Introduction

iced is a cross-platform GUI library for Rust. It is inspired by Elm, a delightful functional language for building web applications.

As a GUI library, iced helps you build *graphical user interfaces* for your Rust applications.

iced is strongly focused on **simplicity** and **type-safety**. As a result, iced tries to provide simple building blocks that can be put together with strong typing to reduce the chance of **runtime errors**.

This book will:

- Introduce you to the fundamental ideas of iced.
- Teach you how to build interactive applications with iced.
- Emphasize principles to scale and grow iced applications.

Before proceeding, you should have some basic familiarity with Rust. If you are new to Rust or feel lost at some point, I recommend you to read the official Rust book.

Philosophy

iced is a *very* opinionated piece of software. It represents the culmination of almost 20 years of my personal battle against the eternal enemy: complexity[1].

Complexity is bad. It creeps in. Silently. Unnoticed. And then it kills everything. Slowly. I have seen it happen time and time again. Many of my codebases have died a slow and painful death in its hands.

Amidst rotten code, I learned to unlearn[2]. Each time, a little piece of a bigger puzzle was revealed. And one day, thanks to Elm, it all seemed to click into place.

I don't claim I have all the answers, but I rarely struggle with complexity nowadays. I have developed a certain intuition—a coding philosophy—that I follow instinctively. When I design libraries, I imbue as much of this learned philosophy into them.

iced is an attempt at creating a GUI toolkit that embodies my coding philosophy to the utmost extent. If these principles do not resonate with you, I recommend you to use a different library; otherwise, frustration will certainly ensue.

In this chapter I describe the most important (and controversial!) design principles that apply to iced, both technically and ideologically.

Open Source is Not About You

> Open source is a licensing and delivery mechanism, period. It means you get the source for software and the right to use and modify it. All social impositions associated with it, including the idea of 'community-driven-development' are part of a recently-invented mythology with little basis in how things actually work. A mythology that embodies, cult-like, both a lack of support for diversity in the ways things can work and a pervasive sense of communal entitlement.
>
> — Rich Hickey, Open Source is Not About You

iced is not your run-of-the-mill popular open source project. It is not owned by a corporation. It is not a brand. It is not a business. And most importantly, it is not a community effort.

iced is just my personal project. I work on it on my own terms, on my time, and give it all away for free as open source.

I do it because I enjoy building useful things **on my own**. I don't do it to meet new people. I don't do it to collaborate with others. I don't do it to make friends. It's great if any of that happens; but it's not the main reason I share my work.

This is fairly uncommon, so certain expectations that may be taken for granted in other open source projects do not make sense here. For instance:

- I do not try to be specially welcoming to new contributors.
- I do not care about promoting the library for widespread adoption.
- I do not keep a “professional” image at all times.
- I do not delegate work to land new features quickly.
- I do not hesitate to introduce breaking changes.

“My lord! Is that legal?”[3] You bet! So if you are coming here expecting a stable, professional, politically correct, and contributor-friendly open source project… You will likely be very disappointed.

But before you leave all flustered, give me a chance to tell you about the upsides of this philosophy of work. Yes, there are some:

- Instead of handholding new contributors, I have more time to spend doing what I do best: **coding**.
- Instead of trying to promote the library, I use a completely honest and direct approach of communication. **What you see is what you get.**
- Instead of being “professional”, I can be approachable, emotional, blunt, honest, controversial, and—most importantly—fun. I am a **human**, not a cold machine trying to keep appearances.
- Instead of delegating work, I build and review every single feature myself; potentially enabling a **high degree of cohesion** not possible in many other projects.
- Instead of maintaining a stable API, I can redesign freely and let the codebase **grow organically** as new solutions are discovered.

Pretty cool, huh? Everything has tradeoffs, and these are the ones that work for me and my projects.

With all that said, it would be reasonable for you to infer that iced must not have a great community of users. After all, I don’t actively promote the library nor welcome contributors. Why would anyone stick around? And yet… Plenty of people do![4]

In my experience, and quite ironically, this unusual approach to open source leads to the best kind of community. The extreme level of opinionation, honesty, and transparency lets users make a quick judgement, reduces frustration, and those that stay tend to share similar values.

Rust is All You Need

iced embraces Rust to its full extent. You will write *everything* in plain Rust.

There is no **D**omain-**S**pecific **L**anguage for defining view logic. No crazy procedural macro magic to make view code easier to write.

Macros, while a part of Rust, are a escape hatch. Creating a macro to meaningfully extend the language is equivalent to admitting that Rust by itself is not powerful enough. It’s the same as giving up. The language has failed.

I believe Rust itself is powerful and elegant enough to express user interface code in a beautiful way, and I am not willing to give up on it without putting up a fight first. If I wanted to code using a different language, then I would have chosen a different language.

Small declarative macros are fine, but only as long as they are very limited in scope and there is an analogous, widespread counterpart that can be considered a core part of the language. For instance, iced defines `row!` and `column!` macros with the same semantics as the ubiquitous `vec!` macro—they all build a collection of items in a certain order.

Long story short, iced expects you to write all your GUI code in Rust, leveraging variables, match statements, nested scopes, iterators, functions, and generics.

The Web is Madness

What once started as a simple invention for sharing hyper-text documents around the world has now evolved into an amalgamation of incohesive patches and standards; most shoe-horned in for the sake of velocity and backwards compatibility.

At the core of it all: a terrible crux. A seemingly benign separation of structure, style, and logic into three different languages: HTML, CSS, and JavaScript.

Except, it doesn’t work. Never did, and never will. Concerns constantly leak from one layer to the other with no consistent ethos. The three different parts need to know about each other *all the time*. And so, the job of a web developer consists in creating abstractions to cope and juggle these three different layers while trying to create an illusion of cohesion; like a magician in a circus.

This crux of separation is the main issue fueling most of the web frameworks ever made. Solutions for a made up problem introduced a long time ago. An opiate for a self-inflicted illness. The Web is built on top of a bunch of awful legacy.

I don’t know about you, but I have had enough of this circus.

iced completely and utterly rejects this separation—as it is simply non-sensical and insanity-inducing. Furthermore, it actively tries to stay away from anything that remotely resembles HTML, CSS, or JavaScript; for they are all incomplete solutions on their own.

When using iced, all of layout, styling, and logic are written directly in Rust as a cohesive whole—leveraging type safety and compiler guarantees as much as possible.

Code Must be Discovered

Coding is ultimately a process of **discovery**.

The problem is your starting point, the language is your map, the compiler is your compass, and the solution is your target.

Your job as a programmer is to figure out your starting point as accurately as possible, and then let the compass guide you.

Stating and defining the problem in your language is paramount. Any mistakes here will set you down the incorrect path—likely leading you to the wrong place after smashing into walls for a while.

In order to formalize the problem, it is important to distill it to its very essence. You must avoid mixing into it any pre-conceived notion of a potential solution you may have in your mind. Failing to do so will ruin your chances of achieving simplicity—which can only ever emerge organically.

iced is a specific set of solutions for problems that involve presenting interactive information to a user. As a consequence, the library tries to stay out of the way during the first stages of the discovery process—only chiming in once a solution has started to form—by decoupling itself from your types, data structures, and business logic.

Note From the Author

This chapter is not finished yet.

There are still some principles I would like to write about:

- Plumbing Code is Foundational
- Generic Problems are Illusions
- Purity Protects Sanity
- Be Competent or be Better

It takes inspiration to turn intuition into words, so I don’t know when I will get to it. Stay tuned!

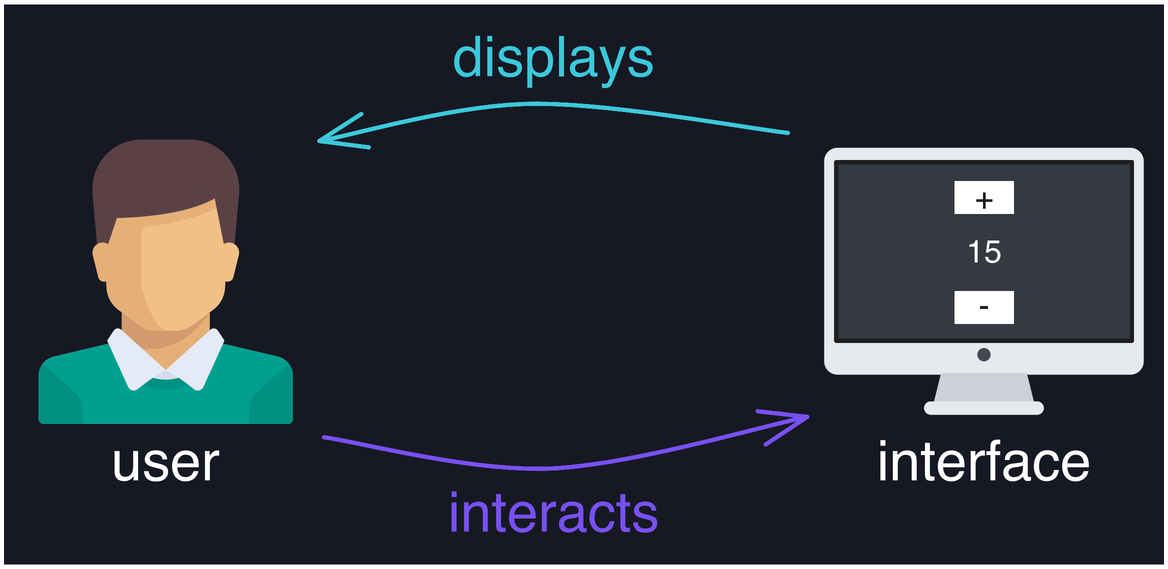
---

1. The Grug Brained Developer ↩
2. Learning to Unlearn ↩
3. I will make it legal ↩
4. Join us in the Discord server! ↩

Architecture

Let's start from the basics! You are probably very familiar with graphical user interfaces already. You can find them on your phone, computer, and most interactive electronic devices. In fact, you are most likely reading this book using one.

At their essence, graphical user interfaces are applications that **display** some information graphically to a user. This user can then choose to **interact** with the application—normally using some kind of device; like a keyboard, mouse, or touchscreen.

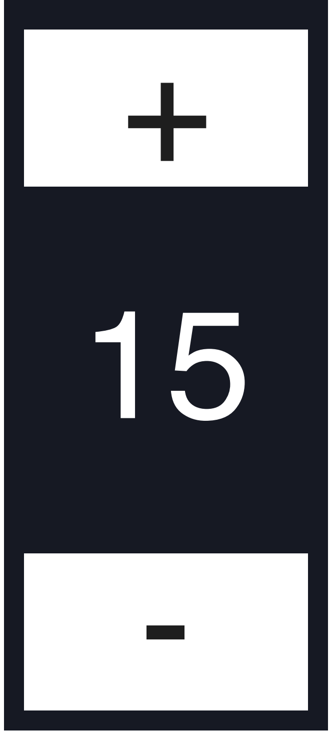


The user interactions may cause the application to update and display new information as a result, which in turn may cause further user interactions, which in turn cause further updates… And so on. This quick feedback loop is what causes the feeling of *interactivity*.

> Note: In this book, we will refer to graphical user interfaces as **GUIs**, **UIs**, **user interfaces**, or simply **interfaces**. Technically, not all interfaces are graphical nor user-oriented; but, given the context of this book, we will use all of these terms interchangeably.

Dissecting an Interface

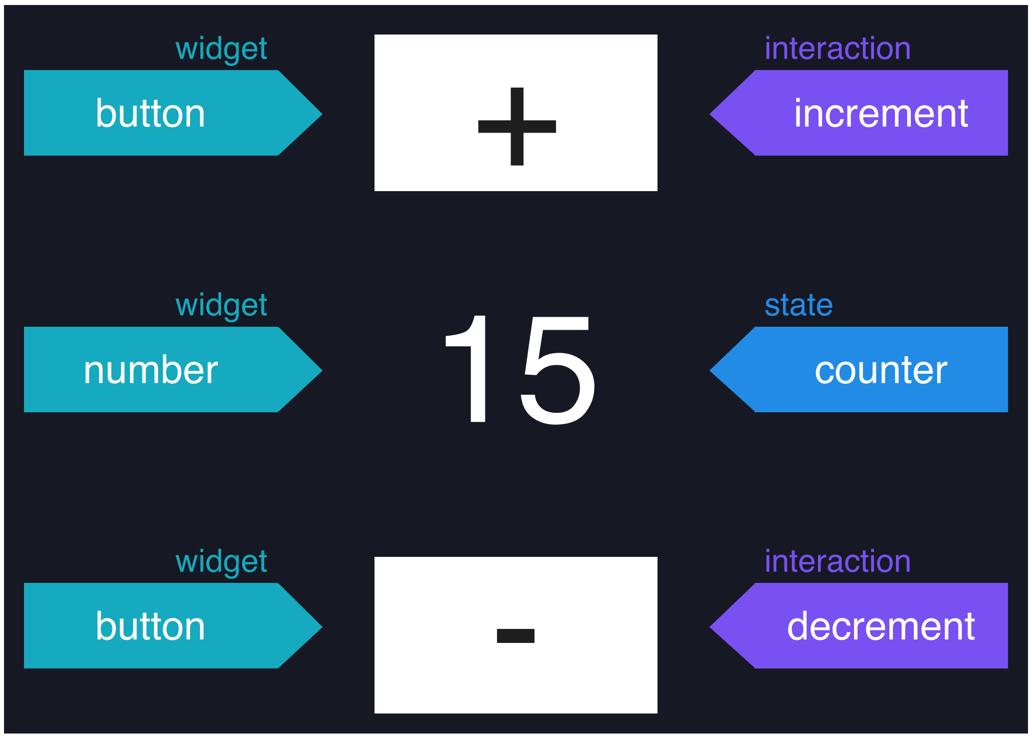
Since we are interested in creating user interfaces, let's take a closer look at them. We will start with a very simple one: the classical counter interface. What is it made of?



As we can clearly see, this interface has three visibly distinct elements: two buttons with a number in between. We refer to these visibly distinct elements of a user interface as **widgets** or **elements**.

Some **widgets** may be interactive, like a button. In the counter interface, the buttons can be used to trigger certain **interactions**. Specifically, the button at the top can be used to increment the counter value, while the button at the bottom can be used to decrement it.

We can also say that user interfaces are *stateful*—there is some **state** that persists between interactions. The counter interface displays a number representing the counter value. The number displayed will change depending on the amount of times we press the buttons. Pressing the increment button once will result in a different displayed value compared to pressing it twice.



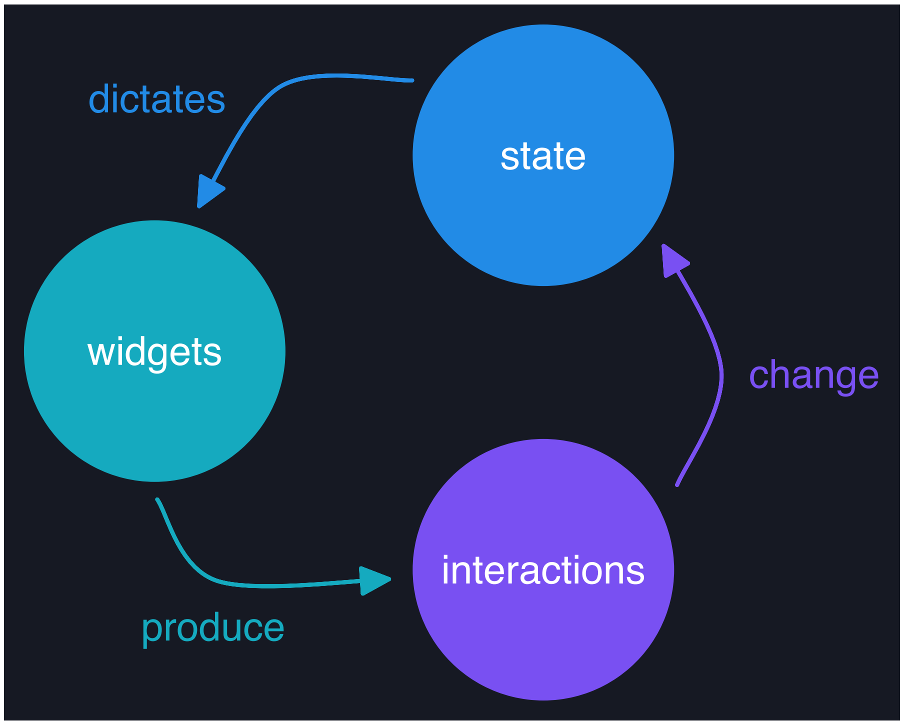
The GUI Trinity

Our quick dissection has successfully identified three foundational ideas in a user interface:

- **Widgets** — the distinct visual elements of an interface.
- **Interactions** — the actions that may be triggered by some widgets.
- **State** — the underlying condition or information of an interface.

These ideas are connected to each other, forming another feedback loop!

Widgets produce **interactions** when a user interacts with them. These **interactions** then change the **state** of the interface. The changed **state** propagates and dictates the new **widgets** that must be displayed. These new **widgets** may then produce new **interactions**, which can change the **state** again... And so on.



These ideas and their connections make up the fundamental architecture of a user interface. Therefore, creating a user interface must inevitably consist in defining these **widgets**, **interactions**, and **state**; as well as the connections between them.

Different Ideas, Different Nature

The three foundational ideas of an interface differ quite a bit when it comes to reusability.

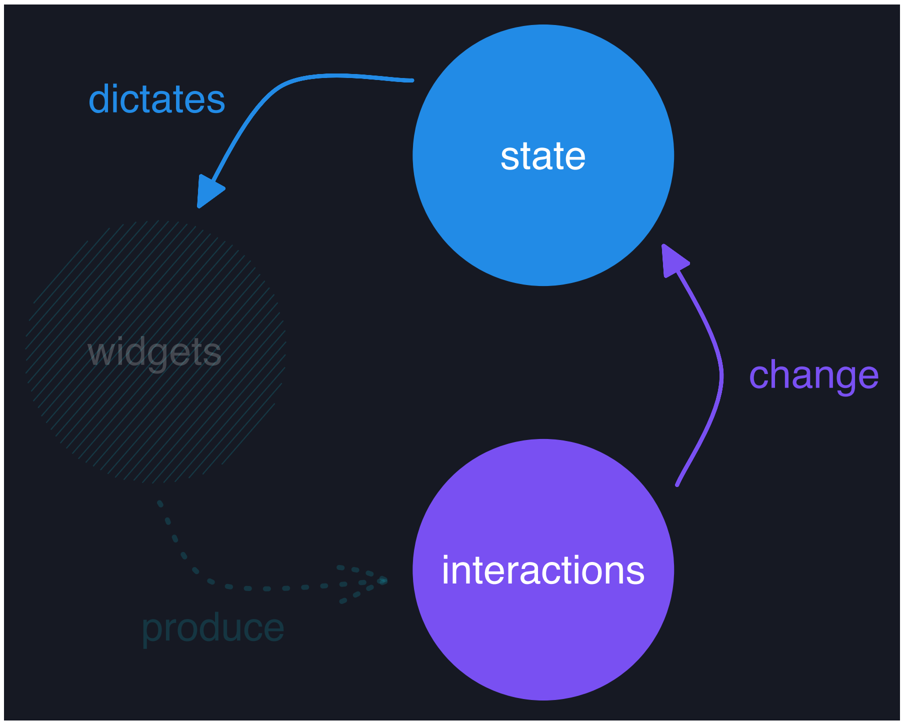
The state and the interactions of an interface are very specific to the application and its purpose. If I tell you that I have an interface with a numeric value and increment and decrement interactions, you will very easily guess I am talking about a counter interface.

However, if I tell you I have an interface with two buttons and a number… It's quite trickier for you to guess the kind of interface I am talking about. It could be anything!

This is because widgets are generally very generic and, therefore, more reusable. Most interfaces display a combination of familiar widgets—like buttons and numbers. In fact, users expect familiar widgets to always behave a certain way. If they don't behave properly, the interface will be unintuitive and have poor user experience.

While widgets are generally very reusable; the specific widget configuration dictated by the application state and its interactions is very application-specific. A button is generic; but a button that has a "+" label and causes a value increment when pressed is very specific.

All of this means that, when we are creating a specific user interface, we don't want to focus on implementing every familiar widget and its behavior. Instead, we want to leverage widgets as reusable building blocks—independent of our application and provided by some library—while placing our focus on the application-specific parts of the fundamental architecture: state, interactions, how the interactions change the state, and how the state dictates the widgets.



The Elm Architecture

It turns out that the four application-specific parts of the architecture of an interface are also the four foundational ideas of The Elm Architecture.

> The Elm Architecture is a pattern for architecting interactive programs that emerges naturally in Elm, a delightful purely functional programming language for reliable web applications.

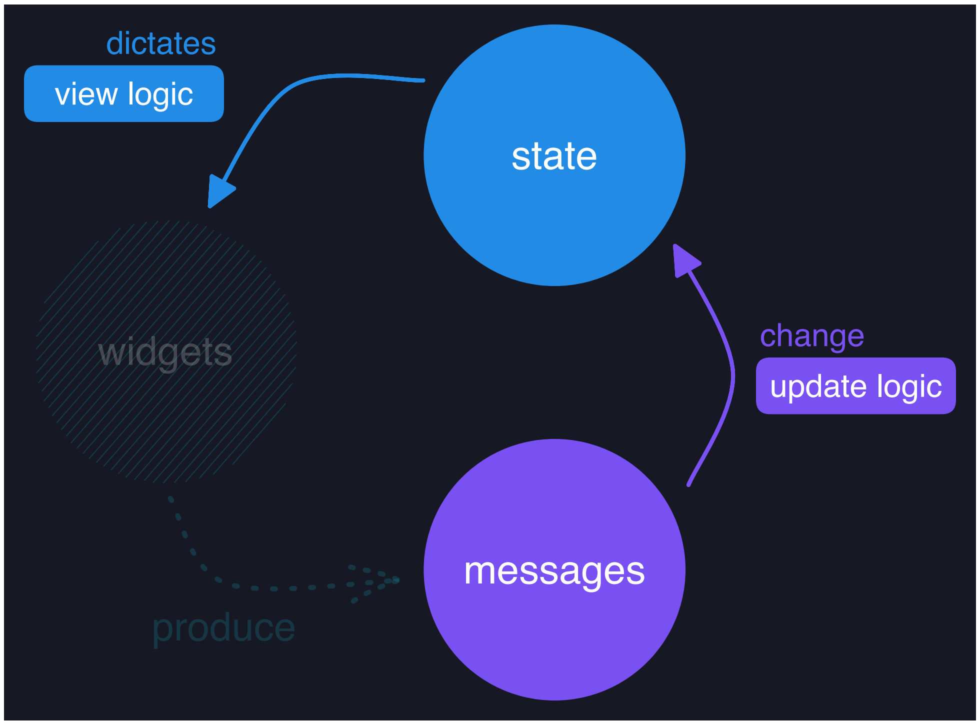
Patterns and ideas that emerge in purely functional programming languages tend to work very well in Rust because they leverage immutability and referential transparency—both very desirable properties that not only make code easy to reason about, but also play nicely with the borrow checker.

Furthermore, The Elm Architecture not only emerges naturally in Elm, but also when simply dissecting user interfaces and formalizing their inner workings; like we just did in this chapter.

The Elm Architecture uses a different—if not more precise—nomenclature for its fundamental parts:

- **Model** — the state of the application.
- **Messages** — the interactions of the application.
- **Update logic** — how the messages change the state.
- **View logic** — how the state dictates the widgets.

These are different names, but they point to the same exact fundamental ideas we have already discovered and, therefore, can be used interchangeably.



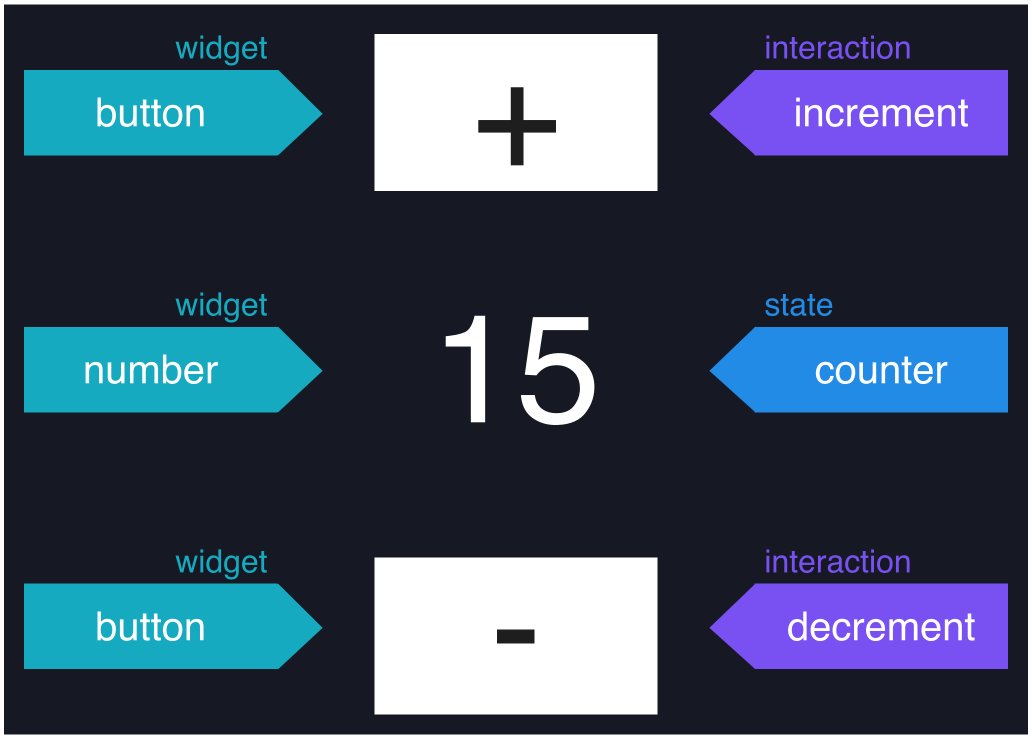
Note: In iced, the names **state** and **messages** are used more often than **model** and **interactions**, respectively.

First Steps

But enough with the theory. It's about time we start writing some code!

iced embraces The Elm Architecture as the most natural approach for architecting interactive applications. Therefore, when using iced, we will be dealing with the four main ideas we introduced in the previous chapter: **state**, **messages**, **update logic**, and **view logic**.

In the previous chapter, we dissected and studied the classical counter interface. Let's try to build it in Rust while leveraging The Elm Architecture.



State

Let's start with the **state**—the underlying data of the application.

In Rust, given the ownership and borrowing rules, it is extremely important to think carefully about the data model of your application.

> I encourage you to always start by pondering about the data of your application and its different states—not only those that are possible, but also those that must be impossible. Then try to leverage the type system as much as you can to *Make Impossible States Impossible*.

For our counter interface, all we need is a counter value. Since we have both increment and decrement interactions, the number could potentially be negative. This means we need a signed integer.

Also, we know some users are crazy and they may want to count a lot of things. Let's give them 64 bits to play with:

```rust
struct Counter {
    value: i64,
}
```

If a crazy user counted 1000 things every second, it would take them ~300 million years to run out of numbers. Let's hope that's enough.

Messages

Next, we need to define our **messages**—the interactions of the application.

Our counter interface has two interactions: **increment** and **decrement**. Technically, we could use a simple boolean to encode these interactions: `true` for increment and `false` for decrement, for instance.

But... we can do better in Rust! Interactions are mutually exclusive—when we have an interaction, what we really have is one value of a possible set of values. It turns out that Rust has the perfect data type for modeling this kind of idea: the *enum*.

Thus, we can define our messages like this:

```rust
enum Message {
    Increment,
    Decrement,
}
```

Simple enough! This also sets us up for the long-term. If we ever wanted to add additional interactions to our application—like a `Reset` interaction, for instance—we could just introduce additional variants to this type. Enums are very powerful and convenient.

Update Logic

Now, it's time for our **update logic**—how messages change the state of the application.

Basically, we need to write some logic that given any message can update any state of the application accordingly. The simplest and most idiomatic way to express this logic in Rust is by defining a method named `update` in our application state.

For our counter interface, we only need to properly increment or decrement the `value` of our `Counter` struct based on the `Message` we just defined:

```rust
impl Counter {
    fn update(&mut self, message: Message) {
        match message {
            Message::Increment => {
                self.value += 1;
            }
            Message::Decrement => {
                self.value -= 1;
            }
        }
    }
}
```

Great! Now we are ready to process user interactions. For instance, imagine we initialized our counter like this:

```rust
let mut counter = Counter { value: 0 };
```

And let's say we wanted to simulate a user playing with our interface for a bit—pressing the increment button twice and then the decrement button once. We could easily compute the final state of our counter with our **update logic**:

```rust
counter.update(Message::Increment);
counter.update(Message::Increment);
counter.update(Message::Decrement);
```

This would cause our `Counter` to end up with a `value` of `1`:

```rust
assert_eq!(counter.value, 1);
```

In fact, we have just written a simple test for our application logic:

```rust
#[test]
fn it_counts_properly() {
    let mut counter = Counter { value: 0 };

    counter.update(Message::Increment);
    counter.update(Message::Increment);
    counter.update(Message::Decrement);

    assert_eq!(counter.value, 1);
}
```

Notice how easy this was to write! So far, we are just leveraging very simple Rust concepts. No dependencies in sight! You may even be wondering... "Where is the GUI code?!"

This is one of the main advantages of The Elm Architecture. As we discovered in the previous chapter, widgets are the only fundamental idea of an interface that is reusable in nature. All the parts we have defined so far are application-specific and, therefore, do not need to know about the UI library at all!

The Elm Architecture properly embraces the different nature of each part of a user interface—decoupling **state**, **messages**, and **update logic** from **widgets** and **view logic**.

View Logic

Finally, the only part left for us to define is our **view logic**—how state dictates the widgets of the application.

Here is where the magic happens! In view logic, we bring together the state of the application and its possible interactions to produce a visual representation of the user interface that must be displayed to the user.

As we have already learned, this visual representation is made of widgets—the visibly distinct units of an interface. Most widgets are not application-specific and they can be abstracted and packaged into reusable libraries. These libraries are normally called *widget toolkits*, *GUI frameworks*, or simply *GUI libraries*.

And this is where **iced** comes in—finally! iced is a cross-platform GUI library for Rust. It packages a fair collection of ready-to-use widgets; buttons and numbers included. Exactly what we need for our counter.

The Buttons

Our counter interface has two **buttons**. Let's see how we can define them using iced.

In iced, widgets are independent values. The same way you can have an integer in a variable, you can have a widget as well. These values are normally created using a *helper function* from the `widget` module.

For our buttons, we can use the `button` helper:

```rust
use iced::widget::button;

let increment = button("+");
let decrement = button("-");
```

That's quite simple, isn't it? For now, we have just defined a couple of variables for our buttons.

As we can see, widget helpers may take arguments for configuring parts of the widgets to our liking. In this case, the `button` function takes a single argument used to describe the contents of the button.

The Number

We have our buttons sitting nicely in our `increment` and `decrement` variables. How about we do the same for our counter value?

While iced does not really have a `number` widget, it does have a more generic `text` widget that can be used to display any kind of text—numbers included:

```rust
use iced::widget::text;

let counter = text(15);
```

Sweet! Like `button`, `text` also takes an argument used to describe its contents. Since we are just getting started, let's simply hardcode `15` for now.

The Layout

Alright! We have our two buttons in `increment` and `decrement`, and our counter value in `counter`. That should be everything, right?

Not so fast! The widgets in our counter interface are displayed in a specific **order**. Given our three widgets, there is a total of **six** different ways to order them. However, the order we want is: `increment`, `counter`, and `decrement`.

A very simple way of describing this order is to create a list with our widgets:

```rust
let interface = vec![increment, counter, decrement];
```

But we are still missing something! It's not only the order that is specific, our interface also has a specific visual **layout**.

The widgets are positioned on top of each other, but they could very well be positioned from left to right instead. There is nothing in our description so far that talks about the **layout** of our widgets.

In iced, layout is described using... well, more widgets! That's right. Not all widgets produce visual results directly; some may simply manage the position of existing widgets. And since widgets are just values, they can be nested and composed nicely.

The kind of vertical layout that we need for our counter can be achieved with the `column` widget:

```rust
use iced::widget::column;

let interface = column![increment, counter, decrement];
```

This is very similar to our previous snippet. iced provides a `column!` macro for creating a `column` out of some widgets in a particular **order**—analogous to `vec!`.

The Interactions

At this point, we have in our `interface` variable a `column` representing our counter interface. But if we actually tried to run it, we would quickly find out that something is wrong.

Our buttons would be completely disabled. Of course! We have not defined any **interactions** for them. Notice that we have yet to use our `Message` enum in our view logic. How is our user interface supposed to produce **messages** if we don't specify them? Let's do that now.

In iced, every widget has a specific type that enables further configuration using simple builder methods. The `button` helper returns an instance of the `Button` type, which has an `on_press` method we can use to define the message it must **produce** when a user presses the button:

```rust
use iced::widget::button;

let increment = button("+").on_press(Message::Increment);
let decrement = button("-").on_press(Message::Decrement);
```

Awesome! Our interactions are wired up. But there is still a small detail left. A button can be pressed multiple times. Therefore, the same button may need to produce multiple instances of the same `Message`. As a result, we need our `Message` type to be cloneable.

We can easily *derive* the `Clone` trait—as well as `Debug` and `Copy` for good measure:

```rust
#[derive(Debug, Clone, Copy)]
enum Message {
    Increment,
    Decrement,
}
```

In The Elm Architecture, messages represent **events** that have occurred—made of pure data. As a consequence, it should always be easy to derive `Debug` and `Clone` for our `Message` type.

The View

We are almost there! There is only one thing left to do: connecting our application **state** to the view logic.

Let's bring together all the view logic we have written so far:

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

// The buttons
let increment = button("+").on_press(Message::Increment);
let decrement = button("-").on_press(Message::Decrement);

// The number
let counter = text(15);

// The layout
let interface = column![increment, counter, decrement];
```

If we ran this view logic, we would now be able to press the buttons. However, nothing would happen as a result. The counter would be stuck—always showing the number `15`. Our interface is completely stateless!

Obviously, the issue here is that our `counter` variable contains a text widget with a hardcoded `15`. Instead, what we want is to actually display the `value` field of our `Counter` state. This way, when a button is pressed and our update logic is triggered, the text widget will display the new `value`.

We can easily do this by running our view logic in a method of our `Counter`—just like we did with our update logic:

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

impl Counter {
    fn view(&self) {
        // The buttons
        let increment = button("+").on_press(Message::Increment);
        let decrement = button("-").on_press(Message::Decrement);

        // The number
        let counter = text(self.value);

        // The layout
        let interface = column![increment, counter, decrement];
    }
}
```

Our `counter` variable now will always have a `text` widget with the current `value` of our `Counter`. Great!

However, and as you may have noticed, this `view` method is completely useless—it constructs an `interface`, but then... It does nothing with it and throws it away!

> In iced, constructing and configuring widgets has no side effects. There is no “global context” you need to worry about in your view code.

Instead of throwing the `interface` away, we need to return it. Remember, the purpose of our **view logic** is to dictate the widgets of our user interface; and the content of the `interface` variable is precisely the description of the interface we want:

```rust
use iced::widget::{button, column, text, Column};

impl Counter {
    fn view(&self) -> Column<Message> {
        // The buttons
        let increment = button("+").on_press(Message::Increment);
        let decrement = button("-").on_press(Message::Decrement);

        // The number
        let counter = text(self.value);

        // The layout
        let interface = column![increment, counter, decrement];

        interface
    }
}
```

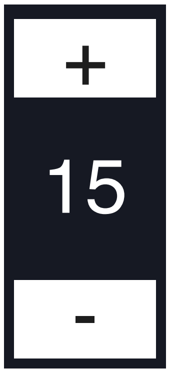
Tada! Notice how the `view` method needs a return type now. The returned type is `Column` because the `column!` macro produces a widget of this type—just like `button` produces a widget of the `Button` type.

You may also have noticed that this `Column` type has a generic type parameter. This type parameter simply specifies the type of messages the widget may produce. In this case, it takes our `Message` because the `increment` and `decrement` buttons inside the column produce messages of this type.

> iced has a strong focus on type safety—leveraging the type system and compile-time guarantees to minimize runtime errors as much as possible.

And well... That's it! Our view logic is done! But wait... It's a bit verbose right now. Since it's such a simple interface, let's just inline everything:

```rust
use iced::widget::{button, column, text, Column};

impl Counter {
    fn view(&self) -> Column<Message> {
        column![
            button("+").on_press(Message::Increment),
            text(self.value),
            button("-").on_press(Message::Decrement),
        ]
    }
}
```



That's much more concise. It even resembles the actual interface! Since creating widgets just yields values with no side effects; we can move things around in our view logic without worrying about breaking other stuff. No spooky action at a distance!

And that’s all there is to our counter interface. I am sure you can’t wait to **run** it. Shall we?

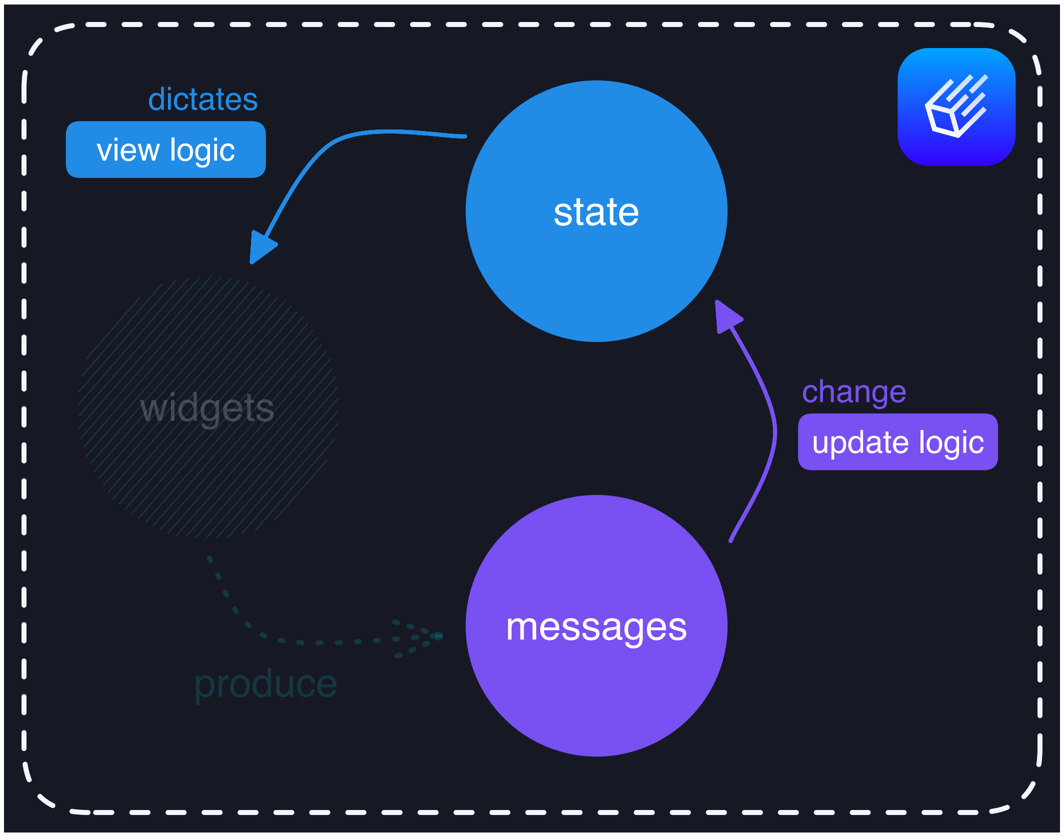
The Runtime

In the previous chapter we built the classical counter interface using iced and The Elm Architecture. We focused on each fundamental part—one at a time: **state**, **messages**, **update logic**, and **view logic**.

But now what? Yes, we have all the fundamental parts of a user interface—as we learned during our dissection—but it is unclear how we are supposed to bring it to life.

It seems we are missing *something* that can put all the parts together and *run* them in unison. *Something* that creates and runs the fundamental loop of a user interface—displaying widgets to a user and reacting to any interactions.

This *something* is called the **runtime**. You can think of it as the environment where the feedback loop of a user interface takes place. The runtime is in charge of every part of the loop: initializing the **state**, producing **messages**, executing the **update logic**, and running our **view logic**.



Another way to picture the runtime is by imagining a huge engine with four fundamental parts missing. Our job is to fill in these parts—and then the engine can run!

A Magical Runtime

Let's try to get a better understanding of the lifetime of an interface by exploring the internals of a basic (although very magical!) runtime.

In fact, we have actually started writing a runtime already! When we implemented the update logic of our counter, we wrote a very small test that simulated a user:

```rust
#[test]
fn it_counts_properly() {
    let mut counter = Counter { value: 0 };

    counter.update(Message::Increment);
    counter.update(Message::Increment);
    counter.update(Message::Decrement);

    assert_eq!(counter.value, 1);
}
```

This is technically a very bare-bones runtime. It initializes the **state**, produces some **interactions**, and executes the **update logic**.

Of course, the interactions are made up, it is very short-lived, and there is no **view logic** involved—far from what we actually want. Still, it's a great start! Let's try to extend it, step by step.

Initializing the State

Our small runtime is already initializing the application state properly:

```rust
// Initialize the state
let mut counter = Counter { value: 0 };
```

However, we can avoid hardcoding the initial state by leveraging the `Default` trait. Let's just derive it:

```rust
#[derive(Default)]
struct Counter {
    value: i64
}
```

And then, we simply use `Counter::default` in our runtime:

```rust
// Initialize the state
let mut counter = Counter::default();
```

The difference may be subtle, but we are separating concerns—we keep the initial state of our application close to the state definition and separated from the runtime. This way, we may eventually be able to make our runtime work with *any* application!

Displaying the Interface

Alright! We have our **state** initialized. What's next? Well, before a user can **interact** with our interface, we need to **display** it to them.

That's easy! We just need to open a window in whatever OS the user is running, initialize a proper graphics backend, and then render the widgets returned by our **view logic**—properly laid out, of course!

What? You have no clue of how to do that? Don't worry, I have this magical function: `display`. It takes a reference to any interface and displays it to the user. It totally works!

```rust
use magic::display;

// Run our view logic to obtain our interface
let interface = counter.view();

// Display the interface to the user
display(&interface);
```

See? Easy! Jokes aside, the purpose of this chapter is not for us to learn graphics programming; but for us to get a better understanding of how a runtime works. A little bit of magic doesn't hurt!

Gathering the Interactions

The user is seeing our interface and is now interacting with it. We need to pay very good attention to all the interactions and produce all the relevant **messages** that our widgets specify.

How? With some more magic, of course! I just found this `interact` function inside of my top hat—it takes an interface and produces the **messages** that correspond to the latest interactions of the user.

```rust
use magic::{display, interact};

// Process the user interactions and obtain our messages
let messages = interact(&interface);
```

Great! `interact` returns a list of **messages** for us—ready to be iterated.

Reacting to the Interactions

At this point, we have gathered the user interactions and we have turned them into a bunch of **messages**. In order to react properly to the user, we need to update our **state** accordingly for each message.

Luckily, there are no more magic tricks involved in this step—we can just use our **update logic**:

```
// Update our state by processing each message
for message in messages {
    counter.update(message);
}
```

That should keep our state completely up-to-date with the latest user interactions.

Looping Around

Okay! Our state has been updated to reflect the user interactions. Now, we need to display the resulting interface again to the user. And after that, we must process any further interactions... And then, update our state once more. And then... Do it all over once again!

This is a loop! And no, loops aren't very magical—not when we write Rust, at least:

```
use magic::{display, interact};

// Initialize the state
let mut counter = Counter::default();

// Be interactive. All the time!
loop {
    // Run our view logic to obtain our interface
    let interface = counter.view();

    // Display the interface to the user
    display(&interface);

    // Process the user interactions and obtain our messages
    let messages = interact(&interface);

    // Update our state by processing each message
    for message in messages {
        counter.update(message);
    }
}
```

Congratulations! We just wrote a perfectly functional runtime—magical properties aside. We can clearly understand here how each fundamental part of The Elm Architecture fits in the lifetime of an application.

Specifically,

- **state** is initialized once,
- **view logic** runs once at startup and then after every batch of interactions,
- and **update logic** runs for every interaction that created a **message**.

The Ice Wizard

“That’s cool and all”, you say, “but I am not a wizard and I still have no clue of how to run the counter interface I wrote. I have things to count!”

Fair enough! iced implements a very similar runtime to the one we just built. It comes bundled with its own magic[1]—so you don’t need to worry about learning the dark arts yourself.

If we want to run our `Counter`, all we have to do is call `run`:

```rust
pub fn main() -> iced::Result {
    iced::run(Counter::update, Counter::view)
}
```

We just provide the **update logic** and **view logic** to the **runtime**—which then figures out the rest!

The runtime is capable of inferring the types for the **state** and **messages** out of the type signatures of our **update logic** and **view logic**. The **state** is initialized leveraging `Default`, as we described earlier.

Notice also that `run` can fail and, therefore, it returns an `iced::Result`. If all we are doing is run the application, we can return this result directly in `main`.

And that should be it! Have fun counting things for 300 million years—at least!

Note From the Author

You reached the end of the book, for now!

I think it should already serve as a quick introduction to the basics of the library. There is a lot more to unravel—but hopefully you are now at a point where you can start playing around, having fun, and experimenting further.

The book is far from finished—there are a lot more topics I want to cover here, namely:

- Layout
- Styling
- Concurrency
- Scaling Applications
- Extending the Runtime
- And More!

Until I get to write them, check out the Additional Resources chapter if you want to explore and learn further.

I hope that you enjoyed the read so far. Stay tuned!

— Héctor

---

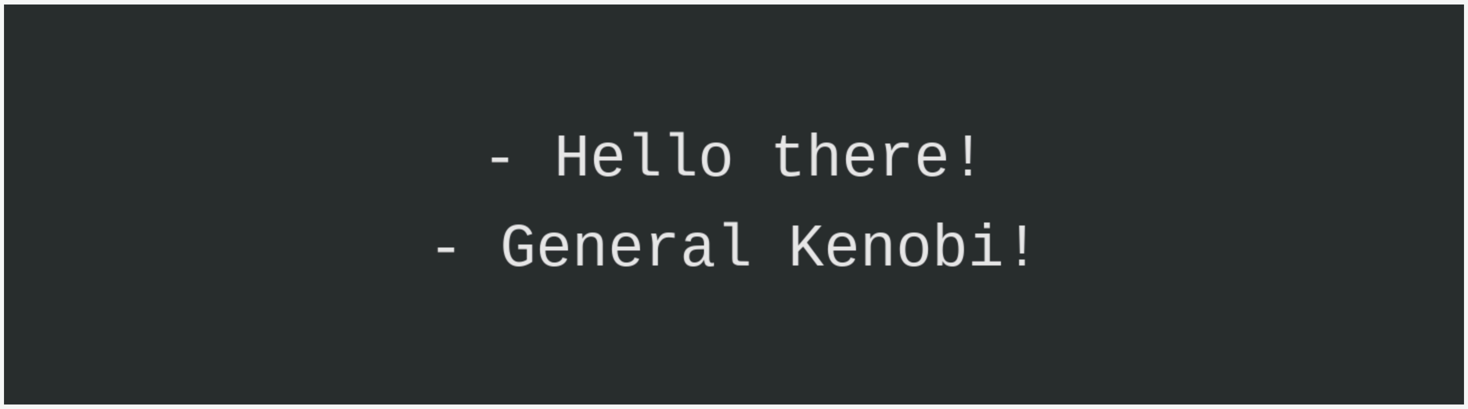
1. Mainly `winit`, `softbuffer`, `wgpu`, `tiny-skia`, and `cosmic-text`. ↩

Text

Text is likely the most essential widget of a graphical user interface.

There are multiple ways to display text in iced, but the most common approach is to use the `text` helper of the `widget` module to build an instance of the `Text` widget.

```rust
use iced::widget::text;
use iced::{Fill, Font};

text("- Hello there!\n- General Kenobi!")
    .font(Font::MONOSPACE)
    .size(30) // in logical pixels
    .line_height(1.5) // relative to the size (=45px)
    .width(Fill)
    .height(Fill)
    .center()
```



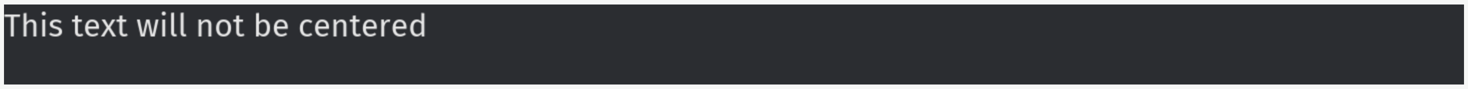
```
- Hello there!
- General Kenobi!
```

Alignment

A `Text` widget aligns its contents inside of its own bounds.

Since, by default, the `Text` widget uses an intrinsic sizing strategy, its bounds will match the content dimensions. Effectively, this means that trying to align text without altering the default sizing strategy will result in a no-op.

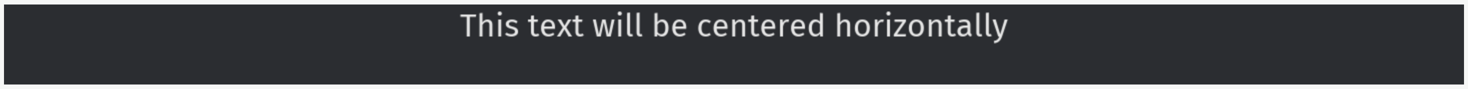
```rust
text("This text will not be centered").center()
```



This text will not be centered

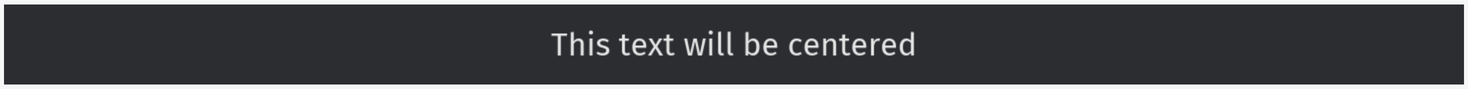
If we explicitly set the `width` to `Fill`, we will get the horizontal alignment we expect:

```rust
use iced::Fill;

text("This text will be centered horizontally")
    .width(Fill)
    .center()
```


This text will be centered horizontally

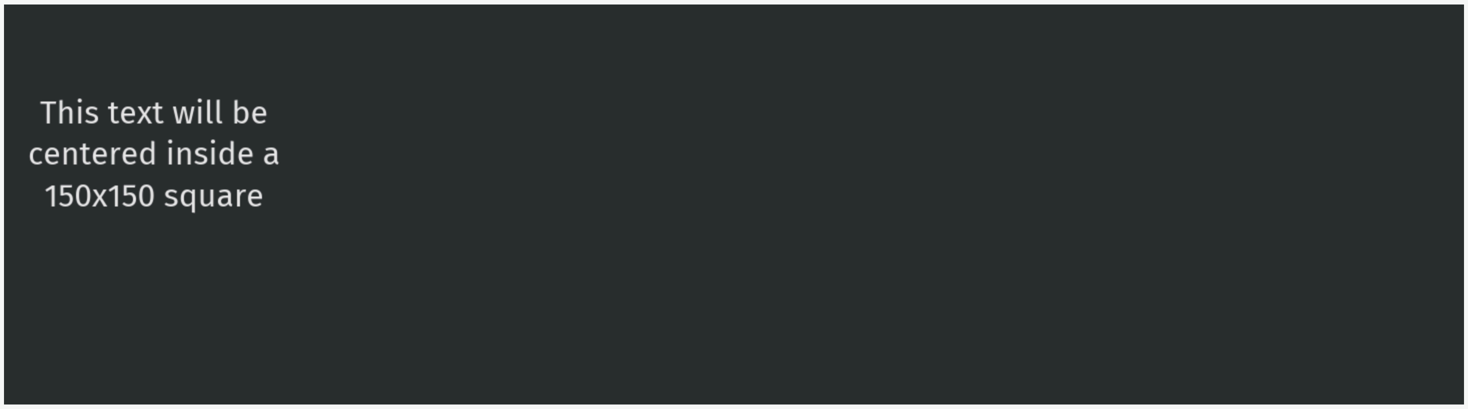
If we do the same for the `height`, we will then align in both axes:

```rust
use iced::Fill;

text("This text will be centered")
    .width(Fill)
    .height(Fill)
    .center()
```


This text will be centered

Of course, we can also align inside fixed dimensions:

```rust
text("This text will be centered inside a 150x150 square")
    .width(150)
    .height(150)
    .center()
```


This text will be
centered inside a
150x150 square

Styling

The different methods of the `Text` widget can be used to change its appearance—from the `font` used and its size to the `color` of the text.

The `style` method, however, lets you leverage the current `Theme` of the application to choose the color of the text:

```rust
use iced::Theme;

text("This is the primary color of the current theme!")
    .style(|theme: &Theme| text::Style {
        color: Some(theme.palette().primary),
    })
```

```
This is the primary color of the current theme!
```

For your convenience, the `widget::text` module has some built-in helpers you can directly provide to `style`:

```rust
text("And this is the warning color!")
    .style(text::warning)
```

```
And this is the warning color!
```

All of the built-in widgets follow this pattern. Keep it in mind!

The `text!` macro

Often, you will find yourself using `format!` to combine a dynamic value with some static text. For instance:

```rust
let name = "Héctor"; // Let's assume this is dynamic

text(format!("Hello, {name}!"))
```

```
Hello, Héctor!
```

The `text!` macro streamlines this use case. It behaves exactly the same as `format!`, but it just returns a `Text` widget instead of a `String`.

```rust
let name = "Héctor"; // Let's assume this is dynamic

text!("Hello, {name}!")
```

```
Hello, Héctor!
```

If you import `iced::widget::text`, you do not need to import the macro separately. Rust will bring you the `text` function helper, the `text` module, and the `text!` macro—all in a single import.

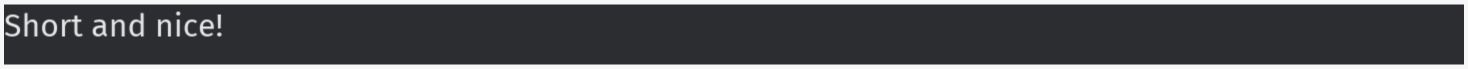
Quite nifty!

String Slices

The `Text` widget provides a `From<&str>` implementation for `Element`.

Since most function helpers for building widgets take an `Into<Element>`, this implementation allows you to skip calling `text()` completely if you have a string slice.

```rust
use iced::widget::container;

// We could just return:
// container(text("Redundant"))
//
// But the following is equivalent:
container("Short and nice!")
```



Short and nice!

This will only work if you want to use the default text appearance. If you need to customize it, you will have to use the `text` helper.

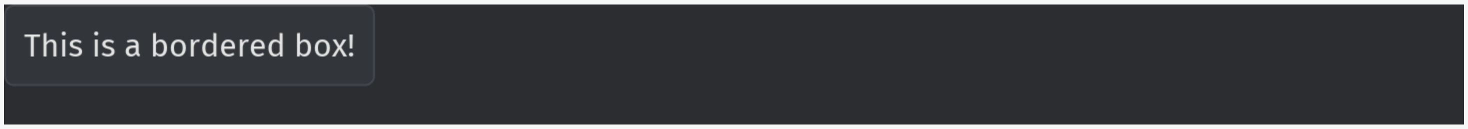
Container

The `Container` widget can apply padding, alignment, and styling to its contents.

Since it allows you to easily position and decorate existing widgets, it is considered one of the main building blocks in iced. You will need to use it often!

As usual, you can build one with the `container` helper found in the `widget` module.

```rust
use iced::widget::container;

container("This is a bordered box!")
    .padding(10)
    .style(container::bordered_box)
```

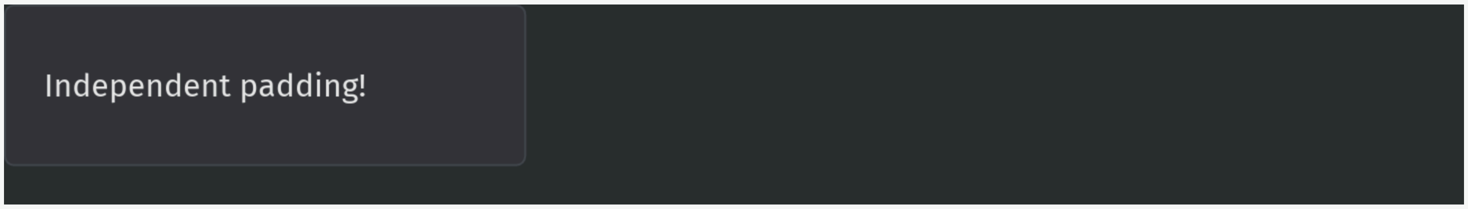

This is a bordered box!

Padding

You can leverage the `padding` method of a `Container` to apply some padding inside its bounds.

Each side can be configured independently. The `padding` module has a bunch of convenient helpers that make this easy.

```rust
use iced::padding;
use iced::widget::container;

container("Independent padding!")
    .padding(padding::vertical(30).left(20).right(80))
    .style(container::bordered_box)
```


Independent padding!

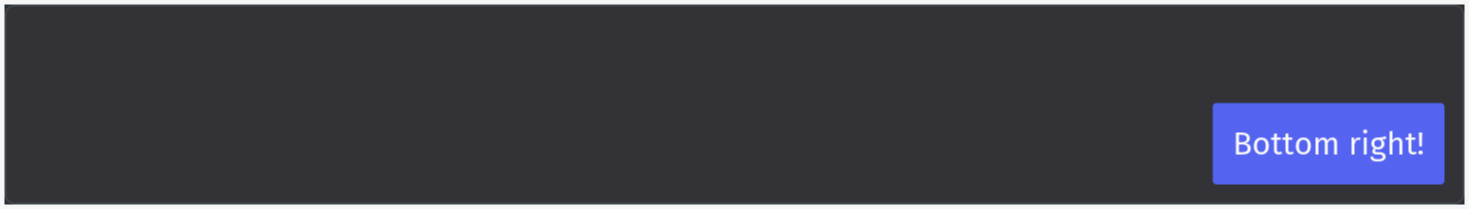
Alignment

Analogously to the `Text` widget, a `Container` will align its contents inside of its bounds.

By default, a `Container` inherits the sizing strategy of its contents. Therefore, all of the alignment methods ask for a new `Length` that will be used to determine the final bounds.

You can think of a `Container` as a framed picture. The frame itself is the `Container` and the picture is the content, which can be aligned inside in different ways.

```rust
use iced::widget::container;
use iced::Fill;

let my_box = container("Bottom right!")
    .padding(10)
    .style(container::primary);

container(my_box)
    .padding(10)
    .align_bottom(Fill)
    .align_right(Fill)
    .style(container::bordered_box)
```



Additional Resources

Here are some further resources you can use to learn more about iced while I am still working on this book:

> Keep in mind that some of these resources may be using an older version of iced. However, while the specifics of the APIs used may change, the fundamental ideas of iced tend to be quite stable.

- The Pocket Guide
- The official examples
- The API Reference
- The unofficial guides
- A step-by-step video guide to building a simple text editor

We also have a very welcoming and active community! Feel free to ask any questions in our Discord server or our Discourse forum.

758 online discourse 547 users

Frequently Asked Questions

Why is the documentation so bad?

The documentation is bad because, unlike most other open-source projects, `iced` unapologetically does not cater to *you*.

> iced is not your run-of-the-mill popular open source project. It is not owned by a corporation. It is not a brand. It is not a business. And most importantly, it is not a community effort.
>
> iced is just my personal project. I work on it on my own terms, on my time, and give it all away for free as open source.

As a result, I have no external incentives to make the library more appealing to newcomers and beginners. I do not care whether you will like the library and I have no reasons to try to convince you to use it.

The gist of it is that

- I enjoy coding more than I enjoy writing documentation,
- I don't like spoonfeeding people as if they are stupid, and
- I ultimately don't care about wide adoption.

When I do write documentation, I don't do it with a promotional intent. Instead, I do it mainly to test whether I can explain the library in simple terms—starting from the problem. This book being a clear attempt at it!

Thus, the existing documentation is not there to cater to a wide audience and convince you to use the library in its current state. It's mainly there for me to cement design choices and have fun. The output is a side effect.

I believe that this approach will eventually lead to a fully fleshed out set of learning materials for everyone, with the least amount of wasted effort on my end. It'll just take a bunch of time!

Why do none of the examples compile?

All of the examples are compiled in a GitHub CI workflow for every single commit pushed to the official repository. They *very* rarely break and, if they do, I fix them quickly. So, you are likely wrong. The examples do compile.

Many newcomers make the rookie mistake of assuming that they can copy code from the `master` branch and then expect it to work with a crates.io release from a year ago. Some even choose to blame the project for their own ignorance and incompetence. I'm always quite baffled by this.

Release tags exist for a reason.

Why do you change the entire API on every release?

Because I am a terrible engineer and I want you all to suffer with me.

Jokes aside, I am always trying to come up with the best way to build GUIs in Rust. Why would I not change the API? I certainly owe absolutely nothing to the users that benefit from my work for free.

If you are not happy with it, feel free to ask for a refund, then fork the project and maintain it yourself.

Is `iced` even used seriously anywhere?

Yes, here and there.

- Kraken has been shipping a desktop application to thousands of users for years.
- System76 maintains a soft fork that powers their new COSMIC desktop environment.
- Plenty of popular open-source projects use it—like Halloy and Sniffnet.
- Many users build and share cool stuff every day in our Discord server.

Check out the project showcase if you want to find out more.

When will `<insert feature>` be developed?

`iced` is a one man project. Every single line of code is either written or reviewed directly by me.

And I am just a dude that enjoys coding and building stuff. I give away some of my work for free. I do this with no timelines. No promises. No expectations. No delegation. I like it this way.

That said, there is a visual roadmap that can give you a certain idea of my current mental model of the future of the library. My mind changes often, though; and so may the roadmap!

When will my PR get reviewed?

I tend to review contributions right before a release. Releases happen rarely; so it may be months until I get to take a look at your code.

I also may choose to prioritize some people, like my friends. I don't care about being fair to everyone.

In any case, if your contribution comes with the expectation that I must eagerly review it, then I kindly ask you to stay away from any of my open-source projects.

Why are you so unprofessional, emotional, and sarcastic?

Why am I having fun, you ask? This is my personal project. I intend to express myself fully.

I don't have the time, the energy, nor the interest to water down my writing in order to make it palatable for everyone.

Did you use any AI or LLM to write this book?

No. I proudly own every ~~terrible~~ piece of writing in here—em dashes included. Beep, boop.

How do I structure a large application?

You split your application into multiple screens, and then use simple composition.

The Pocket Guide has a specific section that showcases this approach.

How can my application receive updates from a channel?

You can use `Task::run` to generate messages from an asynchronous `Stream`.

Alternatively, if you control the creation of the channel; you can use `Subscription::run`.

Does `iced` support Right-To-Left text and/or CJK scripts?

The seeds are planted, but the edge cases may not be fully handled yet.

Specifically, text editing likely has a bunch of issues still. However, Input Method Editors are supported since the `0.14` release.

In any case, proper support is in the roadmap.

When are the `view` and `subscription` functions called?

After every batch of messages and `update` calls.

But this is an implementation detail. You should *never* rely on this. Try to treat these functions as declarative, stateless functions.

Does `iced` redraw all the time?!

No. Not anymore!

It used to be the case that `iced` would redraw the entire window on every tiny mouse movement but, since `0.14`, it supports and enables reactive rendering by default.

I am getting a panic saying there is no reactor running. What is going on?

You are probably using `Task` to execute a `Future` that needs the `tokio` executor:

```
there is no reactor running, must be called from the context of a Tokio 1.x runtime
```

You should be able to fix this issue by enabling the `tokio` feature flag in the `iced` crate:

```
iced = { version = "0.14", features = ["tokio"] }
```