IR compiler & kotlin-wrapper migration guide:

For basic instructions take a look at:

- https://kotlinlang.org/docs/js-ir-migration.html
- https://kotlinlang.org/docs/js-ir-compiler.html
- https://plugins.jetbrains.com/plugin/17183-kotlin-js-inspection-pack/

When using kotlin wrappers with an older version than pre.282:

- Update Kotlin-wrappers (e.g. kotlin-react & kotlin-react-dom) before migrating to the IR compiler because there can be some compatibility issues between IR and kotlin-wrappers when trying to use older legacy wrapper versions (Tested with pre.212)

Migration steps:

- Make interfaces external with structural replace
    - Be careful to not forget interfaces that inherit e.g. from e.g. “FlexboxProps” instead of “Props”
- Put members of external interfaces in a separate config class to circumvent external interface restrictions (receiver function, constructor, val instead of var etc.)
- Rename “Config” class to “<ComponentName>Config” and then move nested classes out of external interface
- Kotlin wrapper migration refactorings
    - Remove attrs. for component properties because it’s no longer required for new DSL
    - jsObject -> jso
    - RBuilder -> ChildrenBuilder

Legacy with RBuilder:

```
private val styled = withStyles<MbLinearProgressProps, MbLinearProgress>(style)

fun RBuilder.renderMbLinearProgress(config: MbLinearProgressConfig) = styled {
 attrs.config = config
}
```

New way with ChildrenBuilder:

```
fun ChildrenBuilder.renderMbLinearProgress(config: MbLinearProgressConfig) {
 MbLinearProgress::class.react {
   this.config = config
 }
}
```

The instantiation stays the same:

```
renderMbLinearProgress(config = MbLinearProgressConfig(show = true))
```

- RState -> State
- RProps -> Props
- RComponent -> Component (Class components: Use custom RComponent for syntactic sugar like it was with RBuilder before)
- Context.Consumer different usage

Legacy:

```
themeContext.Consumer { theme ->
  // Use theme
}
```

New way:

```
themeContext.Consumer {
  children = { theme ->
    // ‘create’ returns a ‘ReactElement’ which ‘children’ expects as a return type
    Fragment.create {
      // Use theme
    }
  }
}
```

As this can get inconvenient really quick, we inject the context instead:

```
companion object : RStatics<dynamic, dynamic, dynamic,
dynamic>(SomeComponent::class) {
 init {
   this.contextType = appContext
 }
}

private val appContext get() = this.asDynamic().context as AppContext
```

- buildElement & buildElements {...} → Fragment.create {...}
- buildElements → createElement(Fragment, jso {}, Fragment.create { ... })
- RClass<IconProps> -> SvgIconComponent
- Since withStyles is legacy and not part of kotlin-mui, use the sx prop for styling
  - Div and other tag definitions no longer allow specifying a class via parameters -> Style classes from a CSS file can now be set like:

```
className = ClassName("someClass")
```

- If a style is used multiple times, define a receiver function in the component that can be used in all sx props like:

Define style:

```
 private fun PropertiesBuilder.drawerStyle() {
   width = 240.px
 }
```

Usage:

```
sx {
 drawerStyle()
}
```

- When using media queries keep in mind that they are only assigned once and defining them multiple times leads to just the last one being used

Wrong: If both if are executed only the styles of the second one will be used

```
sx {
  if (!props.config.hideDrawer) {
     (theme.breakpoints.up(Breakpoint.md)) {
        marginLeft = drawerWidth.px
     }
  }

  if (props.config.smallToolbar) {
    (theme.breakpoints.up(Breakpoint.md)) {
       minHeight = 100.vh - 12.px
       marginLeft = 240.px
    }
  } else {
    (theme.breakpoints.up(Breakpoint.md)) {
       minHeight = 100.vh - 64.px
       marginLeft = 240.px
    }
  }
}
```

Correct: All styles are set in the same media query block

```
sx {
  (theme.breakpoints.up(Breakpoint.md)) {
    if (!props.config.hideDrawer) {
       marginLeft = drawerWidth.px
    }

    if (props.config.smallToolbar) {
       minHeight = 100.vh - 12.px
       marginLeft = drawerWidth.px
    } else {
       minHeight = 100.vh - 64.px
       marginLeft = drawerWidth.px
    }
  }
}
```

- Styling of `InputProps` of MUI `TextField` needs to be done a bit different now because the style cannot be set in the `InputProps` layer:

Legacy:

```
textField {
  attrs.placeholder = "1234"
  attrs.InputProps = js {
    classes = js {
      input = props.classes.codeTextField
    }
  }
}

...
// A class that is used via `withStyles`
codeTextField = js {
      width = 80
      this["&::-webkit-inner-spin-button"] = js {
        this["-webkit-appearance"] = "none"
        margin = 0
      }
      this["-moz-appearance"] = "textfield"
    }
```

New way:

```
TextField<StandardTextFieldProps> {
  variant = "standard"
```

```
  placeholder = "1234"
  sx {
    // Select class via typed class of MuiInput
    MuiInput.input {
      width = 80.px
      // Non typed pseudo elements can be selected by invoking a String
      "&::-webkit-inner-spin-button" {
        WebkitAppearance = None.none
        margin = 0.px
      }
      MozAppearance = Appearance.textfield
    }
  }
}
```

- Some components like Autocomplete require the type of the options to be defined like `Autocomplete<AutocompleteProps<String>>` - This can be seen when looking at the definition. But even if one does not specify the type in the first place an error will pop up in the code and also during compilation.
- `unsafe` attribute of HTML tags not longer exists in the new Kotlin wrapper versions

Legacy:

```
p {
  attrs.unsafe { +item.tip }
}
```

New way:

```
p {
  dangerouslySetInnerHTML = jso { __html = item.tip }
}
```

- Compiling a Kotlin/Js project with the legacy compiler and DCE (processDceKotlinJs) results in the following error:

```
> Task :processDceKotlinJs
error: at /xxx/AriaHasPopup/build/js/packages/AriaHasPopup/kotlin/AriaHasPopup.js
(14, 38): invalid property id
> Task :processDceKotlinJs FAILED
```

This happens due to `JS reserved keywords` being used in `@JsName` for wrapped JS values.

Fix: Add [legacy-union plugin] to build.gradle.kts like:

```
id("io.github.turansky.kfc.legacy-union") version "5.8.0"
```

- When something is unclear and not described in this document -> Search in CampusQR because many common problems have already been handled there
    - For comparison: Last commit before the migration is 8edf8f37c510216389fc9d45815ba639ce235f6f

kotlin-mui issues:

- Some MUI components take a type parameter to specify the property. (E.g. TextField<OutlinedTextFieldProps>) This leads to the error "Type argument is not within its bounds.Expected: ChildrenBuilder Found: OutlinedTextFieldProps" because the context changes in the TextField -> Adding the *@Suppress("UPPER_BOUND_VIOLATED")* annotation mutes this. → Will be solved with context receivers in the future
- Using class selectors in the sx prop like MuiToolbar.root {…} may cause hierarchical issues -> If something does not look as expected try without the class selector and check the dom tree in the browser to investigate the issue
- Theme can no longer be accessed via withStyles
    - → Pass it down via a context when using class components
    - → Use useTheme hook for functional components
- When unclear how to do something specific regarding the kotlin wrapper -> Clone Kotlin-wrappers repo and search for it. E.g. "How to specify "!important"? -> Search for "!important" and you will find the corresponding function (Quite often this is the only viable way since there's not a lot of documentation to refer to)
- Avoid using asDynamic() in normal code because it's technical debt that will most likely require additional work in the future -> Make e.g MuiHelper file and put needed extension functions in it.
    - Once the desired property or element is added to the kotlin wrappers, maintenance work consists of just removing the function from MuiHelper because the call site doesn't change.

Ideally a typed solution is possible like with value here in order to access event.target.value in e.g. onChange function of TextField:

```
val HTMLElement.value: String
 get() = when (this) {
   is HTMLInputElement -> value
   is HTMLTextAreaElement -> value
   else -> throw IllegalArgumentException("There is no value in this HTMLElement")
 }
```

Unfortunately a typed solution is not always possible. The fallback solution is to just use asDynamic() in the receiver function to at least prevent writing asDynamic() outside of the helper functions:

```
var InputBaseComponentProps.min: Any?
 get() = asDynamic().min
 set(value) {
```

```
    asDynamic().min = value
  }
```

- jsStyle can be replaced by using the sx prop of MUI
- div/span/img etc. now need to be refactored to MUI Box component when wanting to style it with the sx prop

Legacy style:

```
img {
  attrs.jsStyle {
    width = 48
  }
  src = props.config.logoUrl
  alt = props.config.logoAlt
}
```

New way:

```
Box {
  component = img
  sx {
    width = 48.px
  }
  // Casting is required to get img attributes
  this as ImgHTMLAttributes<HTMLImageElement>
  src = props.config.logoUrl
  alt = props.config.logoAlt
}
```

Sometimes it might be necessary to target nested classes, which is (at least not yet) a feature of the Kotlin wrappers. To achieve this in a typed way we defined a helper function for that:

```
fun PropertiesBuilder.nested(className: ClassName, style: PropertiesBuilder.() ->
Unit) {
  (Selector("&.${className}")) {
    style()
  }
}
```

Which can then be used like:

```
ListItemButton {
 sx {
   nested(Mui.selected) {
     backgroundColor = Color(“black”)
   }
 }
}
```