GUÍA COMPLETA DE IMPLEMENTACIÓN

Flet para crear aplicaciones en Python

Manual práctico actualizado para construir aplicaciones web, escritorio y móviles con arquitectura modular, clases, componentes reutilizables, estado, rutas, almacenamiento y publicación.

Versión de referencia: Flet 0.85.3 - consultado el 10 de junio de 2026

Cómo usar esta guía

Esta guía está pensada para que puedas crear una aplicación real, no solo copiar ejemplos sueltos. Primero entenderás el modelo mental de Flet; luego montarás una estructura de proyecto mantenible; después aprenderás los controles, propiedades y patrones principales; finalmente tendrás una plantilla completa de app con clases, rutas, estado y publicación.

Las imágenes incluidas son diagramas y capturas esquemáticas generadas para esta guía. Representan resultados visuales esperados y flujos de implementación. No son imágenes copiadas de terceros.

> **Importante:** Flet tiene más de 150 controles y servicios. El PDF cubre los controles y propiedades que se usan en apps reales, además de una tabla de referencia para saber qué objeto usar. Para propiedades extremadamente específicas, usa la referencia oficial como complemento.

Índice rápido

1. Instalación y versión actual
2. Modelo mental: Page, View y Control
3. Estructura profesional de proyecto
4. Plantilla base con clases
5. Propiedades comunes de controles
6. Layouts, formularios y componentes
7. Navegación y rutas
8. Estado, eventos y actualización visual
9. Persistencia y almacenamiento
10. Async, servicios y APIs
11. Temas, responsive y accesibilidad
12. Build, publicación y checklist final
13. Anexos de código y tablas de controles

1. Instalación y versión actual

Flet permite construir aplicaciones web, desktop y móviles usando Python, sin escribir HTML, CSS, JavaScript, Dart, Swift ni Kotlin directamente. Internamente la UI se renderiza con Flutter, pero tú defines la interfaz desde Python.

La versión actual revisada para esta guía es **flet 0.85.3**, publicada en PyPI el 8 de junio de 2026. Para evitar errores por cambios de API, fija la versión en tu proyecto.

> **Evita mezclar tutoriales antiguos.** Flet cambió bastante entre la línea 0.2x y la línea moderna cercana a 1.0. Si un ejemplo usa helpers antiguos como `ft.padding.all()`, conviene traducirlo a la sintaxis moderna, por ejemplo `ft.Padding.all()`.

```
# Crear proyecto
mkdir mi_app_flet
cd mi_app_flet
python -m venv .venv
.venv\Scripts\activate
pip install "flet[all]==0.85.3"

# Ejecutar
python src/main.py
flet run src/main.py
flet run src/main.py --web

# Construir
flet build windows
flet build web
flet build apk
flet build aab
```

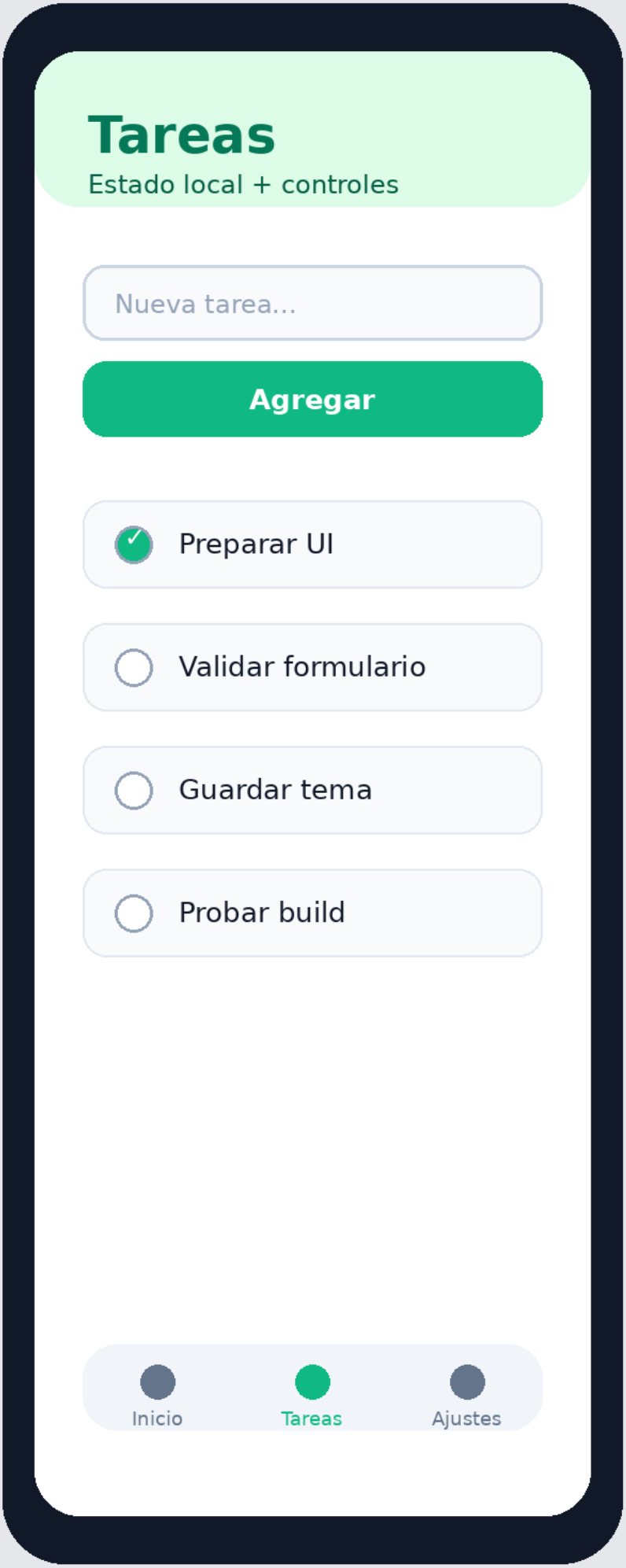
pyproject.toml recomendado

```
[project]
name = "mi-app-flet"
version = "0.1.0"
description = "Aplicación modular creada con Flet"
requires-python = ">=3.10"
dependencies = [
    "flet[all]==0.85.3"
]

[tool.flet.app]
path = "src"
module = "main.py"

[tool.flet]
product = "Mi App Flet"
```

2. Modelo mental de Flet

Una aplicación Flet se entiende como un árbol de controles. La raíz de la sesión es `Page`; dentro de ella puedes tener una o varias `View`; cada vista contiene controles visuales como textos, botones, campos, filas, columnas y contenedores.

Resultado visual esperado para una app modular de escritorio con sidebar, tarjetas, gráfico y lista.

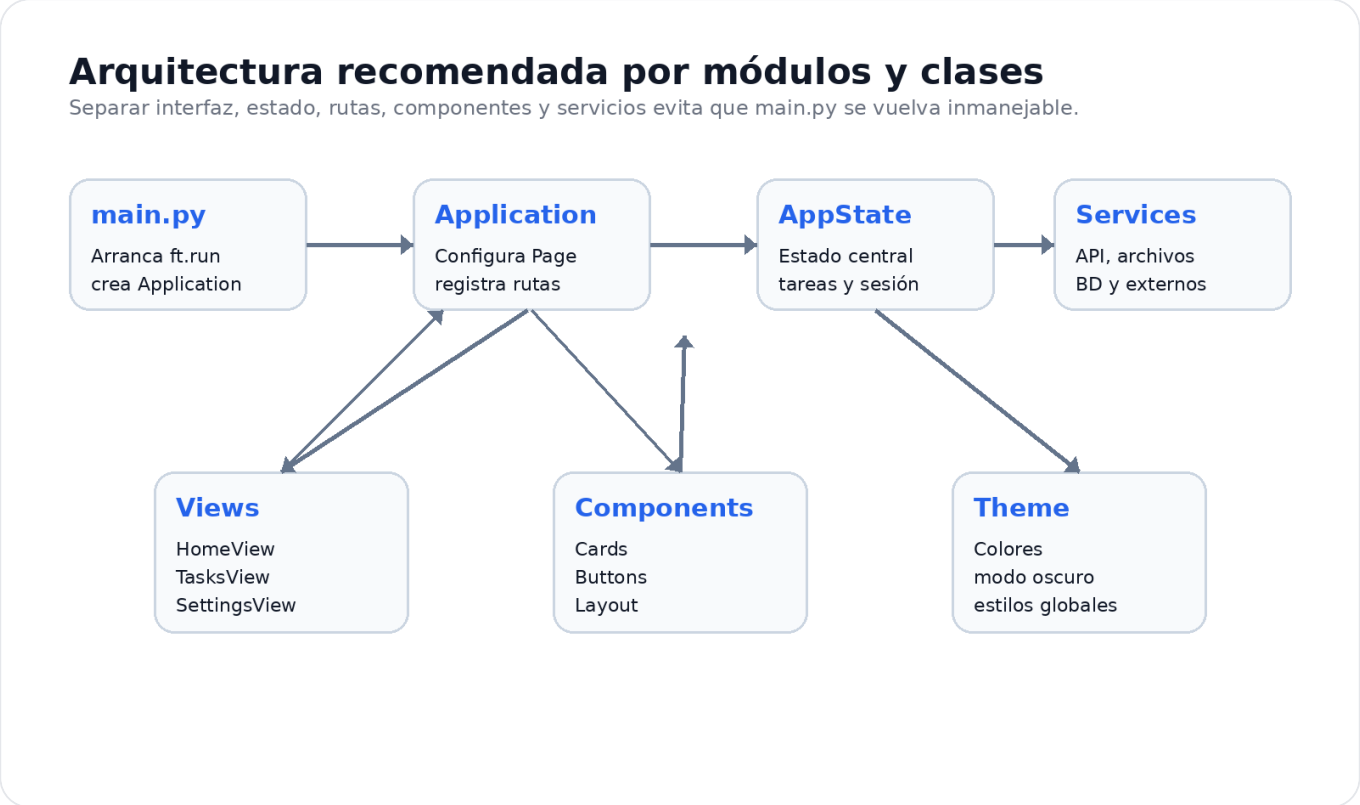

Tareas
Estado local + controles
Nueva tarea...
Agregar
Preparar UI
Validar formulario
Guardar tema
Probar build
Inicio
Tareas
Ajustes

Resultado visual esperado en móvil: el mismo estado y controles adaptados a pantalla vertical.

```
App
└── Page
    ├── page.route
    ├── page.views
    └── View(route="/tasks")
        ├── AppBar
        └── Column
            ├── TextField
            ├── Button
            └── ListView / Column / DataTable
```

3. Arquitectura profesional con módulos y clases

En proyectos pequeños puedes poner todo en `main.py`, pero para una app real conviene separar responsabilidades. Esto permite invocar funciones, componentes y módulos sin duplicar código.



Arquitectura recomendada: main.py solo arranca; Application coordina; Views muestran; Components reutilizan; AppState guarda estado.

Estructura de proyecto sugerida

```
mi_app_flet/
├── pyproject.toml
├── README.md
└── src/
    ├── main.py
    ├── app/
    │   ├── application.py
    │   ├── routes.py
    │   ├── state.py
    │   └── theme.py
    ├── views/
    │   ├── home_view.py
    │   ├── tasks_view.py
    │   └── settings_view.py
    ├── components/
    │   ├── buttons.py
    │   ├── cards.py
    │   └── layout.py
    ├── services/
    │   ├── api.py
    │   └── storage.py
    └── assets/
```

Estructura de carpetas sugerida para un proyecto Flet mantenible.

Módulo	Responsabilidad	Ejemplo
`main.py`	Entrada mínima de la app.	Llama `ft.run(main)`.
`application.py`	Configura Page, rutas, navegación y estado global.	Clase `Application`.
`state.py`	Modelos y datos de la app.	`AppState`, `Task`.
`views/`	Pantallas completas.	`HomeView`, `TasksView`.
`components/`	Piezas reutilizables.	Botones, tarjetas, layouts.
`services/`	APIs, base de datos, archivos, autenticación.	`ApiClient`, `StorageService`.

4. Plantilla base con clases

Esta plantilla usa un estilo imperativo orientado a objetos. Es fácil de entender, probar y extender. Más adelante puedes migrar partes a estilo declarativo si tu UI depende mucho del estado.

src/main.py

```
import flet as ft
from app.application import Application


def main(page: ft.Page):
    app = Application(page)
    app.run()


if __name__ == "__main__":
    ft.run(main)
```

src/app/state.py

```
from dataclasses import dataclass, field


@dataclass
class Task:
    id: int
    title: str
    done: bool = False


@dataclass
class AppState:
    username: str = "Julio"
    tasks: list[Task] = field(default_factory=list)
    next_task_id: int = 1

    def add_task(self, title: str) -> Task:
        title = title.strip()
        if not title:
            raise ValueError("La tarea no puede estar vacía.")
        task = Task(id=self.next_task_id, title=title)
        self.tasks.append(task)
        self.next_task_id += 1
        return task

    def toggle_task(self, task_id: int, value: bool) -> None:
        for task in self.tasks:
            if task.id == task_id:
                task.done = value
                return

    def remove_task(self, task_id: int) -> None:
        self.tasks = [t for t in self.tasks if t.id != task_id]
```

src/app/routes.py

```
HOME = "/"
TASKS = "/tasks"
SETTINGS = "/settings"
```

5. Application: el coordinador de la app

La clase `Application` debe encargarse de configurar la página, aplicar el tema, registrar eventos globales y reconstruir las vistas según la ruta actual.

```
import flet as ft
from app.state import AppState
from app.theme import apply_theme
from app import routes
from views.home_view import HomeView
from views.tasks_view import TasksView
from views.settings_view import SettingsView


class Application:
    def __init__(self, page: ft.Page):
        self.page = page
        self.state = AppState()

    def run(self):
        apply_theme(self.page)
        self.page.on_route_change = self.on_route_change
        self.page.on_view_pop = self.on_view_pop
        self.on_route_change(None)

    def on_route_change(self, e):
        self.page.views.clear()
        self.page.views.append(HomeView(self.state))

        if self.page.route == routes.TASKS:
            self.page.views.append(TasksView(self.state))
        elif self.page.route == routes.SETTINGS:
            self.page.views.append(SettingsView(self.state))

        self.page.update()

    async def on_view_pop(self, e: ft.ViewPopEvent):
        if len(self.page.views) > 1:
            self.page.views.pop()
            await self.page.push_route(self.page.views[-1].route)
```

> **Regla:** una pantalla no debería saber cómo se construye toda la app. Una vista solo debería ocuparse de su contenido y eventos locales.

6. Temas globales y consistencia visual

Define colores, modo claro/oscuro, padding global y título de ventana desde un módulo de tema. Así evitas estilos repetidos en cada vista.

```
import flet as ft


def apply_theme(page: ft.Page) -> None:
    page.title = "Mi App Flet"
    page.theme_mode = ft.ThemeMode.SYSTEM
    page.padding = 0
    page.spacing = 0

    page.theme = ft.Theme(
        color_scheme_seed=ft.Colors.GREEN,
    )
    page.dark_theme = ft.Theme(
        color_scheme_seed=ft.Colors.BLUE,
    )
```

Usa constantes y clases para unificar estilos: radios, sombras, colores, densidad y espaciados.

```
import flet as ft


class AppCard(ft.Container):
    def __init__(self, content: ft.Control, padding: int = 18):
        super().__init__(
            content=content,
            padding=ft.Padding.all(padding),
            border_radius=ft.BorderRadius.all(18),
            bgcolor=ft.Colors.SURFACE_CONTAINER_HIGHEST,
            shadow=ft.BoxShadow(
                blur_radius=16,
                spread_radius=1,
                color=ft.Colors.with_opacity(0.10, ft.Colors.BLACK),
                offset=ft.Offset(0, 4),
            ),
        )
```

7. Controles principales y cuándo usarlos

Mapa de controles por propósito. Te ayuda a elegir el objeto correcto antes de programar.

Control / familia	Uso	Propiedades y eventos frecuentes	Cuándo usarlo
Control	Propiedades base	`visible`, `disabled`, `opacity`, `expand`, `expand_loose`, `col`, `tooltip`, `badge`, `rtl`	Todos los controles.
LayoutControl	Tamaño, posición, animación	`width`, `height`, `aspect_ratio`, `left`, `top`, `right`, `bottom`, `margin`, `align`, `rotate`, `scale`, `offset`, `animate_*`	Controles visuales.
Page	Sesión raíz	`title`, `theme`, `dark_theme`, `theme_mode`, `route`, `views`, `padding`, `scroll`, `window`, `shared_preferences`	Entrada principal.
View	Pantallas/rutas	`route`, `controls`, `appbar`, `navigation_bar`, `drawer`, `floating_action_button`, `bgcolor`, `padding`, `can_pop`	Cada pantalla de la app.
Container	Caja visual	`content`, `padding`, `margin`, `bgcolor`, `border`, `border_radius`, `alignment`, `shadow`, `gradient`, `image`, `ink`	Tarjetas, secciones, fondos.
Row	Layout horizontal	`controls`, `alignment`, `vertical_alignment`, `spacing`, `wrap`, `run_spacing`, `tight`	Controles en fila.
Column	Layout vertical		

Control / familia	Uso	Propiedades y eventos frecuentes	Cuándo usarlo
		`controls`, `alignment`, `horizontal_alignment`, `spacing`, `wrap`, `scroll`, `tight`	Pantallas y formularios.
Stack	Superposición	`controls`, `clip_behavior`, hijos con `left/top/right/bottom`	Mapas, badges, planos, overlays.
Text	Texto	`value`, `size`, `weight`, `italic`, `color`, `text_align`, `max_lines`, `overflow`, `selectable`	Títulos, etiquetas, párrafos.
TextField	Entrada de texto	`value`, `label`, `hint_text`, `password`, `multiline`, `min_lines`, `max_lines`, `keyboard_type`, `prefix`, `suffix`, `error_text`	Formularios.
Button	Acción	`content`, `text`, `icon`, `bgcolor`, `color`, `style`, `disabled`, `url`, `on_click`, `on_hover`	Acciones principales.
IconButton	Acción compacta	`icon`, `selected_icon`, `tooltip`, `style`, `on_click`	Barras, tarjetas, tablas.
Checkbox/ Switch	Booleanos	`label`, `value`, `tristate`, `on_change`, `active_color`	Tareas, preferencias.
Dropdown	Selección	`label`, `options`, `value`, `on_change`, `hint_text`, `disabled`	Catálogos.
Slider	Rangos	`value`, `min`, `max`, `divisions`, `label`, `on_change`, `on_change_end`	Valores numéricos.
ListView	Listas largas	`controls`, `spacing`, `padding`, `auto_scroll`, `expand`, `scroll`	Feeds, registros, mensajes.
GridView	Grillas	`controls`, `runs_count`, `max_extent`, `spacing`, `run_spacing`, `child_aspect_ratio`	Galerías y paneles.
DataTable	Tablas	`columns`, `rows`, `sort_column_index`, `sort_ascending`, `show_checkbox_column`	Datos tabulares.
SnackBar/Dialog	Feedback	`content`, `title`, `actions`, `open`, `modal`, `duration`	Mensajes y confirmaciones.
Image/Video/ WebView	Medios	`src`, `src_base64`, `width`, `height`, `fit`, `repeat`, `border_radius`	Recursos visuales.

8. Propiedades base de todos los controles

Estas propiedades son compartidas por la clase base `Control`. Aprenderlas te sirve para casi cualquier objeto visual o funcional en Flet.

Propiedad	Tipo típico	Uso práctico
`visible`	bool	Oculta o muestra el control. Si es `False`, no se renderiza ni recibe eventos.
`disabled`	bool	Desactiva el control y sus hijos. Útil para bloquear formularios durante carga.
`opacity`	float 0.0-1.0	Transparencia. Combínala con `animate_opacity` para transiciones.
`expand`	bool/int/None	Permite ocupar espacio disponible dentro de `Row`, `Column`, `View` o `Page`.
`expand_loose`	bool	Expande solo si hay espacio, sin forzar ocuparlo todo.
`col`	int/dict	Define columnas en `ResponsiveRow`; por ejemplo `{"xs": 12, "md": 6}`.
`tooltip`	str/Tooltip	Muestra ayuda al pasar el mouse.
`badge`	BadgeValue	Muestra una insignia encima del control.
`rtl`	bool	Dirección de texto derecha a izquierda.

```
ft.Button(
    "Guardar",
    icon=ft.Icons.SAVE,
    tooltip="Guardar cambios",
    disabled=False,
    opacity=1.0,
    expand=False,
    on_click=save_changes,
)
```

9. Propiedades visuales de LayoutControl

Los controles visuales añaden propiedades de tamaño, posición, transformación y animaciones implícitas.

Grupo	Propiedades	Ejemplo de uso
Tamaño	`width`, `height`, `aspect_ratio`	Definir tarjetas, imágenes, inputs y paneles.
Posición absoluta	`left`, `top`, `right`, `bottom`	Solo dentro de `Stack`.
Espacio y alineación	`margin`, `align`	Acomodar controles dentro del padre.
Transformaciones	`rotate`, `scale`, `offset`, `flip`, `transform`	Animaciones, efectos, overlays.
Animaciones	`animate_opacity`, `animate_position`, `animate_scale`, `animate_rotation`	Transiciones suaves.
Eventos	`on_size_change`, `on_animation_end`	Responder a cambios de layout.

```
card = ft.Container(
    width=260,
    height=120,
    padding=ft.Padding.all(16),
    border_radius=ft.BorderRadius.all(18),
    bgcolor=ft.Colors.SURFACE_CONTAINER_HIGHEST,
    animate_opacity=300,
    content=ft.Text("Tarjeta animada"),
)

card.opacity = 0.5
card.update()
```

10. Layout: Container, Row, Column y Stack

El layout es la base de toda app Flet. Normalmente usas `Column` para pantallas, `Row` para acciones en línea, `Container` para tarjetas y `Stack` para superponer elementos.

```
ft.Column(
    expand=True,
    spacing=16,
    controls=[
        ft.Text("Panel", size=28, weight=ft.FontWeight.BOLD),
        ft.Row(
            spacing=12,
            controls=[
                ft.TextField(label="Buscar", expand=True),
                ft.Button("Filtrar"),
            ],
        ),
        ft.Container(
            padding=ft.Padding.all(20),
            border_radius=ft.BorderRadius.all(18),
            bgcolor=ft.Colors.SURFACE_CONTAINER,
            content=ft.Text("Contenido principal"),
        ),
    ],
)
```

Stack para superposición

```
ft.Stack(
    width=400,
    height=260,
    controls=[
        ft.Image(src="plano.png", width=400, height=260),
        ft.Container(
            left=40,
            top=80,
            width=120,
            height=50,
            bgcolor=ft.Colors.AMBER,
            border_radius=ft.BorderRadius.all(12),
            content=ft.Text("Etiqueta"),
        ),
    ],
)
```

11. Formularios realistas

Un formulario bien hecho debe validar, mostrar errores, bloquearse mientras guarda y limpiar estado al completar. No dependas solo de que el usuario escriba bien.

```
class LoginForm(ft.Container):
    def __init__(self):
        self.email = ft.TextField(label="Correo", keyboard_type=ft.KeyboardType.EMAIL)
        self.password = ft.TextField(label="Contraseña", password=True, can_reveal_password=True)
        self.message = ft.Text()
        super().__init__(
            padding=ft.Padding.all(24),
            border_radius=ft.BorderRadius.all(18),
            bgcolor=ft.Colors.SURFACE_CONTAINER,
            content=ft.Column(
                spacing=14,
                controls=[
                    ft.Text("Iniciar sesión", size=26, weight=ft.FontWeight.BOLD),
                    self.email,
                    self.password,
                    ft.Button("Entrar", on_click=self.submit),
                    self.message,
                ],
            ),
        )

    def submit(self, e):
        if not self.email.value or not self.password.value:
            self.message.value = "Completa todos los campos."
            self.message.color = ft.Colors.RED
        else:
            self.message.value = "Formulario válido."
            self.message.color = ft.Colors.GREEN
        self.update()
```

12. Navegación y rutas

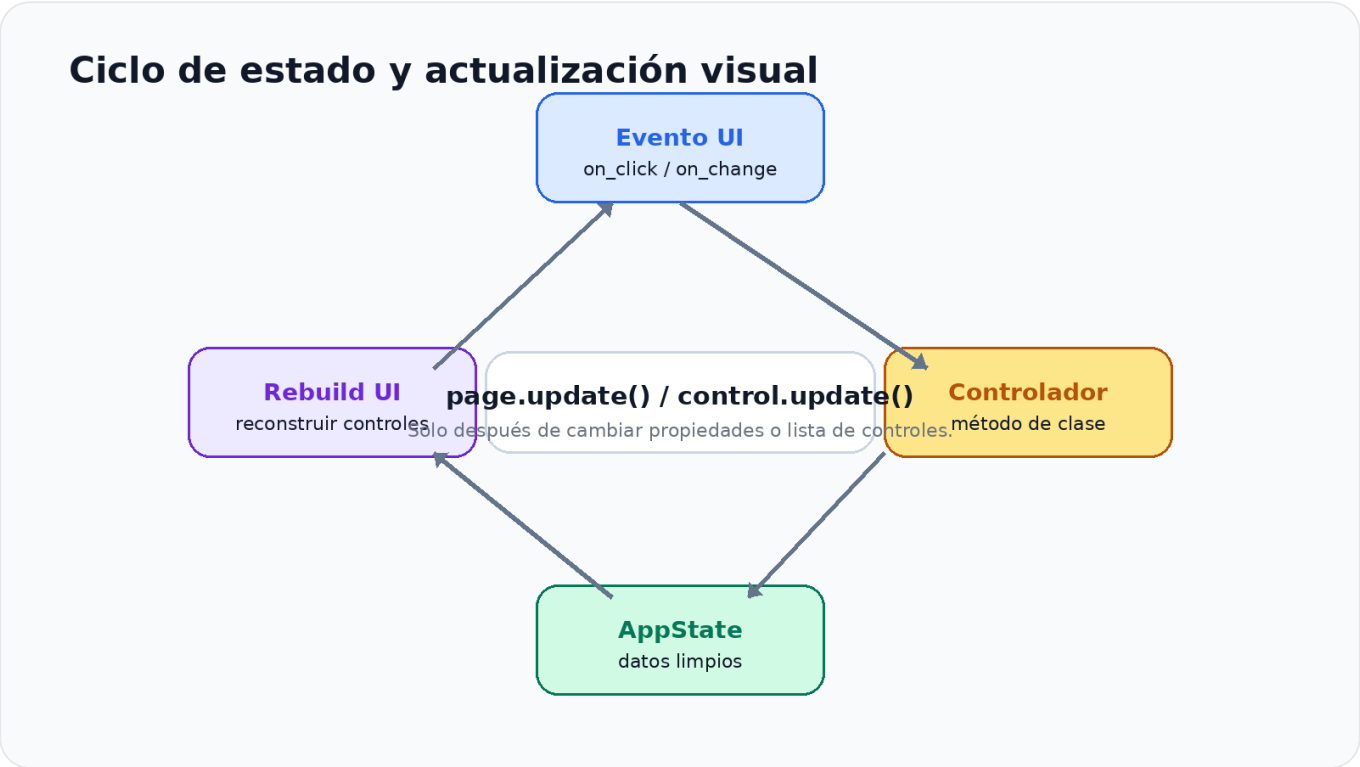
Para aplicaciones con varias pantallas, usa rutas. La ruta actual está en `page.route` y la pila de pantallas está en `page.views`. El patrón más estable es derivar `page.views` desde `page.route`.

Flujo recomendado de navegación: evento, ruta, reconstrucción de vistas y resultado.

```
async def open_settings(e):
    await page.push_route("/settings")


def on_route_change(e):
    page.views.clear()
    page.views.append(HomeView(state))

    if page.route == "/settings":
        page.views.append(SettingsView(state))

    page.update()


async def on_view_pop(e):
    page.views.pop()
    await page.push_route(page.views[-1].route)


page.on_route_change = on_route_change
page.on_view_pop = on_view_pop
```

13. Estado, eventos y actualización visual

El flujo normal de Flet es: ocurre un evento, cambias datos o propiedades, reconstruyes los controles necesarios y llamas `update()`. Si modificas un control específico, puedes llamar `control.update()`; si reconstruyes varias partes, usa `page.update()`.



Ciclo de estado recomendado para evitar inconsistencias visuales.

```
def add_task(self, e):
    try:
        self.state.add_task(self.input_task.value)
        self.input_task.value = ""
        self.rebuild_tasks()
        self.page.update()
    except ValueError as ex:
        self.page.show_dialog(ft.SnackBar(ft.Text(str(ex))))
```

Consejo: no guardes el estado real solo dentro de controles visuales. Los controles muestran datos; el estado debe vivir en modelos, clases o servicios.

14. Vista completa de tareas

Este ejemplo junta estado, TextField, Button, Column, Checkbox, SnackBar y reconstrucción de UI.

```
import flet as ft
from app import routes
from app.state import AppState
from components.cards import AppCard


class TasksView(ft.View):
    def __init__(self, state: AppState):
        self.state = state
        self.input_task = ft.TextField(label="Nueva tarea", expand=True)
        self.tasks_column = ft.Column(spacing=8)

        super().__init__(
            route=routes.TASKS,
            controls=[
                ft.AppBar(title=ft.Text("Tareas")),
                ft.Container(
                    padding=ft.Padding.all(24),
                    content=ft.Column(
                        spacing=16,
                        controls=[
                            AppCard(ft.Row(controls=[
                                self.input_task,
                                ft.Button("Agregar", on_click=self.add_task),
                            ])),
                            self.tasks_column,
                        ],
                    ),
                ),
            ],
        )
        self.rebuild_tasks()

    def add_task(self, e):
        try:
            self.state.add_task(self.input_task.value)
            self.input_task.value = ""
            self.rebuild_tasks()
            self.page.update()
        except ValueError as ex:
            self.page.show_dialog(ft.SnackBar(ft.Text(str(ex))))

    def rebuild_tasks(self):
        self.tasks_column.controls = [
            ft.Checkbox(label=t.title, value=t.done,
                on_change=lambda e, task_id=t.id: self.toggle_task(task_id, e.control.value))
            for t in self.state.tasks
        ] or [ft.Text("No hay tareas todavía.")]

    def toggle_task(self, task_id: int, value: bool):
        self.state.toggle_task(task_id, value)
        self.rebuild_tasks()
        self.page.update()
```

15. Responsive design

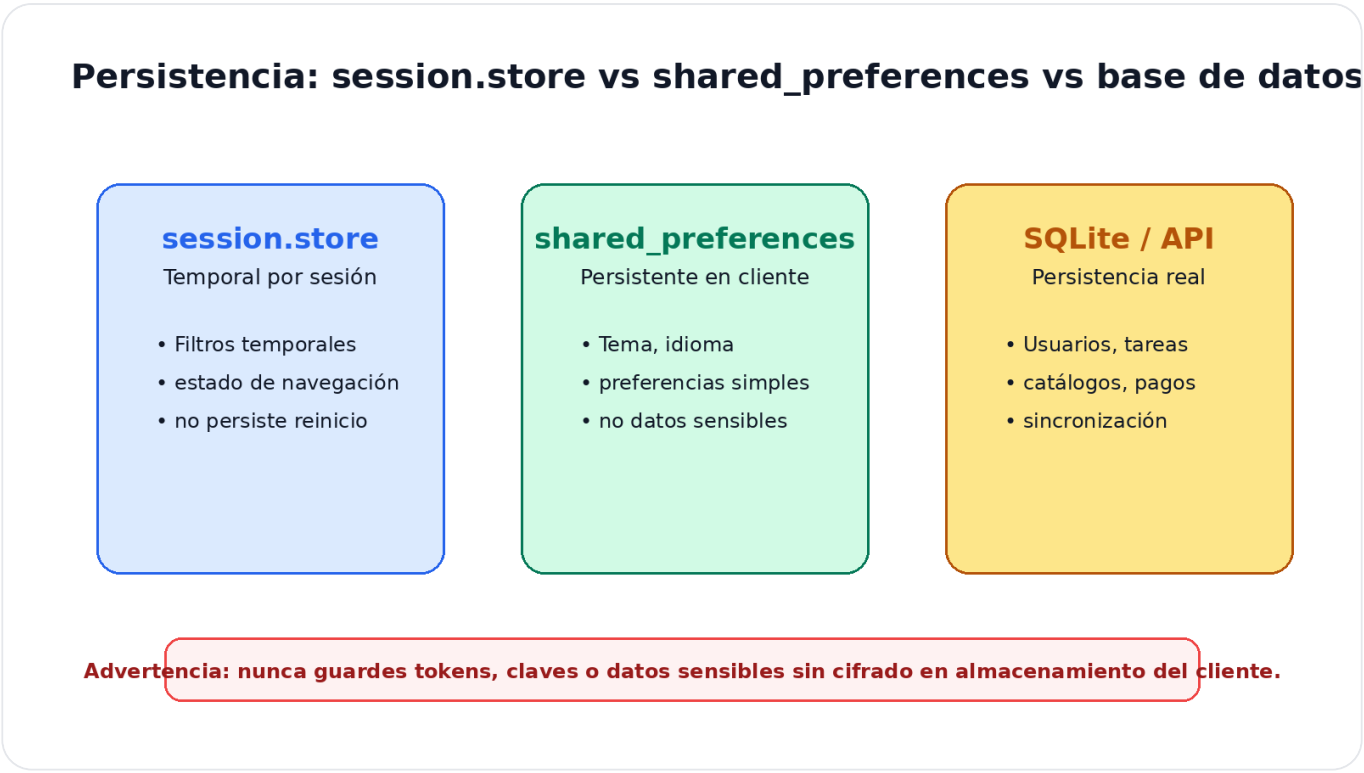
Para que tu app funcione en escritorio, tablet y móvil, usa `ResponsiveRow` y la propiedad `col`. Piensa cada tarjeta como una unidad que puede ocupar 12, 6, 4 o 3 columnas según el ancho.

ResponsiveRow permite definir cómo se comporta cada control por breakpoint.

```
ft.ResponsiveRow(
    controls=[
        ft.Container(
            col={"xs": 12, "md": 6, "lg": 4},
            padding=ft.Padding.all(20),
            content=ft.Text("Tarjeta 1"),
        ),
        ft.Container(
            col={"xs": 12, "md": 6, "lg": 4},
            padding=ft.Padding.all(20),
            content=ft.Text("Tarjeta 2"),
        ),
    ]
)
```

16. Persistencia: qué guardar y dónde

Flet ofrece almacenamiento temporal de sesión y almacenamiento persistente del cliente. Para datos de negocio reales, usa SQLite, archivos, API o backend.



Comparación práctica entre session.store, shared_preferences y base de datos/API.

```
# Guardar preferencia persistente en el cliente
await page.shared_preferences.set("mi_app.theme_mode", "dark")

# Leer preferencia
mode = await page.shared_preferences.get("mi_app.theme_mode")

# Borrar preferencia
await page.shared_preferences.remove("mi_app.theme_mode")
```

17. Async, APIs y operaciones lentas

Si una operación tarda, usa funciones async para no bloquear la interfaz. Esto aplica para APIs, archivos, espera, descargas y procesos de red.

```
import asyncio
import flet as ft


async def main(page: ft.Page):
    status = ft.Text("Sin cargar")

    async def load_data(e):
        status.value = "Cargando..."
        page.update()
        await asyncio.sleep(1)  # aquí iría una API real
        status.value = "Datos cargados"
        page.update()

    page.add(status, ft.Button("Cargar", on_click=load_data))


ft.run(main)
```

Situación	Recomendación
Llamar API HTTP	Usa `async def` y cliente HTTP async.
Guardar archivo grande	Muestra estado de carga y evita bloquear eventos.
Esperas o timers	Usa `await asyncio.sleep()`, no `time.sleep()`.
Publicar como web estática	Prefiere async porque Pyodide tiene limitaciones con threading.

18. Servicios, archivos y capacidades del dispositivo

Flet incluye servicios y extensiones para archivos, cámara, geolocalización, sensores, audio, video y más. La disponibilidad depende de la plataforma, por eso debes manejar errores y permisos.

Servicio/control	Uso típico	Precaución
`FilePicker`	Seleccionar, guardar o subir archivos.	En Linux desktop puede requerir Zenity.
`Camera`	Vista previa, fotos y video.	Requiere permisos móviles y soporte de plataforma.
`Geolocator`	Ubicación del dispositivo.	Verifica permisos y disponibilidad.
`Video`	Reproductor de video.	En Linux puede requerir flavor completo.
`WebView`	Mostrar contenido web embebido.	No todas las plataformas se comportan igual.
`Map`	Mapas interactivos.	Revisa proveedor, tiles y permisos.

```
# Patrón general de manejo de servicios
try:
    # llamar servicio o API de plataforma
    pass
except Exception as ex:
    page.show_dialog(ft.SnackBar(ft.Text(f"No disponible: {ex}")))
```

19. Publicación y build

Para empaquetar usa `flet build`. El proceso genera un proyecto Flutter temporal, copia assets, empaqueta tu app Python y produce el ejecutable o paquete según la plataforma.

Pipeline de publicación con flet build.

Target	Comando	Notas
Windows	`flet build windows`	Ejecutable para Windows.
Linux	`flet build linux`	Prueba dependencias del sistema.
macOS	`flet build macos`	Requiere entorno compatible.
Web	`flet build web`	Para publicar como app web.
Android APK	`flet build apk`	Instalable directo.
Android AAB	`flet build aab`	Formato para Play Store.
iOS	`flet build ipa`	Requiere macOS/Xcode.

20. Checklist de calidad antes de entregar una app

1. La app corre con `python src/main.py`.
2. La app corre con `flet run src/main.py --web`.
3. Las rutas funcionan con botón atrás y recarga del navegador.
4. No hay estado duplicado entre controles y modelos.
5. Los formularios validan campos vacíos, errores de tipo y estados de carga.
6. La UI responde bien en móvil, tablet y desktop.
7. Los datos sensibles no se guardan sin cifrar en cliente.
8. Las dependencias están en `pyproject.toml` y no salen de `pip freeze` sin revisar.
9. El build se prueba en la plataforma objetivo.
10. Las imágenes, iconos y assets están en `src/assets`.

21. Errores comunes y solución

Problema	Causa común	Solución
No cambia la interfaz	Se cambió una propiedad pero no se llamó update.	Usa `control.update()` o `page.update()`.
La ruta no abre	Ruta sin slash inicial o no está en el router.	Usa rutas como `/tasks` y centralízalas.
Se duplican controles	Agregas controles sin limpiar/reconstruir.	Reasigna `controls` o limpia antes de agregar.
App lenta con listas grandes	Demasiados controles visibles.	Usa `ListView`, paginación o carga incremental.
Build falla en móvil	Dependencias no compatibles o permisos faltantes.	Revisa paquetes binarios, permisos y logs.
Diseño se rompe en móvil	Anchos fijos excesivos.	Usa `expand`, `ResponsiveRow` y breakpoints.

22. Anexo: patrón de aplicación por capas

Para proyectos más grandes, usa capas. La UI no debería hablar directamente con base de datos o API. Inserta un servicio o repositorio.

```
class TaskRepository:
    def __init__(self, api_client):
        self.api = api_client

    async def list_tasks(self):
        return await self.api.get("/tasks")

    async def create_task(self, title: str):
        return await self.api.post("/tasks", json={"title": title})


class TaskController:
    def __init__(self, state: AppState, repository: TaskRepository):
        self.state = state
        self.repository = repository

    async def load_tasks(self):
        data = await self.repository.list_tasks()
        self.state.tasks = [Task(**item) for item in data]

    async def create_task(self, title: str):
        item = await self.repository.create_task(title)
        self.state.tasks.append(Task(**item))
```

23. Anexo: guía rápida de estilos

```
class UI:
    PAGE_PADDING = ft.Padding.all(24)
    CARD_PADDING = ft.Padding.all(18)
    CARD_RADIUS = ft.BorderRadius.all(18)

    @staticmethod
    def title(value: str) -> ft.Text:
        return ft.Text(value, size=28, weight=ft.FontWeight.BOLD)

    @staticmethod
    def muted(value: str) -> ft.Text:
        return ft.Text(value, size=14, color=ft.Colors.ON_SURFACE_VARIANT)

    @staticmethod
    def card(content: ft.Control) -> ft.Container:
        return ft.Container(
            padding=UI.CARD_PADDING,
            border_radius=UI.CARD_RADIUS,
            bgcolor=ft.Colors.SURFACE_CONTAINER_HIGHEST,
            content=content,
        )
```

24. Anexo: comandos útiles

```
# Ver versión de Flet
flet --version
python -c "import flet as ft; print(ft.__version__)"

# Crear app base
flet create mi_app

# Ejecutar con hot reload aproximado desde CLI
flet run src/main.py

# Ejecutar en navegador
flet run src/main.py --web

# Build con limpieza de cache si hay problemas
flet build windows --clear-cache

# Ver ayuda de build
flet build --help
```

25. Fuentes consultadas

La guía se basó principalmente en documentación oficial de Flet y PyPI. Las imágenes internas de implementación fueron creadas para este PDF.

- Flet Docs - Introduction: https://docs.flet.dev/
- Flet Docs - Installation: https://docs.flet.dev/getting-started/installation/
- Flet Docs - Controls: https://docs.flet.dev/controls/
- Flet Docs - Control base class: https://docs.flet.dev/controls/control/
- Flet Docs - LayoutControl: https://docs.flet.dev/controls/layoutcontrol/
- Flet Docs - Navigation and Routing: https://docs.flet.dev/cookbook/navigation-and-routing/
- Flet Docs - Client Storage: https://docs.flet.dev/cookbook/client-storage/
- Flet Docs - Async apps: https://docs.flet.dev/cookbook/async-apps/
- Flet Docs - Theming: https://docs.flet.dev/cookbook/theming/
- Flet Docs - Publishing: https://docs.flet.dev/publish/
- PyPI - flet: https://pypi.org/project/flet/
- GitHub Releases - flet-dev/flet: https://github.com/flet-dev/flet/releases