5. JETBACK COMPOSE

Approccio funzionale, funzioni etichettate con annotazione **@Composable**. Notazione è gestita in compilazione.

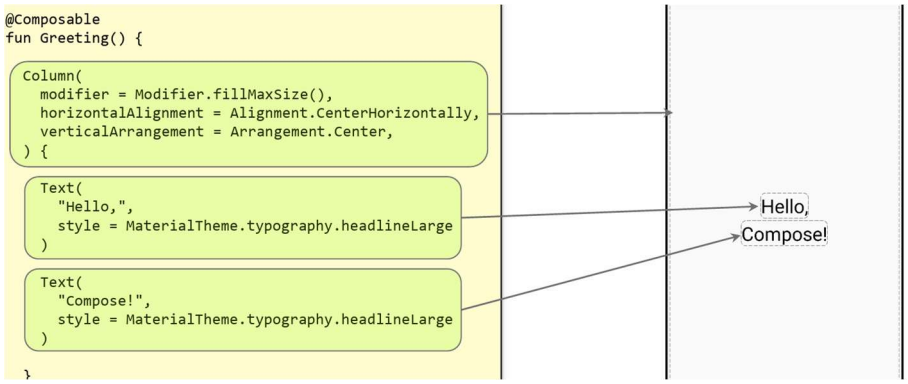
NOTA: una funzione composable può invocare una funzione plain mentre una funzione plain, non può invocare una funzione composable. C'è solo un'eccezione → setContent: accetta come parametro una lambda composable.

Ogni volta che una funzione composable è eseguita, Andorid traccia i parametri forniti alla funzione.

Puramente dichiarativo: funzione di basso livello producono codice che viene eseguito dalla cpu per produrre delle rappresentazioni. Se android si accorge che sono cambiati dei parametri → android riesegue la funzione per la GPU → se cambiano i dati, cambia cosa vedo su schermo → **data driven**.

Android esegue funzioni sofisticate, in modo parallelo → dunque per evitare incongruenze, le funzioni composable **non possono avere "side effect"**, come ad esempio modificare variabili globali, etc… loro possono solo generare del codice.

Le funzioni composable, solitamente, **non ritornano nulla** ma emettono dei blocchi internamente **chiamando altre funzioni composable** primitive fornite da Android o non primitive create → queste funzioni consistono in istruzioni per la GPU.



Scritte in Uppercase →**Column**

- Modifier
- horizontalAlignment
- verticalArrangment
- trailing lambda
    - contiene 2 buildComposable Text

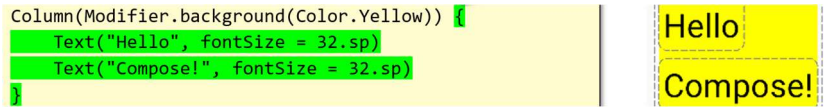
NOTA su column: mette uno sotto l'altro i blocchi che contiene nella lambda, rispettando indicazioni date.
NOTA su row: mette uno a fianco all'altro
NOTA su text: trasforma una stringa in un insieme di comandi grafici

- stringa
- style → mette insieme le informazioni

Compose DSL

Quando è presente una trailing lambda, questa crea una composizione gerarchica: column è padre dei 2 Text

Non avendo specificato posizione e dimensione, il blocco di column sarà grande il minimo che serve per contenere i due text e i due text non avranno un allineamento particolare.



NOTA: Una lambda può avere un receiver → elemento this che fornisce alcune funzionalità aggiuntive.

TEXT → mostrare uno o più linee di testo

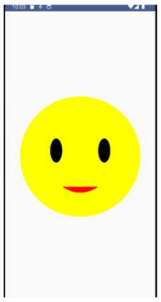
- **text**: obbligatorio, indica la stringa che vogliamo mostrare (il resto sono opzionali)
- **fontsize**: dimensione
- **modifier**: azioni o decorazioni che vogliamo
  - **size**: area del riquadro
  - **background**: colore sfondo
  - **wapContectSize**: se non c' abbastanza spazio per la stringa, questa viene wrappata e ci sarà una rappresentazione su più righe
  - **blur**: nebbia su elemento, sfoca

```
Text(
  text = "Hello",
  fontSize = 64.sp,
  modifier = Modifier.size(160.dp)
                .background(Color.Yellow)
                .wrapContentSize(Alignment.Center)
                .blur(8.dp) )
```

CANVAS → modo per rappresentare una superficie, un'area rettangolare dipingendo qualcosa

Molti metodi.

```
Canvas(modifier = Modifier.fillMaxSize(), onDraw = {
    val (w,h) = size
    val r = 0.4f * min(w,h)
    val offsetL = Offset(-r*0.5f, -r*0.3f)
    val offsetR = Offset( r*0.3f, -r*0.3f)
    val path = Path()
    path.moveTo(w*0.5f-r*0.3f, h*0.5f+r*0.5f)
    path.relativeQuadraticBezierTo(r*0.3f,r*0.2f, r*0.6f, 0f)
    drawCircle(color = Color.Yellow, radius = r, center = center)
    drawOval(Color.Black, size = Size(r*0.2f,r*0.4f), topLeft = center+offsetL )
    drawOval(Color.Black, size = Size(r*0.2f,r*0.4f), topLeft = center+offsetR )
    drawPath(path, Color.Red)
})
```



All'interno del blocco it.polito.mad, aggiungo **un file kotlin chiamato gui.kt** dicendo che è un file.

Creo funzione **@Composable fun MyGui() { }**

- **Text**
  - **style** = MaterialTheme.typography._____ → contiene stili predefiniti
  - **textAlign** = TextAlign._____ → allineamenti orrizzontali
    - allinea all'interno del suo riquadro, non in generale
  - **modifier** = Modifier
    - .background(Color.___) → colorare sfondo blocco
    - .fillMaxWidth() → prende tutto lo spazio possibile orrizzontalmente e solo
      - quello di cui ha bisogno verticalmente
    - .fillMaxWeigth() → prende tutto lo spazio possibile verticalmente
    - .fillMaxSize(.8f) →prendi 80% veritcalmente e 80% orrizontalmente
- **Canvas**(modifier = Modifier._____){
  - val r = minOf(size.width, size.height)*0.4f →raggio
  - drawCircle(Color._____, r, Offset(size.width*0.5f, size.height*0.5f))

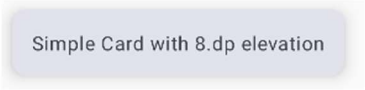
  }
- **Image()** → mostrare immagini forniti da instanza Painter, bisogna aver già scaricato le immagini e devono essere disponibili

```
Image(
    painter = painterResource(id = androidx.core.R.drawable.ic_call_answer),
    contentDescription = "Phone call",
    modifier = Modifier.fillMaxSize()
)
```

NOTA: esiste la libreria coil che mi permette di usare async image `implementation("io.coil-kt:coil-compose:2.6.0")`

- AsyncImage(
  - model="url"
  - contentDescription ="stringa per non vedenti"
  - contentScale= ContentScale.Crop →
  - modifier = _____
  - placeholder = __(qualcosa in res)__ → usato per mostrare qualcosa mentre carica l'immagine
  - occhio che non funzionava

- **Card()** → blocco dove si possono inserire cose in colonna
    - Elevation: assottiglia angoli
    - Padding: spazio interno




```
Card(elevation = CardDefaults.cardElevation(8.dp),
     modifier = Modifier.padding(16.dp)
) {
     Text(text = "Simple Card with 8.dp elevation",
                  modifier = Modifier.padding(16.dp))
}
```

NOTA: aggiungendo **@Preview** ho la possibilità di vedere cosa fa *(Aggiungere sopra a composable*)

BASIC LAYOUTS:

- COLUMN: gerarchico
    - Ogni figlio, uno sotto la'ltro
- RAW: gerarchio
    - Ogni figlio, alla dx del precedente
- BOX: gerarchico
    - Ogni figlio, sopra al precedente → insieme di layer

```
@Composable
fun ColumnLayout() {
          Column(
            Modifier.fillMaxSize(),
             horizontalAlignment = Alignment.CenterHorizontally
             verticalArrangment = Arrangment.SpaceEvenly,
          ){
            YellowBlock()
            RedBlock()
          }
}

@ComposaBble
fun RowLayout(){
          Row(
            modifier = Modifier.fillMaxSize(),
            horizontalArrangment = Arrangment.SpaceEvenly,
            verticalAlignment = Alignment. CenterVertically,
          ){
            YellowBlock()
            RedBlock()
          }
}

@Preview
@Composable
Fun MyGui() {
          BoxWithConstraints {
            if (this.maxHeight > this.maxWidth)
                    ColumnLayout()
            else
                    RowLayout()
          }
```

NOTA: Column e War possono avere un peso **weight**

- Prendono spazio proporzionale al peso

```
fun ColumnLayout() {
    Column(
        Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.SpaceEvenly
    ) { this: ColumnScope
        Box(Modifier.fillMaxWidth().weight(1f).background(Color.Yellow)
        Box(Modifier.fillMaxWidth().weight(2f).background(Color.Red))
    }
}
```

```
olumn(
   Modifier.fillMaxSize(),
   horizontalAlignment = Alignment.CenterHorizontally,
   verticalArrangement = Arrangement.SpaceEvenly
{ this: ColumnScope
   Box(Modifier.aspectRatio( ratio: 1f).weight(2f).background(Color.Yellow)
   Spacer(Modifier.height(8.dp))
   Box(Modifier.aspectRatio( ratio: 1f).weight(2f).background(Color.Red))
```

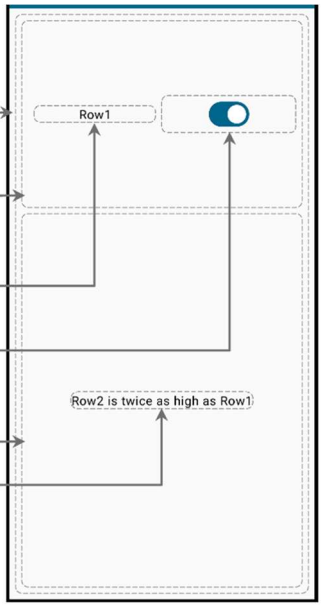
CREATING COMPLEX LAYOUTS

Quando SI CREA UN Composable, conviene aggiungere come parametro un Modifier che è inizializzato di default con una lista Modifier vuota.

```
@Composable
fun ProductCard(modifier: Modifier = Modifier) {
    Column(modifier = modifier, ... ) { ... }
}
```

Layout base

```
@Composable
fun SimpleLayout() {
  Column(Modifier.fillMaxSize()) {

    Row(
      Modifier.fillMaxWidth()
              .weight(1f),
      verticalAlignment =Alignment.CenterVertically ) {
      Text("Row1",
          Modifier.weight(1f),
          textAlign = TextAlign.Center)
      Switch(checked = true, onCheckedChange = {},
            modifier = Modifier.weight(1f) )
    }

    Box(Modifier.fillMaxWidth().weight(2f),
        contentAlignment = Alignment.Center ) {
      Text("Row2 is twice as high as Row1")

    }
  }
}
```



Funzioni composable

- possono ricevere parametri, usare istruzioni condizionali, cicliche, etc
- possono chiamare altre funzioni


```
@Composable
fun SimpleList(l: List<String>, modifier: Modifier) {
    Column(modifier = modifier) {
        l.withIndex().forEach() {
            val mod = if (it.index %2 == 0) Modifier
            else Modifier.background(Color.LightGray).fillMaxWidth()
            Text(text = it.value, modifier = mod, fontSize=32.sp)
        }
    }
}
```

questo metodo per la lista, va bene se ho pochi elementi altrimenti no perché non ho scrolling.

Lists & Grids

Implementa lo **scrolling** nel caso in cui il nr di elementi sia maggior dello spazio disponibile. Inoltre, la maggior parte dei blocchi non vengono eseguiti dalla GPU quando esegue in quanto è **LAZY** → quando inizio a scrollare, istanzio gli elementi che diventano visibili.

Quando si crea un Lazy column/grid, ci sono un insieme di parametri e composizione gerarchica. Accetta un receiver che ha LazyListScope → fornisce una serie di metodi:

- items: accetta una lista o un array di items e una lambda responsabile di emettere il contenuto per ogni elemento visibile nel container
- item() : come sopra ma per un singolo elmemento
- indexedItems(): provvede una lambda function con un parametro extra che riporta l'indice corrente

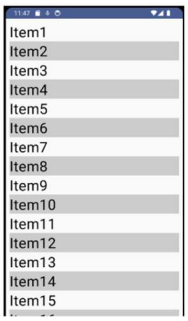
```
@Preview
@Composable
fun MyGui() {
    val l = listOf("A","B","C","D","E","F","G","H","L","M")
    LazyColumn(modifier = Modifier.fillMaxSize()) { this: LazyListScope
        items(l) { this: LazyItemScope it: String
            Text(text = it, style=MaterialTheme.typography.displayLarge
        }
    }
}

items( count: 1000) { this: LazyItemScope
    Text(text = "Element$it", style
```

Element223
Element224

- stickyHeadeR() emette item header che rimane pinnato anche se si scrolla.

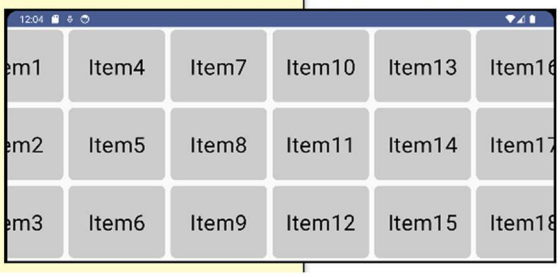
LIST

```
@Composable
fun ScrollingList(l: List<String>, modifier: Modifier) {
    LazyColumn(modifier = modifier) {
        itemsIndexed(l) { index,text ->
            val mod = if (index %2 == 0) Modifier
            else Modifier.background(Color.LightGray).fillMaxWidth()
            Text(text, fontSize = 32.sp, modifier = mod)
        }
    }
}
```



GRID

```
@Composable
fun ScrollingGrid(data: List<String>, modifier: Modifier) {
  LazyHorizontalGrid(rows = GridCells.Fixed(3), modifier = modifier) {
    items(data) {
      Box(modifier = Modifier.padding(4.dp)) {
        Text(it,
             Modifier.fillMaxHeight()
                     .aspectRatio(4f / 3f)
                     .background(
                        Color.LightGray,
                        RoundedCornerShape(8.dp))
                     .wrapContentSize(),
             fontSize = 32.sp)
      }
```



MODIFIERS: forniscono decoratori e comportamenti per elemento a cui applichiamo.

NOTA: l'ordine conta

```
Text(text = "Hello Compose",
     modifier = Modifier.background(Color.Green)
                        .padding(16.dp)
)
```

Hello Compose

```
Text(text = "Hello Compose",
     modifier = Modifier.padding(16.dp)
                        .background(Color.Green)
)
```

Hello Compose

Possono essere forniti ad ogni composable, di default è settato come empty.

```
@Composable
fun CustomImage(imageId: Int, modifier: Modifier = Modifier) {
     Image(painter = painterResource(id = imageId),
           contentDescription = "",
           modifier)
}

CustomImage(R.drawable.vacation, Modifier
     .padding(16.dp)
     .width(270.dp)
     .clip(shape = RoundedCornerShape(30.dp)))
```

- **background(...)**
  - Draws a solid colored shape behind the composable.
- **clickable(...)**
  - Defines a handler that will be invoked when the composable is clicked. When the click is performed, it also causes a ripple effect
- **clip(...)**
  - Clips the composable content to a specified shape
- **fillMaxHeight(...)**
  - The composable will be sized to fit the parent's maximum height.
- **fillMaxSize(...)**
  - The composable will be sized to fit the parent's maximum height and width.
- **fillMaxWidth(...)**
  - The composable will be sized to fit the parent's maximum width
- **offset(...)**
  - Moves the composable the specified distance along the x and y axes from its current position
- **padding(...)**
  - Increases the amount of space around an object.

Then → permette di appendere una ModiferList ad un'altra ModiferList

```
val modifier = Modifier
      .border(width = 2.dp, color = Color.Black)
      .padding(all = 10.dp)
val secondModifier = Modifier.height(100.dp)

Text(
      "Hello Compose",
      modifier.then(secondModifier),
      fontSize = 40.sp,
      fontWeight = FontWeight.Bold
)
```

Custom modifier

Si possono creare dei modifier. Per fare ciò si può partire da drawWithContent().

```
fun Modifier.dashedBorder() = this.drawWithContent {
    val outline = RoundedCornerShape(8.dp).createOutline(size, layoutDirection, this)
    val path = Path().apply { addOutline(outline) }
    val stroke = Stroke(
        cap = StrokeCap.Round,
        width = 1.dp.toPx(),
        pathEffect = PathEffect.dashPathEffect(intervals = floatArrayOf(16.0f, 8.0f))
    )
    this.drawContent()
    drawPath(path = path, style = stroke, color = Color.Gray)
}.padding(4.dp)
```

```
Text(text = "Custom Modifier",
     modifier = Modifier.dashedBorder().background(Color.Yellow),
     fontSize = 32.sp )
```

Custom Modifier

- Direzione viene presa dalla direzione che arriva nella lambda → crea outline
- Outline trasformata in path → per disegnarlo
- Stroke

*Composable possono essere **statefull** → dipendono dai dati → cambiano*

Quando i dati/parametri/stati_interni cambiano, riceve una notifica e deve ridisegnarsi → **Recomposition.** → Android reinvoca la funzione composable con i dati aggiornati.

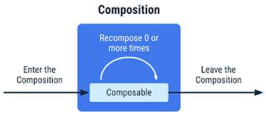
***NOTA**: non posso usare le variabili locali per storare lo stato in quanto scompaiono e le variabili globali sono inadeguate in quando ci sarebbe facilmente incongruenza a causa del parallelismo.*

Android prova ad essere conservativo → prova ad evitare ricomposizione per elementi che non cambiano.

→ No side effect

Recomposition skips as many composable functions and lambdas as possible, based on parameters that did not change

- Moreover, it is optimistic, expecting input parameters not to change before the recomposition phase ends
- In case a change occurs while a recomposition is ongoing, the recomposition is cancelled, the provisional view tree is discarded, and the whole process is started again



Nonostante ciò si possono avere dei side effect nelle callBack → es: OnClick

Android fornisce un container MutableState<T> → monade che wrappa ogni tipo di dato:

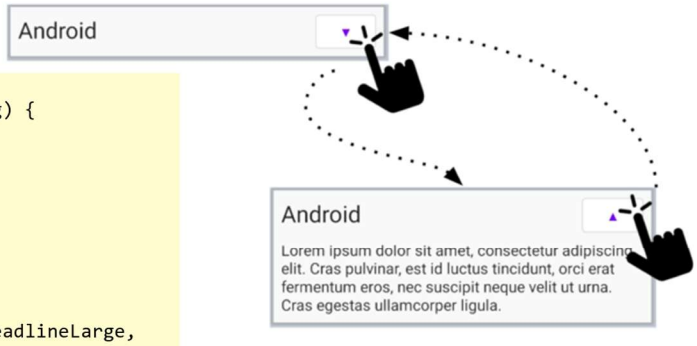
- **State**<T> :incapsula un pezzo di dato
- **remember**(..) : permette di tracciare ogni modifica e chiamare il processo di ricomposizione
  - `val mutableState = remember { mutableStateOf( initialValue ) }`
  - `var value by remember { mutableStateOf( initialValue ) }`
  - `val (value, setValue) = remember { mutableStateOf( initialValue ) }`

```
@Composable
fun MyGui() {
    var counter by remember { mutableStateOf( value: 0) }
    Column(modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.SpaceEvenly,
        horizontalAlignment = Alignment.CenterHorizontally
        ) { this: ColumnScope
        Text( text: "$counter", style=MaterialTheme.typography.displayLarg
        Button(onClick = { counter += 1 }) { this: RowScope
            Text( text: "+")
        }
    }
```

```
fun MyGui() {
    var visible by remember { mutableStateOf( value: false) }
    Column(modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.SpaceEvenly,
        horizontalAlignment = Alignment.CenterHorizontally
        ) { this: ColumnScope
        Button(onClick = { visible = ! visible }) { this: Row
            Text( text: "Toggle visibility")
        }
        if (visible) YellowBlock()
        else RedBlock()
```

espansione

```
@Composable
fun ExpandableBlock(title: String, body:String) {
  val (expanded, setExpanded) = remember {
     mutableStateOf(false)
  }
  val b = if(expanded) "\u25b2" else "\u25bc"
  Column(Modifier.padding(8.dp)) {
    Row {
      Text(text = title,
           style = MaterialTheme.typography.headlineLarge,
           modifier = Modifier.weight(1f) )
      OutlinedButton(
         onClick = { setExpanded(!expanded) }
      ) { Text(b) }
    }
    if (expanded) { Text(body, Modifier.padding(top = 8.dp)) }
  }
}
```



Input widgets

- **TextField** → blocco dove utente può scrivere qualcosa e nel caso in cui non scriva niente, ci sia un placeholder
  È stateful e deve mantenere lo stato in memoria, se l'utente ha scritto qualcosa, deve mostare quello che ha scritto l'utente con lo stile definito per utente

```
@Composable
fun InputPane(onOk: (v:String)->Unit) {
  val (data, setData) = remember { mutableStateOf("") }

  Row(Modifier.fillMaxWidth().padding(8.dp)) {
    TextField(value = data, onValueChange = setData,
              modifier = Modifier.weight(1f))
    Spacer(modifier = Modifier.width(8.dp))
    Button(onClick = { onOk(data) }) { Text("Ok") }
  }
```

- **Switch** → permette all'utente di scegliere tra due stati
  - Richiede cjecked e onCheckedCange

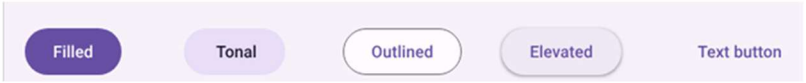
- **Slider** → permette all'utente di selezionare un valore in un range
  - Richiede value e onValueChange

- **RangeSlider** → permette all'utente di selezionare 2 valori in un range
  - Valori ClosedFloatingPointRange<Float>

```
val (value, setValue) = remember { mutableStateOf(0.0f..100f) }
RangeSlider(value = value, onValueChange = setValue,
            steps =  9, valueRange = 0f..100f,
            modifier = Modifier.padding(horizontal = 16.dp)
)
```

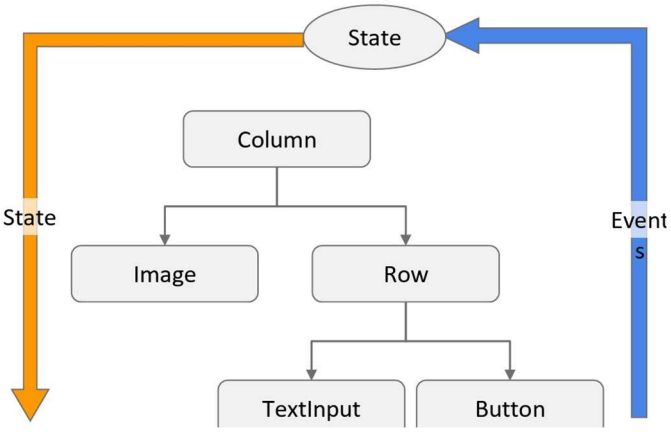
Button widgets: permettono iterazione con utente:

- `Button(...)` classic button with solid background: used for primary actions
- `FilledTonalButton(...)` lighter background: used for significant actions
- `ElevatedButton(...)` cast a shadow to appear prominent: also for significant actions
- `OutlinedButton(...)` has a border but no fill: used for less important actions
- `TextButton(...)` no background or border: used for secondary actions
- All variations requires the `onClick` parameter that specifies the action to execute



Quando si ha uno stato, bisogna capire dove salvarlo (abbiamo capito che non vanno bene variabili locali o globali).

→**UNIDIRECTIONAL DATA FLOW**: lo stato è mantenuto al top dell'albero della logica ed è passato come parametro ad ogni elemento figlio che necessita accedere e ogni funzione. Ogni cambiamento è una conseguenza dell'iterazione con utente, nel caso di cambiamento il top component sarà recomposed e la modifica sarà propagata ai figli.



Quando si crea un'attività, l'attività è fornita con dei **viewModel** che possono essere acceduti da ogni composable. ViewModel contengono lo stato e metodi che modificano lo stato.

Quando un'attività inizia, il viewModel viene creato, se un'attività viene distrutta → viewModel distrutto, Nel caso in cui un'attività viene ditrutta per essere successivamente ricreata → recompose (rotazione schermo), ViewModel non viene distrutto.

- Storiamo mutable state

Material library fornisce guidelines per stile e definire cose.

- Designer sceglie max 2,3 colori.
  - Il primary color deve essere usato per button, headlines, top barr.
  - Secondary Color: molto distante dal primary color → per azioni che devono essere totalmente diverse dal normale.
- UI contiene un insieme di file modificabili