file are unbuffered, so you can retrieve a log in your Python program at any moment with a simple:

```
with open("out.log", "r") as f:
    log = f.read()
```

`AlertDialog` or any other control can be used to display the value of `log` variable.

When the program is terminated by calling `sys.exit()` with exit code `100` (magic code) the entire log will be displayed in a scrollable window.

```
import sys
sys.exit(100)
```

Calling `sys.exit()` with any other exit code will terminate (close) the app without displaying a log.

Packaging app for Android

Introduction

Flet CLI provides `flet build apk` and `flet build aab` commands that allow packaging Flet app into Android APK and Android App Bundle (AAB) respectively.

Prerequisites

Native Python packages

Native Python packages (vs "pure" Python packages written in Python only) are packages that partially written in C, Rust or other languages producing native code. Example packages are `numpy`, `cryptography`, `lxml`, `pydantic`.

When packaging Flet app for Android with `flet build` command such packages cannot be installed from PyPI, because there are no wheels (`.whl`) for Android platform.

Therefore, you have to compile native packages for Android on your computer before running `flet build` command.

> **WORK IN PROGRESS**
>
> We are actively working on automating the process described below - it's #1 item in our backlog.

Flet uses Kivy for Android to build Python and native Python packages for Android.

To build your own Python distributive with custom native packages and use it with `flet build` command you need to use `p4a` tool provided by Kivy for Android.

`p4a` command-line tool can be run on macOS and Linux (WSL on Windows).

To get Android SDK install Android Studio.

On macOS Android SDK will be located at `$HOME/Library/Android/sdk`.

Install Temurin8 to get JRE 1.8 required by `sdkmanager` tool:

```
brew install --cask temurin8
export JAVA_HOME=/Library/Java/JavaVirtualMachines/temurin-
8.jdk/Contents/Home
```

Set the following environment variables:

```
export ANDROID_SDK_ROOT="$HOME/Library/Android/sdk"
export NDK_VERSION=25.2.9519653
export SDK_VERSION=android-33
```

Add path to `sdkmanager` to `PATH`:

```
export PATH=$ANDROID_SDK_ROOT/tools/bin:$PATH
```

Install Android SDK and NDK from https://developer.android.com/ndk/downloads/ or with Android SDK Manager:

```
echo "y" | sdkmanager --install "ndk;$NDK_VERSION" --channel=3
echo "y" | sdkmanager --install "platforms;$SDK_VERSION"
```

Create new Python virtual environment:

```
python3 -m venv .venv
source .venv/bin/activate
```

Install `p4a` from Flet's fork - it has pinned Python 3.11.6 which is compatible with the rest of the code produced by `flet build`:

```
pip3 install git+https://github.com/flet-dev/python-for-
android.git@3.11.6
```

Install `cython`:

```
pip install --upgrade cython
```

Run `p4a` with `--requirements` including your custom Python libraries separated with comma, like `numpy` in the following example:

```
p4a create --requirements numpy --arch arm64-v8a --arch armeabi-v7a --
arch x86_64 --sdk-dir $ANDROID_SDK_ROOT --ndk-dir
$ANDROID_SDK_ROOT/ndk/$NDK_VERSION --dist-name mydist
```

Choose No to "Do you want automatically install prerequisite JDK? [y/N]".

NOTE: The library you want to build with `p4a` command should have a recipe in this folder. You can submit a request to make a recipe for the library you need or create your own recipe and submit a PR.

When `p4a` command completes a Python distributive with your custom libraries will be located at:

```
$HOME/.python-for-android/dists/mydist
```

In the terminal where you run `flet build apk` command to build your Flet Android app run the following command to store distributive full path in `SERIOUS_PYTHON_P4A_DIST` environment variable:

```
export SERIOUS_PYTHON_P4A_DIST=$HOME/.python-for-android/dists/mydist
```

Build your app by running `flet build apk` command to build `.apk`.

You app's bundle now includes custom Python libraries.

`flet build apk`

Build an Android APK file from your app.

This command builds release version. 'release' builds don't support debugging and are suitable for deploying to app stores. If you are deploying the app to the Play Store, it's recommended to use Android App Bundles (AAB) or split the APK to reduce the APK size.

- https://developer.android.com/guide/app-bundle
- https://developer.android.com/studio/build/configure-apk-splits#configure-abi-split

Splash screen

By default, generated Android app will be showing a splash screen with an image from `assets` directory (see below) or Flet logo. You can disable splash screen for Android app with `--no-android-splash` option.

Installing APK to a device

The easiest way to install APK to your device is to use `adb` (Android Debug Bridge) tool.

`adb` is a part of Android SDK. For example, on macOS, if Android SDK was installed with Android Studio the location of `adb` tool will be at `~/Library/Android/sdk/platform-tools/adb`.

Check this article for more information about installing and using `adb` tool on various platforms.

To install APK to a device run the following command:

```
adb install <path-to-your.apk>
```

If more than one device is connected to your computer (say, emulator and a physical phone) you can use `-s` option to specify which device you want to install `.apk` on:

```
adb -s <device> install <path-to-your.apk>
```

where `<device>` can be found with `adb devices` command.

Building platform-specific APK

By default, Flet builds "fat" APK which includes binaries for both `arm64-v8a` and `armeabi-v7a` architectures.

If you know/control Android device your app will be distributed on you can build a smaller APK for a specific architecture.

To build APK for `arm64-v8a`:

```
```

```
flet build apk --flutter-build-args=--target-platform --flutter-build-
args=android-arm64
```

To build APK for `armeabi-v7a`:

```
flet build apk --flutter-build-args=--target-platform --flutter-build-
args=android-arm
```

Troubleshooting Android

To run interactive commands inside simulator or device:

```
adb shell
```

To overcome "permissions denied" error while trying to browse file system in interactive Android shell:

```
su
```

To download a file from a device to your local computer:

```
adb pull <device-path> <local-path>
```

`flet build aab`

Build an Android App Bundle (AAB) file from your app.

This command builds release version. 'release' builds don't support debugging and are suitable for deploying to app stores. App bundle is the recommended way to publish to the Play Store as it improves your app size.

Splash screen

By default, generated Android app will be showing a splash screen with an image from `assets` directory (see below) or Flet logo. You can disable splash screen for Android app

with `--no-android-splash` option.

Packaging app for iOS

Introduction

Flet CLI provides `flet build ipa` command that allows packaging Flet app into an iOS archive bundle and IPA for distribution.

> **NOTE**
>
> The command can be run on macOS only.

Prerequisites

Native Python packages

Native Python packages (vs "pure" Python packages written in Python only) are packages that partially written in C, Rust or other languages producing native code. Example packages are `numpy`, `cryptography`, `lxml`, `pydantic`.

When packaging Flet app for iOS with `flet build` command such packages cannot be installed from PyPI, because there are no wheels (`.whl`) for iOS platform.

Therefore, you have to compile native packages for iOS on your computer before running `flet build` command.

> **WORK IN PROGRESS**
>
> We are actively working on automating the process described below - it's #1 item in our backlog.

Flet uses Kivy for iOS to build Python and native Python packages for iOS.

To build your own Python distributive with custom native packages and use it with `flet build` command you need to use `toolchain` tool provided by Kivy for iOS.

`toolchain` command-line tool can be run on macOS only.

Start with creating a new Python virtual environment and installing `kivy-ios` package from Flet's fork as described here:

```
pip install git+https://github.com/flet-dev/python-for-ios.git
```

Run `toolchain` command with the list of packages you need to build, for example to build `numpy`:

```
toolchain build numpy
```

NOTE: The library you want to build with `toolchain` command should have a recipe in this folder. You can submit a request to make a recipe for the library you need or create your own recipe and submit a PR.

You can also install package that don't require compilation with `pip`:

```
toolchain pip install flask
```

This case you don't need to include that package into `requirements.txt` of your Flet app.

When `toolchain` command is finished you should have everything you need in `dist` directory.

Get the full path to `dist` directory by running `realpath dist` command.

In the terminal where you run `flet build ipa` command to build your Flet iOS app run the following command to store `dist` full path in `SERIOUS_PYTHON_IOS_DIST` environment variable:

```
export SERIOUS_PYTHON_IOS_DIST="<full-path-to-dist-directory>"
```

Build your app by running `flet build ipa` command.

You app's bundle now includes custom Python libraries.

`flet build ipa`

Build an iOS archive bundle and IPA for distribution (macOS host only).

> **WORK IN PROGRESS**
>
> Creating of an iOS package, suitable for running on a device or publishing to AppStore is, in general, a complex process with a lot of moving parts. Let us know if it worked or didn't work for your particular case and there are some changes required into Flutter project template.

To successfully generate IPA you should provide correct values for the following arguments:

- `--org` - organization name in reverse domain name notation, e.g. `com.mycompany` (defaults to `com.flet`). The value is combined with `--project` and used as an iOS and Android bundle ID.
- `--project` - project name in C-style identifier format (lowercase alphanumerics with underscores) used to build bundle ID and as a name for bundle executable. By default, it's the name of Flet app directory.
- `--team` - team ID to locate provisioning profile. If no team ID provided a unsigned iOS archive will be generated.

Packaging app for macOS

Flet CLI provides `flet build macos` command that allows packaging Flet app into a macOS application bundle.

> **NOTE**
>
> The command can be run on macOS only.

`flet build macos`

Creates a macOS application bundle from your Flet app.

Packaging app for Linux

Flet CLI provides `flet build linux` command that allows packaging Flet app into a Linux application.

> **NOTE**
>
> The command can be run on Linux only.

Prerequisites

GStreamer for `Audio`

GStreamer libraries must be installed if your Flet app uses `Audio` control.

To install minimal set of GStreamer libs on Ubuntu/Debian run the following commands:

```
apt install libgtk-3-dev libgstreamer1.0-dev libgstreamer-plugins-
base1.0-dev
```

To install full set of GStreamer libs:

```
apt install libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev
libgstreamer-plugins-bad1.0-dev gstreamer1.0-plugins-base gstreamer1.0-
plugins-good gstreamer1.0-plugins-bad gstreamer1.0-plugins-ugly
gstreamer1.0-libav gstreamer1.0-doc gstreamer1.0-tools gstreamer1.0-x
gstreamer1.0-alsa gstreamer1.0-gl gstreamer1.0-gtk3 gstreamer1.0-qt5
gstreamer1.0-pulseaudio
```

See this guide for installing on other Linux distributives.

To build your Flet app that uses `Audio` control add `--include-packages flet_audio` to `flet build` command, for example:

```
flet build apk --include-packages flet_audio
```

MPV for `Video`

libmpv libraries must be installed if your Flet app uses `Video` control. On Ubuntu/Debian you can install it with:

```
sudo apt install libmpv-dev mpv
```

To build your Flet app that uses `Video` control add `--include-packages flet_video` to `flet build` command, for example:

```
flet build apk --include-packages flet_video
```

`flet build linux`

Creates a Linux application from your Flet app.

Packaging app for Windows

Flet CLI provides `flet build windows` command that allows packaging Flet app into a Windows application.

> **NOTE**
>
> The command can be run on Windows only.

Prerequisites

Enable Developer Mode

While running `flet build` on Windows you may get the following error:

```
Building with plugins requires symlink support.

Please enable Developer Mode in your system settings. Run
  start ms-settings:developers
to open settings.
```

Follow this SO answer for the instructions on how to enable developer mode in Windows 11.

Visual Studio components

```
Unable to find suitable Visual Studio toolchain. Please run `flutter
doctor` for more details.
```

Follow this medium article for the instructions on downloading & installing correct Visual Studio components for desktop development.

`flet build windows`

Creates a Windows application from your Flet app.

Publishing Flet app to web

Flet allows publishing your app as a **static** or **dynamic** website.

Static website - content is not changing and delivered exactly as it's stored. Python code is running in the web browser.

Dynamic website - content is dynamically generated for each user. Python code is running on the server.

Here is a table comparing Flet app running as a static vs dynamic website:

	Static website	Dynamic website
Loading time	⬇️ Slower - Python runtime (Pyodide) along with app's Python code and all its dependencies must be loaded into the browser. Pyodide initialization takes time too.	✅ Faster - the app stays and runs on the server.
Python compatibility	⬇️ Not every program that works with native Python can be run with Pyodide	✅ Any Python package can be used.
Responsiveness	✅ Zero latency between user-generated events (clicks, text field changes, drags) and page updates.	⬇️ Non-zero latency - user-generated events are communicated to a server via WebSockets. UI updates are communicated back.
Performance	⬇️ Slower - Pyodide is currently 3x-5x slower than native Python because of WASM	✅ Faster - the code is running on the server by native Python.

	Static website	Dynamic website
Code protection	⬇️ Low - app's code is loaded into a web browser and can be inspected by a user.	✅ High - the app is running on the server.
Hosting	✅ Cheap/Free - no code is running on the server and thus the app can be hosted anywhere: GitHub Pages, Cloudflare Pages, Replit, Vercel, a shared hosting or your own VPS.	⬇️ Paid - the app requires Python code to run on the server and communicate with a web browser via WebSockets.

Integrating existing Flutter packages into your Flet app

> **WORK IN PROGRESS**
>
> The guide is being updated.

Introduction

While Flet controls leverage many built-in Flutter widgets, enabling the creation of even complex applications, not all Flutter widgets or third-party packages can be directly supported by the Flet team or included within the core Flet framework.

To address this, the Flet framework provides an extensibility mechanism. This allows you to incorporate widgets and APIs from your own custom Flutter packages or third-party libraries directly into your Flet application.

Prerequisites

To integrate custom Flutter package into Flet you need to have basic understanding of how to create Flutter apps and packages in Dart language and have Flutter development environment configured. See Flutter Getting Started for more information about Flutter and Dart.

Creating Flet extension

Flet extension that integrates 3rd-party Flutter package consists of the following parts:

1. Flet Dart package.
2. Flet Python control.

The Flet Dart package includes a mechanism to create Flutter widgets based on control names returned by the Control's `_get_control_name()` function. This mechanism iterates through all third-party packages and returns the first matching widget.

Flet Python control is a Python class that you will use in your Flet program.

For example, take a look at a basic Flet extension for flutter_spinkit package.

Flet Dart package

To create a new Dart package, run the following command:

```
flutter create --template=package <package_name>
```

You will see this folder structure:

```
├── CHANGELOG.md
├── LICENSE
├── README.md
├── analysis_options.yaml
├── lib
│   └── <package_name>.dart
├── pubspec.lock
├── pubspec.yaml
├── test
│   └── <package_name>_test.dart
└── <package_name>.iml
```

In the `lib` folder, you need to create `src` folder with two files: `create_control.dart` and `<control_name>.dart`:

```
└── <package_name>
    ├── lib
    │   ├── <package_name>.dart
    │   └── src
    │       ├── create_control.dart
    │       └── <control_name>.dart
    └── pubspec.yaml
```

`pubspec.yaml`

A yaml file containing metadata that specifies the package's dependencies.

In your `pubspec.yaml` you should add dependency to `flet` and Flutter package for which you are creating your extension.

In the Flet Spinkit example, `pubspec.yaml` contains dependency to `flutter_spinkit`:

```
dependencies:
  flet: ^0.22.0
  flutter_spinkit: ^5.2.1
```

`<package_name>.dart`

Extension package should export two methods:

- `createControl` - called to create a widget that corresponds to a control on Python side.
- `ensureInitialized` - called once on Flet program start.

```
library <package_name>;

export "../src/create_control.dart" show createControl,
ensureInitialized;
```

`create_control.dart`

Flet calls `createControl` for all controls and returns the first matching widget.

```
import 'package:flet/flet.dart';

import 'spinkit.dart';

CreateControlFactory createControl = (CreateControlArgs args) {
  switch (args.control.type) {
    case "spinkit":
      return SpinkitControl(
        parent: args.parent,
        control: args.control,
      );
```

```
    default:
      return null;
  }
};

void ensureInitialized() {
  // nothing to initialize
}
```

`<control-name>.dart`

Here you create Flutter "wrapper" widget that will build Flutter widget or API that you want to use in your Flet app.

Wrapper widget passes the state of Python control down to a Flutter widget, that will be displayed on a page, and provides an API to route events from Flutter widget back to Python control.

```
import 'package:flutter/material.dart';
import 'package:flutter_spinkit/flutter_spinkit.dart';

class SpinkitControl extends StatelessWidget {
  const SpinkitControl({
    super.key,
  });

  @override
  Widget build(BuildContext context) {
    return const SpinKitRotatingCircle(
      color: Colors.red,
      size: 100.0,
    );
  }
}
```

As a proof of concept, we would like to see the hardcoded `SpinKitRotatingCircle` in our Flet program, and later we will get to customizing its properties.

Flet Python control

Flet Python control is a Python class that you can create in your Flet app or as an external Python module. In the Flet Spinkit example, we created Spinkit class in

`/controls/spinkit.py` file:

```
from flet_core.control import Control


class Spinkit(Control):
    """
    Spinkit Control.
    """

    def __init__(self):
        Control.__init__(self)

    def _get_control_name(self):
        return "spinkit"
```

The minumal requirements for this class is that it has to be inherited from Flet `Control` and it has to have `_get_control_name` method that will return the control name. This name should be the same as `args.control.type` we check in the `create_control.dart` file.

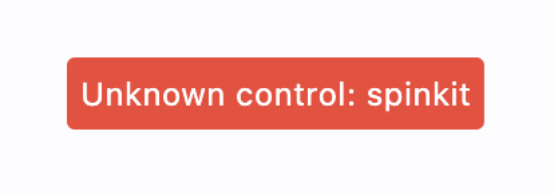
Connect your Python app and Dart package

Once you have created Flet Dart package and Flet Python control, create a Python program in `main.py` that uses it:

```
import flet as ft
from controls.spinkit import Spinkit

def main(page: ft.Page):
    page.vertical_alignment = ft.MainAxisAlignment.CENTER
    page.horizontal_alignment = ft.CrossAxisAlignment.CENTER

    page.add(Spinkit())


ft.app(main)
```

Let's run this simple app with `flet run` command. We expect to see the hardcoded `SpinKitRotatingCircle` on the page but that's not happening yet. Instead, we see this

message in place of the Spinkit control:



Our Flet app doesn't know yet about the new Flet Dart package that we created.

To connect your Python app and new Flet Dart package, you need create to `pubspec.yaml` file on the same level as `main.py`. It should have the following contents:

```
dependencies:
  flet_spinkit:
    path: {absolute-path-to-flet-dart-package-folder}
```

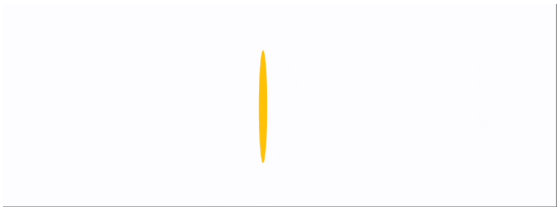
> **WORK IN PROGRESS**
>
> This approach is subject to change and the guide is being updated.

Now you need to build the app for the platform of your choice by running `flet build` command, for example:

```
flet build macos
```

Finally, we open the built app:

```
open build/macos/flet_spinkit_app.app
```



You can find source code for this example here.

> **INFO**

> Every time you need to make changes to Python or Dart part of your extension, you need to re-run build command.

Customize properties

In the example above, Spinkit control creates a hardcoded Flutter widget. Now let's customize its properties.

Flet `Control` properties

When we created Spinkit class in Python, it inherited from Flet `Control` class that has properties common for all controls such as `visible`, `opacity` and `tooltip`, to name a few. See reference for the common Control properties here.

To be able to use these properties for your new control you need to add the Control properties you want to use in the constructor for your new Python control:

```
from typing import Any, Optional

from flet_core.control import Control, OptionalNumber

class Spinkit(Control):
    """
    Spinkit Control.
    """

    def __init__(
        self,
        #
        # Control
        #
        opacity: OptionalNumber = None,
        tooltip: Optional[str] = None,
        visible: Optional[bool] = None,
        data: Any = None,
    ):
        Control.__init__(
            self,
            tooltip=tooltip,
            opacity=opacity,
            visible=visible,
            data=data,
        )
```

```
    def _get_control_name(self):
        return "spinkit"
```

In `<control-name>.dart` file, wrap your widget into `baseControl()` to magically implement all Python's `Control` properties:

```
import 'package:flet/flet.dart';
import 'package:flutter/material.dart';
import 'package:flutter_spinkit/flutter_spinkit.dart';

class SpinkitControl extends StatelessWidget {
  final Control? parent;
  final Control control;

  const SpinkitControl({
    super.key,
    required this.parent,
    required this.control,
  });

  @override
  Widget build(BuildContext context) {
    return baseControl(
        context,
        const SpinKitRotatingCircle(
          color: Colors.green,
          size: 100.0,
        ),
        parent,
        control);
  }
}
```

Finally, use `Control` properties in your app:

```
import flet as ft
from controls.spinkit import Spinkit


def main(page: ft.Page):
    page.vertical_alignment = ft.MainAxisAlignment.CENTER
    page.horizontal_alignment = ft.CrossAxisAlignment.CENTER
```

```
    page.add(Spinkit(opacity=0.5, tooltip="Spinkit tooltip"))


ft.app(main)
```

You can find source code for this example here.

Flet `ConstrainedControl` properties

Generally, there are three types of controls in Flet:

1. Visual Controls that are added to the app/page surface, such as Spinkit.
2. Popup Controls (dialogs, pickers, panels etc.).
3. Non-visual Controls or services that are added to `overlay`, such as Video or Audio.

In the most cases, Visual Controls could inherit from `ConstrainedControl` that has many additional properties such as `top` and `left` for its position within Stack and a bunch of animation properties.

To use those properties, inherit your control from `CostrainedControl` and add those properties to the constructor of your Python control:

```
from typing import Any, Optional

from flet_core.constrained_control import ConstrainedControl
from flet_core.control import OptionalNumber


class Spinkit(ConstrainedControl):
    """
    Spinkit Control.
    """

    def __init__(
        self,
        #
        # Control
        #
        opacity: OptionalNumber = None,
        tooltip: Optional[str] = None,
        visible: Optional[bool] = None,
        data: Any = None,
        #
```

```
        # ConstrainedControl
        #
        left: OptionalNumber = None,
        top: OptionalNumber = None,
        right: OptionalNumber = None,
        bottom: OptionalNumber = None,
    ):
        ConstrainedControl.__init__(
            self,
            tooltip=tooltip,
            opacity=opacity,
            visible=visible,
            data=data,
            left=left,
            top=top,
            right=right,
            bottom=bottom,
        )

    def _get_control_name(self):
        return "spinkit"
```

In `<control-name>.dart` file, use `constrainedControl` method to wrap Flutter widget:

```
import 'package:flet/flet.dart';
import 'package:flutter/material.dart';
import 'package:flutter_spinkit/flutter_spinkit.dart';

class SpinkitControl extends StatelessWidget {
  final Control? parent;
  final Control control;

  const SpinkitControl({
    super.key,
    required this.parent,
    required this.control,
  });

  @override
  Widget build(BuildContext context) {
    return constrainedControl(
        context,
        const SpinKitRotatingCircle(
          color: Colors.green,
          size: 100.0,
        ),
```

```
        parent,
        control);
  }
}
```

Use `ConstrainedControl` properties in your app:

```
import flet as ft
from controls.spinkit import Spinkit


def main(page: ft.Page):
    page.vertical_alignment = ft.MainAxisAlignment.CENTER
    page.horizontal_alignment = ft.CrossAxisAlignment.CENTER

    page.add(
        ft.Stack(
            [
                ft.Container(height=200, width=200,
bgcolor=ft.colors.BLUE_100),
                Spinkit(opacity=0.5, tooltip="Spinkit tooltip", top=0,
left=0),
            ]
        )
    )


ft.app(main)
```

You can find source code for this example here.

Control-specific properties

Now that you have taken full advantage of the properties Flet `Control` and `ConstrainedControl` offer, let's define the properties that are specific to the new Control you are building.

In the Spinkit example, let's define its `color` and `size`.

In Python class, define new `color` and `size` properties:

```
from typing import Any, Optional
```

```python
from flet_core.constrained_control import ConstrainedControl
from flet_core.control import OptionalNumber


class Spinkit(ConstrainedControl):
    """
    Spinkit Control.
    """

    def __init__(
        self,
        #
        # Control
        #
        opacity: OptionalNumber = None,
        tooltip: Optional[str] = None,
        visible: Optional[bool] = None,
        data: Any = None,
        #
        # ConstrainedControl
        #
        left: OptionalNumber = None,
        top: OptionalNumber = None,
        right: OptionalNumber = None,
        bottom: OptionalNumber = None,
        #
        # Spinkit specific
        #
        color: Optional[str] = None,
        size: OptionalNumber = None,
    ):
        ConstrainedControl.__init__(
            self,
            tooltip=tooltip,
            opacity=opacity,
            visible=visible,
            data=data,
            left=left,
            top=top,
            right=right,
            bottom=bottom,
        )

        self.color = color
        self.size = size

    def _get_control_name(self):
```

```
        return "spinkit"

    # color
    @property
    def color(self):
        return self._get_attr("color")

    @color.setter
    def color(self, value):
        self._set_attr("color", value)

    # size
    @property
    def size(self):
        return self._get_attr("size")

    @size.setter
    def size(self, value):
        self._set_attr("size", value)
```

In `<control-name>.dart` file, use helper methods `attrColor` and `attrDouble` to access color and size values:

```
import 'package:flet/flet.dart';
import 'package:flutter/material.dart';
import 'package:flutter_spinkit/flutter_spinkit.dart';

class SpinkitControl extends StatelessWidget {
  final Control? parent;
  final Control control;

  const SpinkitControl({
    super.key,
    required this.parent,
    required this.control,
  });

  @override
  Widget build(BuildContext context) {
    var color = control.attrColor("color", context);
    var size = control.attrDouble("size");

    return constrainedControl(
        context,
        SpinKitRotatingCircle(
```

```
            color: color,
            size: size ?? 50,
        ),
        parent,
        control);
  }
}
```

Use `color` and `size` properties in your app:

```
import flet as ft
from controls.spinkit import Spinkit


def main(page: ft.Page):
    page.vertical_alignment = ft.MainAxisAlignment.CENTER
    page.horizontal_alignment = ft.CrossAxisAlignment.CENTER

    page.add(
        ft.Stack(
            [
                ft.Container(height=200, width=200,
bgcolor=ft.colors.BLUE_100),
                Spinkit(
                    opacity=0.5,
                    tooltip="Spinkit tooltip",
                    top=0,
                    left=0,
                    color=ft.colors.PURPLE,
                    size=150,
                ),
            ]
        )
    )

ft.app(main)
```

You can find source code for this example here.

Examples for different types of properties and events

Enum properties

For example, `clip_behaviour` for `AppBar`.

In Python:

```
# clip_behavior
@property
def clip_behavior(self) -> Optional[ClipBehavior]:
    return self._get_attr("clipBehavior")

@clip_behavior.setter
def clip_behavior(self, value: Optional[ClipBehavior]):
    self._set_attr(
        "clipBehavior",
        value.value if isinstance(value, ClipBehavior) else value,
    )
```

In Dart:

```
var clipBehavior = Clip.values.firstWhere(
    (e) =>
        e.name.toLowerCase() ==
        widget.control.attrString("clipBehavior", "")!.toLowerCase(),
    orElse: () => Clip.none);
```

Json properties

For example, `shape` property for `Card`.

In Python:

```
def before_update(self):
    super().before_update()
    self._set_attr_json("shape", self.__shape)

# shape
@property
def shape(self) -> Optional[OutlinedBorder]:
    return self.__shape

@shape.setter
def shape(self, value: Optional[OutlinedBorder]):
    self.__shape = value
```

In Dart:

```
var shape = parseOutlinedBorder(control, "shape")
```

Children

For example, `content` for `AlertDialog`:

In Python:

```
    def _get_children(self):
        children = []
        if self.__content:
            self.__content._set_attr_internal("n", "content")
            children.append(self.__content)
        return children
```

In Dart:

```
    var contentCtrls =
        widget.children.where((c) => c.name == "content" &&
c.isVisible);
```

Events

For example, `on_click` event for `ElevatedButton`.

In Python:

```
# on_click
@property
def on_click(self):
    return self._get_event_handler("click")

@on_click.setter
def on_click(self, handler):
    self._add_event_handler("click", handler)
```

In Dart:

```
Function()? onPressed = !disabled
    ? () {
```

```
        debugPrint("Button ${widget.control.id} clicked!");
        if (url != "") {
        openWebBrowser(url,
            webWindowName: widget.control.attrString("urlTarget"));
        }
        widget.backend.triggerControlEvent(widget.control.id, "click");
    }
    : null;
```

Examples

A few Flet controls are implemented as in external packages and could serve as a starting point for your own controls:

- `Video` - Python control, Flutter package
- `Audio` - Python control, Flutter package
- `Rive` - Python control, Flutter package

Edit this page

Controls reference

Flet UI is built of controls. Controls are organized into hierarchy, or a tree, where each control has a parent (except Page) and container controls like Column, Dropdown can contain child controls, for example:

```
Page
 ├─ TextField
 ├─ Dropdown
 │   ├─ Option
 │   └─ Option
 └─ Row
     ├─ ElevatedButton
     └─ ElevatedButton
```

Control gallery live demo

Controls by categories

🗃️ Layout

22 items

🗃️ Navigation

8 items

🗃️ Information Displays

12 items

🗃️ Buttons

18 items

🗃️ Input and Selections

16 items

🗃️ Dialogs, Alerts and Panels

13 items

🗃️ Charts

5 items

🗃️ Animations

3 items

🗃️ Utility

22 items

Common control properties

Flet controls have the following properties:

`adaptive`

`adaptive` property can be specified for a control in the following cases:

- A control has matching Cupertino control with similar functionality/presentation and graphics as expected on iOS/macOS. In this case, if `adaptive` is `True`, either Material or Cupertino control will be created depending on the target platform.

  These controls have their Cupertino analogs and `adaptive` property:

  - `AlertDialog`
  - `AppBar`
  - `Checkbox`
  - `ListTile`
  - `NavigationBar`
  - `Radio`
  - `Slider`
  - `Switch`

- A control has child controls. In this case `adaptive` property value is passed on to its children that don't have their `adaptive` property set.

  The following container controls have `adaptive` property:

  - `Card`
  - `Column`
  - `Container`
  - `Dismissible`
  - `ExpansionPanel`
  - `FletApp`
  - `GestureDetector`
  - `GridView`
  - `ListView`

- `Page`
- `Row`
- `SafeArea`
- `Stack`
- `Tabs`
- `View`

`bottom`

Effective inside `Stack` only. The distance that the child's bottom edge is inset from the bottom of the stack.

`data`

Arbitrary data that can be attached to a control.

`disabled`

Every control has `disabled` property which is `False` by default - control and all its children are enabled. `disabled` property is mostly used with data entry controls like `TextField`, `Dropdown`, `Checkbox`, buttons. However, `disabled` could be set to a parent control and its value will be propagated down to all children recursively.

For example, if you have a form with multiple entry controls you can disable them all together by disabling container:

```
c = ft.Column(controls=[
    ft.TextField(),
    ft.TextField()
])
c.disabled = True
page.add(c)
```

`expand`

When a child Control is placed into a `Column` or a `Row` you can "expand" it to fill the available space. `expand` property could be a boolean value (`True` - expand control to fill all

available space) or an integer - an "expand factor" specifying how to divide a free space with other expanded child controls.

For more information and examples about `expand` property see "Expanding children" sections in `Column` or `Row`.

`expand_loose`

Effective only if `expand` is `True`.

If `expand_loose` is `True`, the child control of a `Column` or a `Row` will be given the flexibility to expand to fill the available space in the main axis (e.g., horizontally for a Row or vertically for a Column), but will not be required to fill the available space.

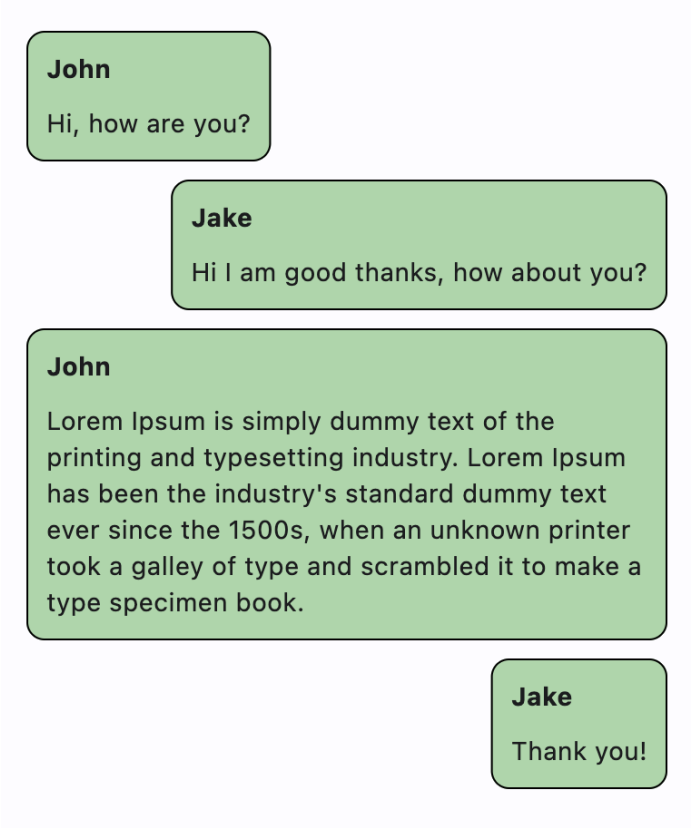
The default value is `False`.

Here is the example of Containers placed in Rows with `expand_loose = True`:

```
import flet as ft


class Message(ft.Container):
    def __init__(self, author, body):
        super().__init__()
        self.content = ft.Column(
            controls=[
                ft.Text(author, weight=ft.FontWeight.BOLD),
                ft.Text(body),
            ],
        )
        self.border = ft.border.all(1, ft.colors.BLACK)
        self.border_radius = ft.border_radius.all(10)
        self.bgcolor = ft.colors.GREEN_200
        self.padding = 10
        self.expand = True
        self.expand_loose = True


def main(page: ft.Page):
    chat = ft.ListView(
        padding=10,
        spacing=10,
        controls=[
            ft.Row(
```

```
                alignment=ft.MainAxisAlignment.START,
                controls=[
                    Message(
                        author="John",
                        body="Hi, how are you?",
                    ),
                ],
            ),
            ft.Row(
                alignment=ft.MainAxisAlignment.END,
                controls=[
                    Message(
                        author="Jake",
                        body="Hi I am good thanks, how about you?",
                    ),
                ],
            ),
            ft.Row(
                alignment=ft.MainAxisAlignment.START,
                controls=[
                    Message(
                        author="John",
                        body="Lorem Ipsum is simply dummy text of the
printing and typesetting industry. Lorem Ipsum has been the industry's
standard dummy text ever since the 1500s, when an unknown printer took
a galley of type and scrambled it to make a type specimen book.",
                    ),
                ],
            ),
            ft.Row(
                alignment=ft.MainAxisAlignment.END,
                controls=[
                    Message(
                        author="Jake",
                        body="Thank you!",
                    ),
                ],
            ),
        ],
    )

    page.window.width = 393
    page.window.height = 600
    page.window.always_on_top = False

    page.add(chat)
```

```
ft.app(main)
```



`height`

Imposed Control height in virtual pixels.

`left`

Effective inside `Stack` only. The distance that the child's left edge is inset from the left of the stack.

`parent`

Points to the direct ancestor(parent) of this control.

It defaults to `None` and will only have a value when this control is mounted (added to the page tree).

The `Page` control (which is the root of the tree) is an exception - it always has `parent=None`.

`right`

Effective inside `Stack` only. The distance that the child's right edge is inset from the right of the stack.

`top`

Effective inside `Stack` only. The distance that the child's top edge is inset from the top of the stack.

`tooltip`

The `tooltip` property (available in almost all controls) now supports both strings and `Tooltip` objects.

`visible`

Every control has `visible` property which is `True` by default - control is rendered on the page. Setting `visible` to `False` completely prevents control (and all its children if any) from rendering on a page canvas. Hidden controls cannot be focused or selected with a keyboard or mouse and they do not emit any events.

`width`

Imposed Control width in virtual pixels.

Transformations

`offset`

Applies a translation transformation before painting the control.

The translation is expressed as a `transform.Offset` scaled to the control's size. For example, an `Offset` with a `x` of `0.25` will result in a horizontal translation of one quarter the width of the control.

The following example displays container at `0, 0` top left corner of a stack as transform applies `-1 * 100, -1 * 100` (`offset * control_size`) horizontal and vertical

translations to the control:

```
import flet as ft

def main(page: ft.Page):

    page.add(
        ft.Stack(
            [
                ft.Container(
                    bgcolor="red",
                    width=100,
                    height=100,
                    left=100,
                    top=100,
                    offset=ft.transform.Offset(-1, -1),
                )
            ],
            width=1000,
            height=1000,
        )
    )

ft.app(main)
```

`opacity`

Defines the transparency of the control.

Value ranges from `0.0` (completely transparent) to `1.0` (completely opaque without any transparency) and defaults to `1.0`.

`rotate`

Transforms control using a rotation around the center.

The value of `rotate` property could be one of the following types:

- `number` - a rotation in clockwise radians. Full circle `360°` is `math.pi * 2` radians, `90°` is `pi / 2`, `45°` is `pi / 4`, etc.
- `transform.Rotate` - allows to specify rotation `angle` as well as `alignment` - the location of rotation center.

For example:

```
ft.Image(
    src="https://picsum.photos/100/100",
    width=100,
    height=100,
    border_radius=5,
    rotate=Rotate(angle=0.25 * pi, alignment=ft.alignment.center_left)
)
```

`scale`

Scale control along the 2D plane. Default scale factor is `1.0` - control is not scaled. `0.5` - the control is twice smaller, `2.0` - the control is twice larger.

Different scale multipliers can be specified for `x` and `y` axis, but setting `Control.scale` property to an instance of `transform.Scale` class:

```
from dataclasses import field

class Scale:
    scale: float = field(default=None)
    scale_x: float = field(default=None)
    scale_y: float = field(default=None)
    alignment: Alignment = field(default=None)
```

Either `scale` or `scale_x` and `scale_y` could be specified, but not all of them, for example:

```
ft.Image(
    src="https://picsum.photos/100/100",
    width=100,
    height=100,
    border_radius=5,
    scale=Scale(scale_x=2, scale_y=0.5)
)
```

Tutorials

Some of the awesome Flet tutorials to get your started:

To-Do app

In this tutorial we will show you, step-by-step, how to create a To-Do app in Python using Flet framew...

Calculator app

In this tutorial we will show you, step-by-step, how to create a Calculator app in Python using Flet fra...

Trello clone

Lets make a clone of Trello in Python with the Flet framework and then deploy it to fly.io!

Solitaire game

In this tutorial we will show you step-by-step creation of a famous Klondike solitaire game in Python w...

Chat app

In this tutorial we are going to create a trivial in-memory Chat app that will help you understand Flet f...

Reference

Below is a list of all available Flet references:

Colors

Color value

CLI

5 items

Environment variables

Below is the list of useful environment variables and their default values:

Icons

Material Icons

Types

5 items