SwiftUI: Declarative, responsive interface

premières interfaces en SwiftUI

POLYTECH
MONTPELLIER

SwiftUI: Introduction

Declarative, responsive interface

SwiftUI: Introduction

SwiftUI uses *View* from UIKit to describe screens, and most of the widgets

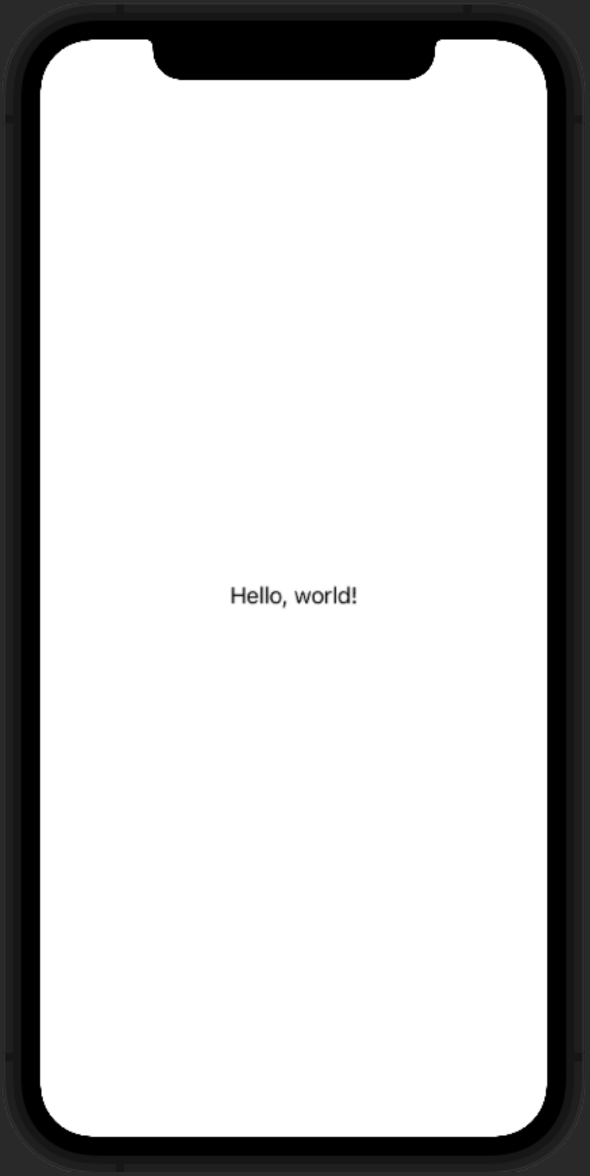
But you no longer design you UI, you *declare* it!

No more layout design but *layout structures* that must declared.

Another important difference: no more `UIViewControler` to implement MVC design pattern but a *reactive link* between data and UI via *Combine* framework.

Create an app with *Playground* et and delete code of `ContentView` to keep only:

```
struct ContentView: View {
  var body: some View {
    Text("Hello, world!")
  }
}
```



Hello, world!

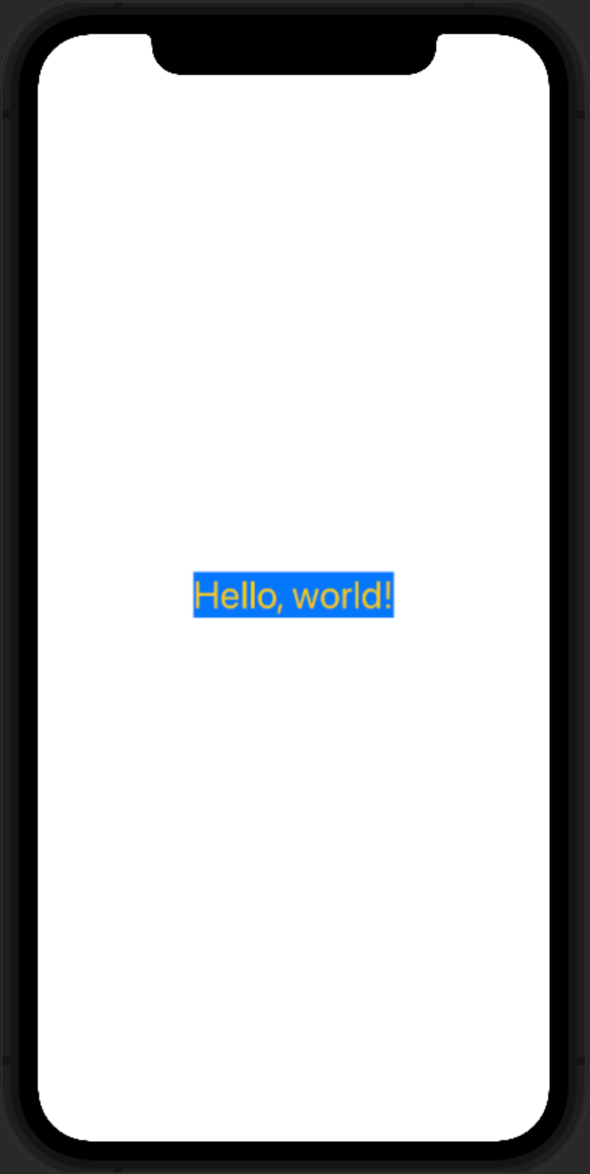
`ContentView`: Struct that declares content of the screen

`Text`: SwiftUI Equivalent to `UILabel` - display a simple text.

`body`: computed property that will be evaluated to define what is displayed

Evaluation of body defines layout and style of what is displayed:

```
struct ContentView: View {
  var body: some View {
    Text("Hello, world!")
      .font(.title)
      .foregroundColor(.yellow)
      .background(.blue)
      .padding()
  }
}
```

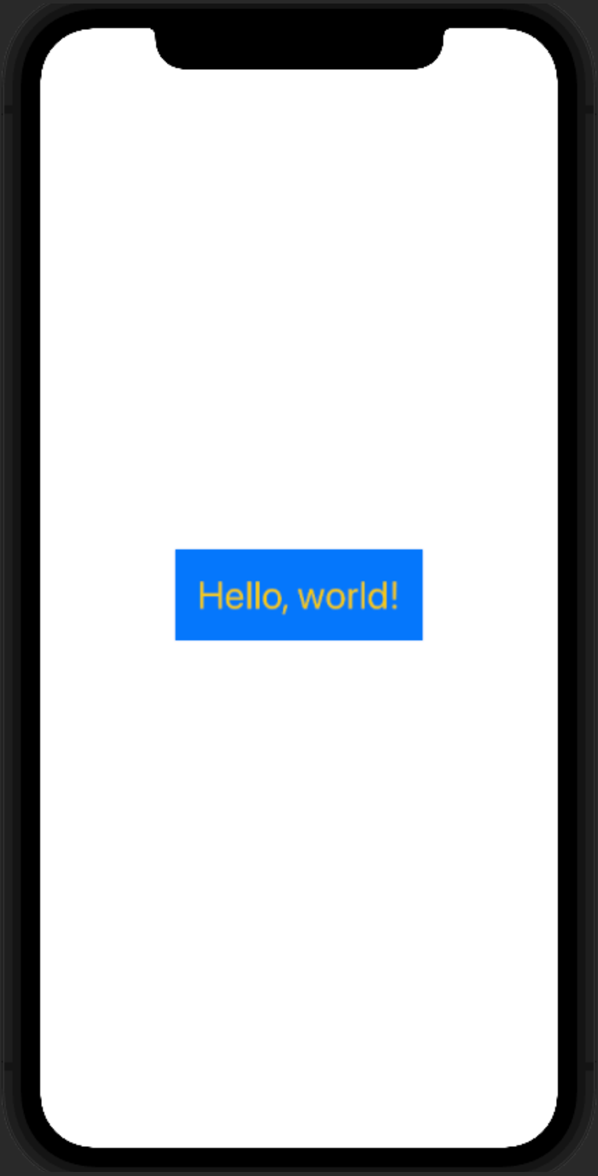


Hello, world!

We just set text as yellow title on blue background and UI changes.

```
struct ContentView: View {
  var body: some View {
    Text("Hello, world!")
      .font(.title)
      .foregroundColor(.yellow)
      .padding()
      .background(.blue)
  }
}
```

Changing the order of the statements changes the result: here the application of a blue background is done after adding a padding.



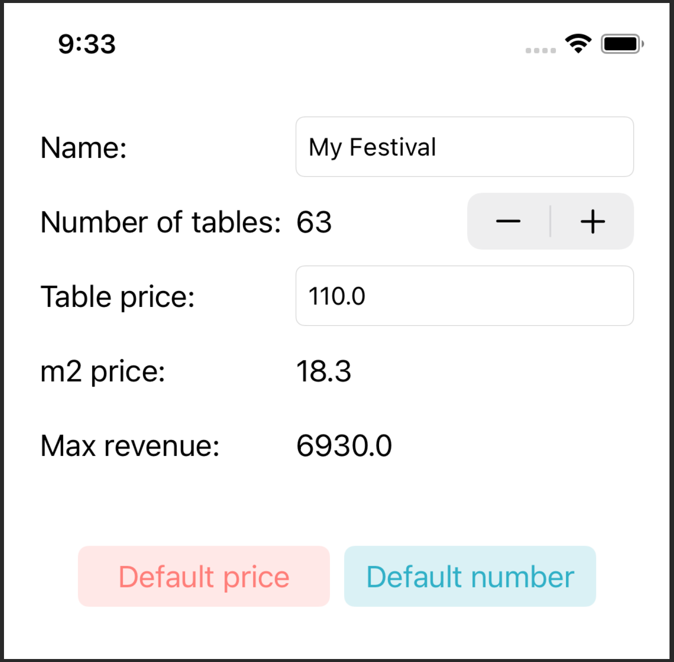
Hello, world!

First UI design with SwiftUI

Objective

Create UI shown to the right:

- Widget to use?
- Layout?
- How to change values?
- Initial reviews...



Name: My Festival

Number of tables: 63

Table price: 110.0

m2 price: 18.3

Max revenue: 6930.0

Basic Widgets

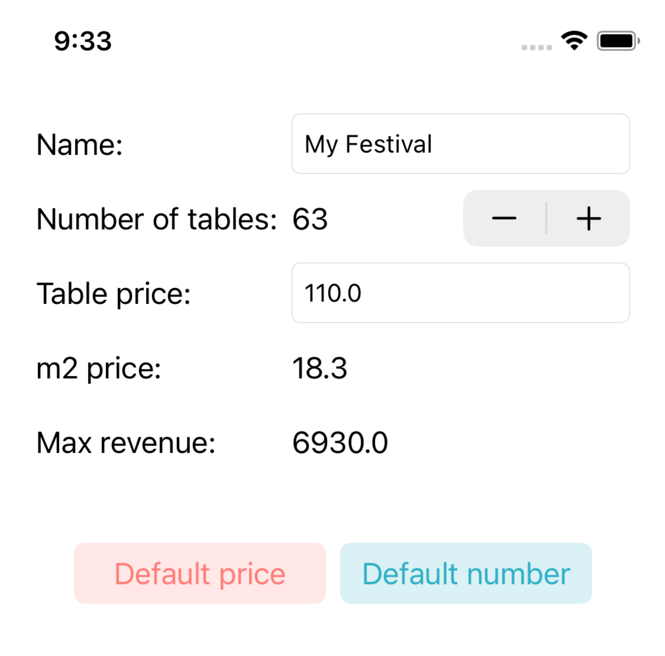
If you knows UIKit, most of widget have an equivalent in SwiftUI, just suppress UI prefix:

- `UILabel` ⟷ `Text`
- `UITextField` ⟷ `TextField`
- `UIStepper` ⟷ `Stepper`
- `UIButton` ⟷ `Button`

First try:

So we need to add:

- `Text("Name:")`
- `TextField()`
- `Text("Number of tables")`
- `Text("64")`
- `Stepper()`
- `Text("Table price")`
- `TextField()`
- `Text("m2 price")`
- `Text("18.3")`
- `Text("Max revenue")`
- `Text("7040.0")`
- `Button("Default price"){}` *// syntax explained later*
- `Button("Default number"){}` *// syntax explained later*



9:33

Name: My Festival

Number of tables: 63

Table price: 110.0

m2 price: 18.3

Max revenue: 6930.0

If we try to add all previous lines of code, we get a lot of compilation errors that comes essentially from

- `TextField`
- `Stepper`

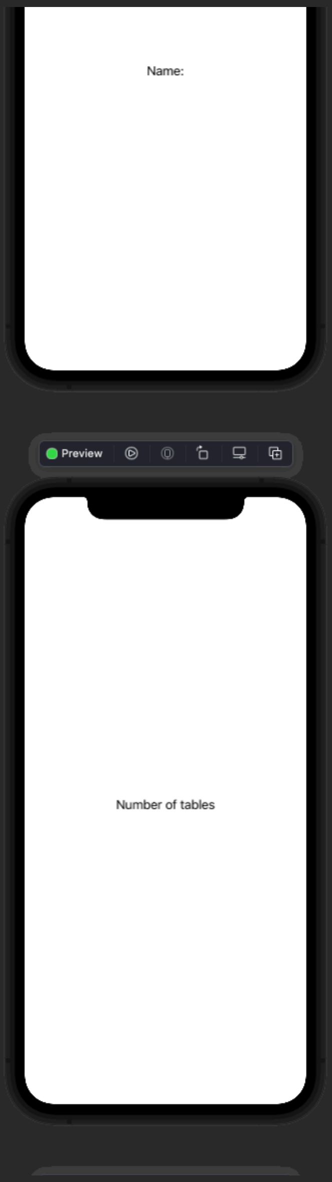
This is because these widgets are made to modify values. So we can't use them without implementing the reactive mechanisms of Swift.

We will see them later. At first, we will create the target UI without these components, possibly replacing them by simple `Text("value")`.

First try

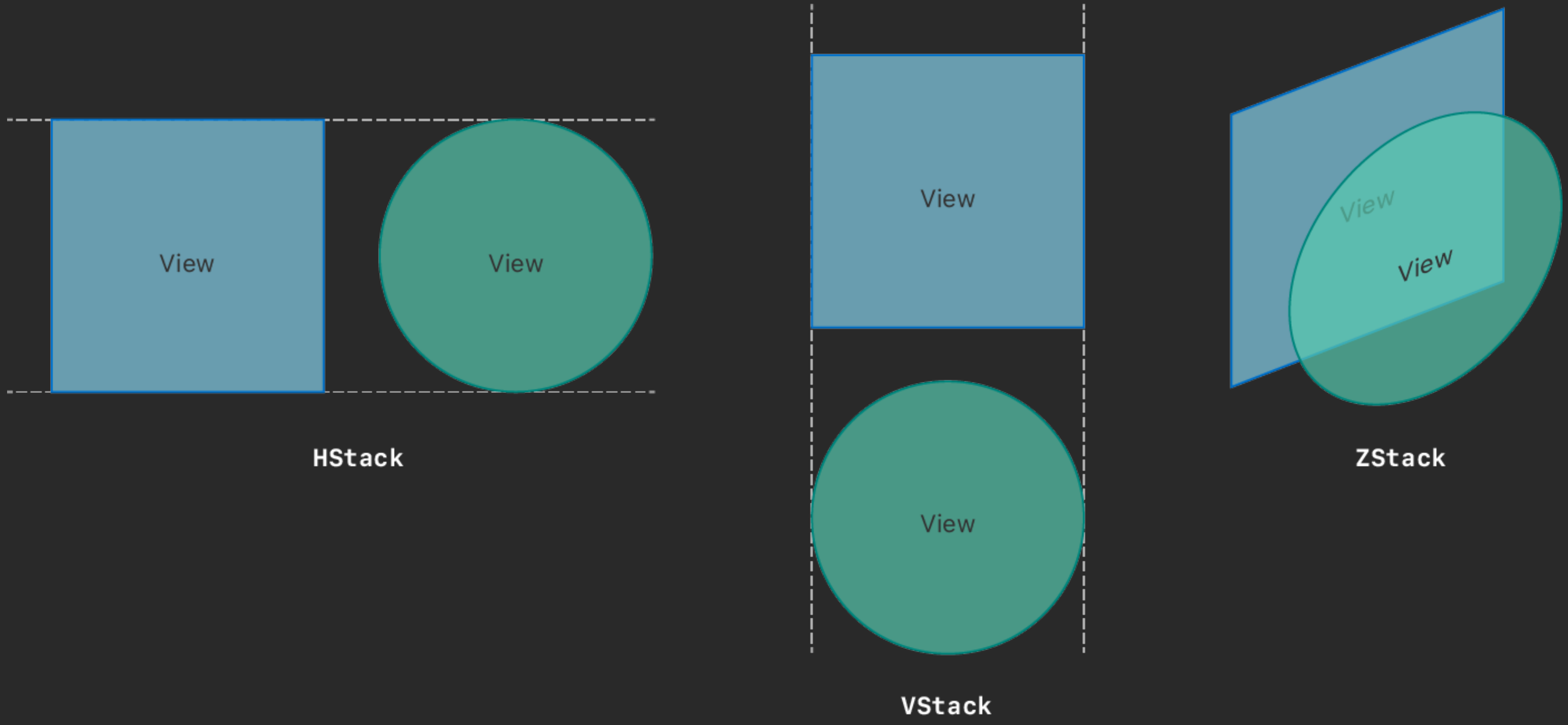
Add widgets to the view:

```
struct ContentView: View {
  var body: some View {
    Text("Name:") ; Text("My Festival")
    Text("Number of tables") ;
    Text("64")
    Text("Table price") ; Text("110.0")
    Text("m2 price") ; Text("18.3")
    Text("Max revenue") ; Text("7040.0")
    Button("Default price"){}
    Button("Default number"){}
  }
}
```



We get as much screens as widgets!

Stacks



Stacks are boxes allowing to stack horizontally, vertically or in depth widgets

```
HStack(alignement: VerticalAlignment) { composants }
```

Places components in a row from left to right, aligned to the top of the stack (.top), the bottom (.bottom), or the center (.center).

```
VStack(alignment: HorizontalAlignment) { composants }
```

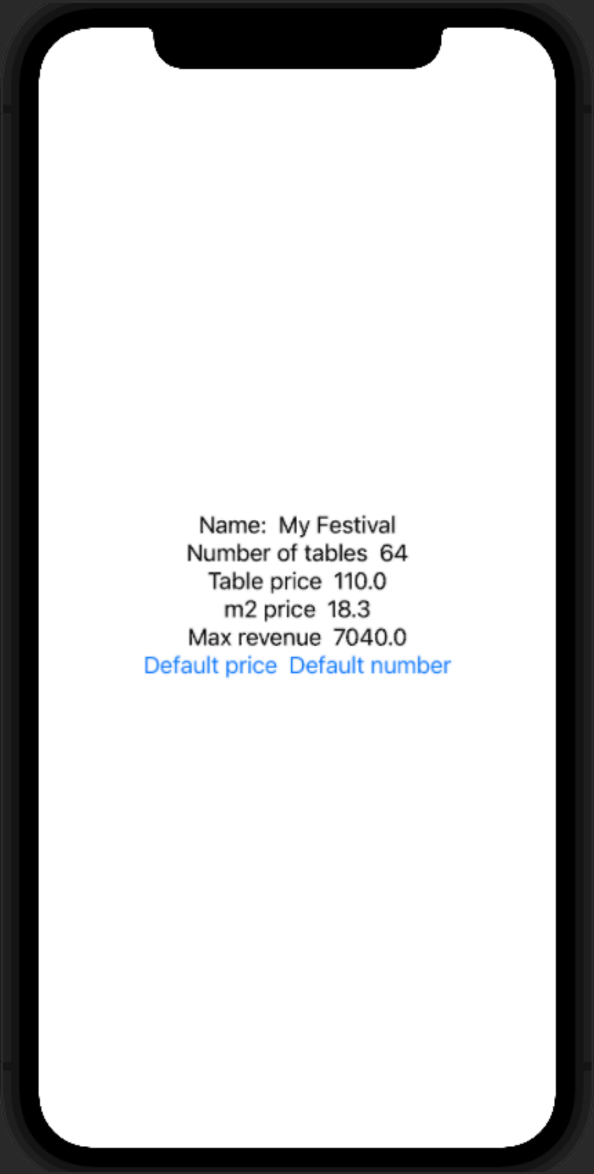
Place components from top to bottom, left justified (.leading), right justified (.trailing), or centered (.center)

```
ZStack(alignement: Alignment) { composants }
```

Place the components from bottom to top, one on top of the other, aligned with their position on a grid

topLeading	top	topTrailing
leading	center	trailing
bottomLeading	bottom	bottomTrailing

```
struct ContentView: View {
  var body: some View {
    VStack{
      HStack{
        Text("Name:") ; Text("My Festival")
      }
      HStack{
        Text("Number of tables");Text("64")
      }
      HStack{ Text("Table price");Text("110.0") }
      HStack{ Text("m2 price") ; Text("18.3") }
      HStack{ Text("Max revenue");Text("7040.0") }
      HStack{
        Button("Default price"){}
        Button("Default number"){}
      }
    }
  }
}
```



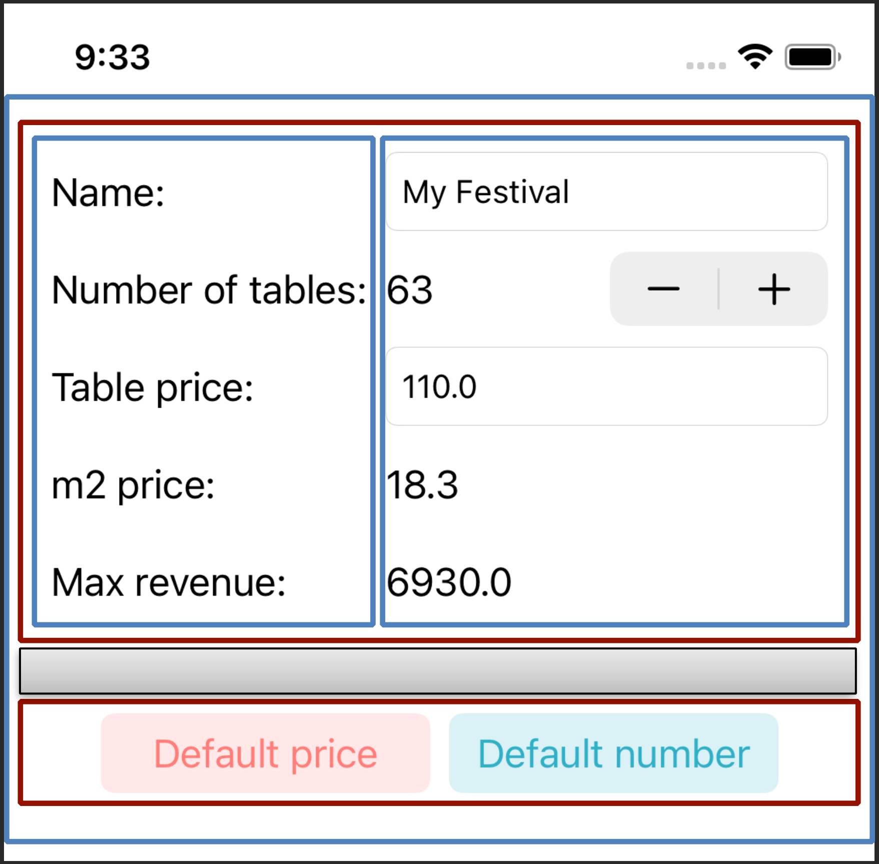
Name: My Festival
Number of tables 64
Table price 110.0
m2 price 18.3
Max revenue 7040.0
Default price Default number

You need to think about your layout structure and choose well your stacks!

In blue: `VStack`

In red: `HStack`

In grey: `Spacer`



9:33

Name: My Festival

Number of tables: 63

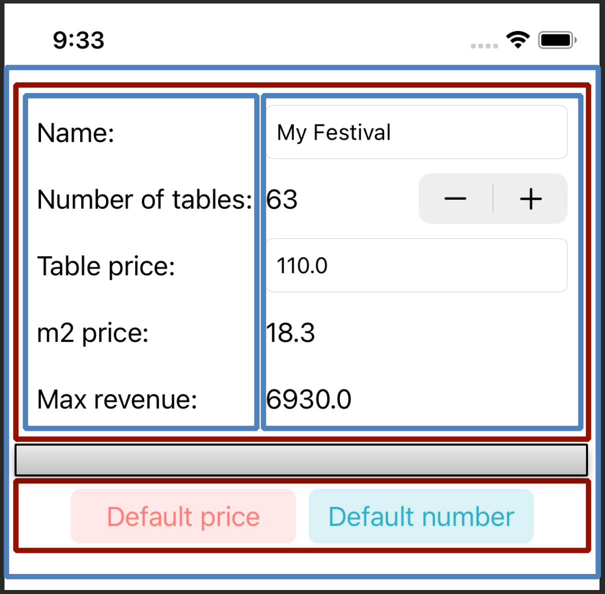
Table price: 110.0

m2 price: 18.3

Max revenue: 6930.0

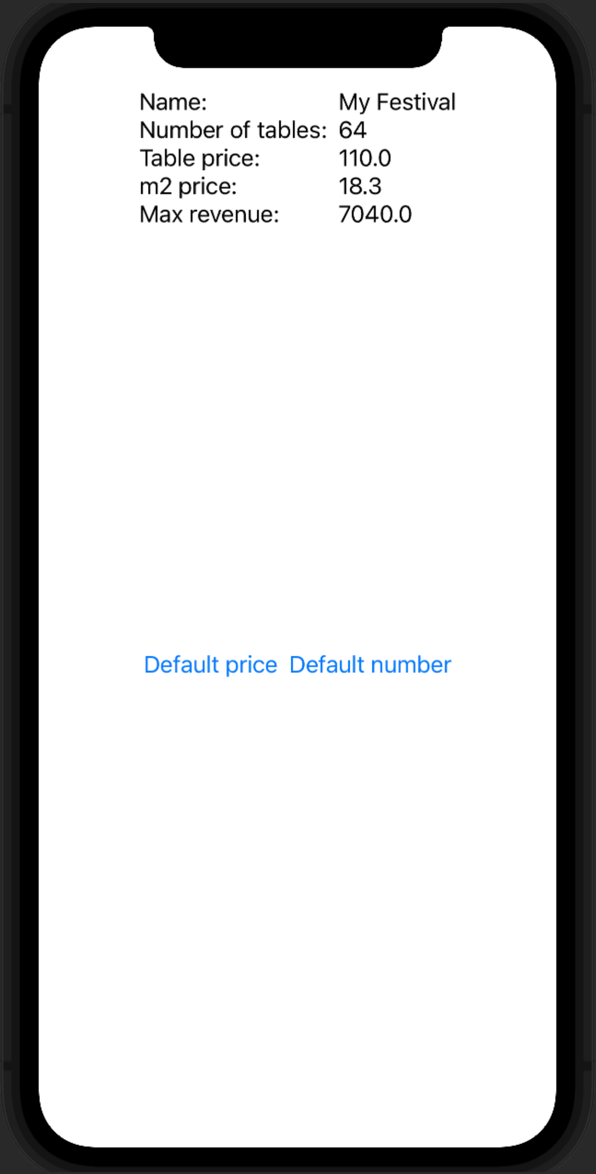
Exercise 1:

Make sure to modify previous code by adding `HStack{}`, `VStack{}`, `Spacer()` corresponding to the diagram below to get the following:



9:33

Name: My Festival
Number of tables: 63
Table price: 110.0
m2 price: 18.3
Max revenue: 6930.0



Name: My Festival
Number of tables: 64
Table price: 110.0
m2 price: 18.3
Max revenue: 7040.0

Note that space between texts and button is too large.

Frames

A Frame is a view of fixed size:

```
frame(minWidth:, maxWidth:, width:, height:, alignment:)
```

You can also make the size maximum with .infinity value

This will allow us to simulate margin constraints

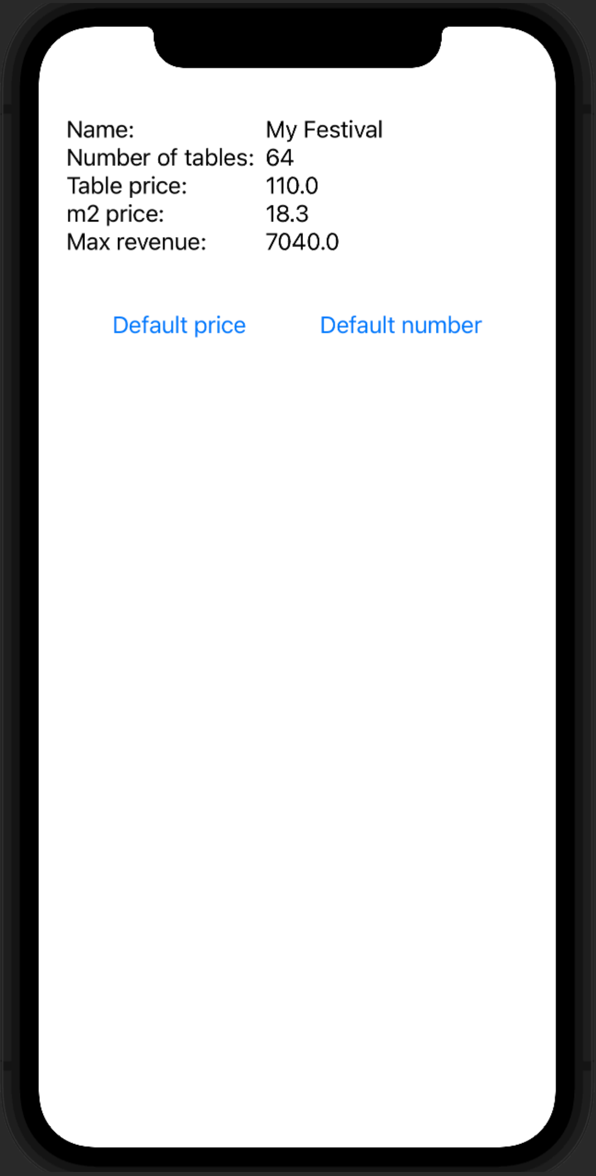
All widgets, including `Spacer` and `Stack` are also views, so we can set them a `frame`.

Example :

```
Spacer().frame(height: 40)// sets height of space between
                          // texts buttons
```

Exercise 2:

Modify you previous answer to obtain result shown to the right:



Name: My Festival
Number of tables: 64
Table price: 110.0
m2 price: 18.3
Max revenue: 7040.0

Buttons

Buttons have three syntaxes that each specify content and action that will be triggered

```
Button(String, action : { }) // button with a text
Button(String){  // button with a text
// action code
}
Button(action: ) {
// content more complex
}
```

You can pass a procedure directly to the action parameter. The procedure must not take any parameters.

Some examples:

With text and an action given in parameter

```
Button("Sign in", action: function_name)
```

With text and code of action

```
Button("Sign in"){
  // action code
}
```

With a content other than text:

```
Button(action: function_name) {
  Label("Add Folder", systemImage: "folder.badge.plus")
}
```

A button is a view and can therefore be associated with the usual properties: `background`, `foregroundColor`, `frame`, `cornerRadius`, `padding`, ...

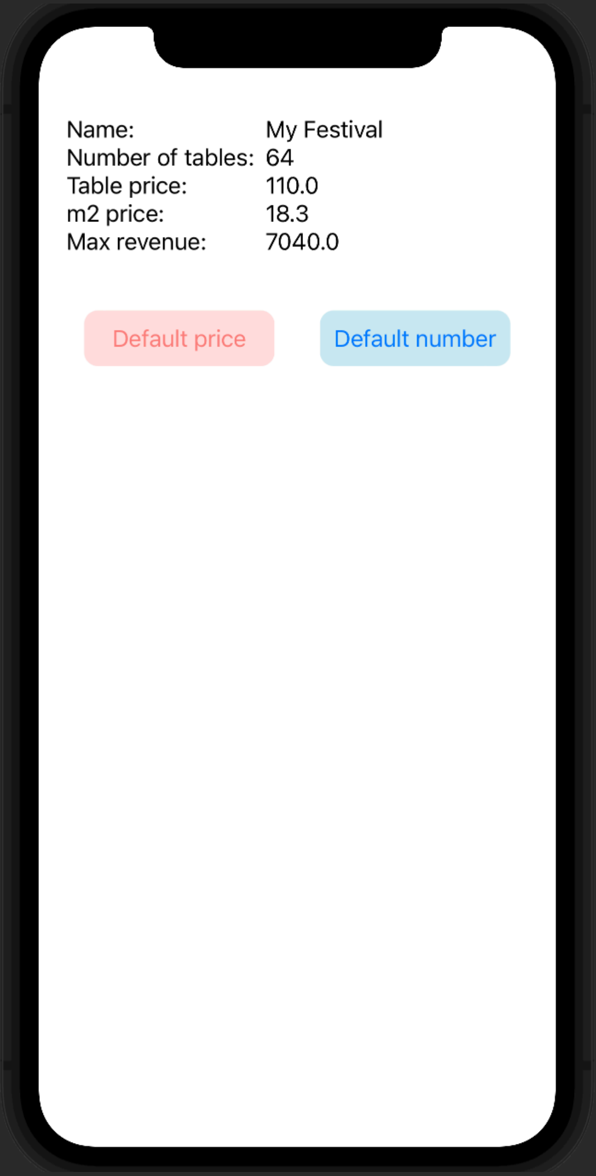
Example :

```
Button("Default number", action: {})
  .padding(10)
  .frame(width: 138)
  .background(.teal.opacity(0.25))
  .cornerRadius(10)
var salmon = Color(red: 1.0, green: 126.0/255.0, blue:
121.0/255.0)
Button("Default price", action: {} )
  .padding(10)
  .frame(width: 138)
  .background(salmon.opacity(0.25))
  .foregroundColor(salmon)
  .cornerRadius(10)
```

Buttons

Exercise 3:

Modify previous code by adapting the buttons to obtain the following display:



Name: My Festival
Number of tables: 64
Table price: 110.0
m2 price: 18.3
Max revenue: 7040.0

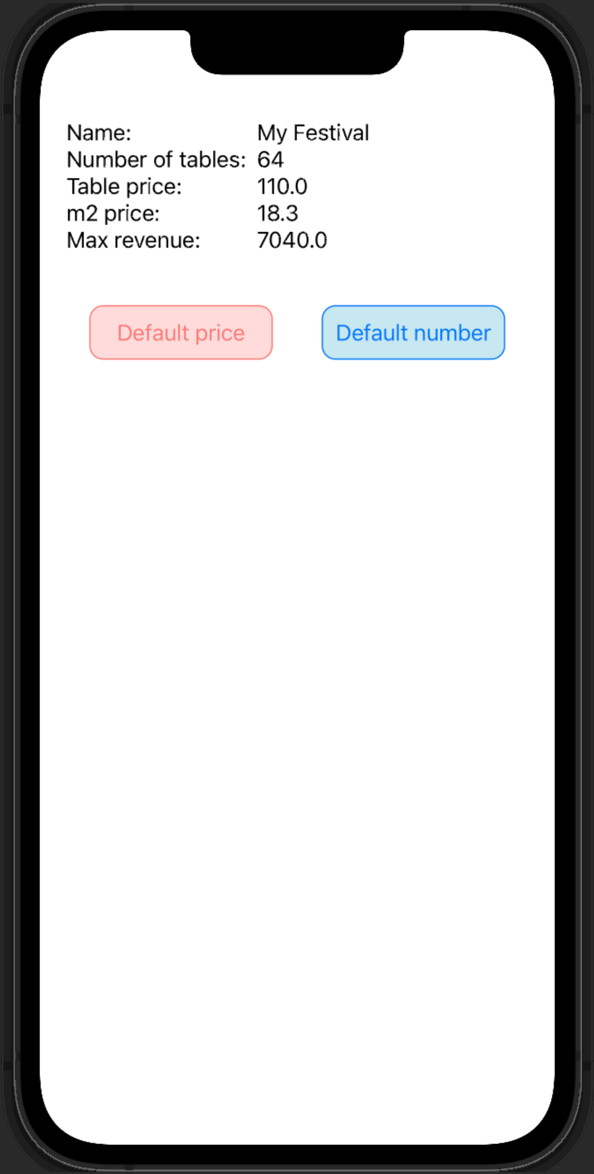
Exercise 4:

Even better, add a border to the buttons

Hint:

Unfortunately borders of button frame do not take cornerRadius into account. So we have to add a border on top of the button.

We won't use a `ZStack` but a `.overlay` to make sure that the button view is perfectly overlaid and we will use a `RoundedRectangle` to create the border.



Name:	My Festival
Number of tables:	64
Table price:	110.0
m2 price:	18.3
Max revenue:	7040.0

Another layout : grids

Another solution would have been to use a grid layout.

For this, SwiftUI provides two structures:

- `LazyVGrid`(columns: , spacing:)
- `LazyHGrid`(rows: , spacing:)

The principle is that a grid will be built vertically, or horizontally using an array of `GridItems` to define columns or rows.

The number of `GridItems` in the array will determine number of columns (resp. rows) of the `LazyVGrid` (resp. `LazyHGrid`), except in the case of adaptive `GridItems`

There are 3 types of `GridItem` :

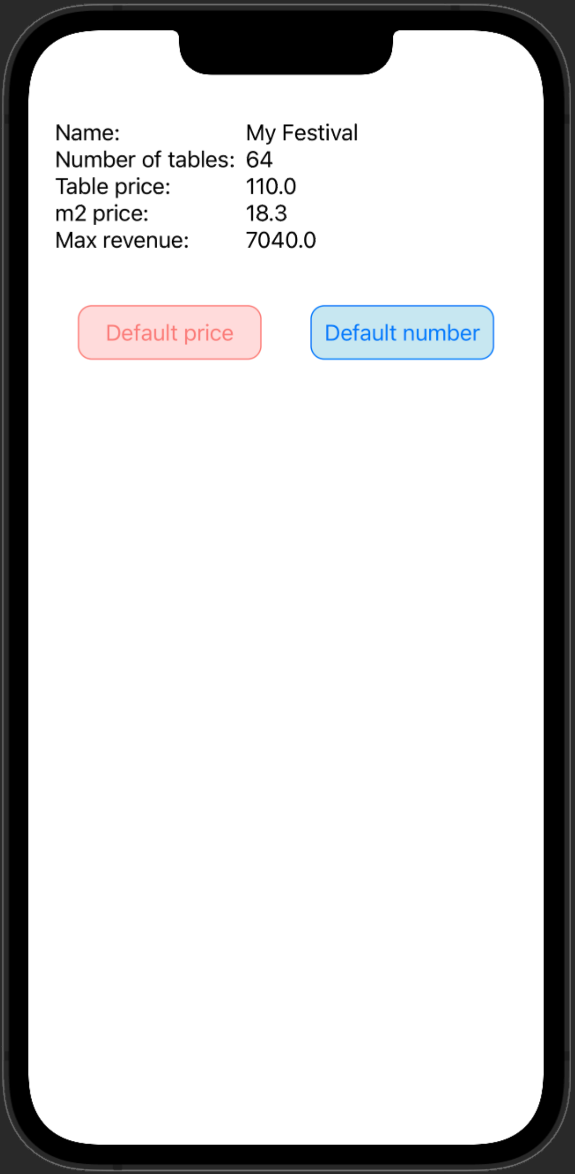
- *Fixed :* of fixed size
- *Flexible :* it adapts to the screen size according to the number of columns (or rows)
- *Adaptative :* it adapts to the screen and will therefore, according to a minimum size, adjust its size to set maximum number of columns (resp. lines)

Examples of [GridItem] used with a LazyVGrid:

```
// 2 columns of fixed size
var cols=[GridItem](repeating:.init(.fixed(120)),count:2)
// 2 columns of the same size adjusting to the width of the screen
var cols=[GridItem](repeating:.init(.flexible()),count:2)
// 2 columns, one of fixed size, the other taking the whole width
var cols=[GridItem(.fixed(120)),GridItem(.flexible)]
// as many columns of minimum size 100 as possible
var cols=[GridItem(.adaptative(minimum:100))]
```

Exercise 5: GridLayout

Using a `LazyVGrid`, modify your code to obtain the layout shown to the left:



Name:	My Festival
Number of tables:	64
Table price:	110.0
m2 price:	18.3
Max revenue:	7040.0