

Contents

Unofficial Tutorial Of Iced Library	4
See Also	5
Contributions	5
License	6
Adding Widgets	7
Batch Commands	9
Batch Subscriptions	12
Button	15
Changing Displaying Content	17
Changing Styles	19
Changing Themes	21
Changing The Window Dynamically	23
Checkbox	26
Closing The Window On Demand	29
Column	31
ComboBox	33
Container	39
Controlling Widgets By Commands	41
Custom Background	44
Custom Styles	49
Drawing Shapes	52

Drawing Widgets	56
Drawing With Caches	62
Executing Custom Commands	66
Explanation of Sandbox Trait	69
First App - Hello World!	71
From Sandbox To Application	72
Image	74
Initializing A Different Window	77
Loading Images Asynchronously	79
Memoryless Pages	86
More Than One Page	91
Mouse Pointer Over Widgets	94
Navigation History	99
On Pressed/Released Of Some Widgets	104
Passing Parameters Across Pages	106
PickList	110
Producing Messages By Keyboard Events	114
Producing Messages By Mouse Events	117
Producing Messages By Timers	120
Producing Widget Messages	122
ProgressBar	128
Radio	130
Row	133
Rule	135
Scrollable	137
Setting Up	141
Slider And VerticalSlider	142

Space	145
Svg	147
Taking Any Children	150
TextInput	157
Text	161
Texts In Widgets	164
Toggler	169
Tooltip	172
Updating Widgets From Events	175
Updating Widgets From Outside	182
Widgets With Children	188
Width And Height	196

Unofficial Tutorial Of Iced Library

Iced is a cross-platform GUI library for Rust. This tutorial serves as a quick start for the library. We try to keep each part of the tutorial as simple as possible.

This tutorial is being updated from Iced version 0.10.0 to 0.12.x. For a complete tutorial for version 0.10.0, please refer to this branch.

Contents:

- Setting Up
- First App - Hello World!
- Explanation of Sandbox Trait
- Adding Widgets
- Changing Displaying Content
- Widgets
 - Text
 - Button
 - TextInput
 - Checkbox
 - Toggler
 - Radio
 - PickList
 - ComboBox
 - Slider And VerticalSlider
 - ProgressBar
 - Tooltip
 - Rule
 - Image
 - Svg
- Layouts
 - Width And Height
 - Column
 - Row
 - Space
 - Container
 - Scrollable
- Changing Themes
- Styles
 - Changing Styles

- Custom Styles
- Multipage Apps
 - More Than One Page
 - Memoryless Pages
 - Passing Parameters Across Pages
 - Navigation History
- Applications
 - From Sandbox To Application
 - Controlling Widgets By Commands
 - Batch Commands
 - Executing Custom Commands
- Windows
 - Initializing A Different Window
 - Changing The Window Dynamically
 - Closing The Window On Demand
- Events
 - On Pressed/Released Of Some Widgets
 - Producing Messages By Mouse Events
 - Producing Messages By Keyboard Events
 - Producing Messages By Timers
 - Batch Subscriptions
- Canvas
 - Drawing Shapes
 - Drawing With Caches
- Custom Widgets
 - Drawing Widgets
 - Updating Widgets From Outside
 - Updating Widgets From Events
 - Producing Widget Messages
 - Mouse Pointer Over Widgets
 - Texts In Widgets
 - Custom Background
 - Widgets With Children
 - Taking Any Children
- Others
 - Loading Images Asynchronously

See Also

- Iced - the Iced library.
- awesome-iced - a list of projects depending on Iced.

Contributions

Pull requests for typos, incorrect information, or other ideas are welcome!

License

All code in the tutorial is provided under the MIT License.

Adding Widgets

Use column! and row! to group multiple widgets such as text and button.

```
use iced::{
    widget::{button, column, row, text},
    Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            text("Yes or no?"),
            row![
                button("Yes"),
                button("No"),
            ],
        ].into()
    }
}
```

:arrow_right: Next: Changing Displaying Content

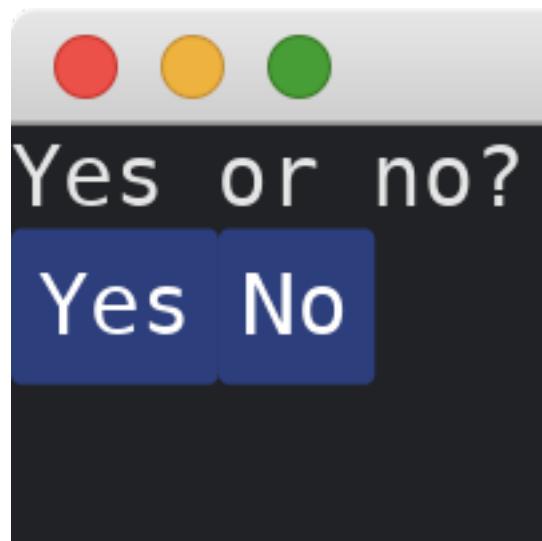


Figure 1: Adding widgets

:blue_book: Back: Table of contents

Batch Commands

Sometimes, we want to execute two or more Commands at the same time. In this case, we can use Command::batch function.

```
use iced::{
    executor,
    widget::{button, column, text_input},
    Application, Command, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

const MY_TEXT_ID: &str = "my_text";

#[derive(Debug, Clone)]
enum MyAppMessage {
    UpdateText(String),
    SelectAll,
}

struct MyApp {
    some_text: String,
}

impl Application for MyApp {
    type Executor = executor::Default;
    type Message = MyAppMessage;
    type Theme = iced::Theme;
    type Flags = ();
}

fn new(_flags: Self::Flags) -> (Self, iced::Command<Self::Message>) {
    (
        Self {
            some_text: String::new(),
        },
        Command::none(),
    )
}
```

```

        )
    }

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, message: Self::Message) -> iced::Command<Self::Message> {
    match message {
        MyAppMessage::UpdateText(s) => self.some_text = s,
        MyAppMessage::SelectAll => {
            return Command::batch(vec![
                text_input::focus(text_input::Id::new(MY_TEXT_ID)),
                text_input::select_all(text_input::Id::new(MY_TEXT_ID)),
            ])
        }
    }
    Command::none()
}

fn view(&self) -> iced::Element<Self::Message> {
    column![
        text_input("", &self.some_text)
            .id(text_input::Id::new(MY_TEXT_ID))
            .on_input(MyAppMessage::UpdateText),
        button("Select all").on_press(MyAppMessage::SelectAll),
    ]
    .into()
}
}

```

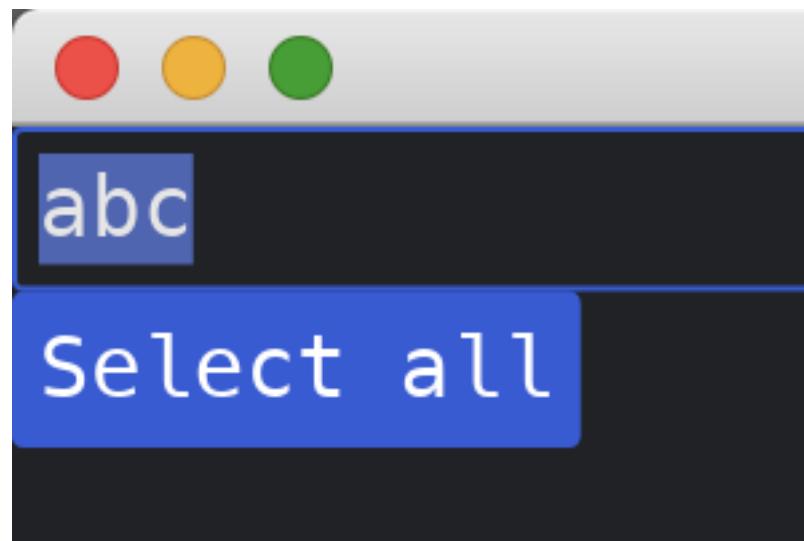


Figure 2: Batch commands

:arrow_right: Next: Executing Custom Commands

:blue_book: Back: Table of contents

Batch Subscriptions

This tutorial follows from the previous two tutorials (keyboard events and timers). We combine the two Subscriptions of keyboard events and timers. This is done by Subscription::batch function.

In the following app, press the space key to start or stop the timer.

```
use iced::*;

event::{self, Status},
executor,
keyboard::{key::Named, Key},
time::{self, Duration},
widget::text,
Application, Command, Event, Settings, Subscription,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    StartOrStop,
    Update,
}

struct MyApp {
    seconds: u32,
    running: bool,
}

impl Application for MyApp {
    type Executor = executor::Default;
    type Message = MyAppMessage;
    type Theme = iced::Theme;
    type Flags = ();
}

fn new(_flags: Self::Flags) -> (Self, iced::Command<Self::Message>) {
    (
        MyApp {
            seconds: 0,
            running: false,
        },
        iced::Command::batch(
            iced::Subscription::batch(
                keyboard::Subscription::new(|key| {
                    if key == Key::Space {
                        if self.running {
                            self.seconds -= 1;
                            self.update();
                        } else {
                            self.seconds = 0;
                            self.start();
                        }
                    }
                })
            )
        )
    )
}
```

```

        Self {
            seconds: 0,
            running: false,
        },
        Command::none(),
    )
}

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, message: Self::Message) -> iced::Command<Self::Message> {
    match message {
        MyAppMessage::StartOrStop => self.running = !self.running,
        MyAppMessage::Update => self.seconds += 1,
    }
    Command::none()
}

fn view(&self) -> iced::Element<Self::Message> {
    text(self.seconds).into()
}

fn subscription(&self) -> iced::Subscription<Self::Message> {
    let subscr_key = event::listen_with(|event, status| match (event, status) {
        (
            Event::Keyboard(iced::keyboard::Event::KeyPressed {
                key: Key::Named(Named::Space),
                ..
            }),
            Status::Ignored,
        ) => Some(MyAppMessage::StartOrStop),
        _ => None,
    });
}

if self.running {
    Subscription::batch(vec![
        subscr_key,
        time::every(Duration::from_secs(1)).map(|_| MyAppMessage::Update),
    ])
} else {
    subscr_key
}
}
}

```

:arrow_right: Next: Drawing Shapes

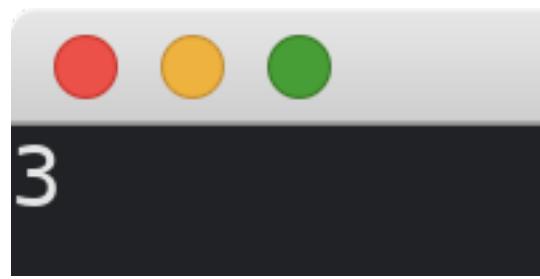


Figure 3: Batch subscriptions

:blue_book: Back: Table of contents

Button

The Button widget supports reactions to pressing/touching events. It has two methods of constructions. If the method `on_press` is set, the button is enabled, and is disabled otherwise. We can also set padding around the text of the button.

```
use iced::{
    widget::{button, column, Button},
    Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    DoSomething,
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            Button::new("Disabled button"),
            button("Construct from function"),
        ]
    }
}
```

```
        button("Enabled button").on_press(MyAppMessage::DoSomething),  
        button("With padding").padding(20),  
    ]  
.into()  
}  
}
```

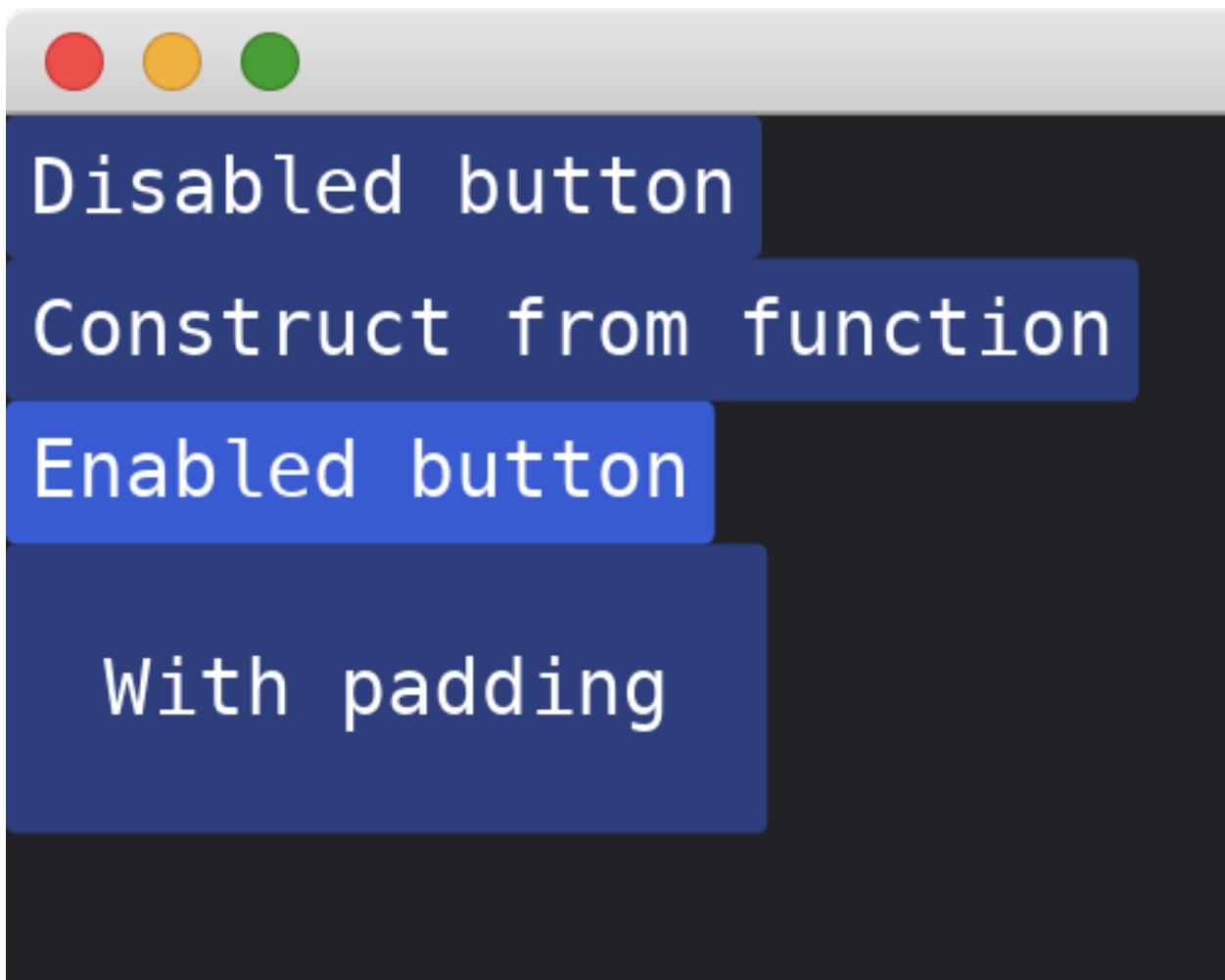


Figure 4: Button

:arrow_right: Next: TextInput

:blue_book: Back: Table of contents

Changing Displaying Content

To change the content in view dynamically, we can do the following:

- Add some fields (e.g., counter) to the main struct MyApp.
- Display the fields in view (e.g., text(self.counter)).
- Produce some messages in view (e.g., button(...).on_press(MyAppMessage::ButtonPressed)).
- Update the fields when messages are received in update (e.g., self.counter += 1).

```
use iced::{
    widget::{button, column, text},
    Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    ButtonPressed,
}

struct MyApp {
    counter: usize,
}

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self { counter: 0 }
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, message: Self::Message) {
```

```

match message {
    MyAppMessage::ButtonPressed => self.counter += 1,
}
}

fn view(&self) -> iced::Element<Self::Message> {
    column![
        text(self.counter),
        button("Increase").on_press(MyAppMessage::ButtonPressed),
    ]
    .into()
}
}

```



Figure 5: Producing and receiving messages

:arrow_right: Next: Text

:blue_book: Back: Table of contents

Changing Styles

Most widgets support style method to change their styles. These methods take parameters from enums of theme module. For example, widget::Text takes theme::Text as the parameter of its style method, and widget::Button takes theme::Button as the parameter of its style method.

Since theme::Text implements From<Color>, we can also use Color struct directly for the style method of widget::Text.

```
use iced::{
    theme,
    widget::{button, column, row, text},
    Color, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    DummyMessage,
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}
}
```

```
fn view(&self) -> iced::Element<Self::Message> {
    column![
        text("Ready?").style(Color::from_rgb(1., 0.6, 0.2)),
        row![
            button("Cancel")
                .style(theme::Button::Secondary)
                .on_press(MyAppMessage::DummyMessage),
            button("Go!~~")
                .style(theme::Button::Primary)
                .on_press(MyAppMessage::DummyMessage),
        ],
    ]
    .into()
}
```

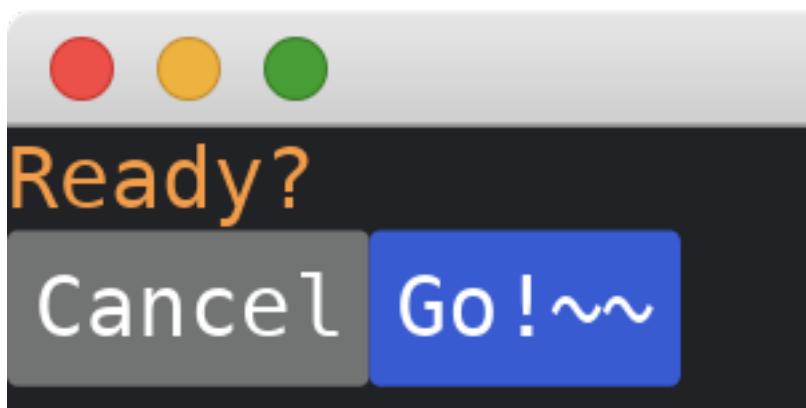


Figure 6: Changing styles

:arrow_right: Next: Custom Styles

:blue_book: Back: Table of contents

Changing Themes

We can implement theme method in Sandbox to return the desired theme.

```
use iced::{Sandbox, Settings};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        "Hello".into()
    }

    fn theme(&self) -> iced::Theme {
        iced::Theme::Dark
        // or
        // iced::Theme::Light
    }
}
```

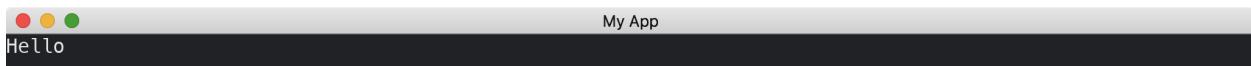


Figure 7: Changing themes

:arrow_right: Next: Changing Styles

:blue_book: Back: Table of contents

Changing The Window Dynamically

We can use functions provided in window module to change the window after it is initialized. For example, to resize the window. These functions return Command, which can be used as the return value in update method. Developers might be interested in other Commands in window module.

The resize function needs an ID of the window we are going to resize. Internally, Iced reserves window::Id::MAIN for the first window spawned.

```
use iced::*;

executor,
widget::{button, row, text_input},
window, Application, Command, Settings, Size,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    UpdateWidth(String),
    UpdateHeight(String),
    ResizeWindow,
}

struct MyApp {
    width: String,
    height: String,
}

impl Application for MyApp {
    type Executor = executor::Default;
    type Message = MyAppMessage;
    type Theme = iced::Theme;
    type Flags = ();
}
```

```

fn new(_flags: Self::Flags) -> (Self, iced::Command<Self::Message>) {
    (
        Self {
            width: "1024".into(),
            height: "768".into(),
        },
        Command::none(),
    )
}

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, message: Self::Message) -> iced::Command<Self::Message> {
    match message {
        MyAppMessage::UpdateWidth(w) => self.width = w,
        MyAppMessage::UpdateHeight(h) => self.height = h,
        MyAppMessage::ResizeWindow => {
            return window::resize(
                iced::window::Id::MAIN,
                Size::new(self.width.parse().unwrap(), self.height.parse().unwrap())
            )
        }
        Command::none()
    }
}

fn view(&self) -> iced::Element<Self::Message> {
    row![
        text_input("Width", &self.width).on_input(MyAppMessage::UpdateWidth),
        text_input("Height", &self.height).on_input(MyAppMessage::UpdateHeight),
        button("Resize window").on_press(MyAppMessage::ResizeWindow),
    ]
    .into()
}
}

```

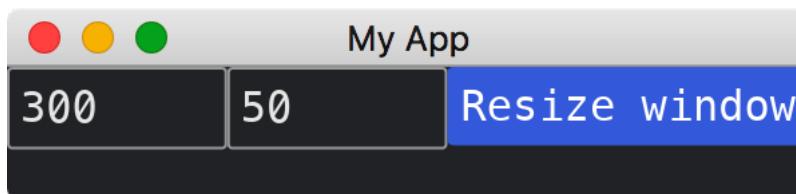


Figure 8: Changing the window dynamically

:arrow_right: Next: Closing The Window On Demand

:blue_book: Back: Table of contents

Checkbox

The Checkbox widget represents a boolean value. It has two methods of constructions. If the `on_toggle` method is set, it is enabled, and is disabled otherwise. It supports reactions to clicking and touching. It is able to change styles of the box and the text. It can also change the space between them.

```
use iced::{
    font::Family,
    widget::{
        checkbox,
        checkbox::Icon,
        column,
        text::{LineHeight, Shaping},
        Checkbox,
    },
    Font, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    DoNothing,
    Update4(bool),
    Update5(bool),
}

#[derive(Default)]
struct MyApp {
    checkbox4: bool,
    checkbox5: bool,
}

impl Sandbox for MyApp {
    type Message = MyAppMessage;
```

```

fn new() -> Self {
    Self::default()
}

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, message: Self::Message) {
    match message {
        MyAppMessage::DoNothing => {}
        MyAppMessage::Update4(b) => self.checkbox4 = b,
        MyAppMessage::Update5(b) => self.checkbox5 = b,
    }
}

fn view(&self) -> iced::Element<Self::Message> {
    column![

        Checkbox::new("Construct from struct", false),
        checkbox("Construct from function", false),
        checkbox("Enabled checkbox", false).on_toggle(|_| MyAppMessage::DoNothing),
        checkbox("Functional checkbox", self.checkbox4).on_toggle(|b| MyAppMessage::Update4(b)),
        checkbox("Shorter parameter", self.checkbox5).on_toggle(MyAppMessage::Update5),
        checkbox("Larger box", false)
            .on_toggle(|_| MyAppMessage::DoNothing)
            .size(30),
        checkbox("Different icon", true)
            .on_toggle(|_| MyAppMessage::DoNothing)
            .icon(Icon {
                font: Font::DEFAULT,
                code_point: '*',
                size: None,
                line_height: LineHeight::default(),
                shaping: Shaping::default()
            }),
        checkbox("Different font", false)
            .on_toggle(|_| MyAppMessage::DoNothing)
            .font(Font {
                family: Family::Fantasy,
                ..Font::DEFAULT
            }),
        checkbox("Larger text", false)
            .on_toggle(|_| MyAppMessage::DoNothing)
            .text_size(24),
        checkbox("Special character ☐", false)
            .on_toggle(|_| MyAppMessage::DoNothing)
            .text_shaping(Shaping::Advanced),
        checkbox("Space between box and text", false)
    ]
}

```

```
        .on_toggle(|_| MyAppMessage::DoNothing)
        .spacing(30),
    ]
    .into()
}
}
```

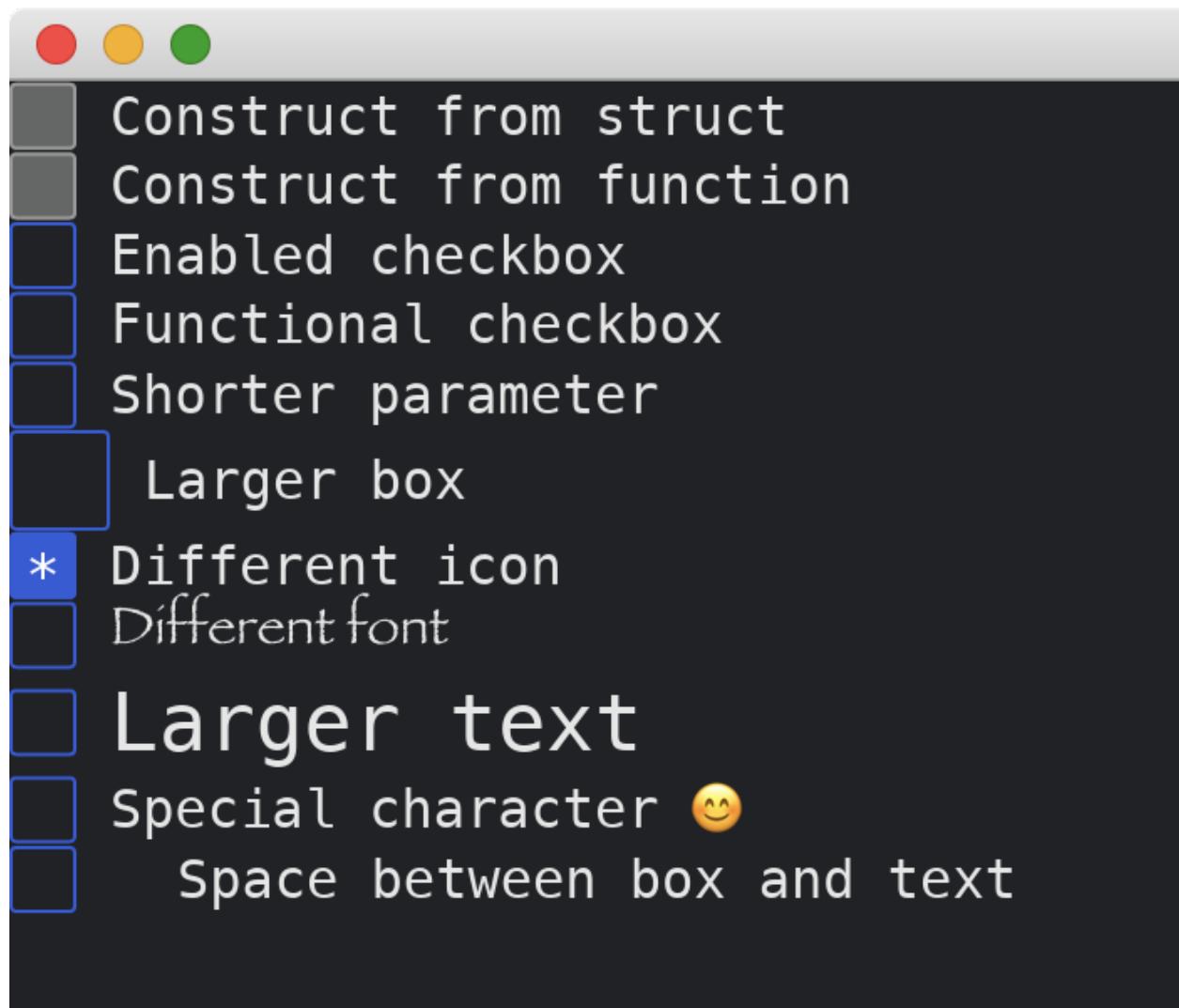


Figure 9: Checkbox

:arrow_right: Next: Toggler

:blue_book: Back: Table of contents

Closing The Window On Demand

This tutorial follows the previous tutorial. We use the close function in window module to close the window. This is also done by returning the Command obtained by the close function.

Similar to the resize function, the close function also needs an ID of the window. We pass window::Id::MAIN for the ID.

```
use iced::{
    executor,
    widget::{button, row},
    window, Application, Command, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    CloseWindow,
}

struct MyApp;

impl Application for MyApp {
    type Executor = executor::Default;
    type Message = MyAppMessage;
    type Theme = iced::Theme;
    type Flags = ();

    fn new(_flags: Self::Flags) -> (Self, iced::Command<Self::Message>) {
        (Self, Command::none())
    }

    fn title(&self) -> String {
        String::from("My App")
    }
}
```

```

fn update(&mut self, message: Self::Message) -> iced::Command<Self::Message> {
    match message {
        MyAppMessage::CloseWindow => window::close(window::Id::MAIN),
    }
}

fn view(&self) -> iced::Element<Self::Message> {
    row![button("Close window").on_press(MyAppMessage::CloseWindow), ].into()
}
}

```

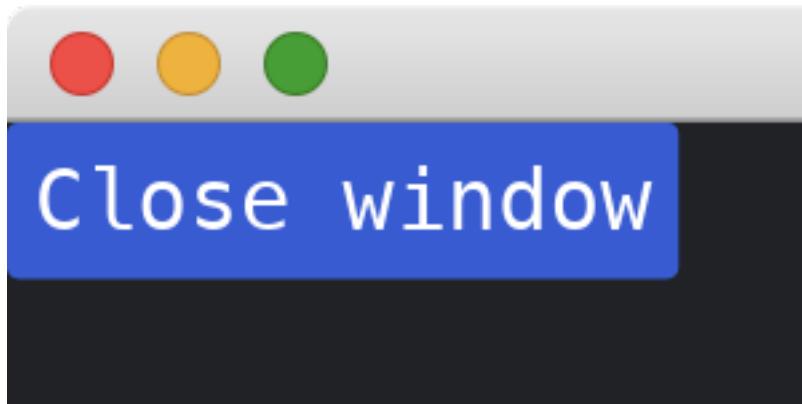


Figure 10: Closing the window on demand

:arrow_right: Next: On Pressed/Released Of Some Widgets

:blue_book: Back: Table of contents

Column

Column helps us placing widgets vertically. It can leave some space between its boundary and its inner content. It can also add spaces among its inner widgets. The inner widgets can be aligned left, middle or right.

```
use iced::{
    widget::{column, Column},
    Alignment, Length, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            Column::with_children(vec![
                "Construct from the with_children function".into(),
                "another element".into(),
            ]),
            Column::new()
                .push("Construct from the new function and the push method")
                .push("another element again"),
        ]
    }
}
```

```

        column(vec!["Construct from function".into()]),
        column!["Construct from macro"],
        column!["With padding"].padding(20),
        column!["Different alignment"]
            .width(Length::Fill)
            .align_items(Alignment::Center),
        column!["Space between elements", "Space between elements"].spacing(20)
    ]
    .into()
}
}

```

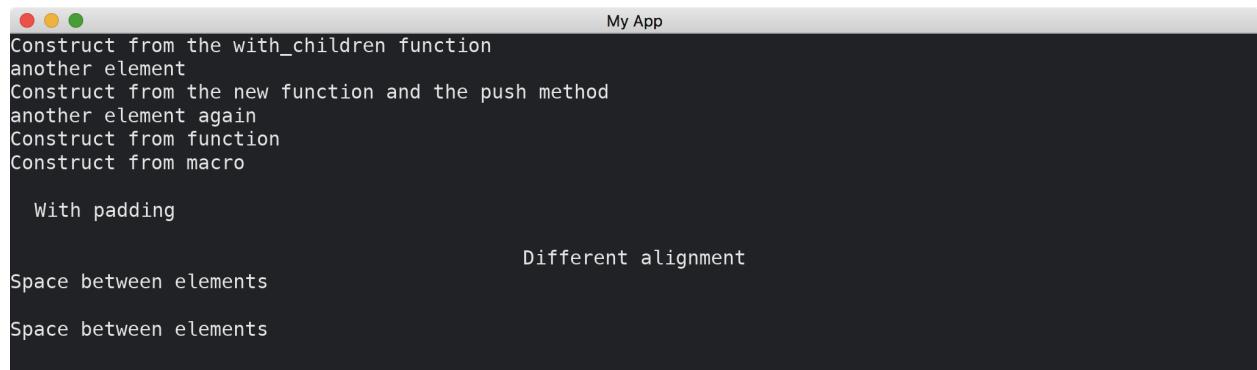


Figure 11: Column

:arrow_right: Next: Row

:blue_book: Back: Table of contents

ComboBox

The ComboBox widget represents a choice among multiple values. The values are shown in a dropdown menu and are searchable. The widget has two methods of constructions. It supports reactions to keyboard inputs and mouse selections. It is able to change fonts. We can add padding around the text inside. We can also add an optional icon.

```
use iced::{
    font::Family,
    widget::{
        column, combo_box,
        combo_box::State,
        text,
        text_input::{Icon, Side},
        ComboBox,
    },
    Font, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    DoNothing,
    Select4(String),
    Select5(String),
    Select6(String),
    Input6(String),
    Select7(String),
    Hover7(String),
    Select8(String),
    Hover8(String),
    Close8,
}

struct MyApp {
```

```

state1: State<u32>,
state2: State<u32>,
state3: State<String>,
state4: State<String>,
select4: Option<String>,
state5: State<String>,
select5: Option<String>,
state6: State<String>,
select6: Option<String>,
input6: String,
state7: State<String>,
select7: Option<String>,
state8: State<String>,
select8: Option<String>,
info8: String,
state9: State<u32>,
state10: State<u32>,
state11: State<u32>,
state12: State<u32>,
}

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self {
            state1: State::new(vec![]),
            state2: State::new(vec![]),
            state3: State::new(["Aa", "Ab", "Ba", "Bb"].map(|s| s.to_string()).to_vec()),
            state4: State::new(["Aa", "Ab", "Ba", "Bb"].map(|s| s.to_string()).to_vec()),
            select4: None,
            state5: State::new(["Aa", "Ab", "Ba", "Bb"].map(|s| s.to_string()).to_vec()),
            select5: None,
            state6: State::new(["Aa", "Ab", "Ba", "Bb"].map(|s| s.to_string()).to_vec()),
            select6: None,
            input6: "".into(),
            state7: State::new(["Aa", "Ab", "Ba", "Bb"].map(|s| s.to_string()).to_vec()),
            select7: None,
            state8: State::new(["Aa", "Ab", "Ba", "Bb"].map(|s| s.to_string()).to_vec()),
            select8: None,
            info8: "".into(),
            state9: State::new(vec![]),
            state10: State::new(vec![]),
            state11: State::new(vec![]),
            state12: State::new(vec![]),
        }
    }
}

```

```

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, message: Self::Message) {
    match message {
        MyAppMessage::DoNothing => {}
        MyAppMessage::Select4(s) => self.select4 = Some(s),
        MyAppMessage::Select5(s) => self.select5 = Some(s),
        MyAppMessage::Select6(s) => self.select6 = Some(s),
        MyAppMessage::Input6(s) => self.input6 = s,
        MyAppMessage::Select7(s) => self.select7 = Some(s),
        MyAppMessage::Hover7(s) => self.select7 = Some(s),
        MyAppMessage::Select8(s) => self.select8 = Some(s),
        MyAppMessage::Hover8(s) => self.select8 = Some(s),
        MyAppMessage::Close8 => self.info8 = "Done!".into(),
    }
}

fn view(&self) -> iced::Element<Self::Message> {
    column![
        ComboBox::new(&self.state1, "Construct from struct", None, |_| {
            MyAppMessage::DoNothing
        }),
        combo_box(&self.state2, "Construct from function", None, |_| {
            MyAppMessage::DoNothing
        }),
        combo_box(&self.state3, "With list of items", None, |_| {
            MyAppMessage::DoNothing
        }),
        combo_box(
            &self.state4,
            "Functional combobox (Press Enter or click an option)",
            self.select4.as_ref(),
            |s| MyAppMessage::Select4(s)
        ),
        combo_box(
            &self.state5,
            "Shorter parameter (Press Enter or click an option)",
            self.select5.as_ref(),
            MyAppMessage::Select5
        ),
        text(&self.input6),
        combo_box(
            &self.state6,
            "Respond to input",
            self.select6.as_ref(),
            MyAppMessage::Select6
        )
    ]
}

```

```

        )
        .on_input(MyAppMessage::Input6),
        combo_box(
            &self.state7,
            "Respond to option hover",
            self.select7.as_ref(),
            MyAppMessage::Select7
        )
        .on_option_hovered(MyAppMessage::Hover7),
        text(&self.info8),
        combo_box(
            &self.state8,
            "Respond to closing menu",
            self.select8.as_ref(),
            MyAppMessage::Select8
        )
        .on_option_hovered(MyAppMessage::Hover8)
        .on_close(MyAppMessage::Close8),
        combo_box(&self.state9, "Different font", None, |_| {
            MyAppMessage::DoNothing
        })
        .font(Font {
            family: Family::Fantasy,
            ..Font::DEFAULT
        }),
        combo_box(&self.state10, "Larger text", None, |_| {
            MyAppMessage::DoNothing
        })
        .size(24.),
        combo_box(&self.state11, "With padding", None, |_| {
            MyAppMessage::DoNothing
        })
        .padding(20),
        combo_box(&self.state12, "Icon", None, |_| MyAppMessage::DoNothing).icon(
            font: Font::DEFAULT,
            code_point: '\u{2705}',
            size: None,
            spacing: 10.,
            side: Side::Left,
        ),
    ],
    .into()
}
}

:arrow_right: Next: Slider And VerticalSlider
:blue_book: Back: Table of contents

```

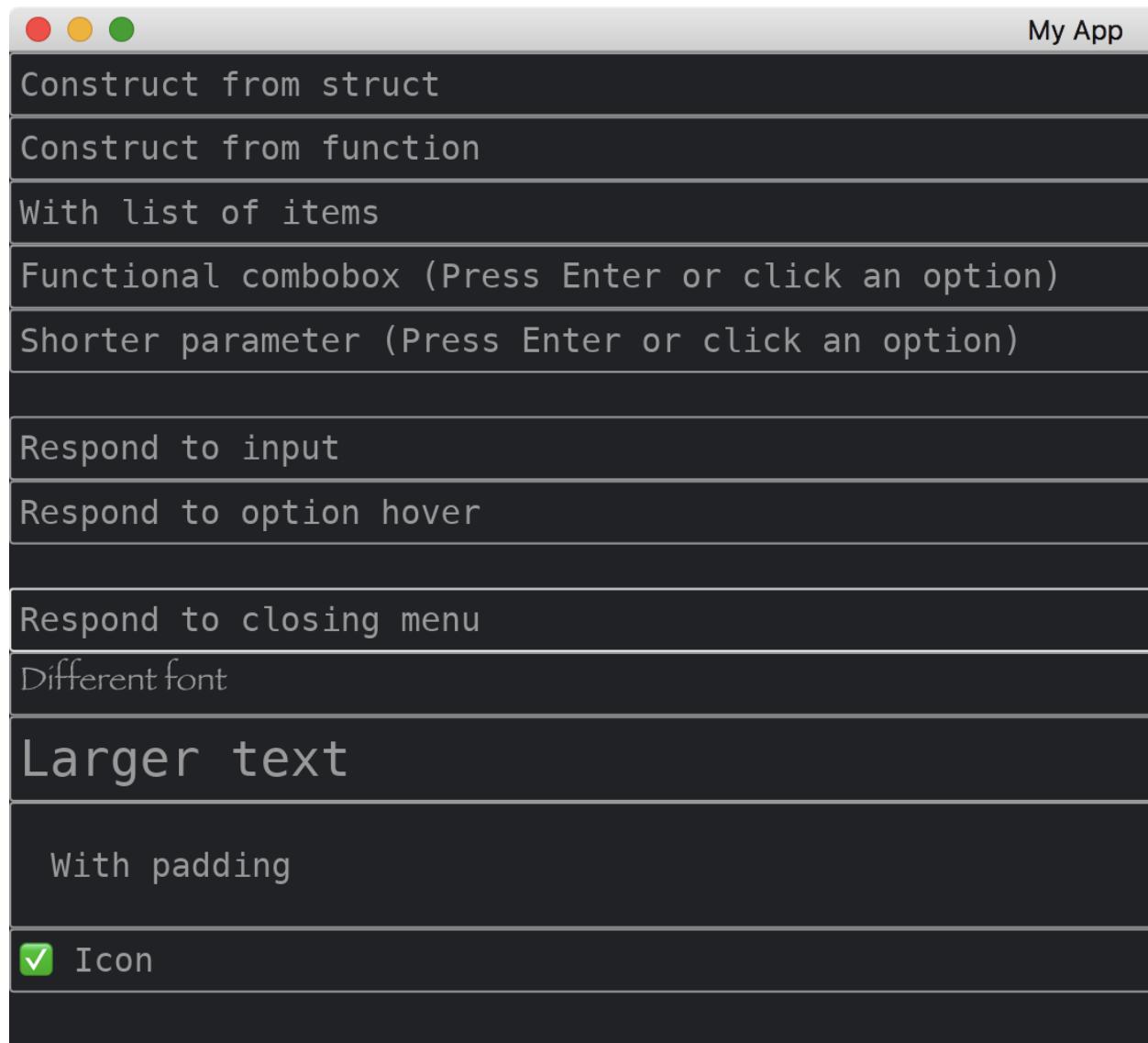


Figure 12: ComboBox

Container

Container is another way to help us laying out widgets, especially when we need to align a widget center both horizontally and vertically. Container accepts any widget including Column and Row.

```
use iced::{
    alignment::{Horizontal, Vertical},
    widget::{column, container, Container},
    Length, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            Container::new("Construct from struct"),
            container("Construct from function"),
            container("With padding").padding(20),
            container("Different alignment")
                .width(Length::Fill)
                .align_x(Horizontal::Center),
        ]
    }
}
```

```

        container("Different alignment for vertical axis")
            .height(Length::Fill)
            .align_y(Vertical::Center),
    ]
    .into()
}
}

```

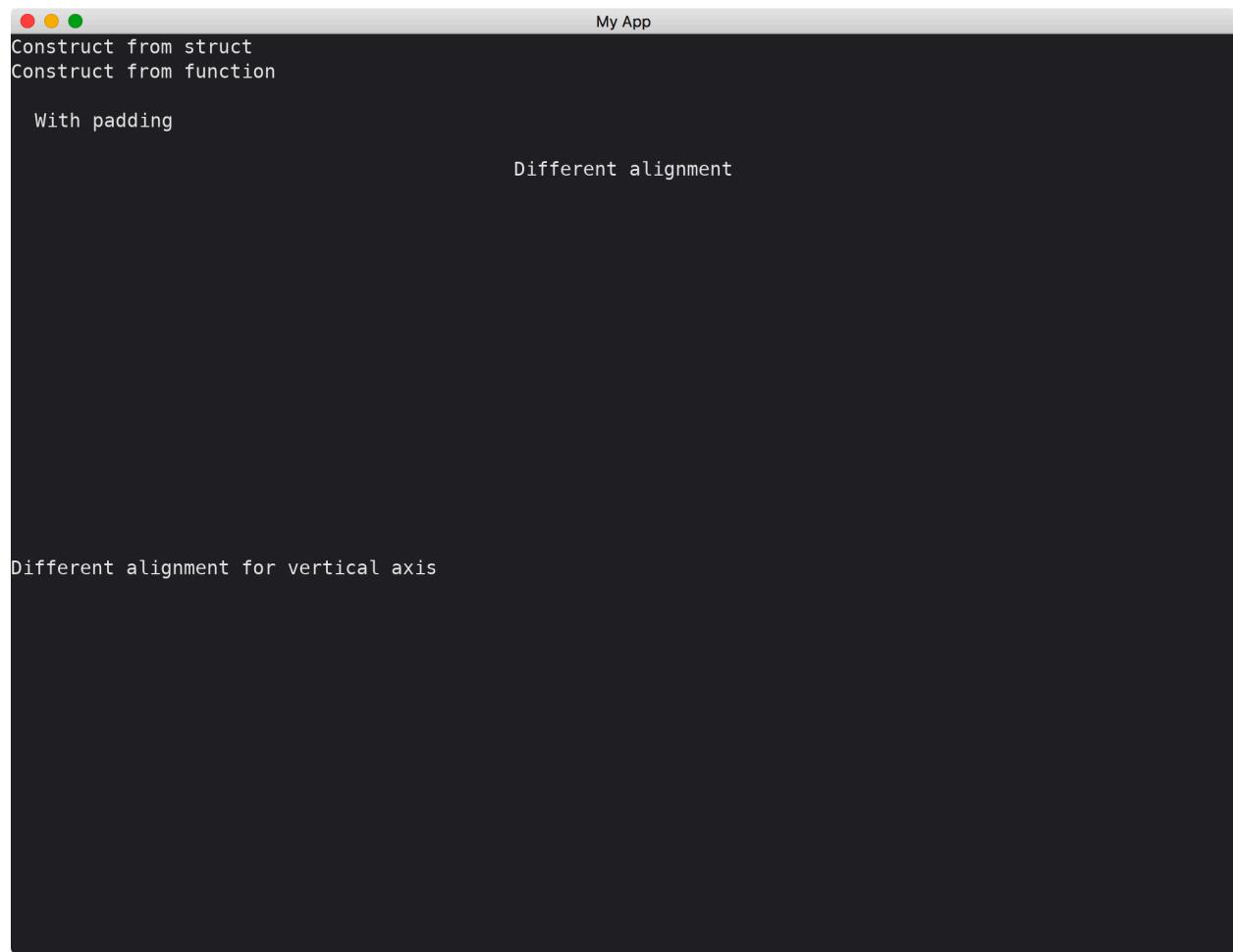


Figure 13: Container

If we want to center a widget both horizontally and vertically, we can use the following code:

```

container("Center").width(Length::Fill).height(Length::Fill).center_x().center_y()
:arrow_right: Next: Scrollable
:blue_book: Back: Table of contents

```

Controlling Widgets By Commands

We can use Command to control widgets. Some widgets have functions to change their behavior. These functions are located at their respective modules. For example, focus is a function in `text_input` module that makes a `TextInput` gaining the focus. This function takes a parameter `text_input::Id`, which can be specified by `id` method of `TextInput`. The function returns a `Command` and we can return the `Command` in `update` or `new` methods of Application.

```
use iced::{
    executor,
    widget::{button, column, text_input},
    Application, Command, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

const MY_TEXT_ID: &str = "my_text";

#[derive(Debug, Clone)]
enum MyAppMessage {
    EditText,
    UpdateText(String),
}

struct MyApp {
    some_text: String,
}

impl Application for MyApp {
    type Executor = executor::Default;
    type Message = MyAppMessage;
    type Theme = iced::Theme;
    type Flags = ();
}
```

```

fn new(_flags: Self::Flags) -> (Self, iced::Command<Self::Message>) {
    (
        Self {
            some_text: String::new(),
        },
        Command::none(),
    )
}

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, message: Self::Message) -> iced::Command<Self::Message> {
    match message {
        MyAppMessage::EditText => return text_input::focus(text_input::Id::new())
        MyAppMessage::UpdateText(s) => self.some_text = s,
    }
    Command::none()
}

fn view(&self) -> iced::Element<Self::Message> {
    column![
        button("Edit text").on_press(MyAppMessage::EditText),
        text_input("", &self.some_text)
            .id(text_input::Id::new(MY_TEXT_ID))
            .on_input(MyAppMessage::UpdateText),
    ]
    .into()
}
}

```

:arrow_right: Next: Batch Commands

:blue_book: Back: Table of contents

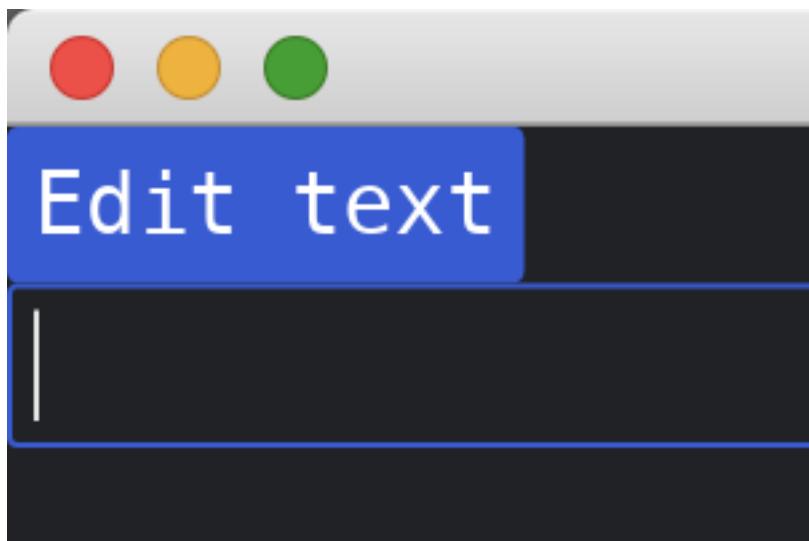


Figure 14: Controlling widgets by commands

Custom Background

We can also draw an image on a Widget.

Similar to the Image widget, we have to enable the image feature.

```
[dependencies]
iced = { version = "0.12.1", features = ["image", "advanced"] }
```

Assume we have an image, named ferris.png, in the Cargo root directory. We load this image when we initialize our widget.

```
struct MyWidgetWithImage {
    handle: Handle,
}

impl MyWidgetWithImage {
    fn new() -> Self {
        Self {
            handle: Handle::from_path("ferris.png"),
        }
    }
}
```

Then, we use the iced::widget::image::layout function to determine the layout of our widget.

```
fn layout(&self, _tree: &mut Tree, renderer: &Renderer, limits: &Layout::Limits) ->
    iced::widget::image::layout(
        renderer,
        limits,
        &self.handle,
        Length::Fixed(200.),
        Length::Fixed(200.),
        iced::ContentFit::Contain,
    )
}
```

And we draw the image by the iced::widget::image::draw function.

```
fn draw(
    &self,
    _state: &Tree,
```

```

    renderer: &mut Renderer,
    _theme: &Renderer::Theme,
    _style: &renderer::Style,
    layout: Layout<'_>,
    _cursor: mouse::Cursor,
    _viewport: &Rectangle,
) {
    // ...
}

iced::widget::image::draw(
    renderer,
    layout,
    &self.handle,
    iced::ContentFit::Contain,
    iced::widget::image::FilterMethod::Linear,
);
}

```

Both functions require the Renderer to implement iced::advanced::image::Renderer.

```
impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidgetWithImage
where
    Renderer: iced::advanced::Renderer + iced::advanced::image::Renderer<Handle = ...>
```

The full code is as follows:

```

use iced::*;

advanced::{
    layout, mouse,
    renderer::{self, Quad},
    widget::Tree,
    Layout, Widget,
},
widget::{container, image::Handle, Theme},
Border, Color, Element, Length, Rectangle, Sandbox, Settings, Shadow, Size,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }
}

```

```

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, _message: Self::Message) {}

fn view(&self) -> iced::Element<Self::Message> {
    container(MyWidgetWithImage::new())
        .width(Length::Fill)
        .height(Length::Fill)
        .center_x()
        .center_y()
        .into()
}
}

struct MyWidgetWithImage {
    handle: Handle,
}

impl MyWidgetWithImage {
    fn new() -> Self {
        Self {
            handle: Handle::from_path("ferris.png"),
        }
    }
}

impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidgetWithImage
where
    Renderer: iced::advanced::Renderer + iced::advanced::image::Renderer<Handle = Handle>
{
    fn size(&self) -> Size<Length> {
        Size {
            width: Length::Shrink,
            height: Length::Shrink,
        }
    }

    fn layout(
        &self,
        _tree: &mut Tree,
        renderer: &Renderer,
        limits: &layout::Limits,
    ) -> layout::Node {
        iced::widget::image::layout(
            renderer,
            limits,
        )
    }
}

```

```

        &self.handle,
        Length::Fixed(200.),
        Length::Fixed(200.),
        iced::ContentFit::Contain,
    )
}

fn draw(
    &self,
    _state: &Tree,
    renderer: &mut Renderer,
    _theme: &Theme,
    _style: &renderer::Style,
    layout: Layout<'_>,
    _cursor: mouse::Cursor,
    _viewport: &Rectangle,
) {
    renderer.fill_quad(
        Quad {
            bounds: layout.bounds(),
            border: Border {
                color: Color::from_rgb(1.0, 0.66, 0.6),
                width: 1.0,
                radius: 10.0.into(),
            },
            shadow: Shadow::default(),
        },
        Color::BLACK,
    );
}

iced::widget::image::draw(
    renderer,
    layout,
    &self.handle,
    iced::ContentFit::Contain,
    iced::widget::image::FilterMethod::Linear,
);
}

impl<'a, Message, Renderer> From<MyWidgetWithImage> for Element<'a, Message, Theme
where
    Renderer: iced::advanced::Renderer + iced::advanced::image::Renderer<Handle = Handle>
{
    fn from(widget: MyWidgetWithImage) -> Self {
        Self::new(widget)
    }
}

```

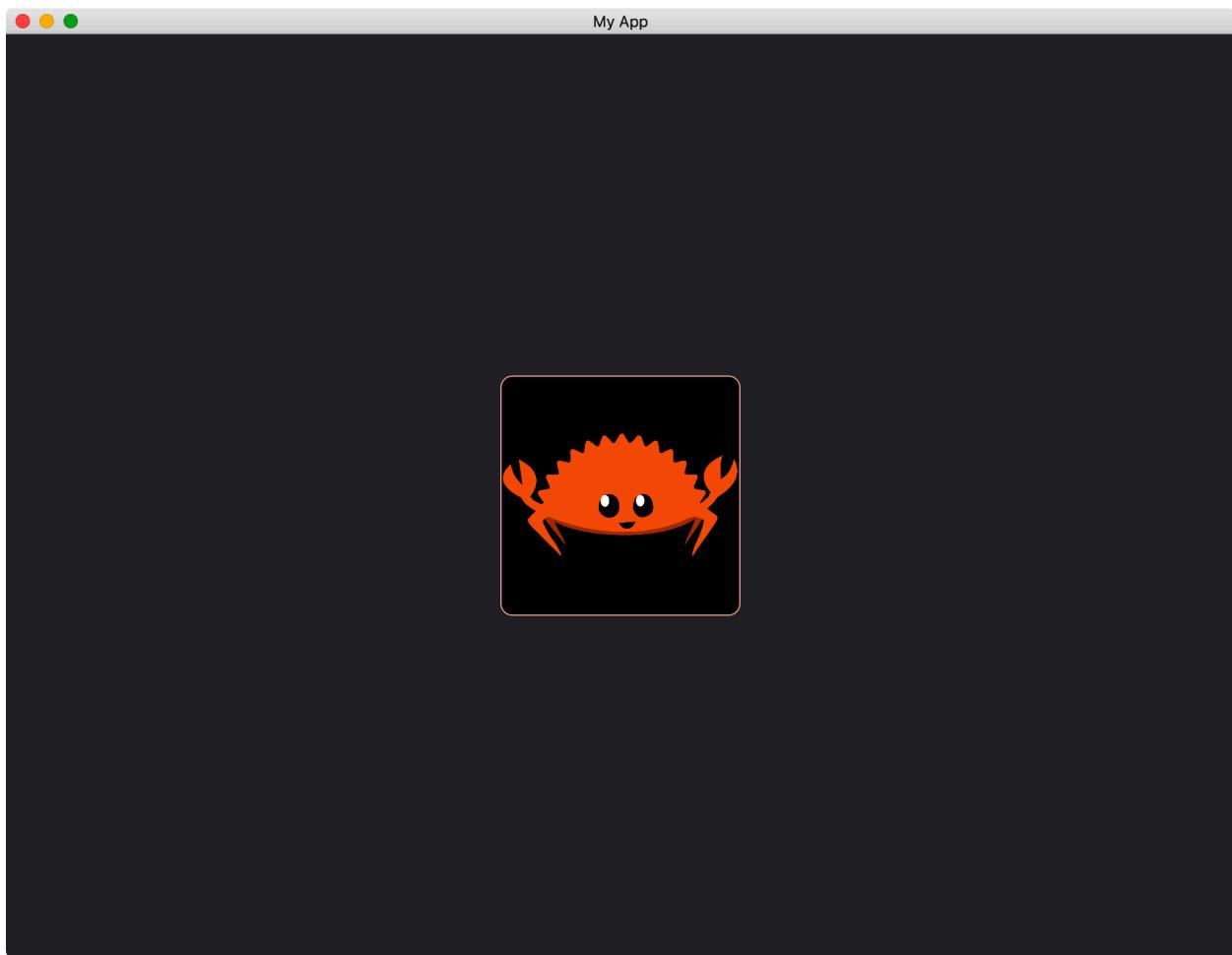


Figure 15: Custom Background

:arrow_right: Next: Widgets With Children

:blue_book: Back: Table of contents

Custom Styles

We mentioned in the previous tutorial about how to change styles of widgets by the enums in theme module. Most enums in theme module support Custom variant, e.g., theme::Radio::Custom(...). This variant takes radio::StyleSheet trait as its parameter. To use the variant, we need to implement the trait (such as RadioStyle struct in the following code). The associated type of the trait should be set to iced::Theme. The methods in the trait return radio::Appearance. We can use theme::Radio::Default to obtain the default value of radio::Appearance, e.g., style.active(&theme::Radio::Default, is_selected). After that, we can modify the default radio::Appearance based on our needs.

```
use iced::{
    theme,
    widget::{column, radio},
    Color, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    Choose(String),
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }
}
```

```

fn update(&mut self, _message: Self::Message) {}

fn view(&self) -> iced::Element<Self::Message> {
    column![
        radio("Choice A", "A", Some("A"), |s| MyAppMessage::Choose(
            s.to_string()
        )),
        .style(theme::Radio::Custom(Box::new(RadioStyle))),
        radio("Choice B", "B", Some("A"), |s| MyAppMessage::Choose(
            s.to_string()
        )),
        .style(theme::Radio::Custom(Box::new(RadioStyle))),
        radio("Choice C", "C", Some("A"), |s| MyAppMessage::Choose(
            s.to_string()
        )),
    ],
    .into()
}

struct RadioStyle;

impl radio::StyleSheet for RadioStyle {
    type Style = iced::Theme;

    fn active(&self, style: &Self::Style, is_selected: bool) -> radio::Appearance {
        radio::Appearance {
            text_color: Some(if is_selected {
                Color::from_rgb(0., 0., 1.)
            } else {
                Color::from_rgb(0.5, 0.5, 0.5)
            }),
            ..style.active(&theme::Radio::Default, is_selected)
        }
    }

    fn hovered(&self, style: &Self::Style, is_selected: bool) -> radio::Appearance {
        style.hovered(&theme::Radio::Default, is_selected)
    }
}

```

:arrow_right: Next: More Than One Page

:blue_book: Back: Table of contents

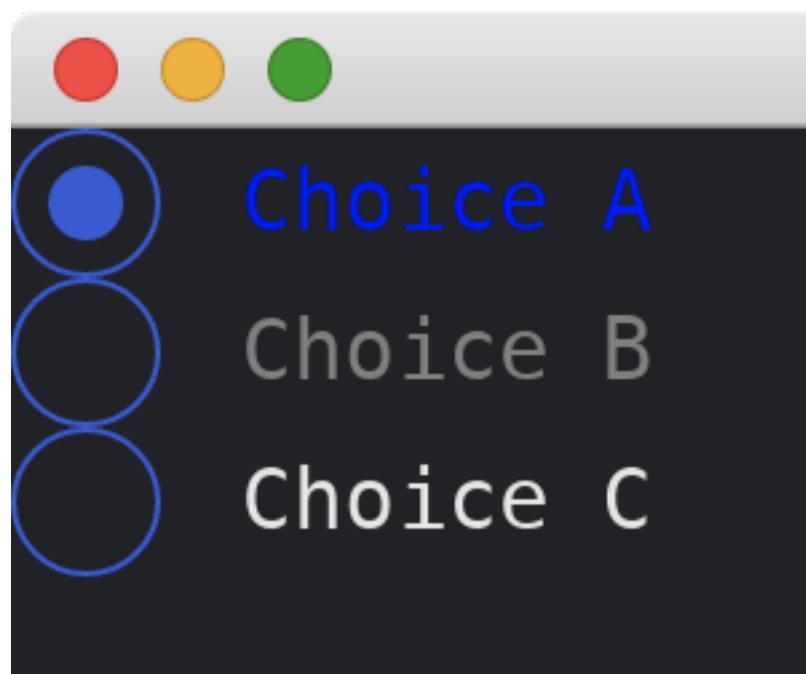


Figure 16: Custom styles

Drawing Shapes

Canvas is a widget that helps us drawing free-style shapes. To use the widget, we need to enable the canvas feature.

```
[dependencies]
iced = { version = "0.12.1", features = ["canvas"] }
```

We use `Canvas::new` to obtain the canvas widget. This function accepts a `Program` trait. We can create a struct (say, `MyProgram`) to implement this trait.

```
impl<Message> Program<Message> for MyProgram {
    type State = ();

    fn draw(
        &self,
        _state: &Self::State,
        renderer: &Renderer,
        _theme: &Theme,
        bounds: Rectangle,
        _cursor: mouse::Cursor,
    ) -> Vec<Geometry> {
        // ...
    }
}
```

There is a generic data type `Message` when we implement the `Program` trait. This helps us adapting to our `Message` in Sandbox.

The associated type `State` is not used in our example, so we set it to `()`.

The key of `Program` is the `draw` method. In the method, we define what shapes we are going to draw. We use `Frame` as a `pen` to draw shapes. For example, we use the `fill_rectangle` method of `Frame` to draw a filled rectangle. Or we can stroke and fill any `Path`. Finally, we use the `into_geometry` method of `Frame` to return the `Geometry` as required by the `draw` method.

```
use iced::*;

mouse,
widget::{
    canvas::{Frame, Geometry, Path, Program, Stroke},
    column, Canvas,
```

```

    },
    Alignment, Color, Length, Point, Rectangle, Renderer, Sandbox, Settings, Theme
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            "A Canvas",
            Canvas::new(MyProgram)
                .width(Length::Fill)
                .height(Length::Fill)
        ]
        .align_items(Alignment::Center)
        .into()
    }
}

struct MyProgram;

impl<Message> Program<Message> for MyProgram {
    type State = ();

    fn draw(
        &self,
        _state: &Self::State,
        renderer: &Renderer,
        _theme: &Theme,
        bounds: Rectangle,
        _cursor: mouse::Cursor,
    ) -> Vec<Geometry> {

```

```

let mut frame = Frame::new(renderer, bounds.size());

frame.fill_rectangle(Point::ORIGIN, bounds.size(), Color::from_rgb(0.0, 0.0, 0.0));

frame.fill(
    &Path::circle(frame.center(), frame.width().min(frame.height()) / 4.0),
    Color::from_rgb(0.6, 0.8, 1.0),
);

frame.stroke(
    &Path::line(
        frame.center() + Vector::new(-250.0, 100.0),
        frame.center() + Vector::new(250.0, -100.0),
    ),
    Stroke {
        style: Color::WHITE.into(),
        width: 50.0,
        ..Default::default()
    },
);

vec![frame.into_geometry()]
}
}

:arrow_right: Next: Drawing With Caches
:blue_book: Back: Table of contents

```

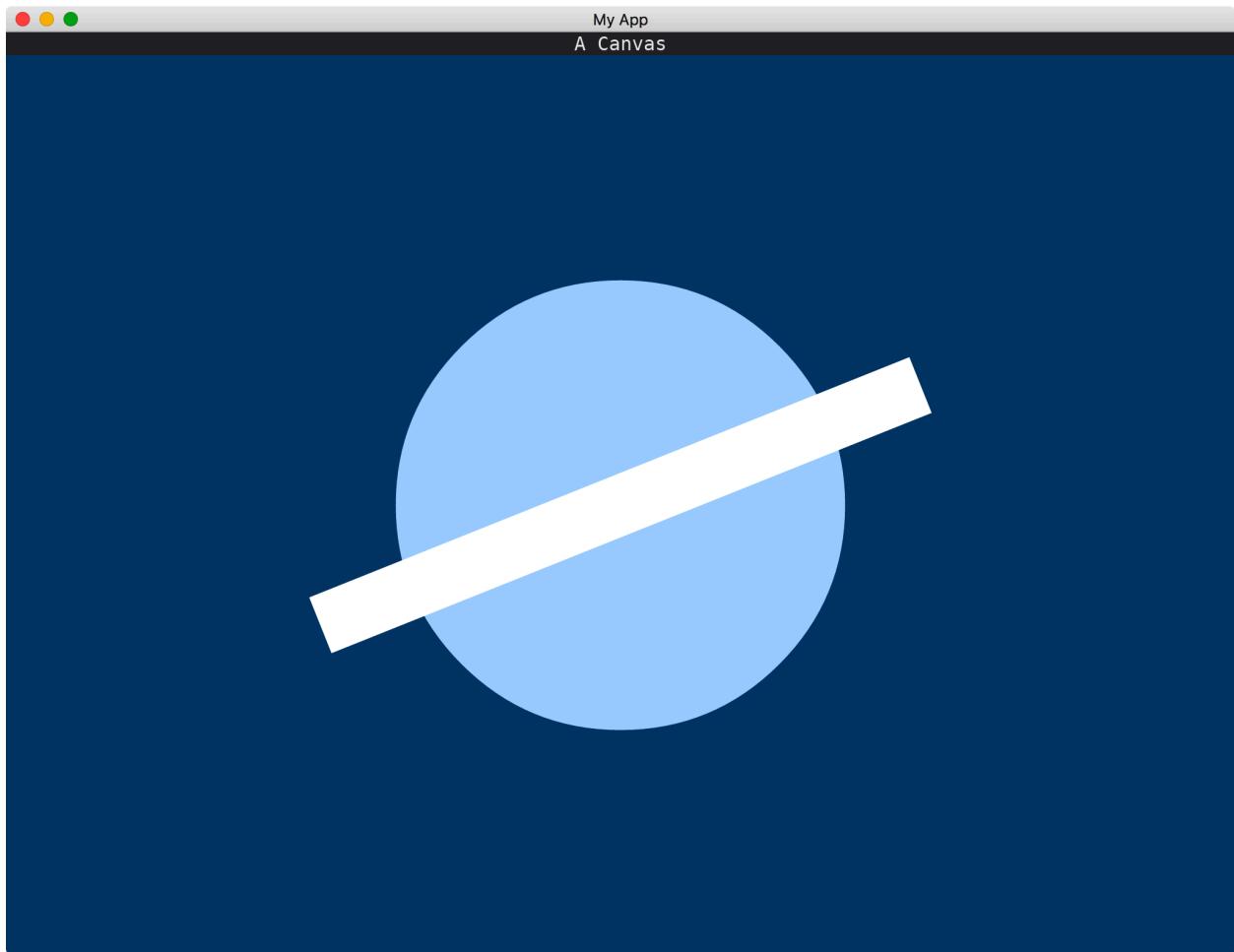


Figure 17: Drawing Shapes

Drawing Widgets

In addition to the build-in widgets, we can also design our own custom widgets. To do so, we need to enable the advanced feature. The dependencies of the Cargo.toml file should look like this:

```
[dependencies]
iced = { version = "0.12.1", features = ["advanced"] }
```

Then, we need a struct that implement Widget trait.

```
struct MyWidget;

impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidget
where
    Renderer: iced::advanced::Renderer,
{
    fn size(&self) -> Size<Length> {
        // ...
    }

    fn layout(
        &self,
        _tree: &mut Tree,
        _renderer: &Renderer,
        _limits: &layout::Limits,
    ) -> layout::Node {
        // ...
    }

    fn draw(
        &self,
        _state: &Tree,
        _renderer: &mut Renderer,
        _theme: &Theme,
        _style: &renderer::Style,
        _layout: Layout<'_>,
        _cursor: mouse::Cursor,
        _viewport: &Rectangle,
    ) {

```

```

        // ...
    }
}

```

We define the size of MyWidget by the methods: size and layout. Currently, we set the width and height to Length::Shrink, to tell the layout system that we use the least space for this widget.

```

fn size(&self) -> Size<Length> {
    Size {
        width: Length::Shrink,
        height: Length::Shrink,
    }
}

```

Then, we tell the layout system the precise size we are going to use for the widget. In this example, our widget is of size (100, 100).

```

fn layout(&self, _renderer: &Renderer, _limits: &layout::Limits) -> layout::Node {
    layout::Node::new([100, 100].into())
}

```

Usually, the layout method would consider the Limits parameter, which is the constraints from the layout system. But now, we ignore it for simplicity.

Next, we draw our widget in the draw method. We use the given Renderer to do so. One may refer to the given Theme and Style for the colors of the widget.

```

fn draw(
    &self,
    _state: &Tree,
    renderer: &mut Renderer,
    _theme: &Theme,
    _style: &renderer::Style,
    layout: Layout<'_>,
    _cursor: mouse::Cursor,
    _viewport: &Rectangle,
) {
    renderer.fill_quad(
        Quad {
            bounds: layout.bounds(),
            border: Border {
                color: Color::from_rgb(0.6, 0.8, 1.0),
                width: 1.0,
                radius: 10.0.into(),
            },
            shadow: Shadow::default(),
        },
        Color::from_rgb(0.0, 0.2, 0.4),
    );
}

```

The given Layout parameter would be calculated automatically by the layout system according to the size and layout methods we defined before.

For convenience, we can implement From<MyWidget> for Element.

```
impl<'a, Message, Renderer> From<MyWidget> for Element<'a, Message, Theme, Renderer>
where
    Renderer: iced::advanced::Renderer,
{
    fn from(widget: MyWidget) -> Self {
        Self::new(widget)
    }
}
```

Finally, the widget can be added to our app by the following code.

```
fn view(&self) -> iced::Element<Self::Message> {
    container(MyWidget)
        .width(Length::Fill)
        .height(Length::Fill)
        .center_x()
        .center_y()
        .into()
}
```

Note that it is not necessary to put MyWidget in a Container. We can add the widget directly into our app.

```
fn view(&self) -> iced::Element<Self::Message> {
    MyWidget.into()
}
```

The full code is as follows:

```
use iced::{
    advanced::{
        layout, mouse,
        renderer::{self, Quad},
        widget::Tree,
        Layout, Widget,
    },
    widget::container,
    Border, Color, Element, Length, Rectangle, Sandbox, Settings, Shadow, Size, Theme
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
```

```

type Message = ();

fn new() -> Self {
    Self
}

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, _message: Self::Message) {}

fn view(&self) -> iced::Element<Self::Message> {
    container(MyWidget)
        .width(Length::Fill)
        .height(Length::Fill)
        .center_x()
        .center_y()
        .into()
}
}

struct MyWidget;

impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidget
where
    Renderer: iced::advanced::Renderer,
{
    fn size(&self) -> Size<Length> {
        Size {
            width: Length::Shrink,
            height: Length::Shrink,
        }
    }

    fn layout(
        &self,
        _tree: &mut Tree,
        _renderer: &Renderer,
        _limits: &layout::Limits,
    ) -> layout::Node {
        layout::Node::new([100, 100].into())
    }

    fn draw(
        &self,
        _state: &Tree,
        renderer: &mut Renderer,
    )
}

```

```

        _theme: &Theme,
        _style: &renderer::Style,
        layout: Layout<'_>,
        _cursor: mouse::Cursor,
        _viewport: &Rectangle,
    ) {
        renderer.fill_quad(
            Quad {
                bounds: layout.bounds(),
                border: Border {
                    color: Color::from_rgb(0.6, 0.8, 1.0),
                    width: 1.0,
                    radius: 10.0.into(),
                },
                shadow: Shadow::default(),
            },
            Color::from_rgb(0.0, 0.2, 0.4),
        );
    }
}

impl<'a, Message, Renderer> From<MyWidget> for Element<'a, Message, Theme, Renderer>
where
    Renderer: iced::advanced::Renderer,
{
    fn from(widget: MyWidget) -> Self {
        Self::new(widget)
    }
}

```

:arrow_right: Next: Updating Widgets From Outside
:blue_book: Back: Table of contents

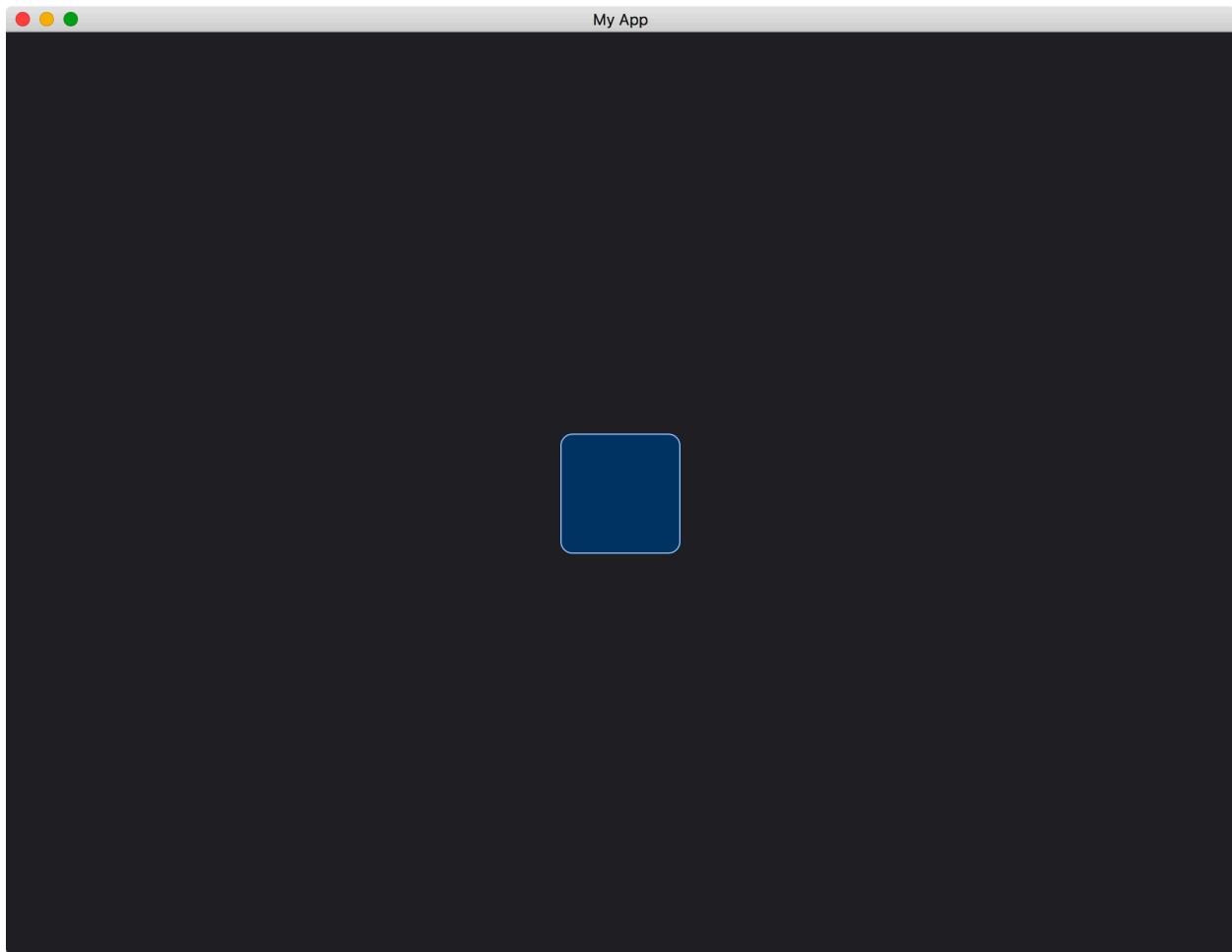


Figure 18: Drawing Widgets

Drawing With Caches

Previously, we mentioned that Program has the method draw that every time the corresponding Canvas needed to be re-drawn, the method is called. However, if the shapes of the Canvas remain unchanged, it is not performant to re-draw all these shapes. Instead, we store an image of these shapes in a Cache, and we draw this cache when the draw method of Program is called.

To do so, we declare a field of type Cache in our app

```
struct MyApp {  
    cache: Cache,  
}
```

and initialize it.

```
fn new() -> Self {  
    Self {  
        cache: Cache::new(),  
    }  
}
```

In the draw method, instead of creating a new Frame

```
let mut frame = Frame::new(renderer, bounds.size());  
// ...  
vec![frame.into_geometry()]
```

we use cache.draw to construct the Geometry.

```
let geometry = self.cache.draw(renderer, bounds.size(), |frame| {  
    // ...  
});  
  
vec![geometry]
```

The closure `|frame| { /* ... */ }` is only called when the dimensions of the frame change or the cache is explicitly cleared.

In addition, previously, we implement Program for the struct MyProgram. But because we need to access the cache field of MyApp, we have to implement Program for MyApp.

The full code is as follows:

```

use iced::*;

mouse,
widget::{
    canvas:{Cache, Geometry, Path, Program, Stroke},
    column, Canvas,
},
Alignment, Color, Length, Point, Rectangle, Renderer, Sandbox, Settings, Theme
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp {
    cache: Cache,
}

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self {
            cache: Cache::new(),
        }
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            "A Canvas",
            Canvas::new(self).width(Length::Fill).height(Length::Fill)
        ]
        .align_items(Alignment::Center)
        .into()
    }
}

impl<Message> Program<Message> for MyApp {
    type State = ();

    fn draw(
        &self,
        _state: &Self::State,

```

```

    renderer: &Renderer,
    _theme: &Theme,
    bounds: Rectangle,
    _cursor: mouse::Cursor,
) -> Vec<Geometry> {
    let geometry = self.cache.draw(renderer, bounds.size(), |frame| {
        frame.fill_rectangle(Point::ORIGIN, bounds.size(), Color::from_rgb(0.0
            frame.fill(
                &Path::circle(frame.center(), frame.width().min(frame.height()) / 2.0,
                Color::from_rgb(0.6, 0.8, 1.0),
            );
            frame.stroke(
                &Path::line(
                    frame.center() + Vector::new(-250.0, 100.0),
                    frame.center() + Vector::new(250.0, -100.0),
                ),
                Stroke {
                    style: Color::WHITE.into(),
                    width: 50.0,
                    ..Default::default()
                },
            );
        );
    });
    vec![geometry]
}
}

```

:arrow_right: Next: Drawing Widgets

:blue_book: Back: Table of contents

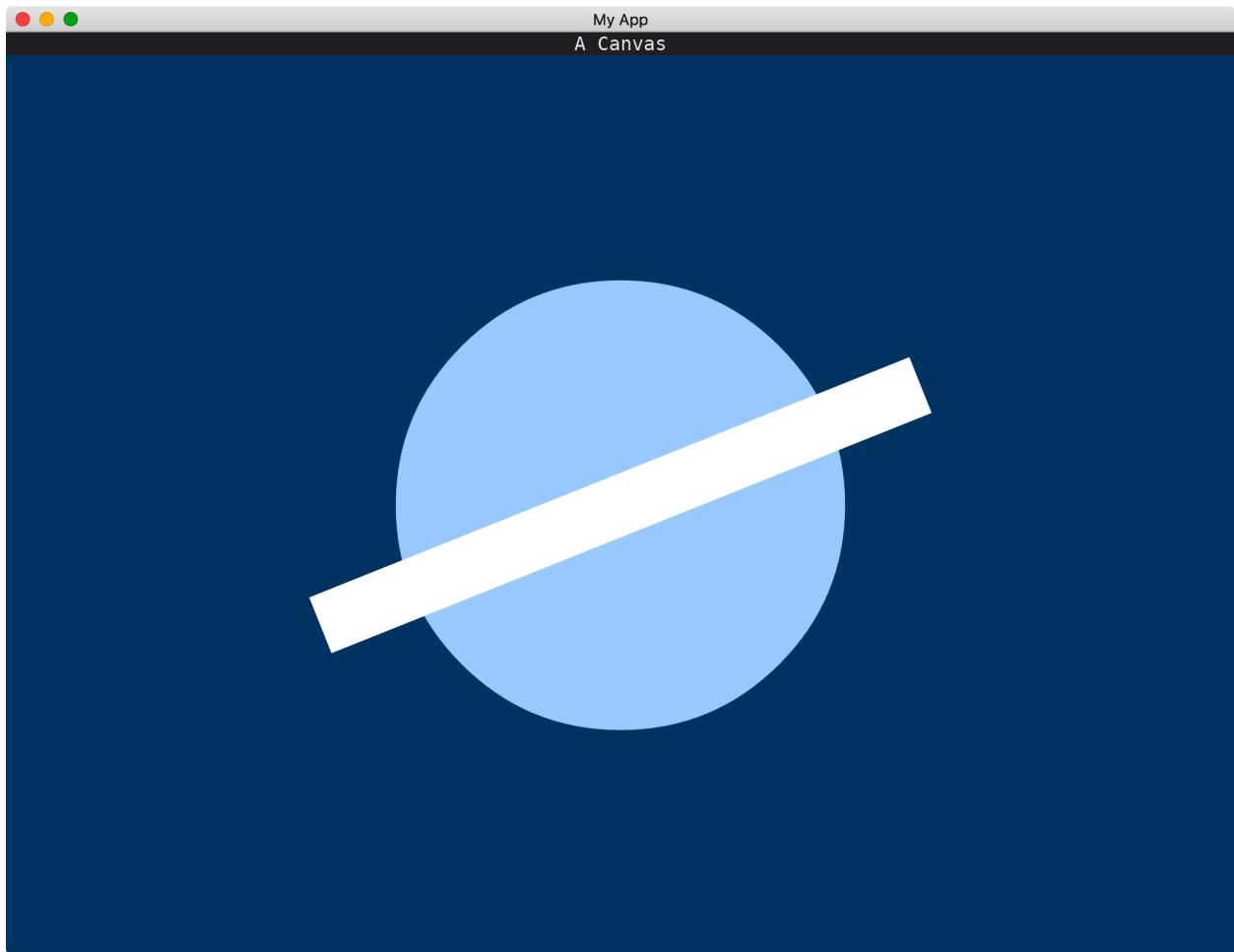


Figure 19: Drawing With Caches

Executing Custom Commands

Commands can help us executing asynchronous functions. To do this, we need to enable one of the following features: tokio, async-std, or smol. The corresponding asynchronous runtime (tokio, async-std, or smol) must also be added.

Here, we use tokio as an example. We enable tokio feature and add tokio crate. The dependencies of Cargo.toml should look like this:

```
[dependencies]
iced = { version = "0.12.1", features = ["tokio"] }
tokio = { version = "1.37.0", features = ["time"] }
```

We use Command::perform to execute an asynchronous function. The first parameter of Command::perform is an asynchronous function, and the second parameter is a function that returns MyAppMessage. The MyAppMessage will be produced once the asynchronous function is done.

In the following code, we use a simple asynchronous function tokio::time::sleep. When the asynchronous function finished, we will receive MyAppMessage::Done.

```
use std::time::Duration;

use iced::{
    executor,
    widget::{button, column, text},
    Application, Command, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    Execute,
    Done,
}

struct MyApp {
    state: String,
```

```

}

impl Application for MyApp {
    type Executor = executor::Default;
    type Message = MyAppMessage;
    type Theme = iced::Theme;
    type Flags = ();

    fn new(_flags: Self::Flags) -> (Self, iced::Command<Self::Message>) {
        (
            Self {
                state: "Ready".into(),
            },
            Command::none(),
        )
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, message: Self::Message) -> iced::Command<Self::Message> {
        match message {
            MyAppMessage::Execute => {
                self.state = "Executing".into();
                return Command::perform(tokio::time::sleep(Duration::from_secs(1))
                    MyAppMessage::Done
                );
            }
            MyAppMessage::Done => self.state = "Done".into(),
        }
        Command::none()
    }

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            button("Execute").on_press(MyAppMessage::Execute),
            text(self.state.as_str()),
        ]
        .into()
    }
}

```

:arrow_right: Next: Initializing A Different Window

:blue_book: Back: Table of contents

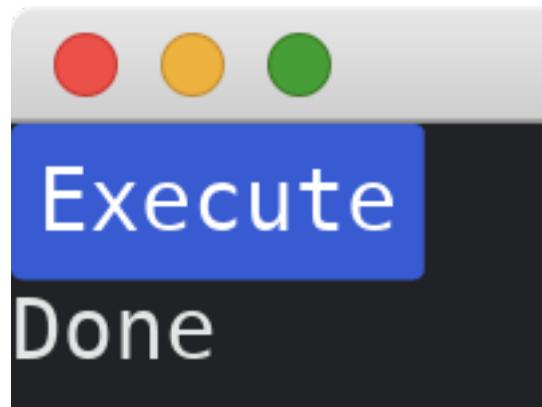


Figure 20: Executing custom commands

Explanation of Sandbox Trait

Sandbox works as follows.

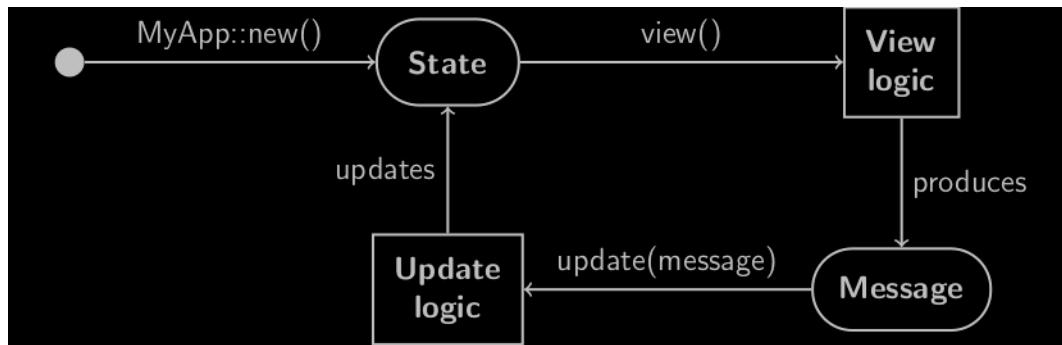


Figure 21: Sandbox trait

This can also be seen in the following code.

```
#[derive(Debug, Clone)]
enum Message {
    // ...
}

struct MyApp {
    // ... (Some fields for the app state)
}

impl Sandbox for MyApp {
    type Message = Message;

    fn new() -> Self {
        Self {
            // ...
        }
    }

    fn title(&self) -> String {
        // Title of the window
    }
}
```

```
fn update(&mut self, message: Self::Message) {
    match message {
        // Update logic
    }
}

fn view(&self) -> iced::Element<Self::Message> {
    // View logic
}
}

:arrow_right: Next: Adding Widgets
:blue_book: Back: Table of contents
```

First App - Hello World!

We need a struct to implement Sandbox, and call its run method from main. All widgets should be placed inside the view method.

```
use iced::{Sandbox, Settings};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        "Hello World!".into()
    }
}
```

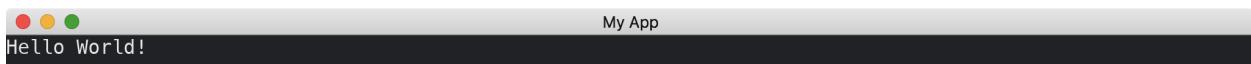


Figure 22: First app

:arrow_right: Next: Explanation of Sandbox Trait

:blue_book: Back: Table of contents

From Sandbox To Application

To have more control over our app, we can use Application trait, which is a generalization of Sandbox trait. There are two main differences between Application and Sandbox. One thing is the associated types. We have to specify Executor, Theme and Flags in addition to Message in Sandbox. Basically, we use the suggested defaults for these associated types. The other is that we have to return Command in new method and update method. We just return Command::none() for both methods.

```
use iced::{executor, Application, Command, Settings};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Application for MyApp {
    type Executor = executor::Default;
    type Message = ();
    type Theme = iced::Theme;
    type Flags = ();

    fn new(_flags: Self::Flags) -> (Self, iced::Command<Self::Message>) {
        (Self, Command::none())
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) -> iced::Command<Self::Message> {
        Command::none()
    }

    fn view(&self) -> iced::Element<Self::Message> {
        "Hello World!".into()
    }
}
```



Figure 23: From sandbox to application

:arrow_right: Next: Controlling Widgets By Commands

:blue_book: Back: Table of contents

Image

The Image widget is able to display an image. It has two methods of constructions. We can set how to fit the image content into the widget bounds.

To use the widget, we have to enable the image feature. The `Cargo.toml` dependencies should look like this:

```
[dependencies]
iced = { version = "0.12.1", features = ["image"] }

Assume we have an image named ferris.png in the project root directory, i.e., the
image has the path my_project/ferris.png where my_project is the name of our
project.

use iced::*;
    widget::column, image, text, Image},
    ContentFit, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
```

```
    text("Construct from struct"),
    Image::new("ferris.png"),
    text("Construct from function"),
    image("ferris.png"),
    text("Different content fit"),
    image("ferris.png").content_fit(ContentFit::Cover),
]
.into()
}
}

:arrow_right: Next: Svg
:blue_book: Back: Table of contents
```

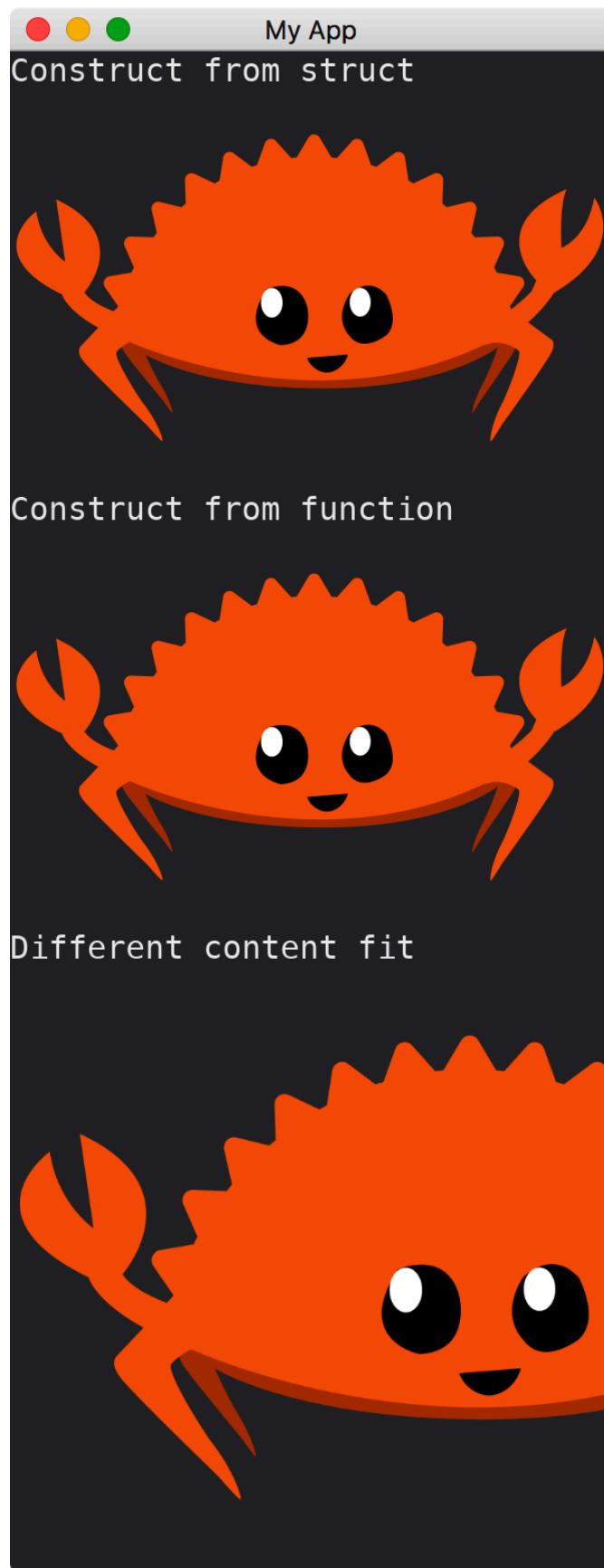


Figure 24: Image
76

Initializing A Different Window

We can use `window::Settings` to change the properties of the window (such as position and size) when we call `run` of a `Sandbox` or `Application`. Developers might be interested in reading the document of `window::Settings` for other properties.

```
use iced::{window, Point, Sandbox, Settings, Size};

fn main() -> iced::Result {
    MyApp::run(Settings {
        window: window::Settings {
            size: Size {
                width: 70.,
                height: 30.,
            },
            position: window::Position::Specific(Point { x: 50., y: 60. }),
            ..window::Settings::default()
        },
        ..Settings::default()
    })
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        "Hello".into()
    }
}
```

```
}
```

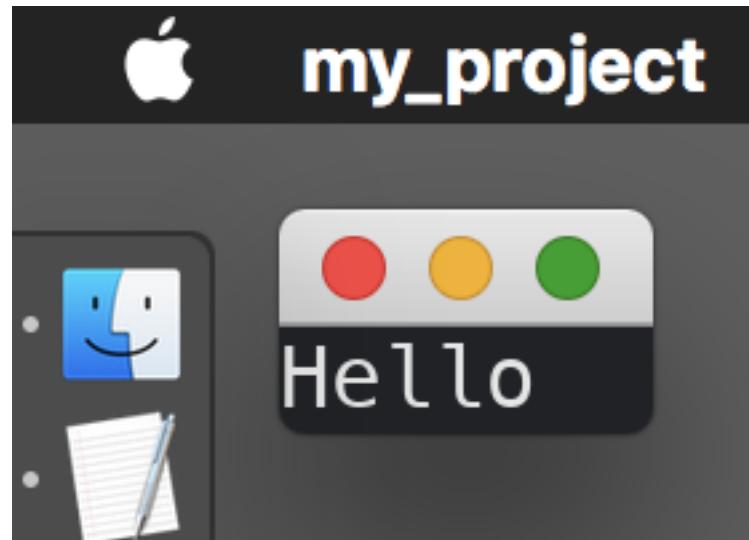


Figure 25: Initializing a different window

:arrow_right: Next: Changing The Window Dynamically

:blue_book: Back: Table of contents

Loading Images Asynchronously

In the previous tutorials, we introduced how to use Application to execute an asynchronous operation and how to display an image. This tutorial combines both and demonstrates how to load an image asynchronously.

It is assumed that there is an image named ferris.png in the Cargo project root directory.

To use asynchronous and image functions, we have to include the corresponding asynchronous library and enable the corresponding features. The dependencies of the Cargo.toml file should look like this:

```
[dependencies]
iced = { version = "0.12.1", features = ["image", "tokio"] }
tokio = { version = "1.36.0", features = ["full"] }
```

Our app will have three states: *start*, *loading* and *loaded*. We use two fields to encode the three states.

```
struct MyApp {
    image_handle: Option<Handle>,
    show_container: bool,
}
```

When the state is

- **start**: image_handle is None and show_container is false;
- **loading**: image_handle is None and show_container is true;
- **loaded**: image_handle is Some(...) and show_container is true.

The app begins in the *start* state.

The app always shows a button that is for loading the ferris.png image. In the *start* state, the app shows no additional widget. In the *loading* state, the app shows the text Loading.... And in the *loaded* state, the app shows the image.

```
fn view(&self) -> iced::Element<Self::Message> {
    column![
        button("Load").on_press(MyMessage::Load),
        if self.show_container {
            match &self.image_handle {
                Some(h) => container(image(h.clone())),
                None => container("Loading..."),
            }
        }
    ]
}
```

```

        }
    } else {
        container(""),
    },
]
.padding(20)
.into()
}

```

We have two messages for the app:

```

#[derive(Debug, Clone)]
enum MyMessage {
    Load,
    Loaded(Vec<u8>),
}

```

When the button is pressed, the app triggers a Load message to load the image. And when the image is loaded, the app triggers a Loaded(...) message.

The image will be loaded asynchronously.

```

fn update(&mut self, message: Self::Message) -> iced::Command<Self::Message> {
    match message {
        MyMessage::Load => {
            self.show_container = true;
            return Command::perform(
                async {
                    let mut file = File::open("ferris.png").await.unwrap();
                    let mut buffer = Vec::new();
                    file.read_to_end(&mut buffer).await.unwrap();
                    buffer
                },
                MyMessage::Loaded,
            );
        }
        MyMessage::Loaded(data) => self.image_handle = Some(Handle::from_memory(data));
        Command::none()
    }
}

```

The full code is as follows:

```

use iced::{
    executor,
    widget::{button, column, container, image, image::Handle},
    Application, Command,
};
use tokio::fs::File;
use tokio::io::AsyncReadExt;

fn main() -> iced::Result {
    Application::run()
}

```

```

        MyApp::run(iced::Settings::default())
    }

#[derive(Debug, Clone)]
enum MyMessage {
    Load,
    Loaded(Vec<u8>),
}

struct MyApp {
    image_handle: Option<Handle>,
    show_container: bool,
}

impl Application for MyApp {
    type Executor = executor::Default;
    type Message = MyMessage;
    type Theme = iced::Theme;
    type Flags = ();

    fn new(_flags: Self::Flags) -> (Self, iced::Command<Self::Message>) {
        (
            Self {
                image_handle: None,
                show_container: false,
            },
            Command::none(),
        )
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, message: Self::Message) -> iced::Command<Self::Message> {
        match message {
            MyMessage::Load => {
                self.show_container = true;
                return Command::perform(
                    async {
                        let mut file = File::open("ferris.png").await.unwrap();
                        let mut buffer = Vec::new();
                        file.read_to_end(&mut buffer).await.unwrap();
                        buffer
                    },
                    MyMessage::Loaded,
                );
            }
        }
    }
}

```

```

        MyMessage::Loaded(data) => self.image_handle = Some(Handle::from_memory(data))
    }
    Command::none()
}

fn view(&self) -> iced::Element<Self::Message> {
    column![
        button("Load").on_press(MyMessage::Load),
        if self.show_container {
            match &self.image_handle {
                Some(h) => container(image(h.clone())),
                None => container("Loading..."),
            }
        } else {
            container("")
        },
    ]
    .padding(20)
    .into()
}
}

```

State of *start*:

State of *loading*:

State of *loaded*:

:blue_book: Back: Table of contents

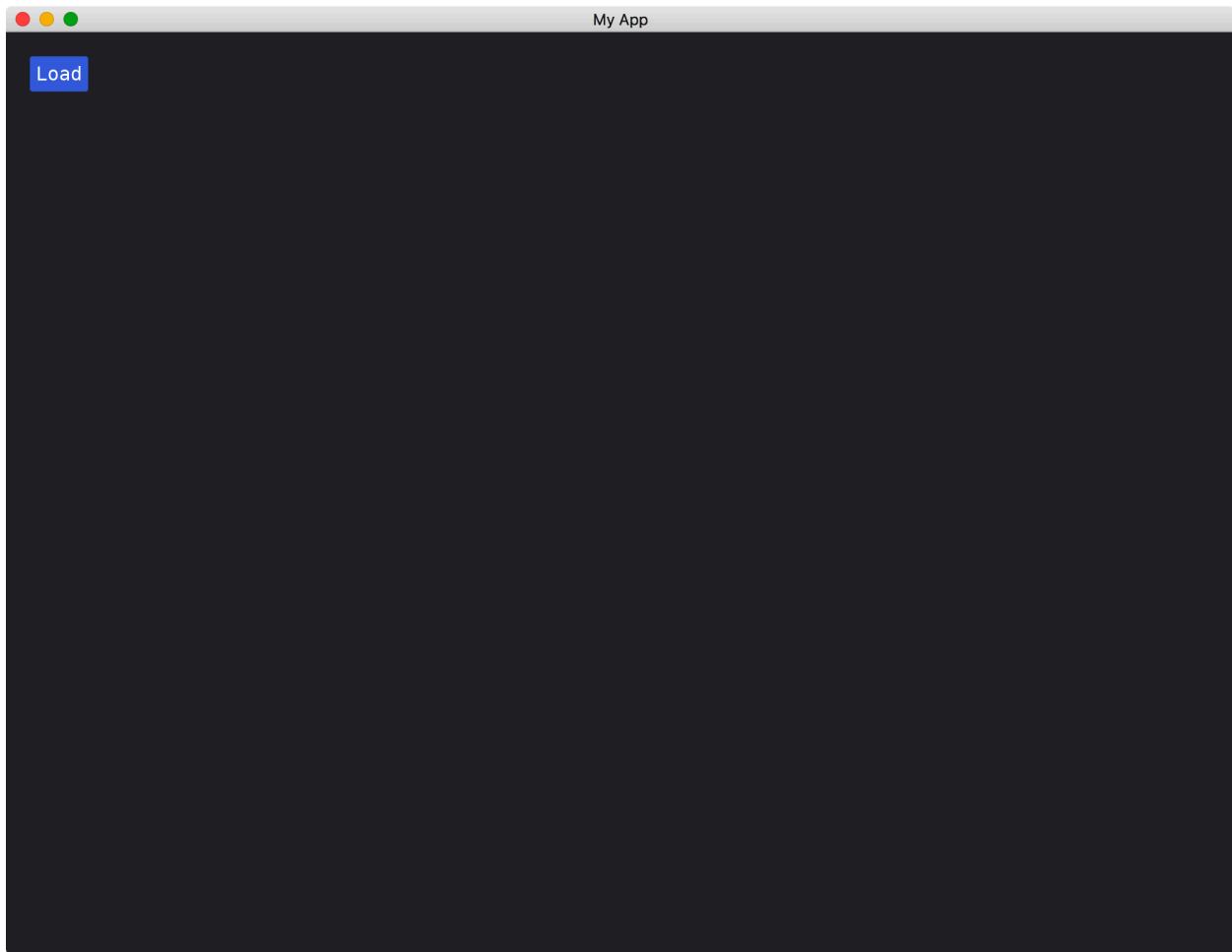


Figure 26: Loading Images Asynchronously 1

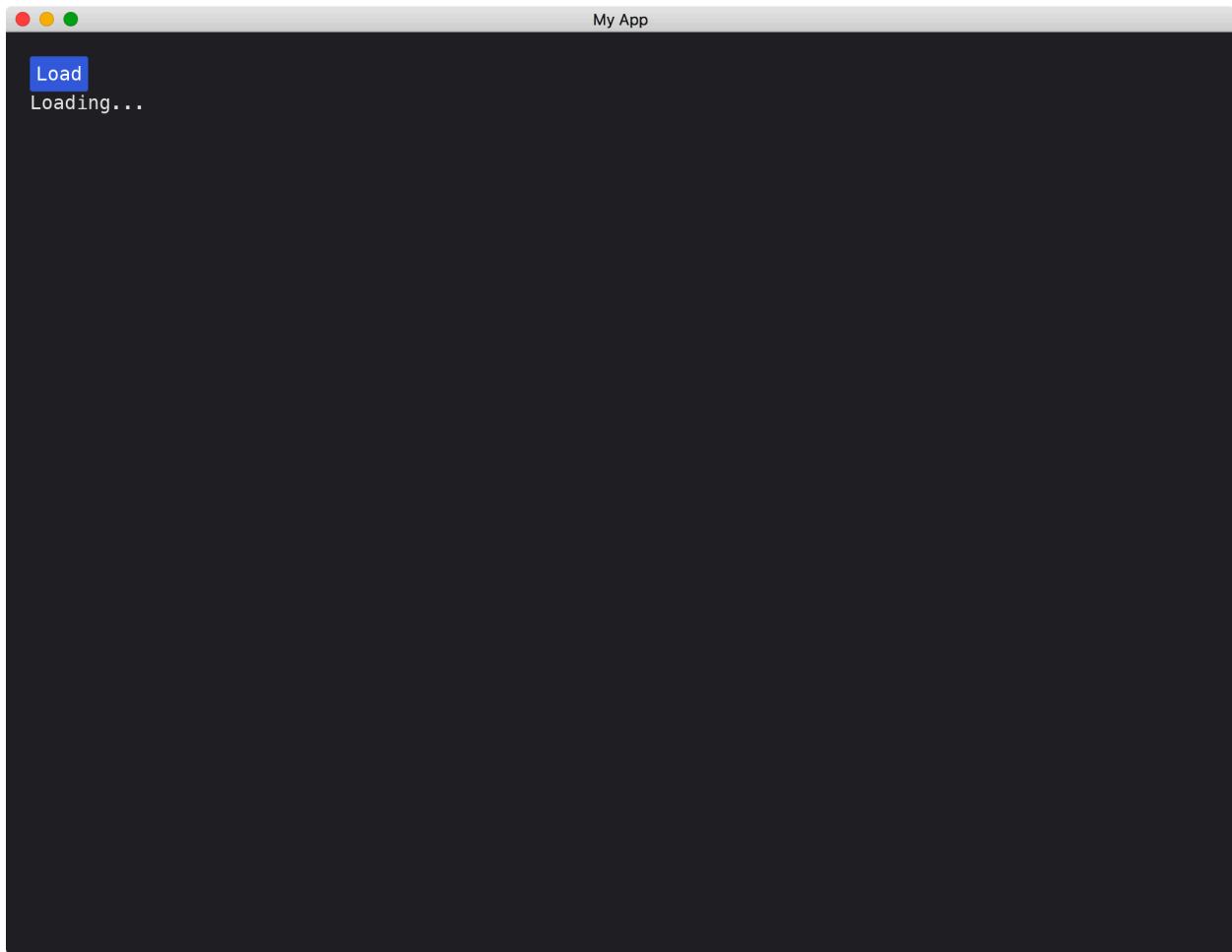


Figure 27: Loading Images Asynchronously 2

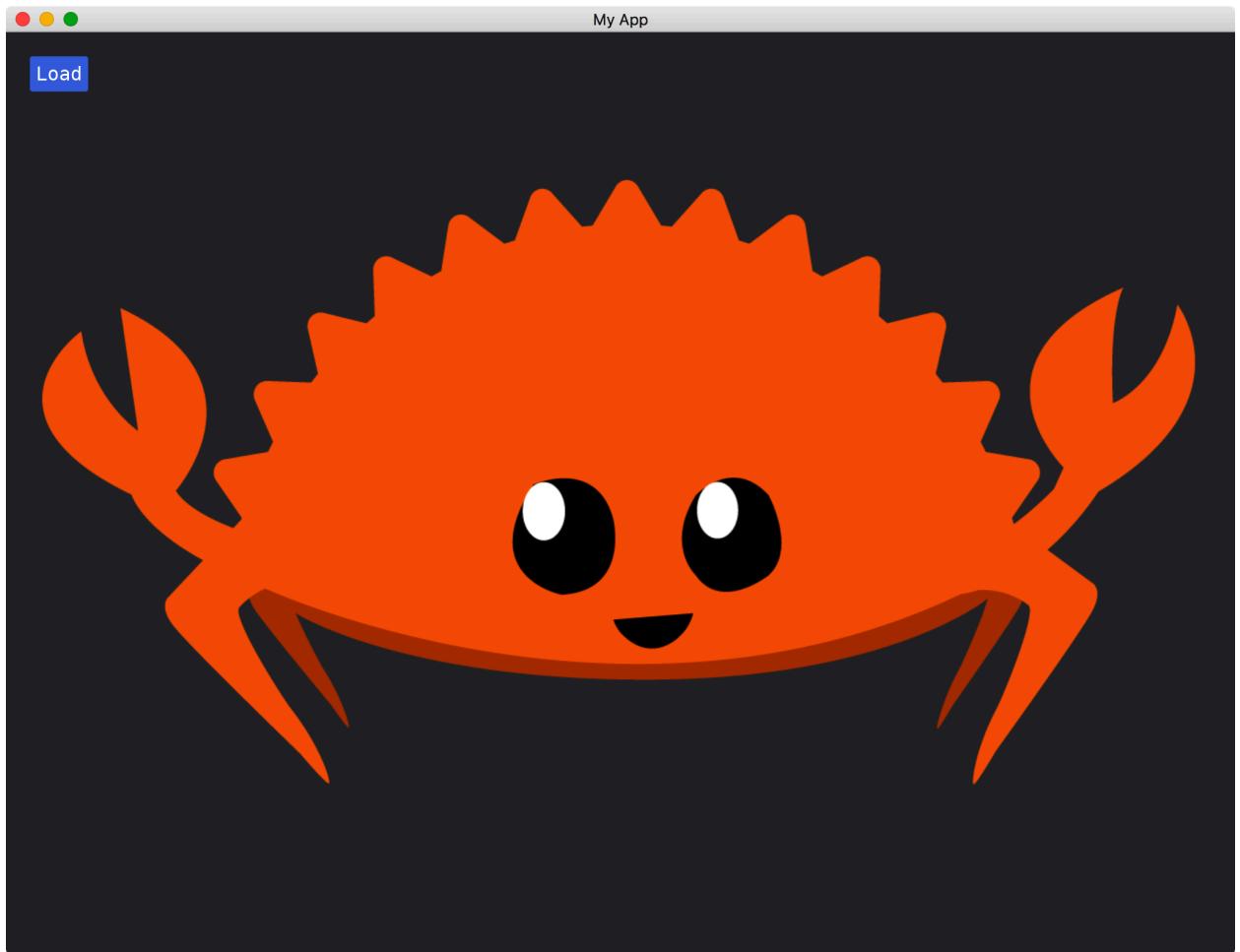


Figure 28: Loading Images Asynchronously 3

Memoryless Pages

The previous tutorial has a problem that the fields can still be accessed when we navigate to other pages. This brings potential security problems, e.g., the input password could be accessed in other pages.

To fix this problem, we can use trait objects. The Page trait below is responsible for update and view for a single page. In the main struct MyApp, we dispatch update and view to the corresponding page that is indicated by page field in MyApp. The update method in MyApp is also responsible for switching pages.

In addition, we explicitly distinguish messages from different pages in MyAppMessage.

```
use iced::{
    widget::{button, column, text, text_input},
    Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    PageA(PageAMessage),
    PageB(PageBMessage),
}

trait Page {
    fn update(&mut self, message: MyAppMessage) -> Option<Box<dyn Page>>;
    fn view(&self) -> iced::Element<'_, MyAppMessage>;
}

struct MyApp {
    page: Box<dyn Page>,
}

impl Sandbox for MyApp {
    type Message = MyAppMessage;
```

```

fn new() -> Self {
    Self {
        page: Box::new(PageA::new()),
    }
}

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, message: Self::Message) {
    let page = self.page.update(message);
    if let Some(p) = page {
        self.page = p;
    }
}

fn view(&self) -> iced::Element<Self::Message> {
    self.page.view()
}
}

```

In this tutorial, we have two pages, PageA and PageB. PageA is a simple login form and PageB is a simple hello page. Let's start with PageB. In its update method, we only care about messages of PageBMessage.

```

#[derive(Debug, Clone)]
enum PageBMessage {
    ButtonPressed,
}

type Mb = PageBMessage;

struct PageB;

impl PageB {
    fn new() -> Self {
        Self
    }
}

impl Page for PageB {
    fn update(&mut self, message: MyAppMessage) -> Option<Box<dyn Page>> {
        if let MyAppMessage::PageB(msg) = message {
            match msg {
                PageBMessage::ButtonPressed => return Some(Box::new(PageA::new())))
            }
        }
        None
    }
}

```

```

fn view(&self) -> iced::Element<MyAppMessage> {
    column![
        text("Hello!"),
        button("Log out").on_press(MyAppMessage::PageB(Mb::ButtonPressed)),
    ]
    .into()
}
}

```

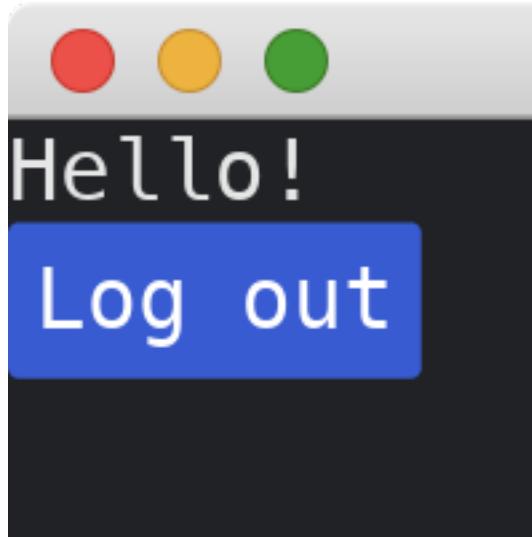


Figure 29: Page B

In PageA, we check the password when the login button is pressed. If it is a valid password, we switch to PageB. Note that PageA (and its password field) is dropped after we switch to PageB. This ensures the password is protected.

```

#[derive(Debug, Clone)]
enum PageAMessage {
    TextChanged(String),
    ButtonPressed,
}
type Ma = PageAMessage;

struct PageA {
    password: String,
}

impl PageA {
    fn new() -> Self {
        Self {
            password: String::new(),
        }
    }
}

```

```

}

impl Page for PageA {
    fn update(&mut self, message: MyAppMessage) -> Option<Box<dyn Page>> {
        if let MyAppMessage::PageA(msg) = message {
            match msg {
                PageAMessage::TextChanged(s) => self.password = s,
                PageAMessage::ButtonPressed => {
                    if self.password == "abc" {
                        return Some(Box::new(PageB::new()));
                    }
                }
            }
        }
        None
    }

    fn view(&self) -> iced::Element<MyAppMessage> {
        column![
            text_input("Password", &self.password)
                .secure(true)
                .on_input(|s| MyAppMessage::PageA(MyAppMessage::PageA(TextChanged(s)))),
            button("Log in").on_press(MyAppMessage::PageA(ButtonPressed)),
        ]
        .into()
    }
}

```

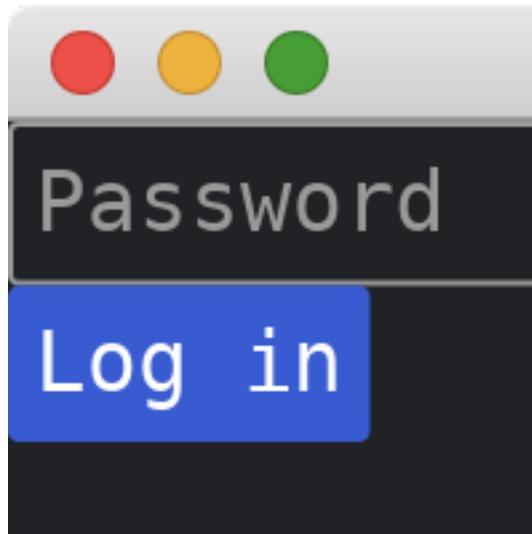


Figure 30: Page A

:arrow_right: Next: Passing Parameters Across Pages

:blue_book: Back: Table of contents

More Than One Page

To have multiple pages, we can add a field page to the main struct MyApp. The field page is an enum defined by us that decides what to display in view method of the Sandbox.

```
use iced::{
    widget::{button, column, text},
    Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

enum Page {
    A,
    B,
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    GoToBButtonPressed,
    GoToAButtonPressed,
}

struct MyApp {
    page: Page,
}

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self { page: Page::A }
    }

    fn title(&self) -> String {
        String::from("My App")
    }
}
```

```

    }

    fn update(&mut self, message: Self::Message) {
        self.page = match message {
            MyAppMessage::GoToBButtonPressed => Page::B,
            MyAppMessage::GoToAButtonPressed => Page::A,
        }
    }

    fn view(&self) -> iced::Element<Self::Message> {
        match self.page {
            Page::A => column![
                text("Page A"),
                button("Go to B").on_press(MyAppMessage::GoToBButtonPressed),
            ],
            Page::B => column![
                text("Page B"),
                button("Go to A").on_press(MyAppMessage::GoToAButtonPressed),
            ],
        }
        .into()
    }
}

```

Page A:

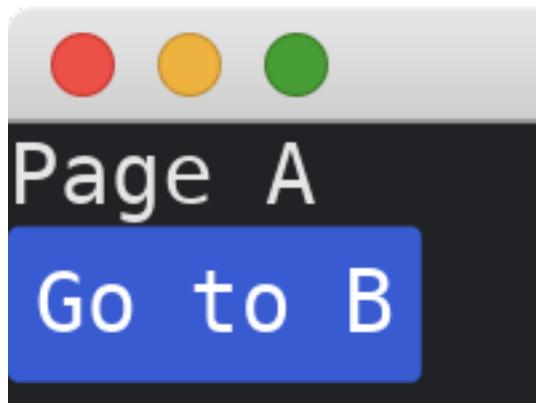


Figure 31: Page A

And page B:

:arrow_right: Next: Memoryless Pages

:blue_book: Back: Table of contents

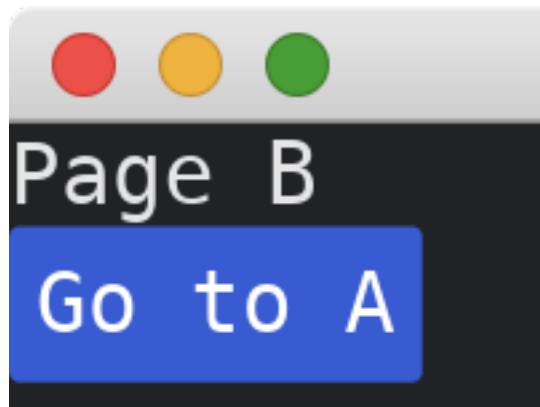


Figure 32: Page B

Mouse Pointer Over Widgets

To change the mouse pointer based on the requirement of our widgets, we can use the `mouse_interaction` method of `Widget`.

```
fn mouse_interaction(
    &self,
    _state: &Tree,
    layout: Layout<'_>,
    cursor: mouse::Cursor,
    _viewport: &Rectangle,
    _renderer: &Renderer,
) -> mouse::Interaction {
    if cursor.is_over(layout.bounds()) {
        mouse::Interaction::Pointer
    } else {
        mouse::Interaction::Idle
    }
}
```

The method returns `Interaction`, which specifies the type of the mouse pointer. In our example, we specify `Interaction::Pointer` when the mouse is over the widget.

The full code is as follows:

```
use iced::{
    advanced::{
        layout, mouse,
        renderer::{self, Quad},
        widget::Tree,
        Layout, Widget,
    },
    widget::container,
    Border, Color, Element, Length, Rectangle, Sandbox, Settings, Shadow, Size, Theme
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;
```

```

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        container(MyWidget)
            .width(Length::Fill)
            .height(Length::Fill)
            .center_x()
            .center_y()
            .into()
    }
}

struct MyWidget;

impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidget
where
    Renderer: iced::advanced::Renderer,
{
    fn size(&self) -> Size<Length> {
        Size {
            width: Length::Shrink,
            height: Length::Shrink,
        }
    }

    fn layout(
        &self,
        _tree: &mut Tree,
        _renderer: &Renderer,
        _limits: &layout::Limits,
    ) -> layout::Node {
        layout::Node::new([100, 100].into())
    }

    fn draw(
        &self,

```

```

        _state: &Tree,
        renderer: &mut Renderer,
        _theme: &Theme,
        _style: &renderer::Style,
        layout: Layout<'_>,
        _cursor: mouse::Cursor,
        _viewport: &Rectangle,
    ) {
        renderer.fill_quad(
            Quad {
                bounds: layout.bounds(),
                border: Border {
                    color: Color::from_rgb(0.6, 0.8, 1.0),
                    width: 1.0,
                    radius: 10.0.into(),
                },
                shadow: Shadow::default(),
            },
            Color::from_rgb(0.0, 0.2, 0.4),
        );
    }

fn mouse_interaction(
    &self,
    _state: &Tree,
    layout: Layout<'_>,
    cursor: mouse::Cursor,
    _viewport: &Rectangle,
    _renderer: &Renderer,
) -> mouse::Interaction {
    if cursor.is_over(layout.bounds()) {
        mouse::Interaction::Pointer
    } else {
        mouse::Interaction::Idle
    }
}

impl<'a, Message, Renderer> From<MyWidget> for Element<'a, Message, Theme, Renderer>
where
    Renderer: iced::advanced::Renderer,
{
    fn from(widget: MyWidget) -> Self {
        Self::new(widget)
    }
}

```

:arrow_right: Next: Texts In Widgets

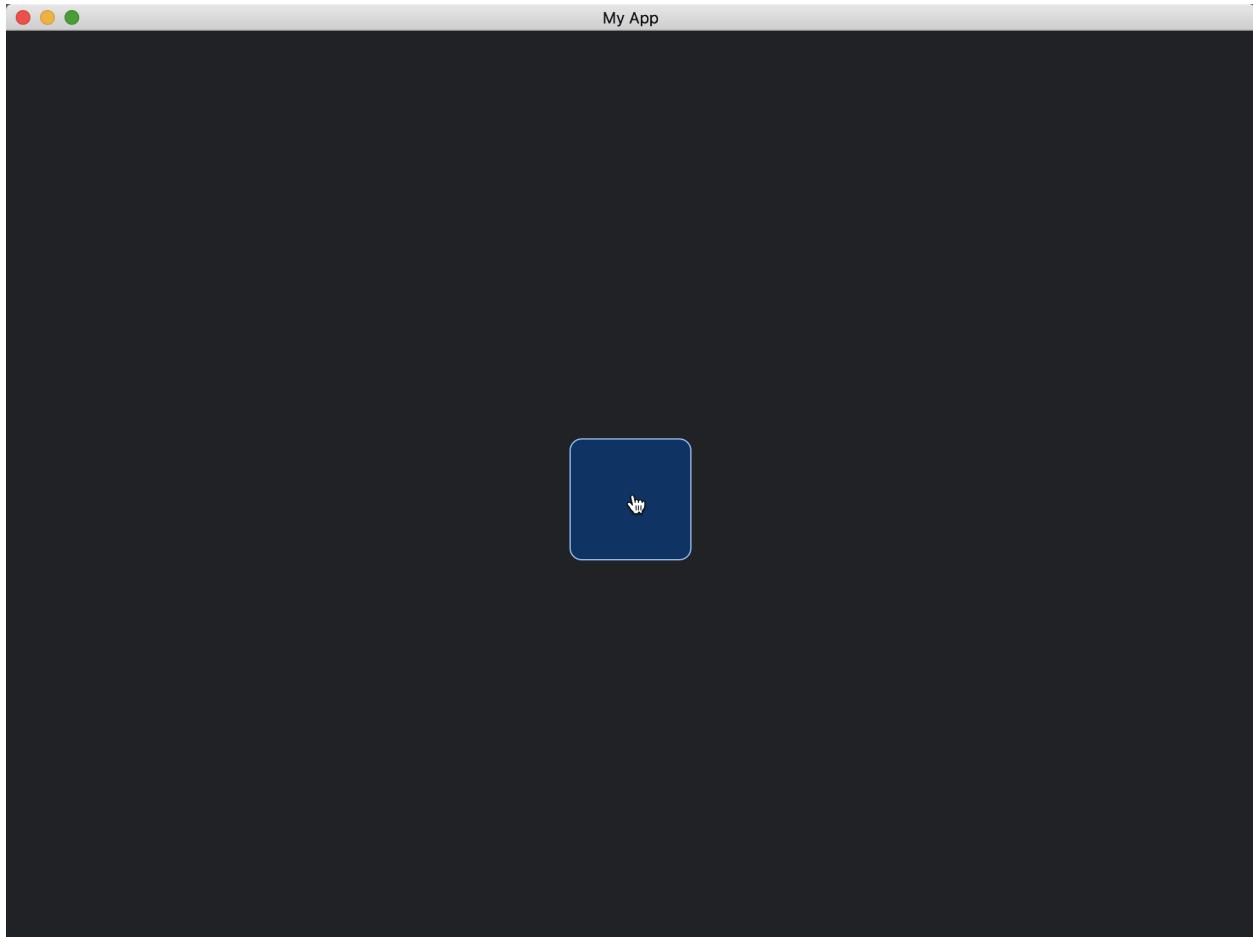


Figure 33: Mouse Pointer Over Widgets

:blue_book: Back: Table of contents

Navigation History

This tutorial follows the previous tutorial. The framework introduced in the previous tutorial can be extended to handle page navigation history, which is capable of restoring past pages.

Instead of keeping a single page in the main struct `MyApp`, we can keep a `Vec` of pages. We control how the `Vec` would change in `update` method of `SandBox`. The communication between `update` of `Sandbox` and `update` of `Page` trait is through a custom enum `Navigation`.

```
use iced::{
    widget::{button, column, row, text},
    Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    PageA(PageAMessage),
    PageB(PageBMessage),
}

enum Navigation {
    GoTo(Box<dyn Page>),
    Back,
    None,
}

trait Page {
    fn update(&mut self, message: MyAppMessage) -> Navigation;
    fn view(&self) -> iced::Element<MyAppMessage>;
}

struct MyApp {
    pages: Vec<Box<dyn Page>>,
}
```

```

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self {
            pages: vec![Box::new(PageA::new())],
        }
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, message: Self::Message) {
        let navigation = self.pages.last_mut().unwrap().update(message);
        match navigation {
            Navigation::GoTo(p) => self.pages.push(p),
            Navigation::Back => {
                if self.pages.len() > 1 {
                    self.pages.pop();
                }
            }
            Navigation::None => {}
        }
    }

    fn view(&self) -> iced::Element<Self::Message> {
        self.pages.last().unwrap().view()
    }
}

```

The following is the first type of pages:

```

#[derive(Debug, Clone)]
enum PageAMessage {
    ButtonPressed,
}

type Ma = PageAMessage;

struct PageA;

impl PageA {
    fn new() -> Self {
        Self
    }
}

impl Page for PageA {

```

```

fn update(&mut self, message: MyAppMessage) -> Navigation {
    if let MyAppMessage::PageA(msg) = message {
        match msg {
            PageAMessage::ButtonPressed => {
                return Navigation::GoTo(Box::new(PageB::new(1)));
            }
        }
    }
    Navigation::None
}

fn view(&self) -> iced::Element<MyAppMessage> {
    column![
        text("Start"),
        button("Next").on_press(MyAppMessage::PageA(ButtonPressed)),
    ]
    .into()
}
}

```

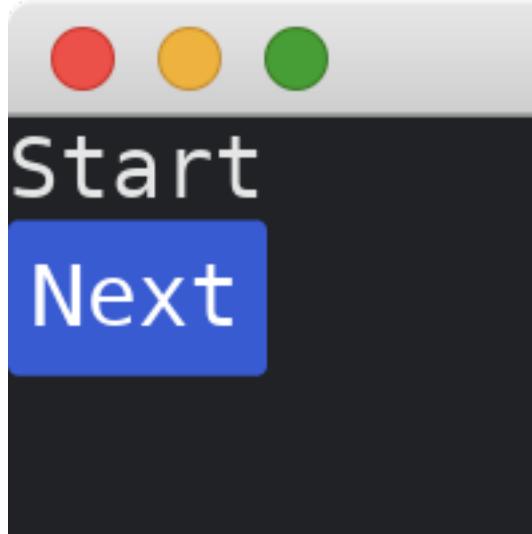


Figure 34: Page A

And the second type of pages:

```

#[derive(Debug, Clone)]
enum PageBMessage {
    BackButtonPressed,
    NextButtonPressed,
}
type Mb = PageBMessage;

struct PageB {
    id: u32,

```

```

}

impl PageB {
    fn new(id: u32) -> Self {
        Self { id }
    }
}

impl Page for PageB {
    fn update(&mut self, message: MyAppMessage) -> Navigation {
        if let MyAppMessage::PageB(msg) = message {
            match msg {
                PageBMessage::BackButtonPressed => return Navigation::Back,
                PageBMessage::NextButtonPressed => {
                    return Navigation::GoTo(Box::new(PageB::new(self.id + 1)))
                }
            }
        }
        Navigation::None
    }

    fn view(&self) -> iced::Element<MyAppMessage> {
        column![
            text(self.id),
            row![
                button("Back").on_press(MyAppMessage::PageB(Mb::BackButtonPressed)),
                button("Next").on_press(MyAppMessage::PageB(Mb::NextButtonPressed))
            ],
        ]
        .into()
    }
}

```

:arrow_right: Next: From Sandbox To Application
:blue_book: Back: Table of contents

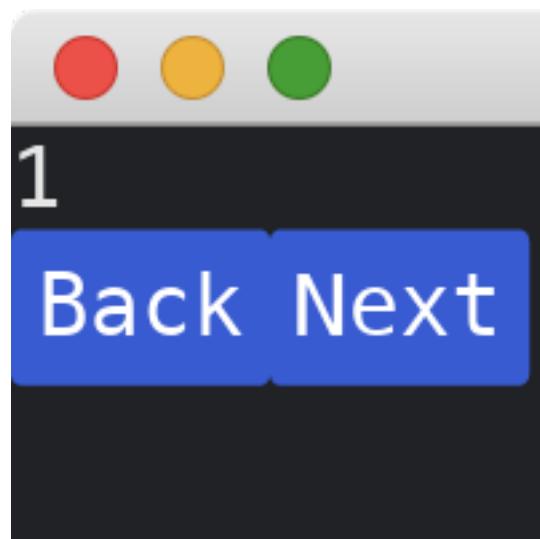


Figure 35: Page B

On Pressed/Released Of Some Widgets

If we only consider mouse pressed or released events, we can use MouseArea. The MouseArea gives the widget being put in it the sense of mouse pressed/released events, even if the widget has no build-in support of the events. For example, we can make a Text to respond to mouse pressed/released events.

```
use iced::widget::mouse_area, Sandbox, Settings;

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    Pressed,
    Released,
}

struct MyApp {
    state: String,
}

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self {
            state: "Start".into(),
        }
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, message: Self::Message) {
```

```

match message {
    MyAppMessage::Pressed => self.state = "Pressed".into(),
    MyAppMessage::Released => self.state = "Released".into(),
}
}

fn view(&self) -> iced::Element<Self::Message> {
    mouse_area(self.state.as_str())
        .on_press(MyAppMessage::Pressed)
        .on_release(MyAppMessage::Released)
        .into()
}
}

```

In addition to `on_press` and `on_release` methods, `MouseArea` also supports `on_middle_press`, `on_right_press`, etc.

When the mouse is pressed:



Figure 36: On pressed/released of some widgets A

And when the mouse is released:



Figure 37: On pressed/released of some widgets B

:arrow_right: Next: Producing Messages By Mouse Events

:blue_book: Back: Table of contents

Passing Parameters Across Pages

This tutorial follows the previous tutorial. We use the same Page trait and MyApp struct.

```
use iced::{
    widget::{button, column, text, text_input},
    Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    PageA(PageAMessage),
    PageB(PageBMessage),
}

trait Page {
    fn update(&mut self, message: MyAppMessage) -> Option<Box<dyn Page>>;
    fn view(&self) -> iced::Element<MyAppMessage>;
}

struct MyApp {
    page: Box<dyn Page>,
}

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self {
            page: Box::new(PageA::new()),
        }
    }
}
```

```

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, message: Self::Message) {
    let page = self.page.update(message);
    if let Some(p) = page {
        self.page = p;
    }
}

fn view(&self) -> iced::Element<Self::Message> {
    self.page.view()
}
}

```

For PageA (the login form), we have a TextInput for names and a submit Button. We pass name field of PageA to new function of PageB when we press the submit button.

```

#[derive(Debug, Clone)]
enum PageAMessage {
    TextChanged(String),
    ButtonPressed,
}
type Ma = PageAMessage;

struct PageA {
    name: String,
}

impl PageA {
    fn new() -> Self {
        Self {
            name: String::new(),
        }
    }
}

impl Page for PageA {
    fn update(&mut self, message: MyAppMessage) -> Option> {
        if let MyAppMessage::PageA(msg) = message {
            match msg {
                PageAMessage::TextChanged(s) => self.name = s,
                PageAMessage::ButtonPressed => {
                    return Some(Box::new(PageB::new(self.name.clone())))
                }
            }
        }
        None
    }
}

```

```

    }

    fn view(&self) -> iced::Element<MyAppMessage> {
        column![
            text_input("Name", &self.name).on_input(|s| MyAppMessage::PageA(Ma::Text(s))),
            button("Log in").on_press(MyAppMessage::PageA(Ma::ButtonPressed)),
        ]
        .into()
    }
}

```

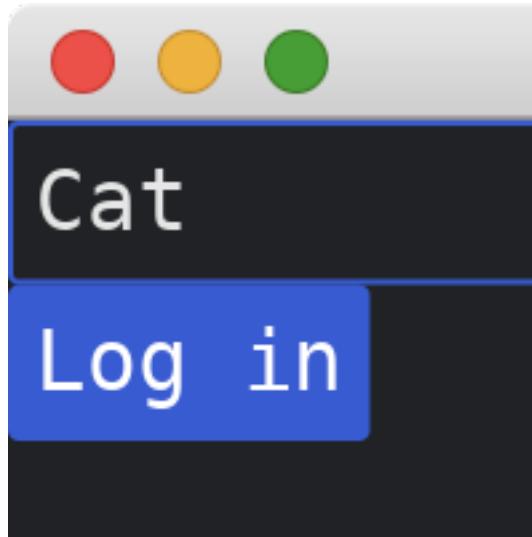


Figure 38: Page A

In PageB, we receive the name from new function and display the name in view.

```

#[derive(Debug, Clone)]
enum PageBMessage {
    ButtonPressed,
}

type Mb = PageBMessage;

struct PageB {
    name: String,
}

impl PageB {
    fn new(name: String) -> Self {
        Self { name }
    }
}

impl Page for PageB {
    fn update(&mut self, message: MyAppMessage) -> Option<Box<dyn Page>> {

```

```

if let MyAppMessage::PageB(msg) = message {
    match msg {
        PageBMessage::ButtonPressed => return Some(Box::new(PageA::new()))
    }
}
None
}

fn view(&self) -> iced::Element<MyAppMessage> {
    column![
        text(format!("Hello {}!", self.name)),
        button("Log out").on_press(MyAppMessage::PageB(Mb::ButtonPressed)),
    ]
    .into()
}
}

```

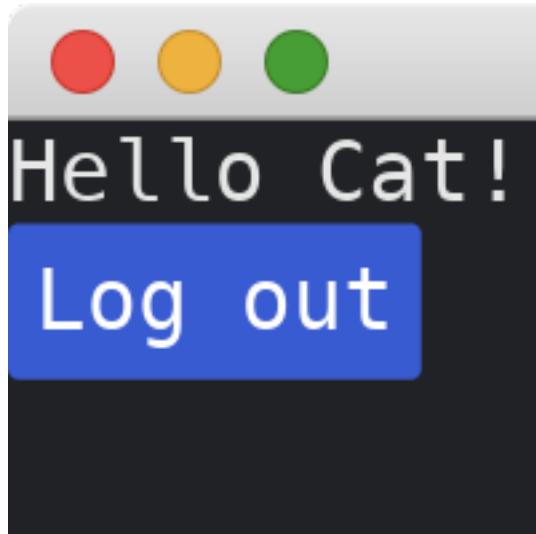


Figure 39: Page B

:arrow_right: Next: Navigation History

:blue_book: Back: Table of contents

PickList

The PickList widget represents a choice among multiple values. It has two methods of constructions. It supports reactions to option selections and menu opening/closing. A placeholder can be set when options are not selected yet. It is able to change styles of the text. We can add padding around the text inside. We can also change the icon of the handle.

```
use iced::{
    font::Family,
    widget::{column, pick_list, pick_list::Handle, row, text, text::Shaping, PickL
    Font, Pixels, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    DoNothing,
    Update3(String),
    Open10,
    Close11,
}

struct MyApp {
    pick_list_3: Option<String>,
    info_10: String,
    info_11: String,
}

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self {
            pick_list_3: Some("Functional pick list".into()),
            info_10: "".into(),
            info_11: "".into(),
        }
    }

    fn update(&mut self, message: MyAppMessage) {
        match message {
            MyAppMessage::Update3(text) => self.info_10 = text,
            MyAppMessage::Close11 => self.info_11 = " ".into(),
            _ => {}
        }
    }

    fn view(&self) -> iced::Element<MyAppMessage> {
        column![

            pick_list::Handle {
                pick_list_3: self.pick_list_3.as_ref().map(|text| text.as_str()),
                info_10: &self.info_10,
                info_11: &self.info_11,
            },
        ]
    }
}
```

```

        info_11: "" .into(),
    }
}

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, message: Self::Message) {
    match message {
        MyAppMessage::DoNothing => {}
        MyAppMessage::Update3(s) => self.pick_list_3 = Some(s),
        MyAppMessage::Open10 => self.info_10 = "Open".into(),
        MyAppMessage::Close11 => self.info_11 = "Close".into(),
    }
}

fn view(&self) -> iced::Element<Self::Message> {
    column![
        PickList::new(
            vec![ "Construct from struct" ],
            Some("Construct from struct"),
            |_| MyAppMessage::DoNothing
        ),
        pick_list(
            vec![ "Construct from function" ],
            Some("Construct from function"),
            |_| MyAppMessage::DoNothing
        ),
        pick_list(
            ["Functional pick list", "Other choices 1", "Other choices 2"]
                .map(|s| s.to_string())
                .to_vec(),
            self.pick_list_3.clone(),
            |s| MyAppMessage::Update3(s)
        ),
        pick_list(vec![ "A", "B", "C" ], None::<&str>, |_| {
            MyAppMessage::DoNothing
        })
        .placeholder("Placeholder"),
        pick_list(vec![ "Different font" ], Some("Different font"), |_| {
            MyAppMessage::DoNothing
        })
        .font(Font {
            family: Family::Fantasy,
            ..Font::DEFAULT
        }),
        pick_list(vec![ "Larger text" ], Some("Larger text"), |_| {

```

```

        MyAppMessage::DoNothing
    })
    .text_size(24),
    pick_list(
        vec!["Special character \u2019"],
        Some("Special character \u2019"),
        |_| MyAppMessage::DoNothing
    )
    .text_shaping(Shaping::Advanced),
    pick_list(vec!["With padding"], Some("With padding"), |_| {
        MyAppMessage::DoNothing
    })
    .padding(20),
    pick_list(vec!["Different handle"], Some("Different handle"), |_| {
        MyAppMessage::DoNothing
    })
    .handle(Handle::Arrow {
        size: Some(Pixels(24.))
    }),
    row![[
        pick_list(vec!["Respond to open"], Some("Respond to open"), |_| {
            MyAppMessage::DoNothing
        })
        .on_open(MyAppMessage::Open10),
        text(&self.info_10),
    ],
    row![[
        pick_list(vec!["Respond to close"], Some("Respond to close"), |_| {
            MyAppMessage::DoNothing
        })
        .on_close(MyAppMessage::Close11),
        text(&self.info_11),
    ],
    ]
    .into()
}
}

:arrow_right: Next: ComboBox
:blue_book: Back: Table of contents

```

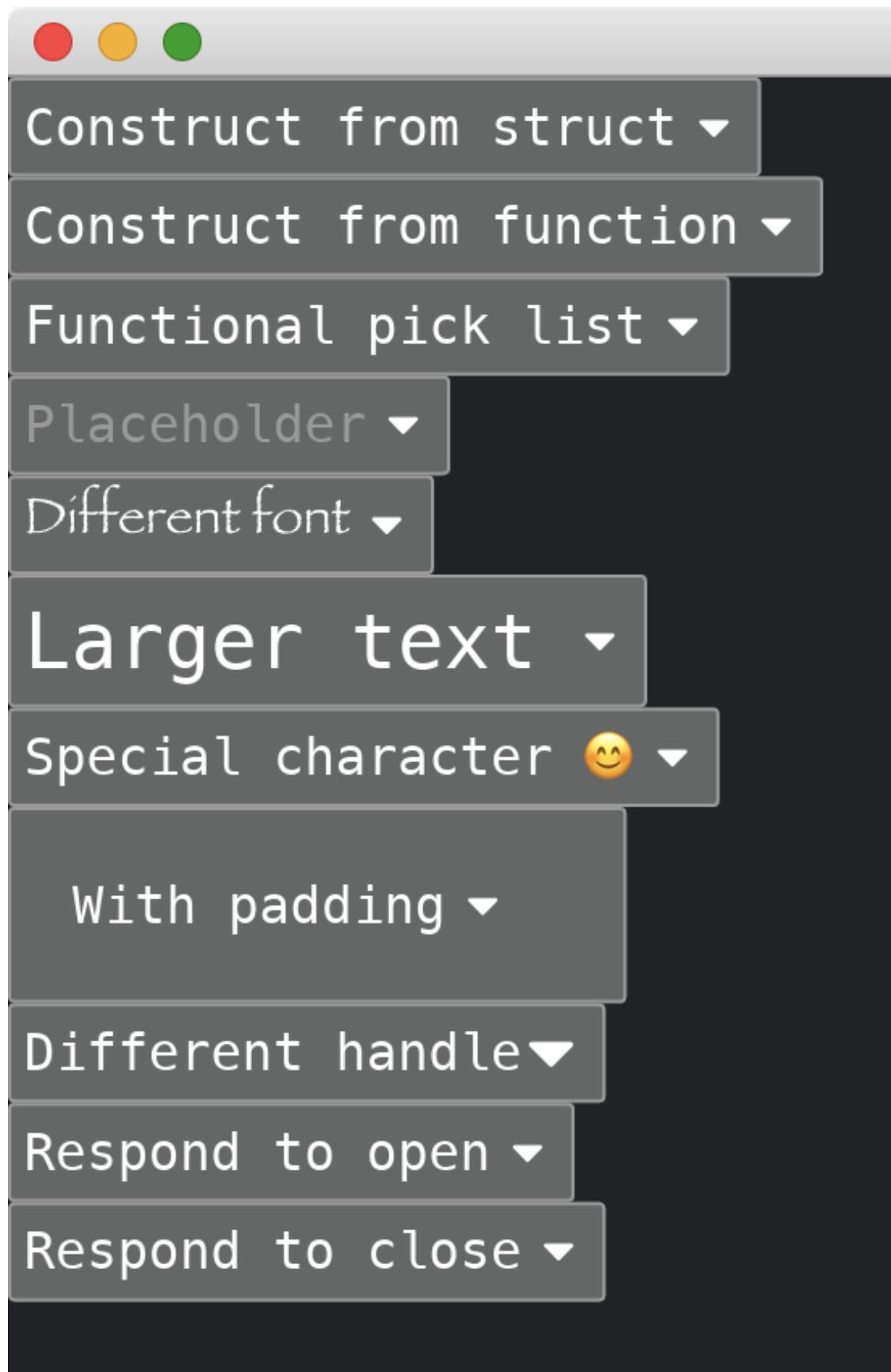


Figure 40: PickList
113

Producing Messages By Keyboard Events

This tutorial follows from the previous tutorial. Instead of capturing Event::Mouse and Event::Touch, we capture Event::Keyboard in the listen_with function.

```
use iced::*;

event::{self, Status},
executor,
keyboard::{key::Named, Event::KeyPressed, Key},
widget::text,
Application, Event, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    KeyPressed(String),
}

struct MyApp {
    pressed_key: String,
}

impl Application for MyApp {
    type Executor = executor::Default;
    type Message = MyAppMessage;
    type Theme = iced::Theme;
    type Flags = ();
}

fn new(_flags: Self::Flags) -> (Self, iced::Command<Self::Message>) {
    (
        Self {
            pressed_key: "".into(),
        },
    )
}
```

```

        iced::Command::none(),
    )
}

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, message: Self::Message) -> iced::Command<Self::Message> {
    match message {
        MyAppMessage::KeyPressed(s) => self.pressed_key = s,
    }
    iced::Command::none()
}

fn view(&self) -> iced::Element<Self::Message> {
    text(self.pressed_key.as_str()).into()
}

fn subscription(&self) -> iced::Subscription<Self::Message> {
    event::listen_with(|event, status| match (event, status) {
        (
            Event::Keyboard(KeyPressed {
                key: Key::Named(Named::Enter),
                ..
            }),
            Status::Ignored,
        ) => Some(MyAppMessage::KeyPressed("Enter".into())),
        (
            Event::Keyboard(KeyPressed {
                key: Key::Named(Named::Space),
                ..
            }),
            Status::Ignored,
        ) => Some(MyAppMessage::KeyPressed("Space".into())),
        _ => None,
    })
}
}

```

:arrow_right: Next: Producing Messages By Timers

:blue_book: Back: Table of contents

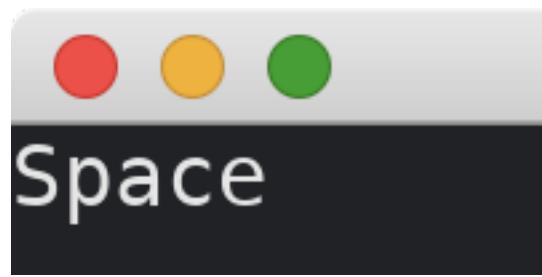


Figure 41: Producing messages by keyboard events

Producing Messages By Mouse Events

To capture events of the window, we implement subscription method in Application. This method returns Subscription struct, which allows us to specify how to handle events. We can use listen_with function to construct a Subscription. The listen_with function takes a function as its input. The input function takes two parameters, Event and Status, and returns Option<MyAppMessage>, which means this function is capable of transforming Event to MyAppMessage. We then receive the transformed MyAppMessage in update method.

In the input function, we only care about ignored events (i.e., events that is not handled by widgets) by checking if Status is Status::Ignored.

In this tutorial, we capture Event::Mouse(...) and Event::Touch(...) and produce messages.

```
use iced::{
    event::{self, Event, Status},
    executor,
    mouse::Event::CursorMoved,
    touch::Event::FingerMoved,
    widget::text,
    Application, Point, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    PointUpdated(Point),
}

struct MyApp {
    mouse_point: Point,
}
```

```

impl Application for MyApp {
    type Executor = executor::Default;
    type Message = MyAppMessage;
    type Theme = iced::Theme;
    type Flags = ();

    fn new(_flags: Self::Flags) -> (Self, iced::Command<Self::Message>) {
        (
            Self {
                mouse_point: Point::ORIGIN,
            },
            iced::Command::none(),
        )
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, message: Self::Message) -> iced::Command<Self::Message> {
        match message {
            MyAppMessage::PointUpdated(p) => self.mouse_point = p,
        }
        iced::Command::none()
    }

    fn view(&self) -> iced::Element<Self::Message> {
        text(format!("{}: {}", self.mouse_point)).into()
    }

    fn subscription(&self) -> iced::Subscription<Self::Message> {
        event::listen_with(|event, status| match (event, status) {
            (Event::Mouse(CursorMoved { position }), Status::Ignored)
            | (Event::Touch(FingerMoved { position, .. }), Status::Ignored) => {
                Some(MyAppMessage::PointUpdated(position))
            }
            _ => None,
        })
    }
}

```

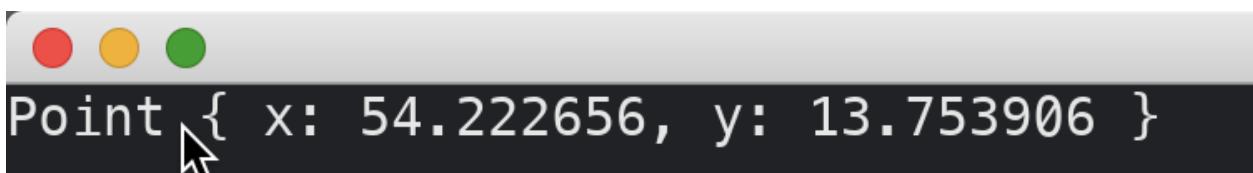


Figure 42: Producing messages by mouse events

:arrow_right: Next: Producing Messages By Keyboard Events

:blue_book: Back: Table of contents

Producing Messages By Timers

To use build-in timers, we need to enable one of the following features: tokio, async-std, or smol. In this tutorial, we use tokio feature. The dependencies of Cargo.toml should look like this:

```
[dependencies]
iced = { version = "0.12.1", features = ["tokio"] }
```

We use time::every function to obtain Subscription<Instant> struct. Then we map the struct to Subscription<MyAppMessage> by Subscription::map method. The result will be returned in the subscription method of Application. The corresponding MyAppMessage will be received in the update method.

```
use iced::*;

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    Update,
}

struct MyApp {
    seconds: u32,
}

impl Application for MyApp {
    type Executor = executor::Default;
    type Message = MyAppMessage;
    type Theme = iced::Theme;
    type Flags = ();
}
```

```

fn new(_flags: Self::Flags) -> (Self, iced::Command<Self::Message>) {
    (Self { seconds: 0 }, Command::none())
}

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, message: Self::Message) -> iced::Command<Self::Message> {
    match message {
        MyAppMessage::Update => self.seconds += 1,
    }
    Command::none()
}

fn view(&self) -> iced::Element<Self::Message> {
    text(self.seconds).into()
}

fn subscription(&self) -> iced::Subscription<Self::Message> {
    time::every(Duration::from_secs(1)).map(|_| MyAppMessage::Update)
}
}

```



Figure 43: Producing messages by timers

:arrow_right: Next: Batch Subscriptions

:blue_book: Back: Table of contents

Producing Widget Messages

Our custom widgets are able to send Message.

To do so, we need to store the Message we are going to send in the widget.

```
struct MyWidget<Message> {
    pressed_message: Message,
}

impl<Message> MyWidget<Message> {
    fn new(pressed_message: Message) -> Self {
        Self { pressed_message }
    }
}
```

We use a generic type Message for MyWidget, so that the sent pressed_message in MyWidget will match the associated type Message of Sandbox.

The message pressed_message will be sent when the widget is pressed.

```
fn on_event(
    &mut self,
    _state: &mut Tree,
    event: Event,
    layout: Layout<'_>,
    cursor: mouse::Cursor,
    _renderer: &Renderer,
    _clipboard: &mut dyn Clipboard,
    shell: &mut Shell<'_, Message>,
    _viewport: &Rectangle,
) -> event::Status {
    if cursor.is_over(layout.bounds()) {
        match event {
            Event::Mouse(mouse::Event::ButtonPressed(_)) => {
                shell.publish(self.pressed_message.clone());
                event::Status::Captured
            }
            _ => event::Status::Ignored,
        }
    } else {

```

```

        event::Status::Ignored
    }
}

```

We use `shell.publish(self.pressed_message.clone())` to send `pressed_message` to our app. To ensure the mouse pressed event happens within the range of the widget, we use `cursor.is_over(layout.bounds())` to check the mouse position and match the event to `Event::Mouse(mouse::Event::ButtonPressed(_))` to check the mouse button state.

Finally, we pass our Message to the widget.

```

#[derive(Debug, Clone)]
enum MyMessage {
    MyWidgetPressed,
}

// ...

impl Sandbox for MyApp {
    type Message = MyMessage;

    // ...

    fn view(&self) -> iced::Element<'_, Self::Message> {
        container(
            column![
                MyWidget::new(MyMessage::MyWidgetPressed),
                // ...
            ]
            // ...
        )
        // ...
    }
}

```

The full code is as follows:

```

use iced::*;

advanced::{
    graphics::core::event,
    layout, mouse,
    renderer::{self, Quad},
    widget::Tree,
    Clipboard, Layout, Shell, Widget,
},
widget::{column, container, text},
Border, Color, Element, Event, Length, Rectangle, Sandbox, Settings, Shadow, S
};

```

```

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyMessage {
    MyWidgetPressed,
}

struct MyApp {
    count: u32,
}

impl Sandbox for MyApp {
    type Message = MyMessage;

    fn new() -> Self {
        Self { count: 0 }
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, message: Self::Message) {
        match message {
            MyMessage::MyWidgetPressed => self.count += 1,
        }
    }

    fn view(&self) -> iced::Element<Self::Message> {
        container(
            column![MyWidget::new(MyMessage::MyWidgetPressed), text(self.count)]
                .spacing(20)
                .align_items(iced::Alignment::Center),
        )
        .width(Length::Fill)
        .height(Length::Fill)
        .center_x()
        .center_y()
        .into()
    }
}

struct MyWidget<Message> {
    pressed_message: Message,
}

```

```

impl<Message> MyWidget<Message> {
    fn new(pressed_message: Message) -> Self {
        Self { pressed_message }
    }
}

impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidget<Message>
where
    Renderer: iced::advanced::Renderer,
    Message: Clone,
{
    fn size(&self) -> Size<Length> {
        Size {
            width: Length::Shrink,
            height: Length::Shrink,
        }
    }

    fn layout(
        &self,
        _tree: &mut Tree,
        _renderer: &Renderer,
        _limits: &layout::Limits,
        _layout::Node {
            layout::Node::new([100, 100].into())
        }
    )

    fn draw(
        &self,
        _state: &Tree,
        renderer: &mut Renderer,
        _theme: &Theme,
        _style: &renderer::Style,
        layout: Layout<'_>,
        _cursor: mouse::Cursor,
        _viewport: &Rectangle,
    ) {
        renderer.fill_quad(
            Quad {
                bounds: layout.bounds(),
                border: Border {
                    color: Color::from_rgb(0.6, 0.8, 1.0),
                    width: 1.0,
                    radius: 10.0.into(),
                },
                shadow: Shadow::default(),
            },
            Color::from_rgb(0.0, 0.2, 0.4),
        )
    }
}

```

```

    );
}

fn on_event(
    &mut self,
    _state: &mut Tree,
    event: Event,
    layout: Layout<'_>,
    cursor: mouse::Cursor,
    _renderer: &Renderer,
    _clipboard: &mut dyn Clipboard,
    shell: &mut Shell<'_>, Message>,
    _viewport: &Rectangle,
) -> event::Status {
    if cursor.is_over(layout.bounds()) {
        match event {
            Event::Mouse(mouse::Event::ButtonPressed(_)) => {
                shell.publish(self.pressed_message.clone());
                event::Status::Captured
            }
            _ => event::Status::Ignored,
        }
    } else {
        event::Status::Ignored
    }
}
}

impl<'a, Message, Renderer> From<MyWidget<Message>> for Element<'a, Message, Theme>
where
    Message: 'a + Clone,
    Renderer: iced::advanced::Renderer,
{
    fn from(widget: MyWidget<Message>) -> Self {
        Self::new(widget)
    }
}

:arrow_right: Next: Mouse Pointer Over Widgets
:blue_book: Back: Table of contents

```

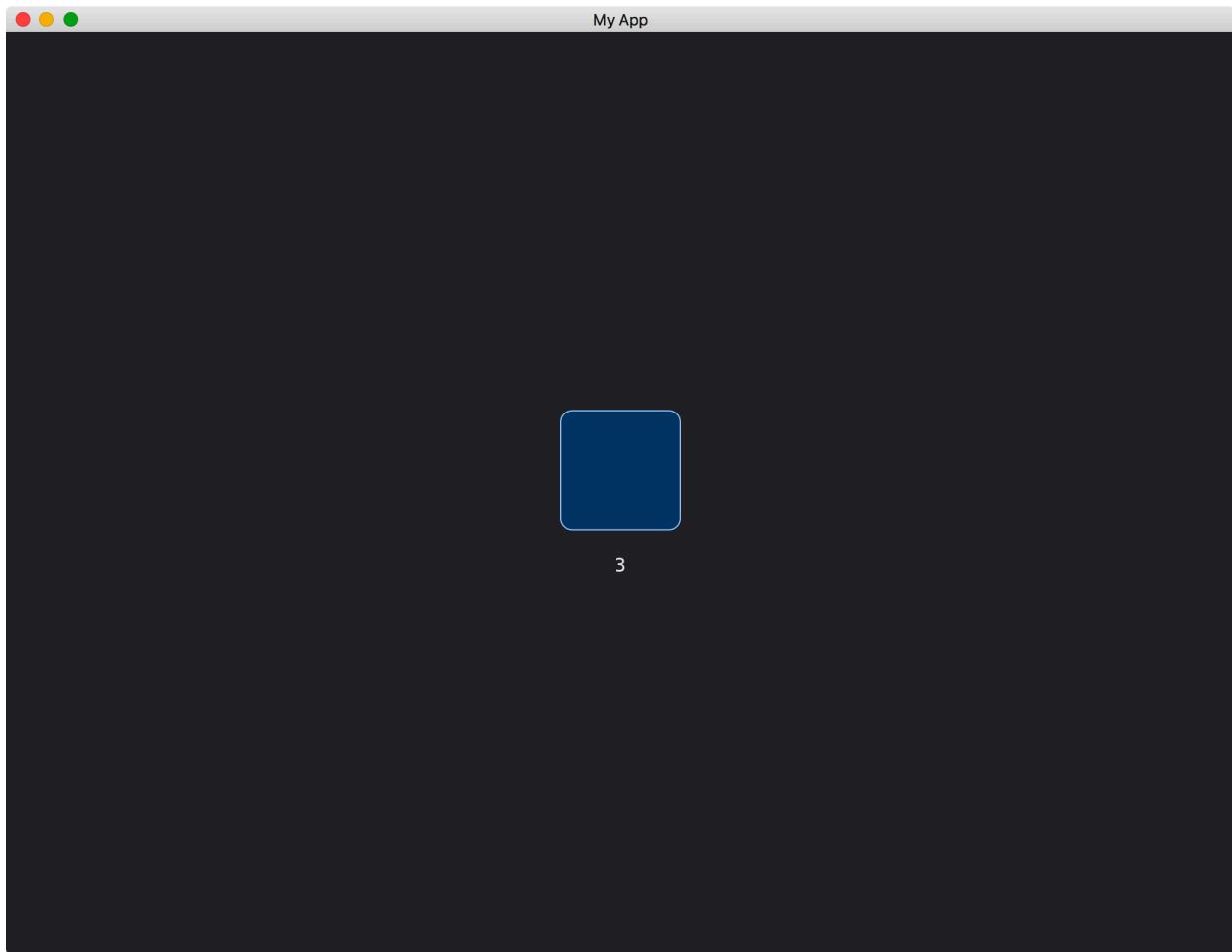


Figure 44: Producing Widget Messages

ProgressBar

The ProgressBar widget represents a value in a given range. It has two methods of constructions.

```
use iced::{
    widget::{column, progress_bar, text, ProgressBar},
    Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            text("Construct from struct"),
            ProgressBar::new(0.0..=100.0, 50.),
            text("Construct from function"),
            progress_bar(0.0..=100.0, 30.),
        ]
        .into()
    }
}
```

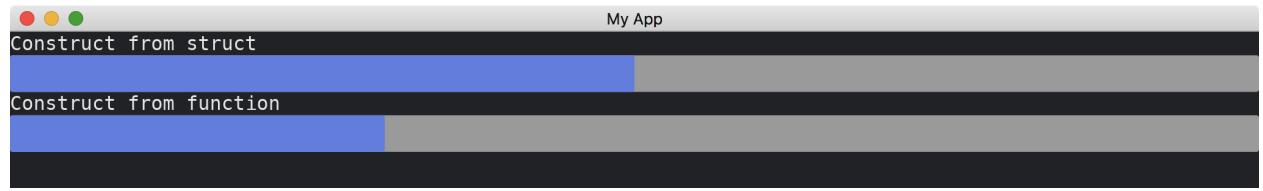


Figure 45: ProgressBar

:arrow_right: Next: Tooltip

:blue_book: Back: Table of contents

Radio

The Radio widget represents a choice among multiple values. It has two methods of constructions. It supports reactions to clicking and touching. It is able to change styles of the button and the text. It can also change the space between them.

```
use iced::{
    font::Family,
    widget::{column, radio, row, text, text::Shaping, Radio},
    Font, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    DoNothing,
    Update3(u32),
    Update4(String),
}

struct MyApp {
    radio3: Option<u32>,
    radio4: Option<String>,
}

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self {
            radio3: Some(1),
            radio4: Some("a".into()),
        }
    }

    fn title(&self) -> String {

```

```

    String::from("My App")
}

fn update(&mut self, message: Self::Message) {
    match message {
        MyAppMessage::DoNothing => {}
        MyAppMessage::Update3(i) => self.radio3 = Some(i),
        MyAppMessage::Update4(s) => self.radio4 = Some(s),
    }
}

fn view(&self) -> iced::Element<Self::Message> {
    column![
        Radio::new("Construct from struct", 0, None, |_| {
            MyAppMessage::DoNothing
        }),
        radio("Construct from function", 0, None, |_| {
            MyAppMessage::DoNothing
        }),
        row![text("Functional radio"),
            radio("A", 1, self.radio3, |i| MyAppMessage::Update3(i)),
            radio("B", 2, self.radio3, |i| MyAppMessage::Update3(i)),
            radio("C", 3, self.radio3, |i| MyAppMessage::Update3(i)),
        ],
        row![text("Radio of String values"),
            radio("A", &"a".to_string(), self.radio4.as_ref(), |s| {
                MyAppMessage::Update4(s.into())
            }),
            radio("B", &"b".to_string(), self.radio4.as_ref(), |s| {
                MyAppMessage::Update4(s.into())
            }),
            radio("C", &"c".to_string(), self.radio4.as_ref(), |s| {
                MyAppMessage::Update4(s.into())
            }),
        ],
        radio("Larger button", 0, None, |_| MyAppMessage::DoNothing).size(40),
        radio("Different font", 0, None, |_| MyAppMessage::DoNothing).font(Font{
            family: Family::Fantasy,
            ..Font::DEFAULT
        }),
        radio("Larger text", 0, None, |_| MyAppMessage::DoNothing).text_size(24),
        radio("Special character ☀", 0, None, |_| {
            MyAppMessage::DoNothing
        })
        .text_shaping(Shaping::Advanced),
        radio("Space between button and text", 0, None, |_| {

```

```
        MyAppMessage::DoNothing  
    })  
    .spacing(30),  
]  
.into()  
}  
}
```

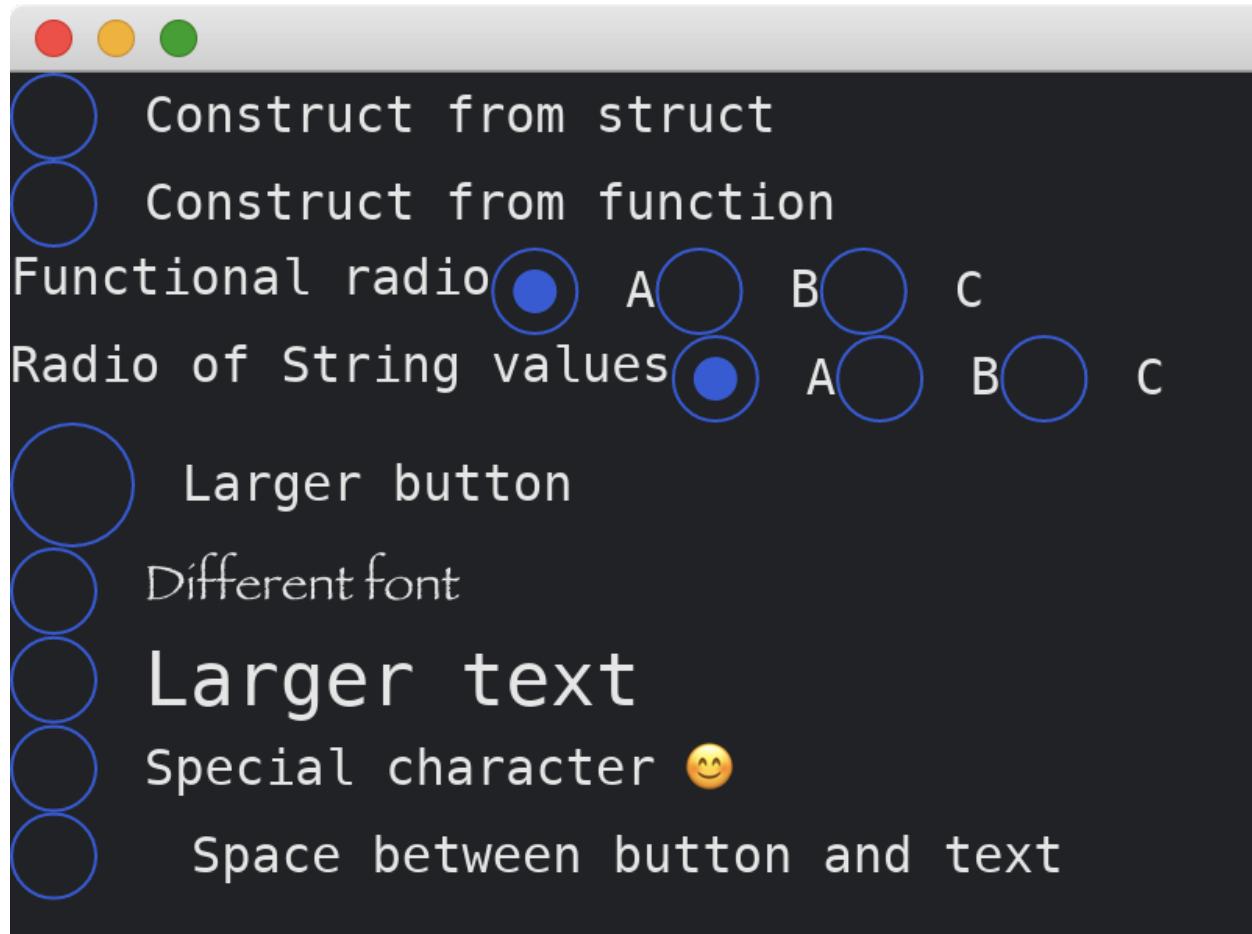


Figure 46: Radio

:arrow_right: Next: PickList

:blue_book: Back: Table of contents

Row

Similar to Column, Row helps us placing widgets horizontally. It can leave some space between its boundary and its inner content. It can also add spaces among its inner widgets. The inner widgets can be aligned top, middle or bottom.

```
use iced::{
    widget::{column, row, Row},
    Alignment, Length, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            Row::with_children(vec![
                "Construct from the with_children function.".into(),
                "another element".into(),
            ]),
            Row::new()
                .push("Construct from the new function and the push method.")
                .push("another element again"),
        ]
    }
}
```

```

        row(vec!["Construct from function".into()]),
        row!["Construct from macro"],
        row!["With padding"].padding(20),
        row![ "Space between elements", "Space between elements", ].spacing(20),
        row![ "Different alignment"]
            .height(Length::Fill)
            .align_items(Alignment::Center),
        ]
    .into()
}
}

```

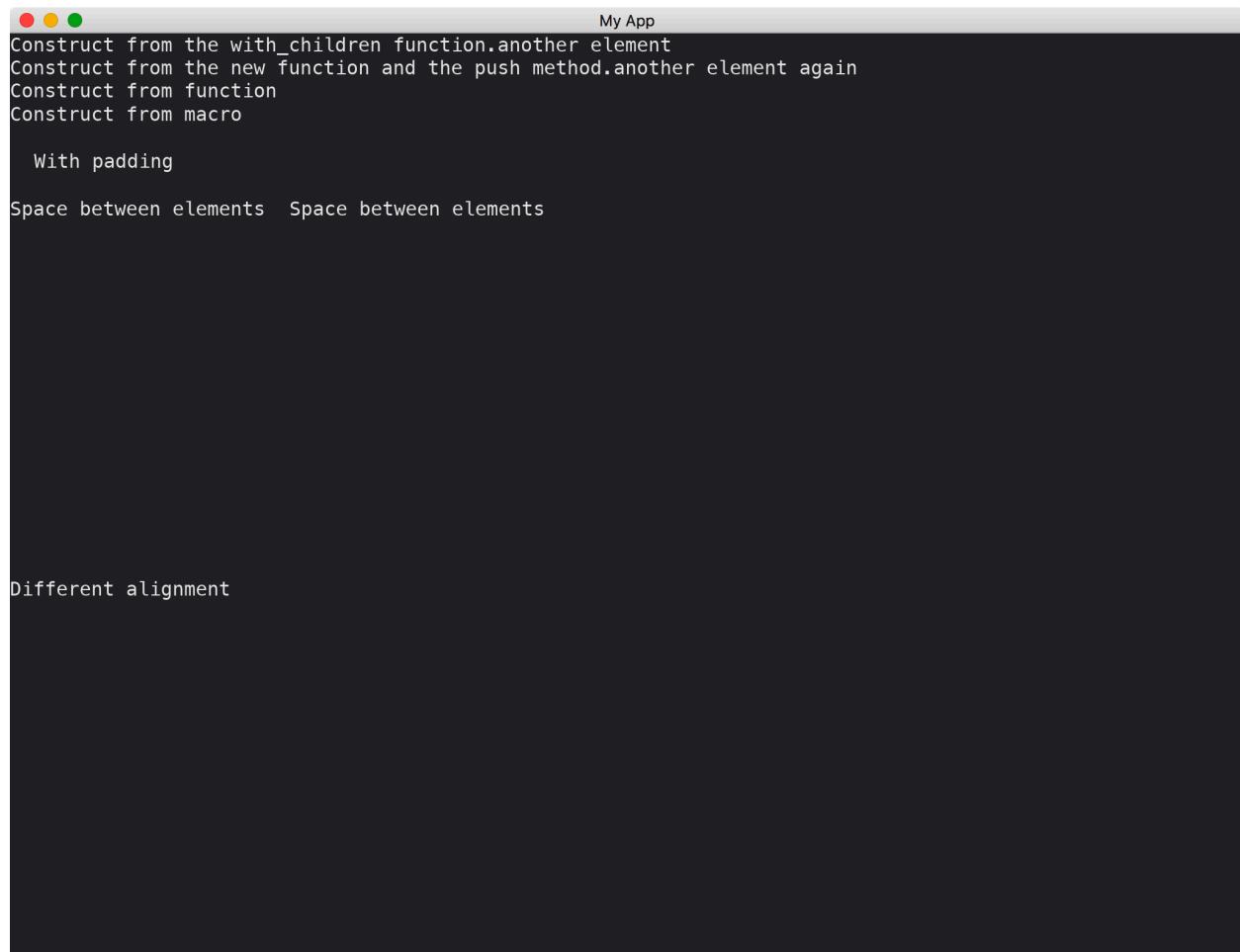


Figure 47: Row

:arrow_right: Next: Space

:blue_book: Back: Table of contents

Rule

The Rule widget is a horizontal (or vertical) line for separating widgets clearly. It has two methods of constructions. We can change the space around it. The widget can be set to be either horizontal or vertical.

```
use iced::{
    widget::{column, horizontal_rule, text, vertical_rule, Rule},
    Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            text("Construct from struct"),
            Rule::horizontal(0),
            text("Construct from function"),
            horizontal_rule(0),
            text("Different space"),
            horizontal_rule(50),
            text("Vertical rule"),
        ]
    }
}
```

```
        vertical_rule(100),  
    ]  
.into()  
}  
}
```

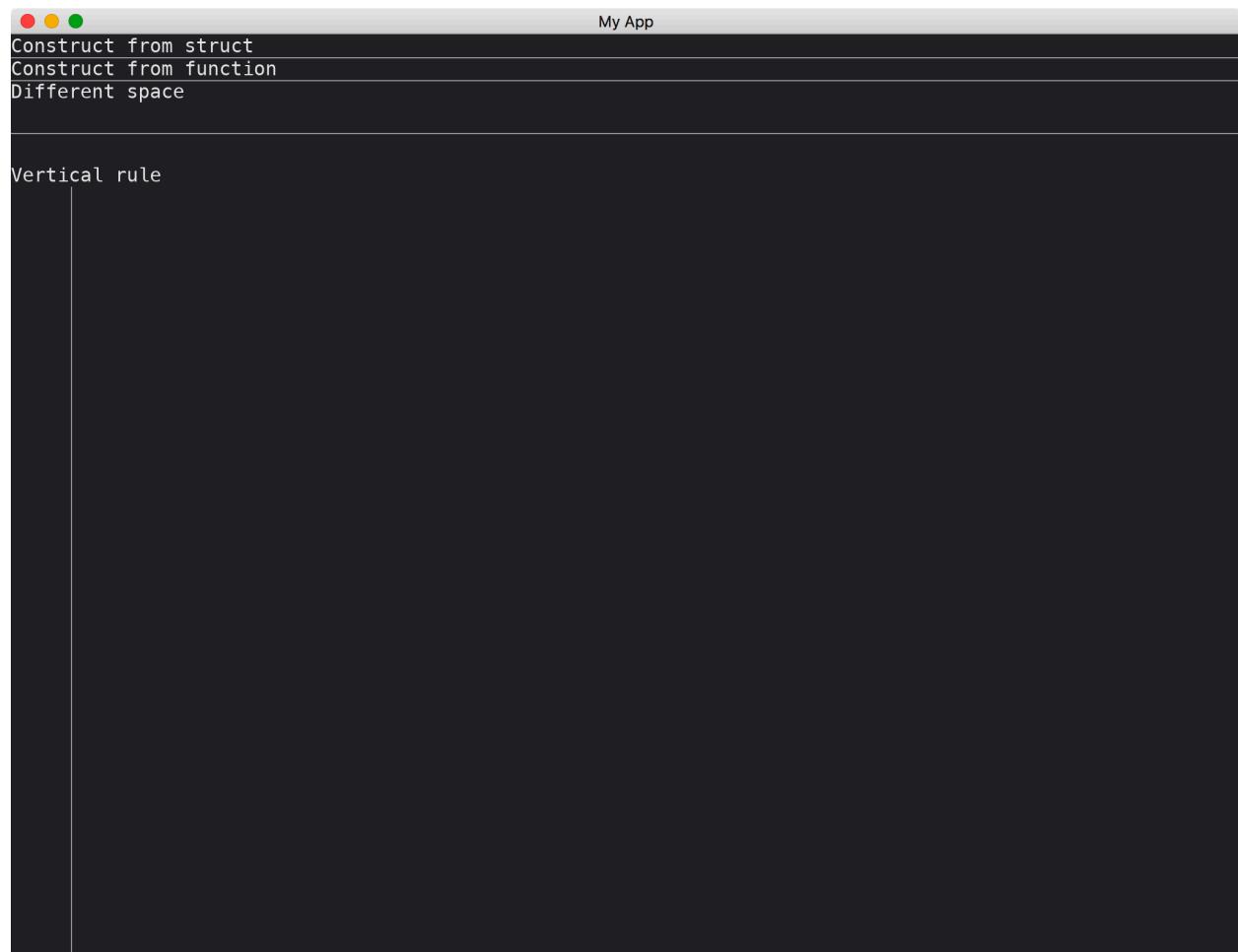


Figure 48: Rule

:arrow_right: Next: Image

:blue_book: Back: Table of contents

Scrolled

When there are too many widgets, they may go beyond the boundary of the window. Scrolled provides an infinite space that widgets can be navigated by scroll bars. The scroll bars can be vertical, horizontal or both. When the scroll bars are changed, we can also receive their scroll positions and update other widgets.

```
use iced::{
    widget::{
        column, row,
        scrollable::{Direction, Properties, Viewport},
        text, Scrolled,
    },
    Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyMessage {
    Scrolled4(Viewport),
}

struct MyApp {
    offset4: String,
}

impl Sandbox for MyApp {
    type Message = MyMessage;

    fn new() -> Self {
        Self { offset4: "".into() }
    }

    fn title(&self) -> String {
        String::from("My App")
    }
}
```

```

fn update(&mut self, message: Self::Message) {
    match message {
        MyMessage::Scrolled4(v) => {
            self.offset4 = format!("{} {}", v.absolute_offset().x, v.absolute_offset().y);
        }
    }
}

fn view(&self) -> iced::Element<Self::Message> {
    let long_vertical_texts =
        column((0..10).map(|i| text(format!("{} vertical scrollable", i + 1))));
    let long_horizontal_texts =
        row((0..10).map(|i| text(format!("{} horizontal scrollable ", i + 1))));
    let long_both_texts = column(
        (0..10).map(|i| text(format!("{} vertical and horizontal scrollable", i + 1))));
    let long_both_texts_2 = column(
        (0..10).map(|i| text(format!("{} vertical and horizontal scrollable", i + 1))));

    column![
        Scrollable::new(long_vertical_texts)
            .width(230)
            .height(105)
            .direction(Direction::Vertical(Properties::new())),
        Scrollable::new(long_horizontal_texts)
            .width(500)
            .height(30)
            .direction(Direction::Horizontal(Properties::new())),
        Scrollable::new(long_both_texts)
            .width(230)
            .height(105)
            .direction(Direction::Both {
                vertical: Properties::new(),
                horizontal: Properties::new()
            }),
        column![
            Scrollable::new(long_both_texts_2)
                .width(230)
                .height(105)
                .direction(Direction::Both {
                    vertical: Properties::new(),
                    horizontal: Properties::new()
                })
                .on_scroll(MyMessage::Scrolled4),
            text(&self.offset4),
        ],
    ],
}

```

```
        ]
        .spacing(50)
        .into()
    }
}
```

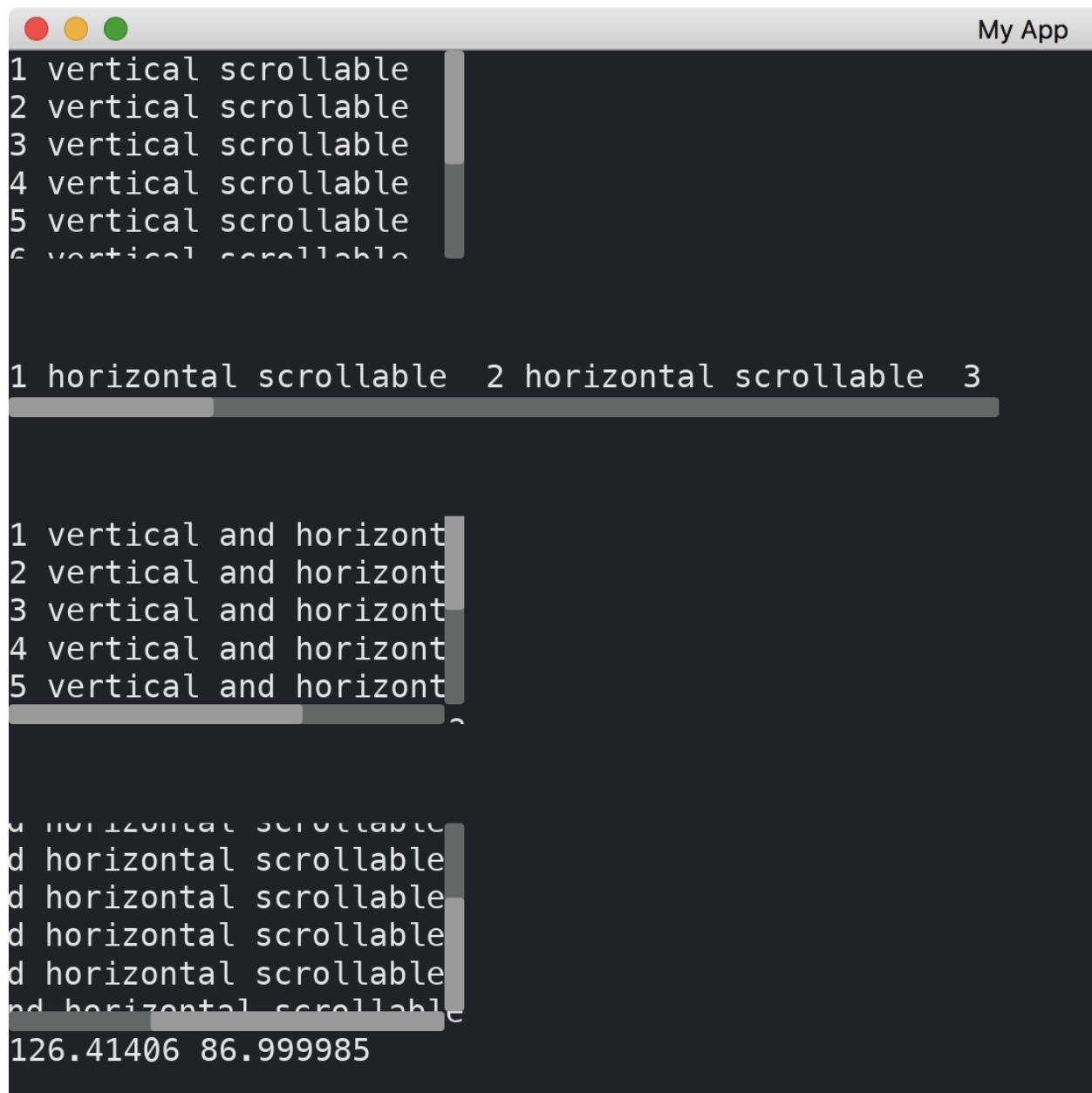


Figure 49: Scrollable

Instead of using `Scrollable::new`, we can also use the `scrollable` function.

:arrow_right: Next: Changing Themes

:blue_book: Back: Table of contents

Setting Up

Initialize a Cargo project.

```
cargo new my_project
```

where `my_project` is the name of the project.

Add Iced to the project dependencies.

```
cd my_project  
cargo add iced
```

You should see the dependency in the end of `Cargo.toml` file.

```
[dependencies]  
iced = "0.12.1"
```

Note: If you encounter `WGPU` Error, you can disable `wgpu` in `Cargo.toml` file.

```
[dependencies]  
iced = { version = "0.12.1", default-features = false }
```

:arrow_right: Next: First App - Hello World!

:blue_book: Back: Table of contents

Slider And VerticalSlider

The Slider widget represents a chosen value in a given range. It has two methods of constructions. It supports reactions to mouse pressing/releasing and touching. The selected value can be snapped to a given step. The widget can be set to be either horizontal or vertical.

```
use iced::{
    widget::{column, slider, text, vertical_slider, Slider},
    Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    DoNothing,
    Update3(u32),
    Update4(u32),
    Update5(u32),
    Update6(u32),
    Update7(u32),
    Release7,
    Update8(u32),
}

struct MyApp {
    value3: u32,
    value4: u32,
    value5: u32,
    value6: u32,
    value7: u32,
    released_value_7: u32,
    value8: u32,
}

impl Sandbox for MyApp {
```

```

type Message = MyAppMessage;

fn new() -> Self {
    Self {
        value3: 50,
        value4: 50,
        value5: 50,
        value6: 50,
        value7: 50,
        released_value_7: 50,
        value8: 50,
    }
}

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, message: Self::Message) {
    match message {
        MyAppMessage::DoNothing => {}
        MyAppMessage::Update3(v) => self.value3 = v,
        MyAppMessage::Update4(v) => self.value4 = v,
        MyAppMessage::Update5(v) => self.value5 = v,
        MyAppMessage::Update6(v) => self.value6 = v,
        MyAppMessage::Update7(v) => self.value7 = v,
        MyAppMessage::Release7 => self.released_value_7 = self.value7,
        MyAppMessage::Update8(v) => self.value8 = v,
    }
}

fn view(&self) -> iced::Element<Self::Message> {
    column![
        text("Construct from struct"),
        Slider::new(0..=100, 50, |_| MyAppMessage::DoNothing),
        text("Construct from function"),
        slider(0..=100, 50, |_| MyAppMessage::DoNothing),
        text("Functional slider"),
        slider(0..=100, self.value3, |v| MyAppMessage::Update3(v)),
        text("Shorter parameter"),
        slider(0..=100, self.value4, MyAppMessage::Update4),
        text("Different step"),
        slider(0..=100, self.value5, MyAppMessage::Update5).step(10u32),
        text("Different step when a shift key is pressed"),
        slider(0..=100, self.value6, MyAppMessage::Update6).shift_step(10u32),
        text(format!("React to mouse release: {}", self.released_value_7)),
        slider(0..=100, self.value7, MyAppMessage::Update7).on_release(MyAppMe
text("Press Ctrl (or Command) and click to return to the default value")
    ]
}

```

```

        slider(0..=100, self.value8, MyAppMessage::Update8).default(30u32),
        text("Vertical slider"),
        vertical_slider(0..=100, 50, |_| MyAppMessage::DoNothing),
    ]
    .into()
}
}

```

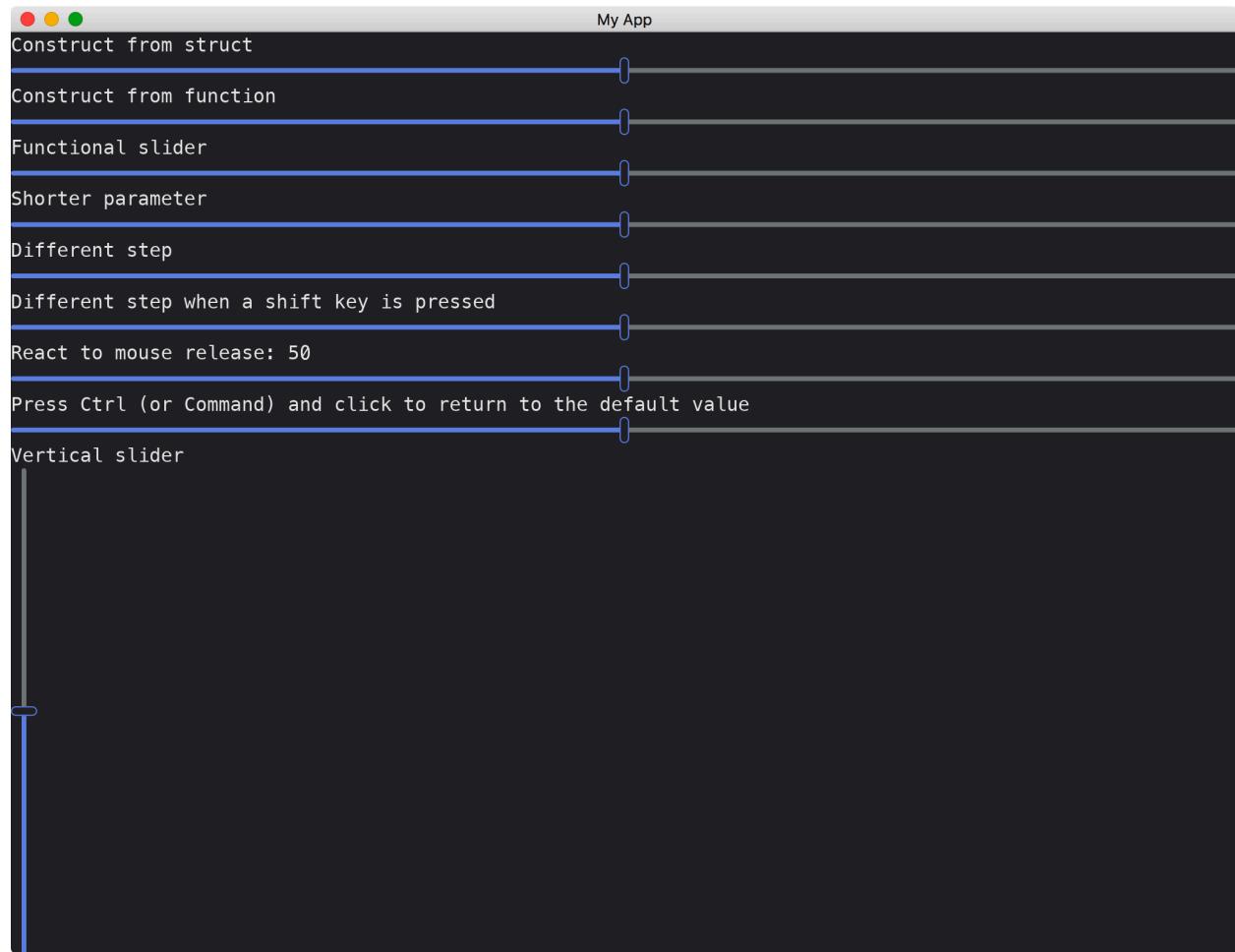


Figure 50: Slider

:arrow_right: Next: ProgressBar

:blue_book: Back: Table of contents

Space

Space is a convenient widget that helps us laying out our widgets. It is an empty widget that occupies a space. It has several constructions to help us allocating spaces horizontally, vertically or both.

```
use iced::{
    widget::{button, column, horizontal_space, row, vertical_space, Space},
    Alignment, Length, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            row![
                button("Horizontal space 1A"),
                Space::with_width(50),
                button("Horizontal space 1B"),
            ],
            row![
                button("Horizontal space 2A"),
            ],
        ]
    }
}
```

```

        Space::with_width(Length::Fill),
        button("Horizontal space 2B"),
    ],
    row![
        button("Horizontal space 3A"),
        horizontal_space(),
        button("Horizontal space 3B"),
    ],
    button("Vertical space 1A"),
    Space::with_height(50),
    button("Vertical space 1B"),
    Space::with_height(Length::Fill),
    button("Vertical space 2A"),
    vertical_space(),
    button("Vertical space 2B"),
    button("Diagonal space A"),
    row![Space::new(50, 50), button("Diagonal space B")].align_items(Align
]
    .into()
}
}

```

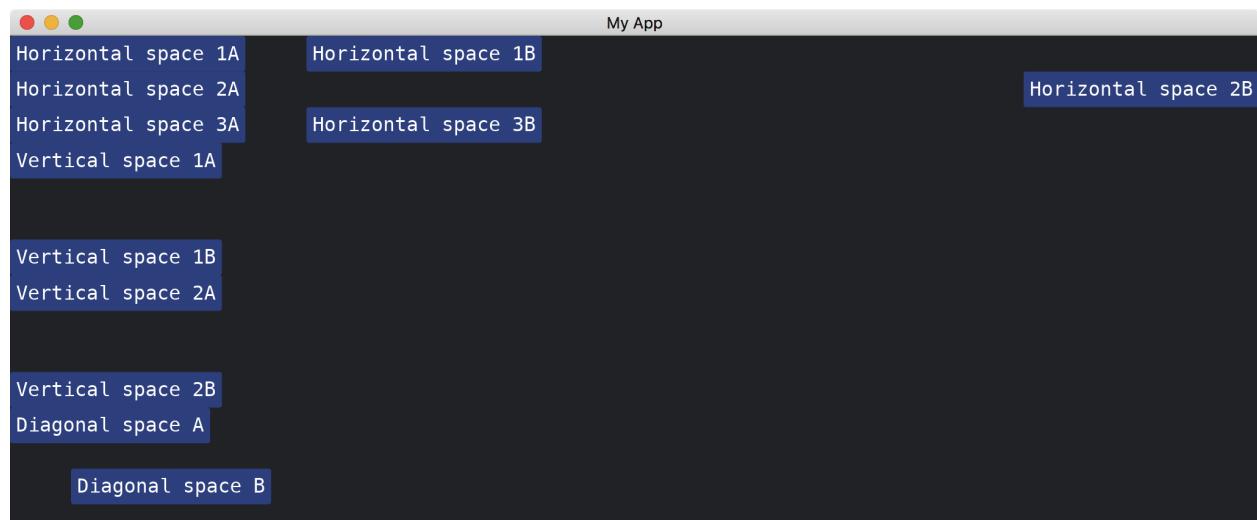


Figure 51: Space

:arrow_right: Next: Container

:blue_book: Back: Table of contents

Svg

The Svg widget is able to display an SVG image. It has two methods of constructions. We can set how to fit the image content into the widget bounds.

To use the widget, we have to enable the svg feature. The Cargo.toml dependencies should look like this:

```
[dependencies]
iced = { version = "0.12.1", features = ["svg"] }
```

Let's add an SVG image named pic.svg into the project root directory, i.e., the image has the path my_project/pic.svg where my_project is the name of our project. The file pic.svg contains the following content:

```
<svg viewBox="0 0 400 300" xmlns="http://www.w3.org/2000/svg">
    <rect width="400" height="300" style="fill:rgb(100,130,160)"/>
    <circle cx="200" cy="150" r="100" style="fill:rgb(180,210,240)"/>
</svg>
```

Our example is as follows:

```
use iced::*;

widget::{column, svg, svg::Handle, text, Svg},
ContentFit, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }
}
```

```
}

fn update(&mut self, _message: Self::Message) {}

fn view(&self) -> iced::Element<Self::Message> {
    column![  
        text("Construct from struct"),  
        Svg::from_path("pic.svg"),  
        text("Construct from function"),  
        svg(Handle::from_path("pic.svg")),  
        text("Different content fit"),  
        Svg::from_path("pic.svg").content_fit(ContentFit::None),  
    ]  
.into()  
}
}
```

:arrow_right: Next: Width And Height

:blue_book: Back: Table of contents

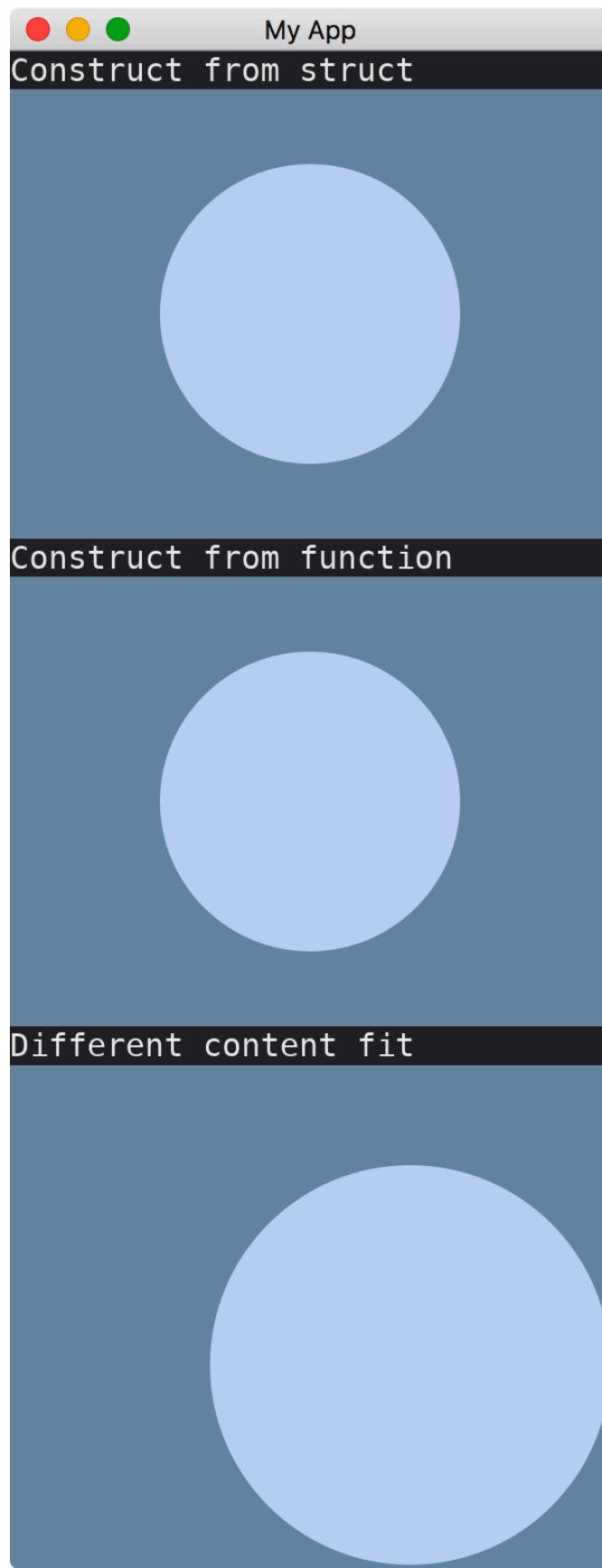


Figure 52: Svg
149

Taking Any Children

Since all Widget can be transformed to Element, our custom widget is able to take any Widget as its children.

This time, our MyWidgetOuter will take an Element as its inner widget when it is initialized.

```
struct MyWidgetOuter<'a, Message, Renderer> {
    inner_widget: Element<'a, Message, Theme, Renderer>,
}

impl<'a, Message, Renderer> MyWidgetOuter<'a, Message, Renderer>
where
    Renderer: iced::advanced::Renderer,
{
    fn new(inner_widget: Element<'a, Message, Theme, Renderer>) -> Self {
        Self { inner_widget }
    }
}
```

When we draw or layout the `inner_widget`, we will use its methods from Widget. Yet, the `inner_widget` is of type Element. So, we have to cast it as Widget by the `as_widget` method.

```
fn layout(
    &self,
    tree: &mut Tree,
    renderer: &Renderer,
    limits: &layout::Limits,
) -> layout::Node {
    let mut child_node =
        self.inner_widget
            .as_widget()
            .layout(&mut tree.children[0], renderer, limits);

    let size_of_this_node = child_node.size().expand(Size::new(50., 50.));

    child_node = child_node.align(Alignment::Center, Alignment::Center, size_of_th
        layout::Node::with_children(size_of_this_node, vec![child_node]))
```

```
}
```

In the code above, we make the size of `MyWidgetOuter` relative to its `inner_widget`. More precisely, we retrieve the size of `inner_widget` and pad the size as the size of `MyWidgetOuter`.

Then, in the `draw` method of `MyWidgetOuter`, we also draw the `inner_widget`.

```
fn draw(
    &self,
    state: &Tree,
    renderer: &mut Renderer,
    theme: &Theme,
    style: &renderer::Style,
    layout: Layout<'_>,
    cursor: mouse::Cursor,
    viewport: &Rectangle,
) {
    renderer.fill_quad(
        Quad {
            bounds: layout.bounds(),
            border: Border {
                color: Color::from_rgb(0.6, 0.93, 1.0),
                width: 1.0,
                radius: 10.0.into(),
            },
            shadow: Shadow::default(),
        },
        Color::from_rgb(0.0, 0.33, 0.4),
    );
    self.inner_widget.as_widget().draw(
        &state.children[0],
        renderer,
        theme,
        style,
        layout.children().next().unwrap(),
        cursor,
        viewport,
    );
}
```

Note that we have to pass the child state `&state.children[0]` to `inner_widget` since the anonymous widget may need the information about its state.

To make the underlying system aware of the child state, we have to explicitly tell the system the existence of the child. Otherwise, `state.children` in `draw` will be empty.

```
fn children(&self) -> Vec<Tree> {
    vec![Tree::new(self.inner_widget.as_widget())]
}
```

```

fn diff(&self, tree: &mut Tree) {
    tree.diff_children(std::slice::from_ref(&self.inner_widget));
}

```

The full code is as follows:

```

use iced::*;

advanced::{
    layout, mouse,
    renderer::{self, Quad},
    widget::Tree,
    Layout, Widget,
},
widget::{button, container},
Alignment, Border, Color, Element, Length, Rectangle, Sandbox, Settings, Shadow
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        container(MyWidgetOuter::new(button("Other widget").into()))
            .width(Length::Fill)
            .height(Length::Fill)
            .center_x()
            .center_y()
            .into()
    }
}

struct MyWidgetOuter<'a, Message, Renderer> {
    inner_widget: Element<'a, Message, Theme, Renderer>,
}

```

```

}

impl<'a, Message, Renderer> MyWidgetOuter<'a, Message, Renderer>
where
    Renderer: iced::advanced::Renderer,
{
    fn new(inner_widget: Element<'a, Message, Theme, Renderer>) -> Self {
        Self { inner_widget }
    }
}

impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidgetOuter<'_, Message, Renderer> {
    Renderer: iced::advanced::Renderer,
{
    fn size(&self) -> Size<Length> {
        Size {
            width: Length::Shrink,
            height: Length::Shrink,
        }
    }

    fn diff(&self, tree: &mut Tree) {
        tree.diff_children(std::slice::from_ref(&self.inner_widget));
    }

    fn layout(
        &self,
        tree: &mut Tree,
        renderer: &Renderer,
        limits: &layout::Limits,
    ) -> layout::Node {
        let mut child_node =
            self.inner_widget
                .as_widget()
                .layout(&mut tree.children[0], renderer, limits);

        let size_of_this_node = child_node.size().expand(Size::new(50., 50.));

        child_node = child_node.align(Alignment::Center, Alignment::Center, size_of_this_node);
        layout::Node::with_children(size_of_this_node, vec![child_node])
    }

    fn children(&self) -> Vec<Tree> {
        vec![Tree::new(self.inner_widget.as_widget())]
    }
}

```

```

fn draw(
    &self,
    state: &Tree,
    renderer: &mut Renderer,
    theme: &Theme,
    style: &renderer::Style,
    layout: Layout<'_>,
    cursor: mouse::Cursor,
    viewport: &Rectangle,
) {
    renderer.fill_quad(
        Quad {
            bounds: layout.bounds(),
            border: Border {
                color: Color::from_rgb(0.6, 0.93, 1.0),
                width: 1.0,
                radius: 10.0.into(),
            },
            shadow: Shadow::default(),
        },
        Color::from_rgb(0.0, 0.33, 0.4),
    );
}

self.inner_widget.as_widget().draw(
    &state.children[0],
    renderer,
    theme,
    style,
    layout.children().next().unwrap(),
    cursor,
    viewport,
);
}
}

impl<'a, Message, Renderer> From<MyWidgetOuter<'a, Message, Renderer>>
for Element<'a, Message, Theme, Renderer>
where
    Message: 'a,
    Renderer: iced::advanced::Renderer + 'a,
{
    fn from(widget: MyWidgetOuter<'a, Message, Renderer>) -> Self {
        Self::new(widget)
    }
}

:arrow_right: Next: Loading Images Asynchronously
:blue_book: Back: Table of contents

```

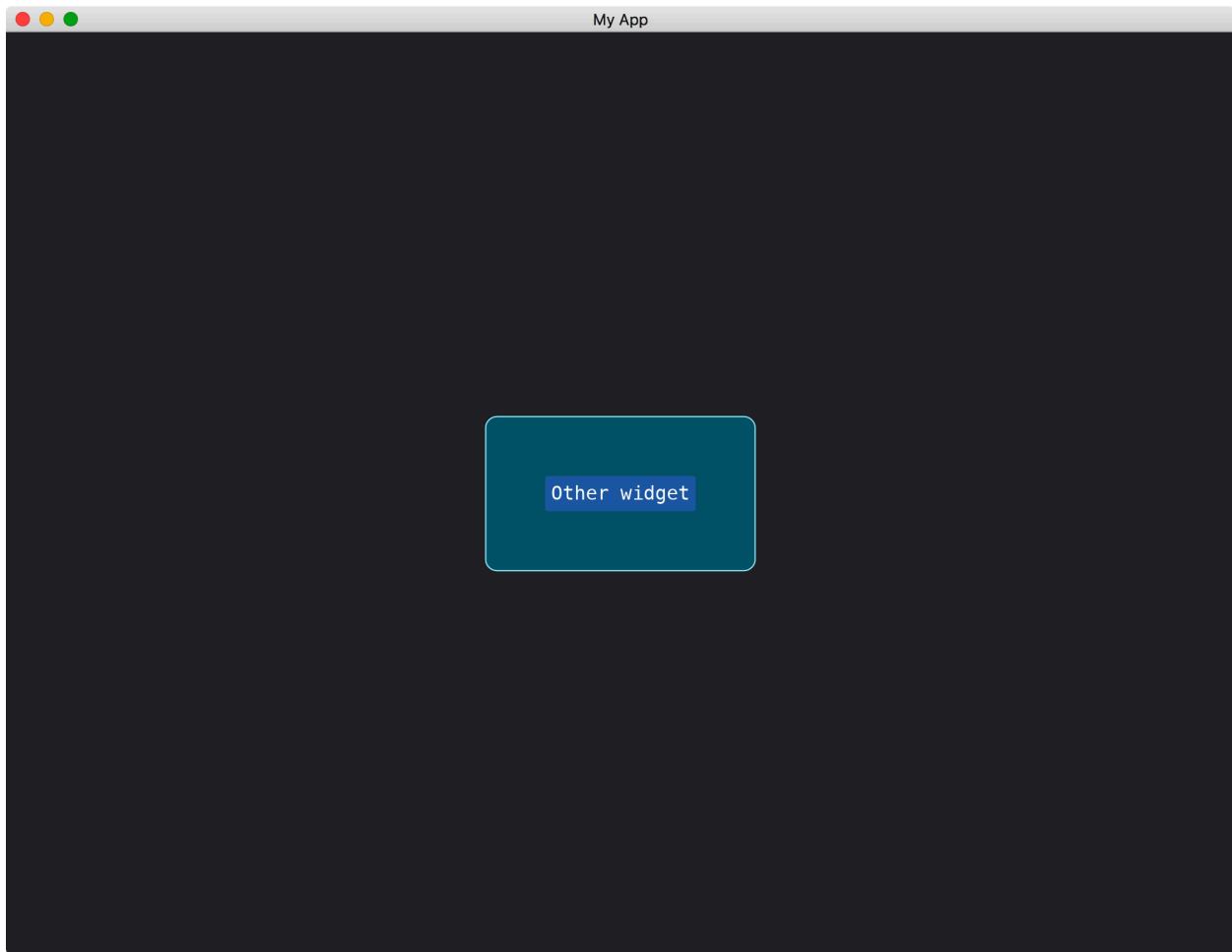


Figure 53: Taking Any Children

TextInput

The TextInput widget let users to input texts. It has two methods of constructions. If the on_input method is set, it is enabled, and is disabled otherwise. It supports reactions to pasting texts or keyboard submissions. It is able to change fonts and text sizes. We can add padding around the text inside. We can also add an optional icon.

```
use iced::{
    font::Family,
    widget::{
        column, text, text_input,
        text_input::{Icon, Side},
        TextInput,
    },
    Font, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    Update3(String),
    Update4(String),
    Update5(String),
    Paste5(String),
    Update6(String),
    Submit6,
    Update7(String),
    Update11(String),
}

#[derive(Default)]
struct MyApp {
    text3: String,
    text4: String,
    text5: String,
    info5: String,
```

```

    text6: String,
    info6: String,
    text7: String,
    text11: String,
}

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self::default()
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, message: Self::Message) {
        match message {
            MyAppMessage::Update3(s) => self.text3 = s,
            MyAppMessage::Update4(s) => self.text4 = s,
            MyAppMessage::Update5(s) => {
                self.text5 = s;
                self.info5 = "".into();
            }
            MyAppMessage::Paste5(s) => {
                self.text5 = s;
                self.info5 = "Pasted".into();
            }
            MyAppMessage::Update6(s) => {
                self.text6 = s;
                self.info6 = "".into();
            }
            MyAppMessage::Submit6 => self.info6 = "Submitted".into(),
            MyAppMessage::Update7(s) => self.text7 = s,
            MyAppMessage::Update11(s) => self.text11 = s,
        }
    }

    fn view(&self) -> iced::Element<Self::Message> {
        column![

            text_input("Construct from function", ""),
            TextInput::new("Construct from struct", ""),
            text_input("Enabled text input", self.text3.as_str())
                .on_input(|s| MyAppMessage::Update3(s)),
            text_input("Shorter on_input", self.text4.as_str()).on_input(MyAppMessage::Update4),
            text_input("Press Ctrl/Cmd + V", self.text5.as_str())
                .on_input(MyAppMessage::Update5)
        ]
    }
}

```

```

        .on_paste(MyAppMessage::Paste5),
        text(self.info5.as_str()),
        text_input("Press enter", self.text6.as_str())
            .on_input(MyAppMessage::Update6)
            .on_submit(MyAppMessage::Submit6),
        text(self.info6.as_str()),
        text_input("Password", self.text7.as_str())
            .secure(true)
            .on_input(MyAppMessage::Update7),
        text_input("Different font", "").font(Font {
            family: Family::Fantasy,
            ..Font::DEFAULT
        }),
        text_input("Larger text", "").size(24),
        text_input("With padding", "").padding(20),
        text_input("Icon", self.text11.as_str())
            .icon(Icon {
                font: Font::DEFAULT,
                code_point: '\u{2705}',
                size: None,
                spacing: 10.,
                side: Side::Left,
            })
            .on_input(MyAppMessage::Update11),
    ]
    .into()
}
}

```

:arrow_right: Next: Checkbox

:blue_book: Back: Table of contents

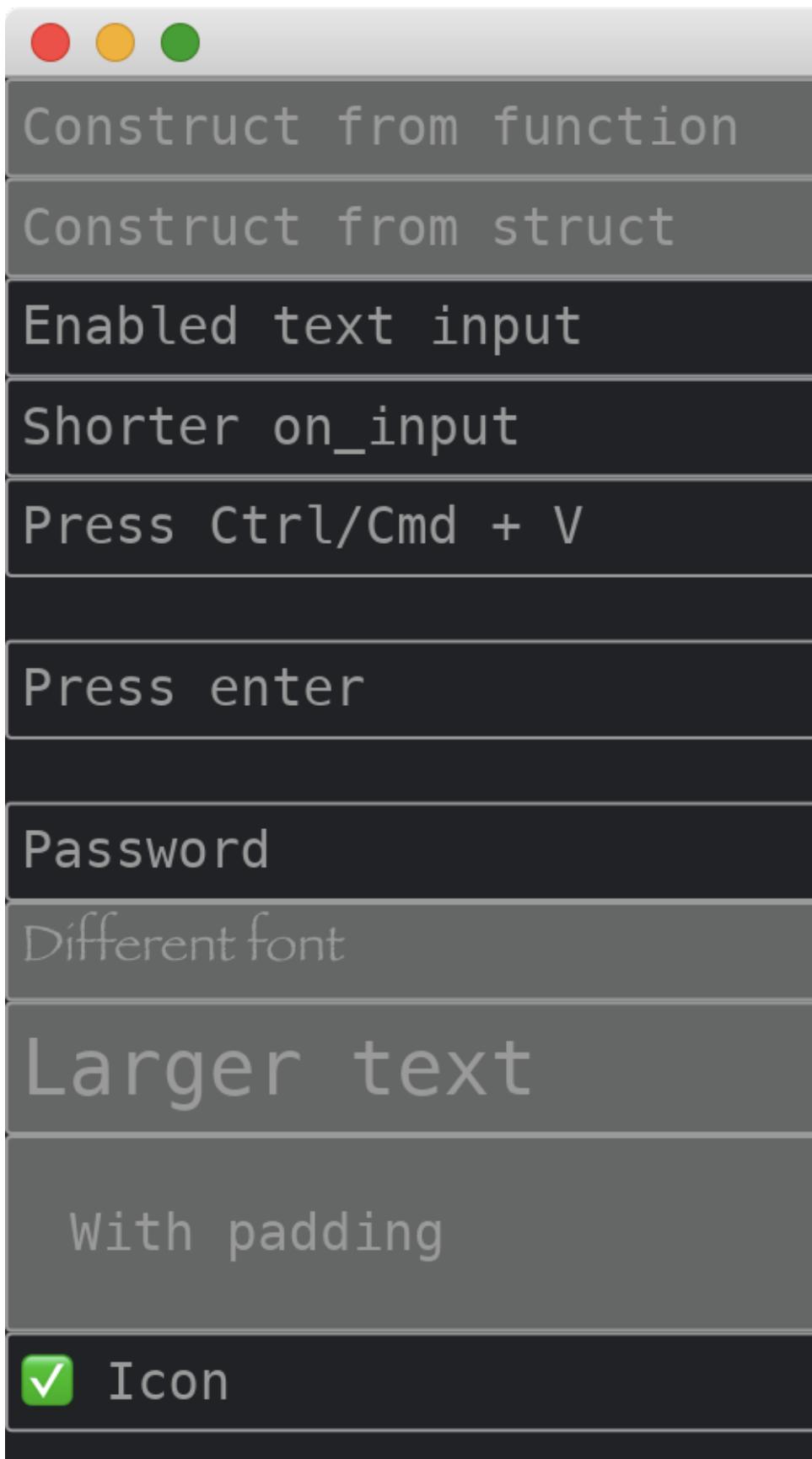


Figure 54: TextInput
160

Text

The Text widget is able to display texts. It has three methods of constructions. It is able to change the font, the size of the font, and display special characters. The text inside the widget can be horizontally or vertically centered.

```
use iced::{
    alignment::{Horizontal, Vertical},
    font::Family,
    widget::{column, text, text::Shaping, Text},
    Font, Length, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            "Construct from &str",
            text("Construct from function"),
            Text::new("Construct from struct"),
            text("Different font").font(Font {
                family: Family::Fantasy,
```

```

        ..Font:::DEFAULT
    }) ,
text("Larger text").size(24),
text("Special character ☺").shaping(Shaping::Advanced),
text("Center")
    .width(Length::Fill)
    .horizontal_alignment(Horizontal::Center),
text("Vertical center")
    .height(Length::Fill)
    .vertical_alignment(Vertical::Center),
]
.into()
}
}

```

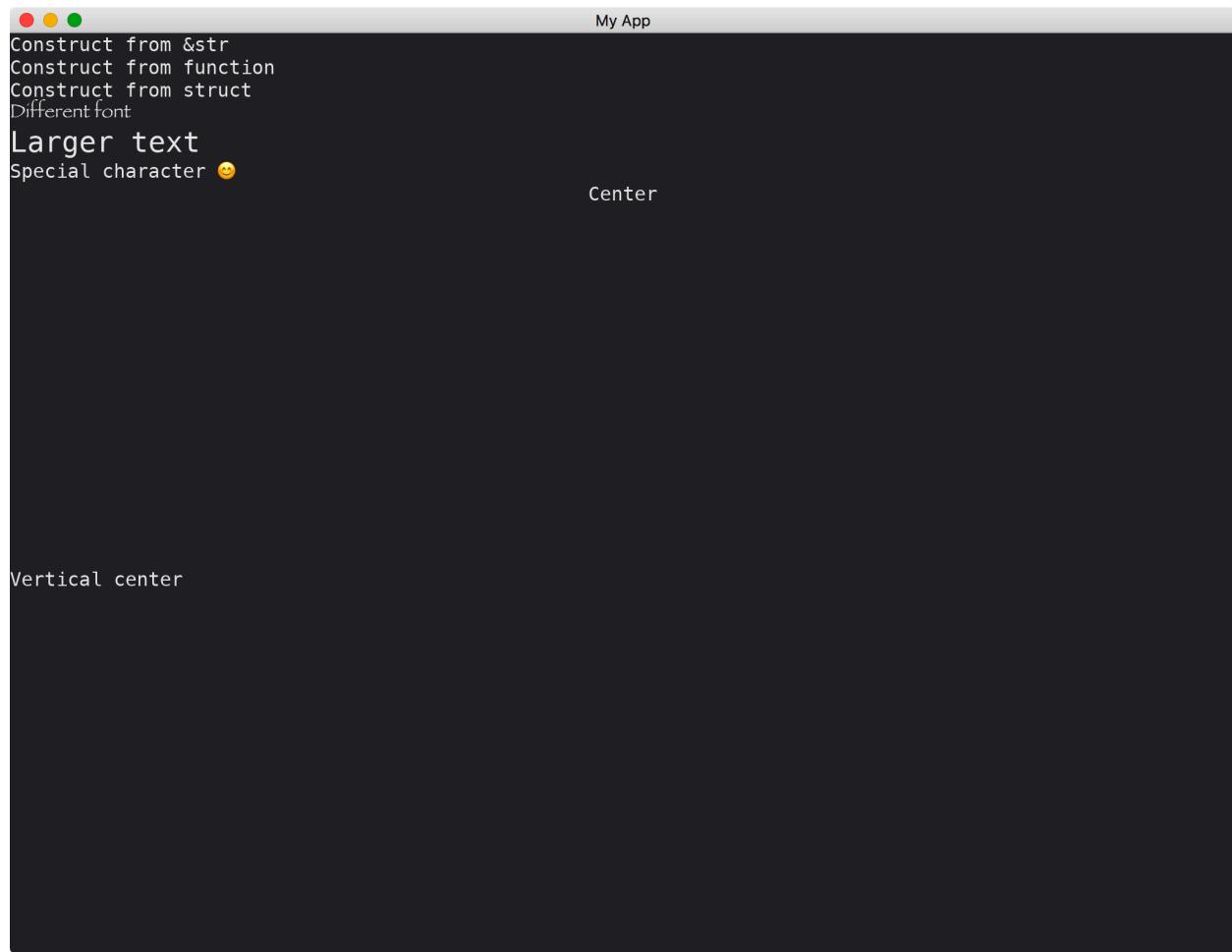


Figure 55: Text

:arrow_right: Next: Button

:blue_book: Back: Table of contents

Texts In Widgets

In addition to draw a Quad, we can also draw texts in our widgets.

For example, suppose we would like to draw a string slice named CONTENT.

```
struct MyWidgetWithText;

impl MyWidgetWithText {
    const CONTENT: &'static str = " My Widget ";

    fn new() -> Self {
        Self
    }
}
```

We use the Renderer's fill_text method to draw the text.

```
fn draw(
    &self,
    _state: &Tree,
    renderer: &mut Renderer,
    _theme: &Theme,
    _style: &renderer::Style,
    layout: Layout<'_>,
    _cursor: mouse::Cursor,
    viewport: &Rectangle,
) {
    // ...

    let bounds = layout.bounds();

    renderer.fill_text(
        Text {
            content: Self::CONTENT,
            bounds: bounds.size(),
            size: renderer.default_size(),
            line_height: LineHeight::default(),
            font: renderer.default_font(),
            horizontal_alignment: Horizontal::Center,
            vertical_alignment: Vertical::Center,
        }
    );
}
```

```

        shaping: Shaping::default(),
    },
    bounds.center(),
    Color::from_rgb(0.6, 0.8, 1.0),
    *viewport,
);
}
}

```

The `fill_text` method needs the `Renderer` type to implement `iced::advanced::text::Renderer`. Thus we have to require this in our `Widget` implementation.

```
impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidgetWithText
where
```

```
    Renderer: iced::advanced::Renderer + iced::advanced::text::Renderer,
```

Since the requirement of the `Renderer` type is changed, we have to change the requirement in `From<MyWidgetWithText>` for `Element`, too.

```
impl'a, Message, Renderer> From<MyWidgetWithText> for Element<'a, Message, Theme,
where
```

```
    Renderer: iced::advanced::Renderer + iced::advanced::text::Renderer,
```

The full code is as follows:

```

use iced::*;

advanced: {
    layout, mouse,
    renderer::{self}, Quad},
    widget::Tree,
    Layout, Text, Widget,
},
alignment:: {Horizontal, Vertical},
widget: {
    container,
    text:: {LineHeight, Shaping},
},
Border, Color, Element, Length, Rectangle, Sandbox, Settings, Shadow, Size, Theme
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }
}

```

```

fn title(&self) -> String {
    String::from("My App")
}

fn update(&mut self, _message: Self::Message) {}

fn view(&self) -> iced::Element<'_, Self::Message> {
    container(MyWidgetWithText::new())
        .width(Length::Fill)
        .height(Length::Fill)
        .center_x()
        .center_y()
        .into()
}
}

struct MyWidgetWithText;

impl MyWidgetWithText {
    const CONTENT: &'static str = "My Widget";

    fn new() -> Self {
        Self
    }
}

impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidgetWithText
where
    Renderer: iced::advanced::Renderer + iced::advanced::text::Renderer,
{
    fn size(&self) -> Size<Length> {
        Size {
            width: Length::Shrink,
            height: Length::Shrink,
        }
    }

    fn layout(
        &self,
        _tree: &mut Tree,
        _renderer: &Renderer,
        _limits: &layout::Limits,
    ) -> layout::Node {
        layout::Node::new([200, 100].into())
    }

    fn draw(

```

```

        &self,
        _state: &Tree,
        renderer: &mut Renderer,
        _theme: &Theme,
        _style: &renderer::Style,
        layout: Layout<'_>,
        _cursor: mouse::Cursor,
        viewport: &Rectangle,
    ) {
        renderer.fill_quad(
            Quad {
                bounds: layout.bounds(),
                border: Border {
                    color: Color::from_rgb(0.6, 0.8, 1.0),
                    width: 1.0,
                    radius: 10.0.into(),
                },
                shadow: Shadow::default(),
            },
            Color::from_rgb(0.0, 0.2, 0.4),
        );
    }

    let bounds = layout.bounds();

    renderer.fill_text(
        Text {
            content: Self::CONTENT,
            bounds: bounds.size(),
            size: renderer.default_size(),
            line_height: LineHeight::default(),
            font: renderer.default_font(),
            horizontal_alignment: Horizontal::Center,
            vertical_alignment: Vertical::Center,
            shaping: Shaping::default(),
        },
        bounds.center(),
        Color::from_rgb(0.6, 0.8, 1.0),
        *viewport,
    );
}
}

impl<'a, Message, Renderer> From<MyWidgetWithText> for Element<'a, Message, Theme, Renderer>
{
    fn from(widget: MyWidgetWithText) -> Self {
        Self::new(widget)
    }
}

```

```
    }  
}
```

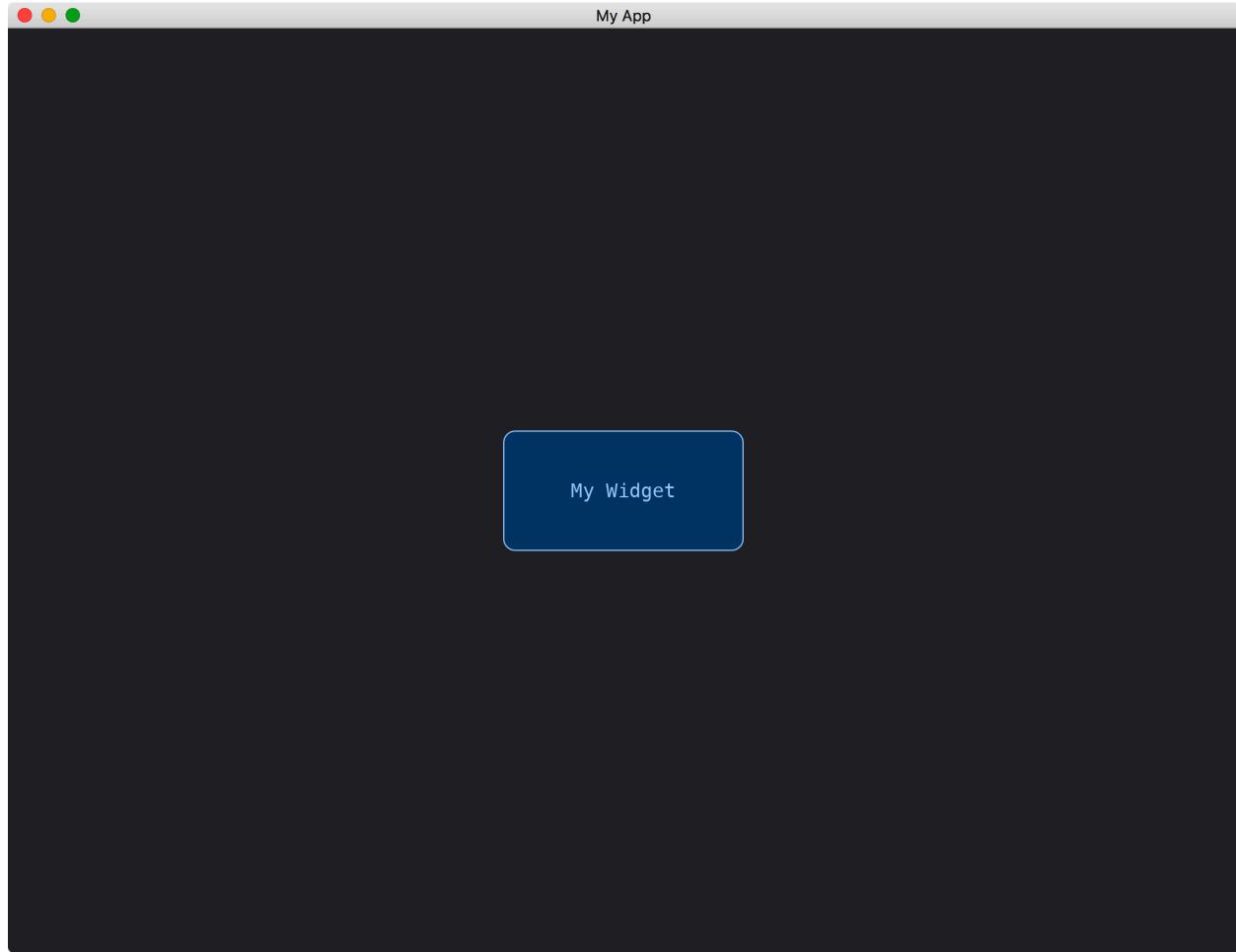


Figure 56: Texts In Widgets

:arrow_right: Next: Custom Background

:blue_book: Back: Table of contents

Toggler

The Toggler widget represents a boolean value. It has two methods of constructions. It supports reactions to clicking and touching. It is able to change styles of the button and the text and the space between them. It can also align its text.

```
use iced::{
    alignment::Horizontal,
    font::Family,
    widget::{column, text::Shaping, toggler, Toggler},
    Font, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyAppMessage {
    DoNothing,
    Update3(bool),
    Update4(bool),
}

#[derive(Default)]
struct MyApp {
    toggler3: bool,
    toggler4: bool,
}

impl Sandbox for MyApp {
    type Message = MyAppMessage;

    fn new() -> Self {
        Self::default()
    }

    fn title(&self) -> String {
        String::from("My App")
    }
}
```

```

}

fn update(&mut self, message: Self::Message) {
    match message {
        MyAppMessage::DoNothing => {}
        MyAppMessage::Update3(b) => self.toggler3 = b,
        MyAppMessage::Update4(b) => self.toggler4 = b,
    }
}

fn view(&self) -> iced::Element<Self::Message> {
    column![
        Toggler::new(Some("Construct from struct".into()), false, |_| {
            MyAppMessage::DoNothing
        }),
        toggler(Some("Construct from function".into()), false, |_| {
            MyAppMessage::DoNothing
        }),
        toggler(Some("Functional toggler".into()), self.toggler3, |b| {
            MyAppMessage::Update3(b)
        }),
        toggler(
            Some("Shorter parameter".into()),
            self.toggler4,
            MyAppMessage::Update4
        ),
        toggler(Some("Larger button".into()), false, |_| {
            MyAppMessage::DoNothing
        })
        .size(30),
        toggler(Some("Different font".into()), false, |_| {
            MyAppMessage::DoNothing
        })
        .font(Font {
            family: Family::Fantasy,
            ..Font::DEFAULT
        }),
        toggler(Some("Larger text".into()), false, |_| {
            MyAppMessage::DoNothing
        })
        .text_size(24),
        toggler(Some("Special character ☐".into()), false, |_| {
            MyAppMessage::DoNothing
        })
        .text_shaping(Shaping::Advanced),
        toggler(Some("Space between button and text".into()), false, |_| {
            MyAppMessage::DoNothing
        })
    ]
}

```

```

        .spacing(30),
        toggler(Some("Centered text".into()), false, |_| {
            MyAppMessage::DoNothing
        })
        .text_alignment(Horizontal::Center),
    ]
    .into()
}
}

```

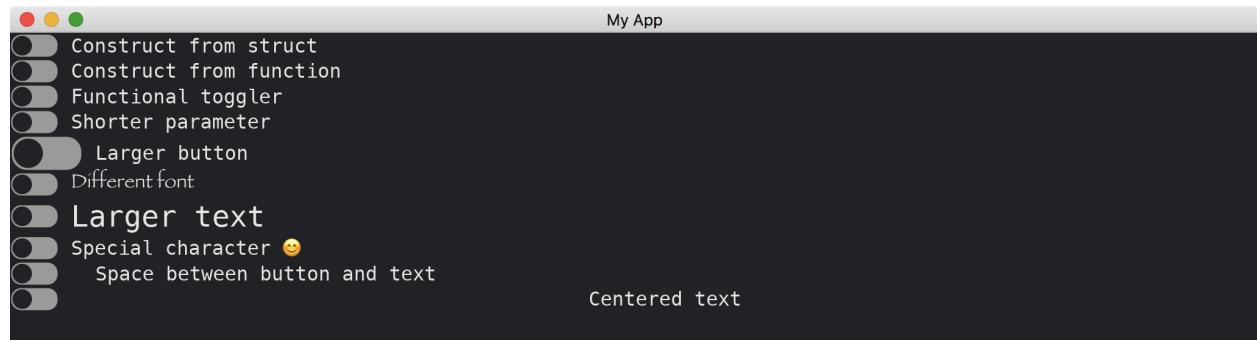


Figure 57: Toggler

:arrow_right: Next: Radio

:blue_book: Back: Table of contents

Tooltip

The Tooltip widget displays a text when the mouse is over a specified widget. It has two methods of constructions. It is able to change styles of the text. We can add padding around the text inside. We can also change the space between the tooltip and the target widget. If the tooltip is allowed to be out of the window, the parts outside are clipped.

```
use iced::{
    widget::{button, column, tooltip, tooltip::Position, Tooltip},
    Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            Tooltip::new(
                button("Mouseover to see the tooltip"),
                "Construct from struct",
                Position::Right
            ),
        ]
    }
}
```

```

        tooltip(
            button("Mouseover to see the tooltip"),
            "Construct from function",
            Position::Right
        ),
        tooltip(
            button("Mouseover to see the tooltip"),
            "With padding",
            Position::Right
        )
        .padding(20),
        tooltip(
            button("Mouseover to see the tooltip"),
            "Far away from the widget",
            Position::Right
        )
        .gap(50),
        tooltip(
            button("Mouseover to see the tooltip"),
            "Parts out of the window are clipped",
            Position::Right
        )
        .snap_within_viewport(false),
        tooltip(
            button("Mouseover to see the tooltip"),
            "Follow the cursor",
            Position::FollowCursor
        )
    ]
    .into()
}
}

```

:arrow_right: Next: Rule

:blue_book: Back: Table of contents

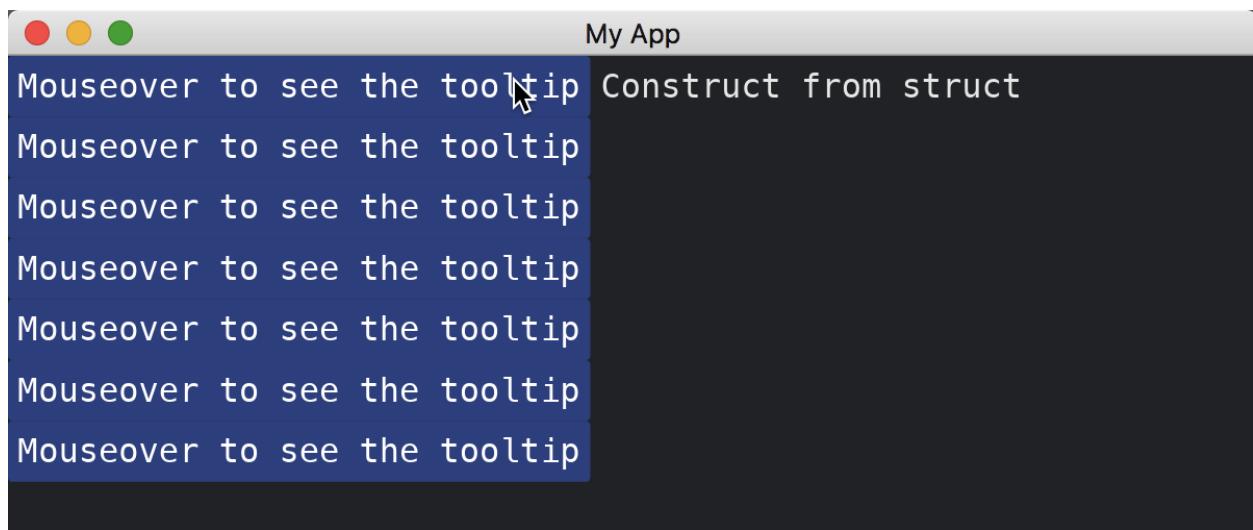


Figure 58: Tooltip

Updating Widgets From Events

Sometimes, we would like our widgets to handle their states by themselves. For example, a widget might change its states when receiving an Event.

To do so, we implement the `on_event` method of Widget.

```
fn on_event(
    &mut self,
    _state: &mut Tree,
    event: Event,
    _layout: Layout<'_>,
    _cursor: mouse::Cursor,
    _renderer: &Renderer,
    _clipboard: &mut dyn Clipboard,
    _shell: &mut Shell<'_, Message>,
    _viewport: &Rectangle,
) -> event::Status {
    match event {
        Event::Keyboard(keyboard::Event::KeyPressed {
            key: keyboard::Key::Named(Named::Space),
            ..
        }) => {
            self.highlight = !self.highlight;
            event::Status::Captured
        }
        _ => event::Status::Ignored,
    }
}
```

Our widget changes its `highlight` field every time when the space bar is pressed.

If the Event passed to the `on_event` method is what the widget needs, we return `Status::Captured`. Otherwise, we return `Status::Ignored` to tell the system the event can be used by other widgets.

Since our widget maintains its own state, we do not need to pass the state from our app.

```
struct MyWidget {
    highlight: bool,
}
```

```

impl MyWidget {
    fn new() -> Self {
        Self { highlight: false }
    }
}

```

The full code is as follows:

```

use iced::{
    advanced::{
        graphics::core::event,
        layout, mouse,
        renderer::{self, Quad},
        widget::Tree,
        Clipboard, Layout, Shell, Widget,
    },
    keyboard::{self, key::Named},
    widget::container,
    Border, Color, Element, Event, Length, Rectangle, Sandbox, Settings, Shadow, S
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        container(MyWidget::new()
            .width(Length::Fill)
            .height(Length::Fill)
            .center_x()
            .center_y()
            .into())
    }
}

```

```

}

struct MyWidget {
    highlight: bool,
}

impl MyWidget {
    fn new() -> Self {
        Self { highlight: false }
    }
}

impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidget
where
    Renderer: iced::advanced::Renderer,
{
    fn size(&self) -> Size<Length> {
        Size {
            width: Length::Shrink,
            height: Length::Shrink,
        }
    }

    fn layout(
        &self,
        _tree: &mut Tree,
        _renderer: &Renderer,
        _limits: &layout::Limits,
    ) -> layout::Node {
        layout::Node::new([100, 100].into())
    }

    fn draw(
        &self,
        _state: &Tree,
        renderer: &mut Renderer,
        _theme: &Theme,
        _style: &renderer::Style,
        layout: Layout<'_>,
        _cursor: mouse::Cursor,
        _viewport: &Rectangle,
    ) {
        renderer.fill_quad(
            Quad {
                bounds: layout.bounds(),
                border: Border {
                    color: Color::from_rgb(0.6, 0.8, 1.0),
                    width: 1.0,
                }
            }
        )
    }
}

```

```

                radius: 10.0.into(),
            },
            shadow: Shadow::default(),
        },
        if self.highlight {
            Color::from_rgb(0.6, 0.8, 1.0)
        } else {
            Color::from_rgb(0.0, 0.2, 0.4)
        },
    ),
);
}

fn on_event(
    &mut self,
    _state: &mut Tree,
    event: Event,
    _layout: Layout<'_>,
    _cursor: mouse::Cursor,
    _renderer: &Renderer,
    _clipboard: &mut dyn Clipboard,
    _shell: &mut Shell<'_, Message>,
    _viewport: &Rectangle,
) -> event::Status {
    match event {
        Event::Keyboard(keyboard::Event::KeyPressed {
            key: keyboard::Key::Named(Named::Space),
            ..
        }) => {
            self.highlight = !self.highlight;
            event::Status::Captured
        }
        _ => event::Status::Ignored,
    }
}
}

impl<'a, Message, Renderer> From<MyWidget> for Element<'a, Message, Theme, Renderer>
where
    Renderer: iced::advanced::Renderer,
{
    fn from(widget: MyWidget) -> Self {
        Self::new(widget)
    }
}

```

When the space bar is pressed, the widget color switches between light and dark.

[:arrow_right:](#) Next: Producing Widget Messages

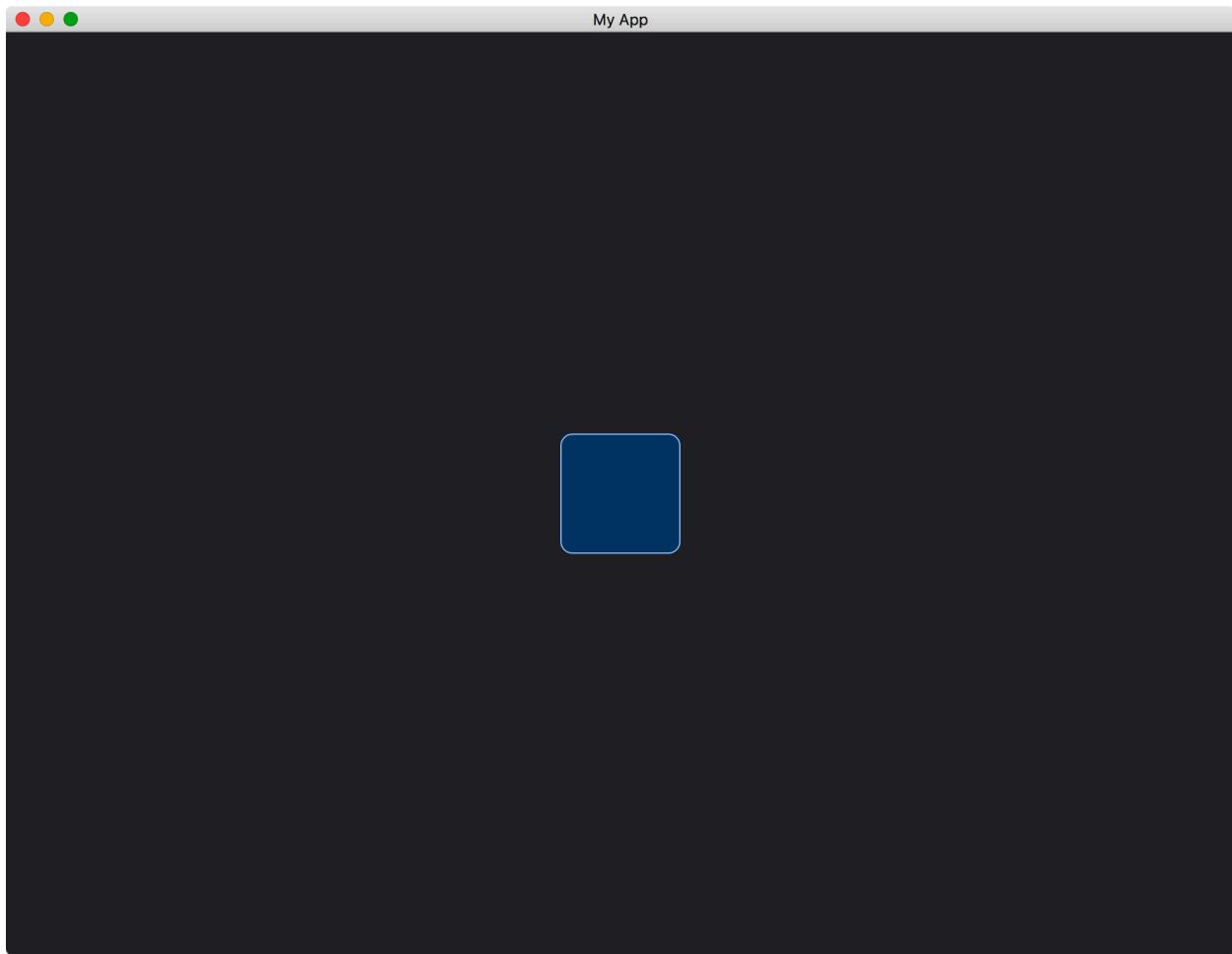


Figure 59: Updating Widgets From Events 1

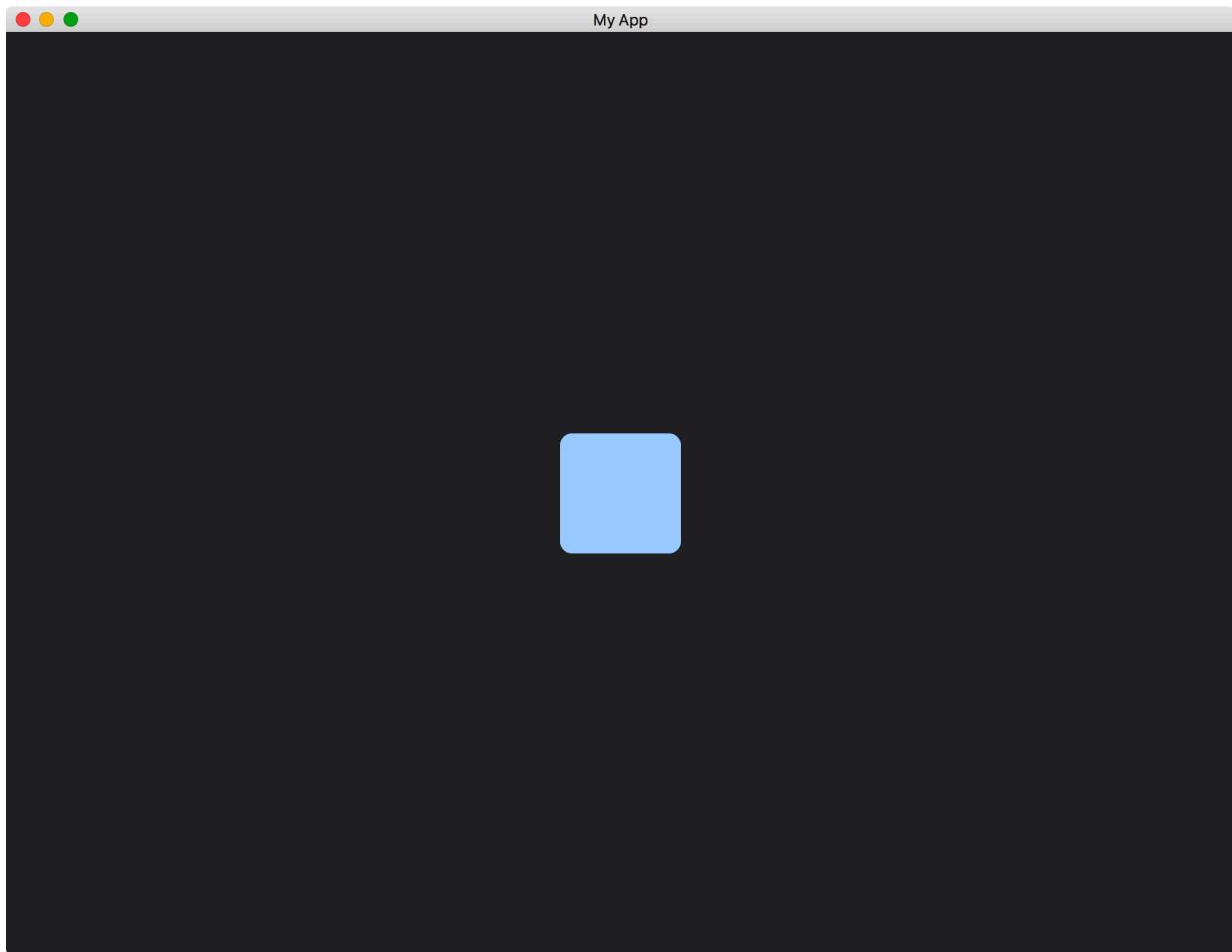


Figure 60: Updating Widgets From Events 2

:blue_book: Back: Table of contents

Updating Widgets From Outside

Consider that our widget has an internal state:

```
struct MyWidget {  
    highlight: bool,  
}
```

We use the `highlight` variable to change the color of our widget in the `draw` method.

```
fn draw(  
    &self,  
    _state: &Tree,  
    renderer: &mut Renderer,  
    _theme: &Theme,  
    _style: &renderer::Style,  
    layout: Layout<'_>,  
    _cursor: mouse::Cursor,  
    _viewport: &Rectangle,  
) {  
    renderer.fill_quad(  
        Quad {  
            bounds: layout.bounds(),  
            border: Border {  
                color: Color::from_rgb(0.6, 0.8, 1.0),  
                width: 1.0,  
                radius: 10.0.into(),  
            },  
            shadow: Shadow::default(),  
        },  
        if self.highlight {  
            Color::from_rgb(0.6, 0.8, 1.0)  
        } else {  
            Color::from_rgb(0.0, 0.2, 0.4)  
        },  
    );  
}
```

We would like to control the `highlight` variable from our app.

To do so, we make `MyWidget` to accept the `highlight` variable when the widget is

constructed.

```
impl MyWidget {
    fn new(highlight: bool) -> Self {
        Self { highlight }
    }
}
```

Then, we initialize MyWidget in the view method of Sandbox with an input value for the highlight variable.

```
struct MyApp {
    highlight: bool,
}

impl Sandbox for MyApp {
    // ...
    fn view(&self) -> iced::Element<'_, Self::Message> {
        container(
            column![
                MyWidget::new(self.highlight),
                // ...
            ]
            // ...
        )
        // ...
    }
}
```

In this example, we control the highlight variable by a checkbox.

The full code is as follows:

```
use iced::*;

advanced::{
    layout, mouse,
    renderer::{self, Quad},
    widget::Tree,
    Layout, Widget,
},
widget::{checkbox, column, container},
Border, Color, Element, Length, Rectangle, Sandbox, Settings, Shadow, Size, Theme
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

#[derive(Debug, Clone)]
enum MyMessage {
    Highlight(bool),
```

```

}

struct MyApp {
    highlight: bool,
}

impl Sandbox for MyApp {
    type Message = MyMessage;

    fn new() -> Self {
        Self { highlight: false }
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, message: Self::Message) {
        match message {
            MyMessage::Highlight(h) => self.highlight = h,
        }
    }

    fn view(&self) -> iced::Element<Self::Message> {
        container(
            column![
                MyWidget::new(self.highlight),
                checkbox("Highlight", self.highlight).on_toggle(MyMessage::Highlight)
            ]
            .spacing(20),
        )
        .width(Length::Fill)
        .height(Length::Fill)
        .center_x()
        .center_y()
        .into()
    }
}

struct MyWidget {
    highlight: bool,
}

impl MyWidget {
    fn new(highlight: bool) -> Self {
        Self { highlight }
    }
}

```

```

impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidget
where
    Renderer: iced::advanced::Renderer,
{
    fn size(&self) -> Size<Length> {
        Size {
            width: Length::Shrink,
            height: Length::Shrink,
        }
    }

    fn layout(
        &self,
        _tree: &mut Tree,
        _renderer: &Renderer,
        _limits: &layout::Limits,
    ) -> layout::Node {
        layout::Node::new([100, 100].into())
    }

    fn draw(
        &self,
        _state: &Tree,
        renderer: &mut Renderer,
        _theme: &Theme,
        _style: &renderer::Style,
        layout: Layout<'_>,
        _cursor: mouse::Cursor,
        _viewport: &Rectangle,
    ) {
        renderer.fill_quad(
            Quad {
                bounds: layout.bounds(),
                border: Border {
                    color: Color::from_rgb(0.6, 0.8, 1.0),
                    width: 1.0,
                    radius: 10.0.into(),
                },
                shadow: Shadow::default(),
            },
            if self.highlight {
                Color::from_rgb(0.6, 0.8, 1.0)
            } else {
                Color::from_rgb(0.0, 0.2, 0.4)
            },
        );
    }
}

```

```
}

impl<'a, Message, Renderer> From<MyWidget> for Element<'a, Message, Theme, Renderer>
where
    Renderer: iced::advanced::Renderer,
{
    fn from(widget: MyWidget) -> Self {
        Self::new(widget)
    }
}
```

When highlight is false:

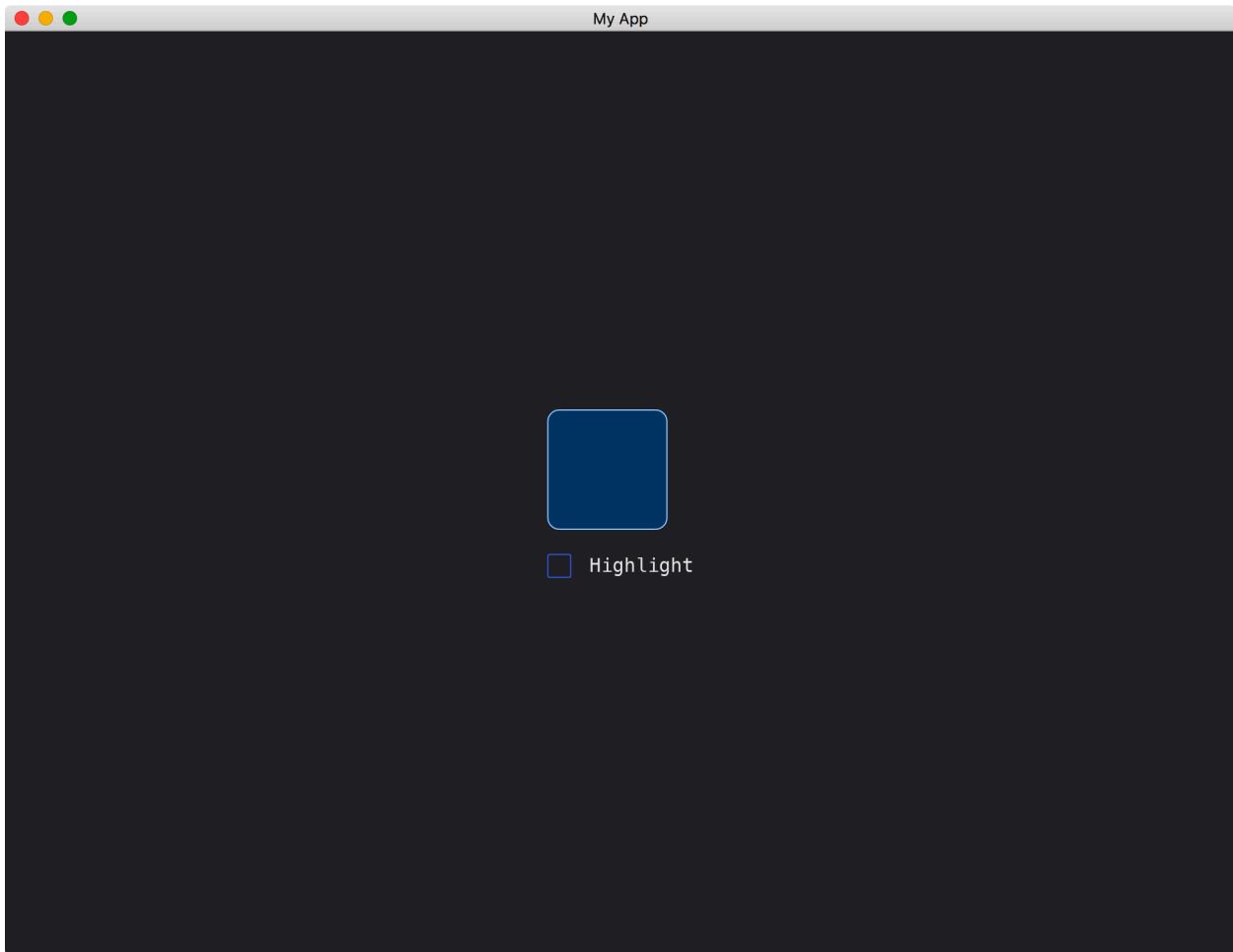


Figure 61: Updating Widgets From Outside 1

When highlight is true:

:arrow_right: Next: Updating Widgets From Events

:blue_book: Back: Table of contents

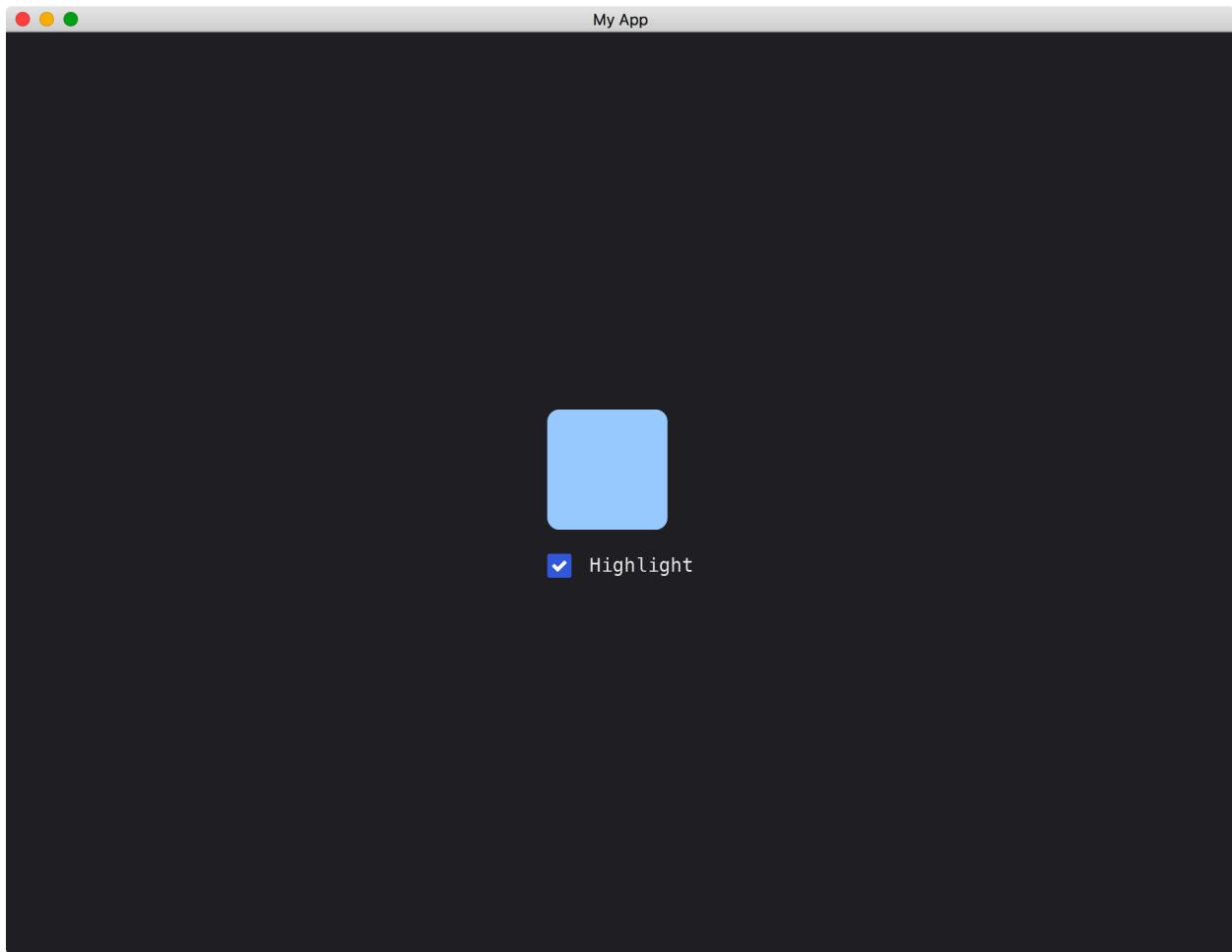


Figure 62: Updating Widgets From Outside 2

Widgets With Children

A custom widget can contain another custom widget.

In the following example, we have a MyWidgetOuter that contains a MyWidgetInner. The MyWidgetInner, which is the same as before, is a simple rectangle.

```
struct MyWidgetInner;

impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidgetInner
where
    Renderer: iced::advanced::Renderer,
{
    // ...
}

impl<'a, Message, Renderer> From<MyWidgetInner> for Element<'a, Message, Theme, Renderer>
where
    Renderer: iced::advanced::Renderer,
{
    // ...
}
```

The MyWidgetOuter contains MyWidgetInner.

```
struct MyWidgetOuter {
    inner_widget: MyWidgetInner,
}

impl MyWidgetOuter {
    fn new() -> Self {
        Self {
            inner_widget: MyWidgetInner,
        }
    }
}
```

In the draw method of MyWidgetOuter, we draw the MyWidgetInner as well.

```
fn draw(
    &self,
    state: &Tree,
```

```

        renderer: &mut Renderer,
        theme: &Theme,
        style: &renderer::Style,
        layout: Layout<'_>,
        cursor: mouse::Cursor,
        viewport: &Rectangle,
    ) {
        renderer.fill_quad(
            Quad {
                bounds: layout.bounds(),
                border_radius: 10.0.into(),
                border_width: 1.0,
                border_color: Color::from_rgb(0.6, 0.93, 1.0),
            },
            Color::from_rgb(0.0, 0.33, 0.4),
        );
    }

    let inner_widget = &self.inner_widget as &dyn Widget<Message, Renderer>;
    inner_widget.draw(
        state,
        renderer,
        theme,
        style,
        layout.children().next().unwrap(),
        cursor,
        viewport,
    );
}

```

When drawing the `MyWidgetInner` in `MyWidgetOuter`, we need to pass the layout of the `MyWidgetInner`. This layout information can be obtained by `layout.children().next().unwrap()`. `layout.children()` is an iterator that stores all the child layouts of `MyWidgetOuter`.

To make the underlying system aware of the child layouts of `MyWidgetOuter`, we have to explicitly tell the system in the `layout` method. Otherwise, the `layout.children()` in the `draw` method will be empty.

```

fn layout(&self, tree: &mut Tree, renderer: &Renderer, limits: &layout::Limits) ->
    let inner_widget = &self.inner_widget as &dyn Widget<Message, Renderer>;
    let mut child_node = inner_widget.layout(tree, renderer, limits);

    let size_of_this_node = Size::new(200., 200.);

    child_node = child_node.align(Alignment::Center, Alignment::Center, size_of_this_node);
}

```

We use the `Node::with_children` function to bind the parent layout and its child layouts.

The full code is as follows:

```
use iced::{
    advanced::{
        layout, mouse,
        renderer::{self, Quad},
        widget::Tree,
        Layout, Widget,
    },
    widget::{container, Theme},
    Alignment, Border, Color, Element, Length, Rectangle, Sandbox, Settings, Shadow
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        container(MyWidgetOuter::new())
            .width(Length::Fill)
            .height(Length::Fill)
            .center_x()
            .center_y()
            .into()
    }
}

struct MyWidgetInner;

impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidgetInner
where
    Renderer: iced::advanced::Renderer,
{
```

```

fn size(&self) -> Size<Length> {
    Size {
        width: Length::Shrink,
        height: Length::Shrink,
    }
}

fn layout(
    &self,
    _tree: &mut Tree,
    _renderer: &Renderer,
    _limits: &layout::Limits,
) -> layout::Node {
    layout::Node::new([100, 100].into())
}

fn draw(
    &self,
    _state: &Tree,
    renderer: &mut Renderer,
    _theme: &Theme,
    _style: &renderer::Style,
    layout: Layout<'_>,
    _cursor: mouse::Cursor,
    _viewport: &Rectangle,
) {
    renderer.fill_quad(
        Quad {
            bounds: layout.bounds(),
            border: Border {
                color: Color::from_rgb(0.6, 0.8, 1.0),
                width: 1.0,
                radius: 10.0.into(),
            },
            shadow: Shadow::default(),
        },
        Color::from_rgb(0.0, 0.2, 0.4),
    );
}
}

impl<'a, Message, Renderer> From<MyWidgetInner> for Element<'a, Message, Theme, Renderer>
where
    Renderer: iced::advanced::Renderer,
{
    fn from(widget: MyWidgetInner) -> Self {
        Self::new(widget)
    }
}

```

```

}

struct MyWidgetOuter {
    inner_widget: MyWidgetInner,
}

impl MyWidgetOuter {
    fn new() -> Self {
        Self {
            inner_widget: MyWidgetInner,
        }
    }
}

impl<Message, Renderer> Widget<Message, Theme, Renderer> for MyWidgetOuter
where
    Renderer: iced::advanced::Renderer,
{
    fn size(&self) -> Size<Length> {
        Size {
            width: Length::Shrink,
            height: Length::Shrink,
        }
    }

    fn layout(
        &self,
        tree: &mut Tree,
        renderer: &Renderer,
        limits: &layout::Limits,
    ) -> layout::Node {
        let inner_widget = &self.inner_widget as &dyn Widget<Message, Theme, Render
        let mut child_node = inner_widget.layout(tree, renderer, limits);

        let size_of_this_node = Size::new(200., 200.);

        child_node = child_node.align(Alignment::Center, Alignment::Center, size_o
            layout::Node::with_children(size_of_this_node, vec![child_node])
    }

    fn draw(
        &self,
        state: &Tree,
        renderer: &mut Renderer,
        theme: &Theme,
        style: &renderer::Style,
        layout: Layout<'_>,
    )
}

```

```

        cursor: mouse::Cursor,
        viewport: &Rectangle,
    ) {
        renderer.fill_quad(
            Quad {
                bounds: layout.bounds(),
                border: Border {
                    color: Color::from_rgb(0.6, 0.93, 1.0),
                    width: 1.0,
                    radius: 10.0.into(),
                },
                shadow: Shadow::default(),
            },
            Color::from_rgb(0.0, 0.33, 0.4),
        );
    }

    let inner_widget = &self.inner_widget as &dyn Widget<Message, Theme, Renderer>;
    inner_widget.draw(
        state,
        renderer,
        theme,
        style,
        layout.children().next().unwrap(),
        cursor,
        viewport,
    );
}
}

impl<'a, Message, Renderer> From<MyWidgetOuter> for Element<'a, Message, Theme, Renderer>
where
    Renderer: iced::advanced::Renderer,
{
    fn from(widget: MyWidgetOuter) -> Self {
        Self::new(widget)
    }
}

```

If `MyWidgetInner` receives events (i.e., implementing `on_event`), we have to call this `on_event` method from `MyWidgetOuter`'s `on_event` method. This ensures the Event is passing from `MyWidgetOuter` to `MyWidgetInner`.

```

fn on_event(
    &mut self,
    state: &mut Tree,
    event: Event,
    layout: Layout<'_>,
    cursor: mouse::Cursor,
    renderer: &Renderer,

```

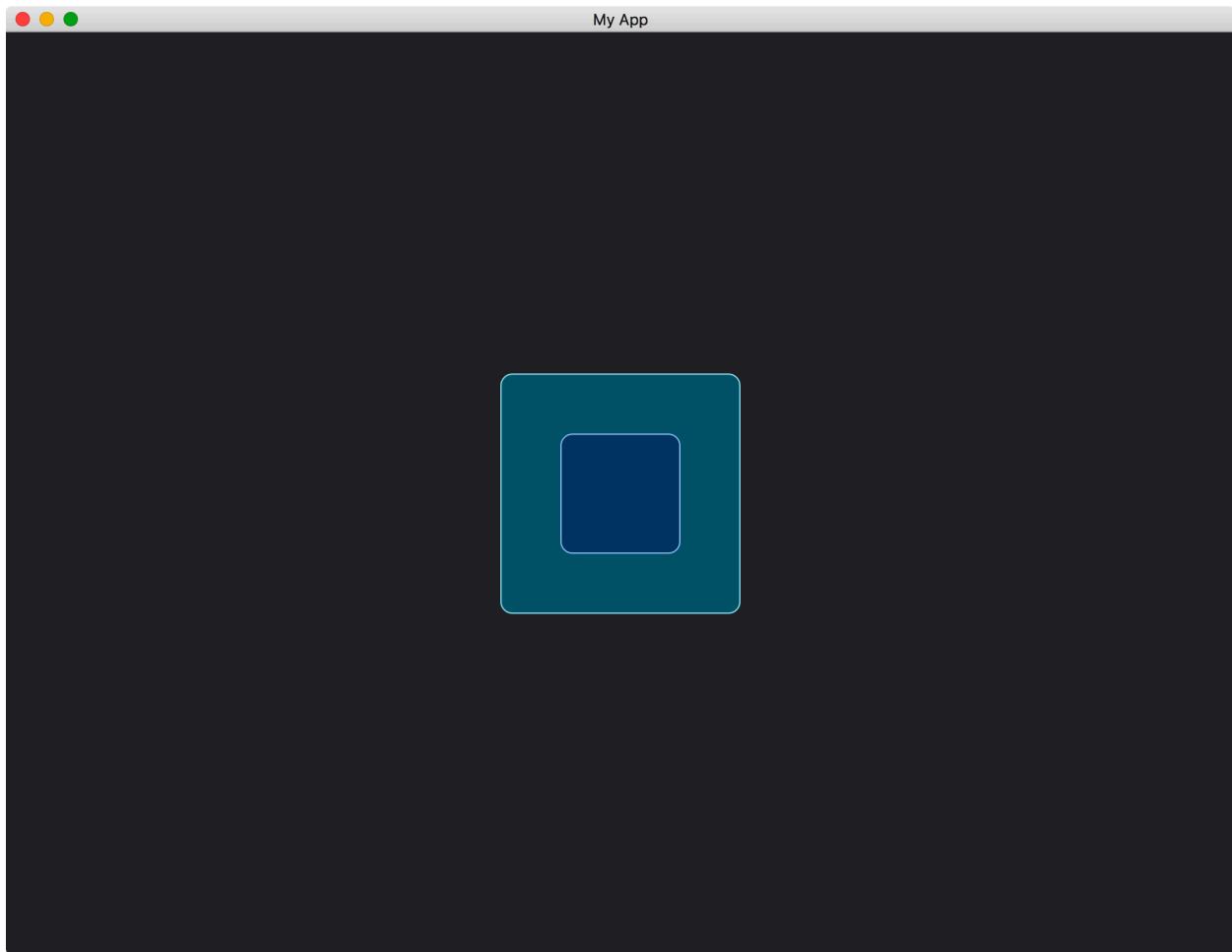


Figure 63: Widgets With Children

```
        clipboard: &mut dyn Clipboard,
        shell: &mut Shell<'_, Message>,
        viewport: &Rectangle,
    ) -> event::Status {
    let inner_widget = &mut self.inner_widget as &mut dyn Widget<Message, Theme, R>;
    inner_widget.on_event(
        state,
        event,
        layout.children().next().unwrap(),
        cursor,
        renderer,
        clipboard,
        shell,
        viewport,
    )
}
```

:arrow_right: Next: Taking Any Children

:blue_book: Back: Table of contents

Width And Height

Most widgets have the width and height methods to control their sizes. The methods accept a parameter Length. There are four types of Length:

- Shrink: occupy the least space.
- Fill: occupy all the rest of space.
- FillPortion: occupy the space relative to other widgets with FillPortion.
- Fixed: occupy a fixed space.

```
use iced::*;

widget::button, column, row},
Length, Sandbox, Settings,
};

fn main() -> iced::Result {
    MyApp::run(Settings::default())
}

struct MyApp;

impl Sandbox for MyApp {
    type Message = ();

    fn new() -> Self {
        Self
    }

    fn title(&self) -> String {
        String::from("My App")
    }

    fn update(&mut self, _message: Self::Message) {}

    fn view(&self) -> iced::Element<Self::Message> {
        column![
            button("Shrink").width(Length::Shrink),
            button("Fill").width(Length::Fill),
            row![
                button("FillPortion 2").width(Length::FillPortion(2)),
            ]
        ]
    }
}
```

```

        button("FillPortion 1").width(Length::FillPortion(1)),
    ]
    .spacing(10),
    button("Fixed").width(Length::Fixed(100.)),
    button("Fill (height)").height(Length::Fill),
]
.spacing(10)
.into()
}
}

```

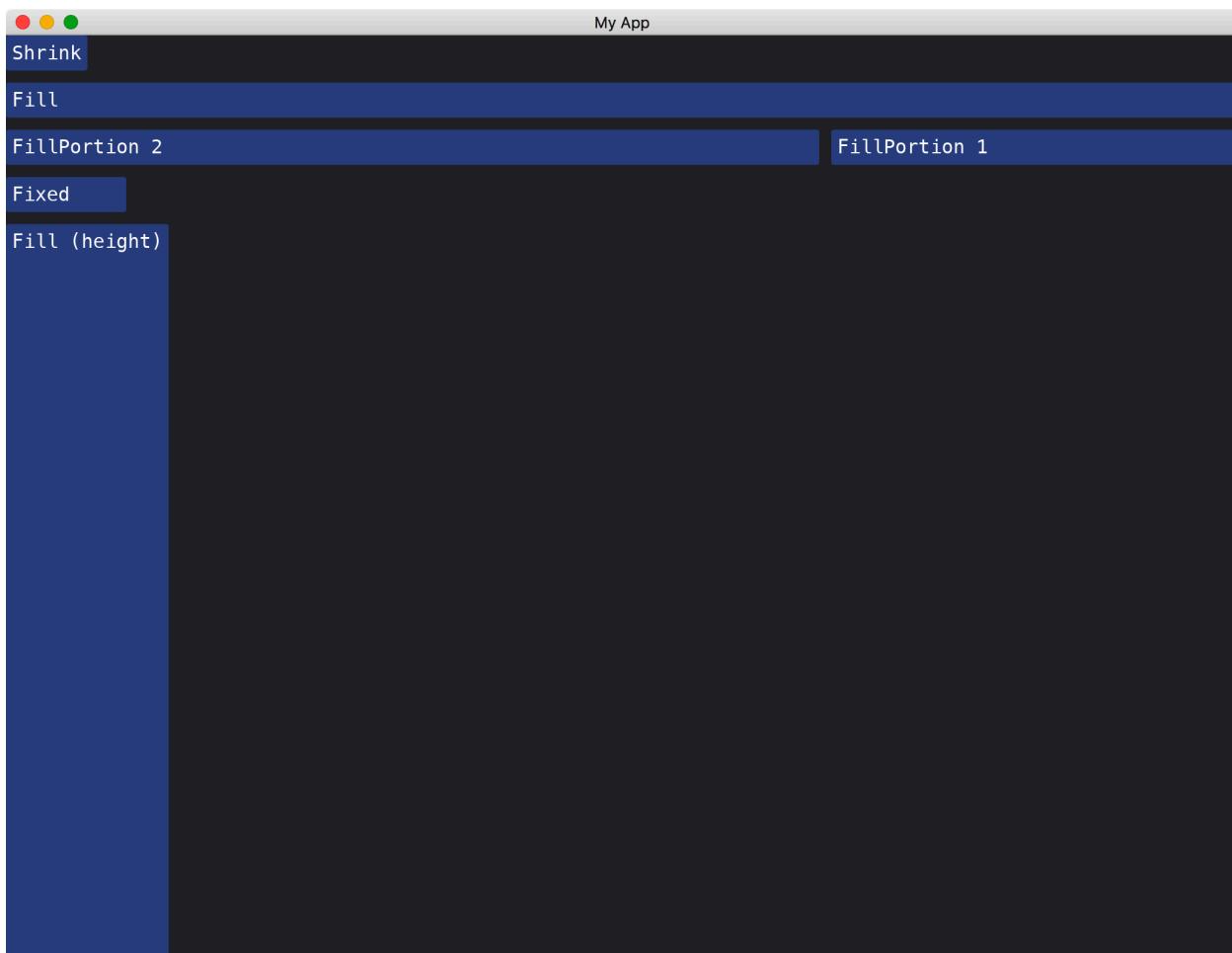


Figure 64: Width And Height

:arrow_right: Next: Column

:blue_book: Back: Table of contents