Upgrading user interfaces for the future

Kongsberg is a company in the maritime industry – it is heavily regulated and in general, it does not spend too much time on 'how things look' - as long as the solution is functional. In the past, large legacy desktop systems have been built for Kongsberg's maritime simulation and training division. These systems use WPF (or older!) to show and control its state.

Author Albert Brand

Recently Kongsberg started to deliver a cloud-based training platform for maritime students, in which a view on the ship's bridge with all instruments is accessible from a browser. Together with the transformation to a web platform there was a great opportunity to rethink how to compose the user interface with reusable elements, how these user interfaces are connected to the simulation services, and how to achieve a maintainable system that may be compiled into something entirely different in a couple of years.

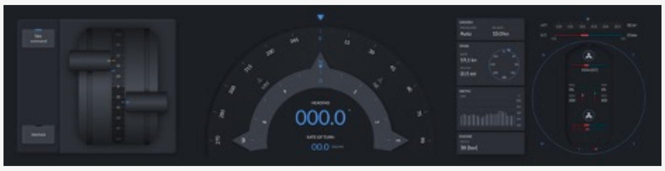
This article will discuss some details of this transformation to the web. If you are interested in the 'cloud side', make sure to read the article by Roy Cornelissen and Sander Aernouts in this magazine.

Rethinking the design

Kongsberg brought in a design agency to create a fresh new look for their entire simulation product suite called K-Sim. This covers:

> the simulated ship controls called **instruments**;
> the virtual ship's bridge where these instruments are shown to the user called **PanoramaWeb**;
> the portal to start a simulation, see assessment results, and buy licenses for specific instruments called **Connect**.

Initially there was a focus on the instrument design. These were crafted as a replica of the physical world (which is called skeuomorphic in experts terms). However, after several iterations it became clear that in some cases, a real life design is hard to manipulate using a display or touch screen. Also, creating components from these designs was deemed to be pretty complicated (although we managed to deliver some!).



After a number of iterations, the agency took these learnings and they made the distinction between **replicas**, **abstractions** and **digital screens**. When a physical replica is too constraining, abstractions are used to present a design that is recognizable but does not exist in real life. For example, the heading repeater instrument has traits of a compass rose that add a relation to its functionality. The third distinct design type is digital screens. Today, some instruments on a ship already use a touch screen instead of a custom hardware panel. It makes sense to give a similar representation to a student.

Creating composable UI elements

I joined the K-Sim Connect team in April 2020 as a Xebia frontend architect. One of the goals was to coach the current team in building modern web frontends. Of course they also wanted me to help build some of the user interfaces, fast! That seemed like a job that suited my skills pretty well.

Some teams already created web versions of instruments (before they hired a design agency). The instruments were built using vanilla JavaScript with CSS and did only use some really low-level libraries such as jQuery to help render the output. The build quality of the components was lacking in several areas: minimal tests, no proper separation of concerns, no reusable parts and of course the visual design was pretty old-school as well.

Together with one of the simulation software architects of Kongsberg I discussed several topics:

> we should create small components to compose larger ones;
> the components should use a modern web standard to expose and isolate itself, and allow for data ingestion and event publishing;
> we shouldn't build everything ourselves but use the best libraries out there to achieve it.

The architect was also thinking about a domain-specific language that expresses how the user interface is laid out in a platform-independent manner. He liked what he heard about *Web Components*[1] as it is the official set of web standards for creating components that encapsulate their presentation and behavior.

So we went forward and started to create a Web Component library based on the initial designs, with many composable elements such as buttons, areas and text elements. But what does composability mean in the context of a user interface? To give an example, let's say that you want to show a big button with a flashing text on it. One way of building such a component is by creating a new one from scratch with exactly that behavior. However, such a solution does not scale: you're probably copy-pasting parts of a similar button, and you need to repeat that process over and over again for new variants of the button. An improved way would be to add parameters to an existing button, such as "size" and "flashing". However, that would still not scale very well, as your component would keep on growing with all kinds of variations which get harder and harder to reason about, let alone write tests for all permutations.

A better way to solve this is by creating an extensible component, which allows for injecting other components that only bother about their own concerns. For instance, the flashing button could be created by the following structure:

```
<StyledButton>
  <Flashing colors="[red,white]">
    <SimpleText size="big">
      Emergency!
    </SimpleText>
  <Flashing>
<StyledButton>
```

And this is exactly what you can do with web components. It offers you custom elements that provide a 'slot' mechanism to pass in other elements, making your components composable from smaller parts.

[1] https://developer.mozilla.org/en-US/docs/Web/Web_Components
[2] https://open-wc.org
[3] https://lit-element.polymer-project.org

Libraries? Yes please.

While implementing the first components it became clear quickly that the Web Component standard is a little bare-boned. This is actually often the case for web standards in general: the standard committee is pressed to agree on a generic solution, and they often choose low-level APIs.
It is up to the web community to pick them up and use them as a foundation for modern libraries.

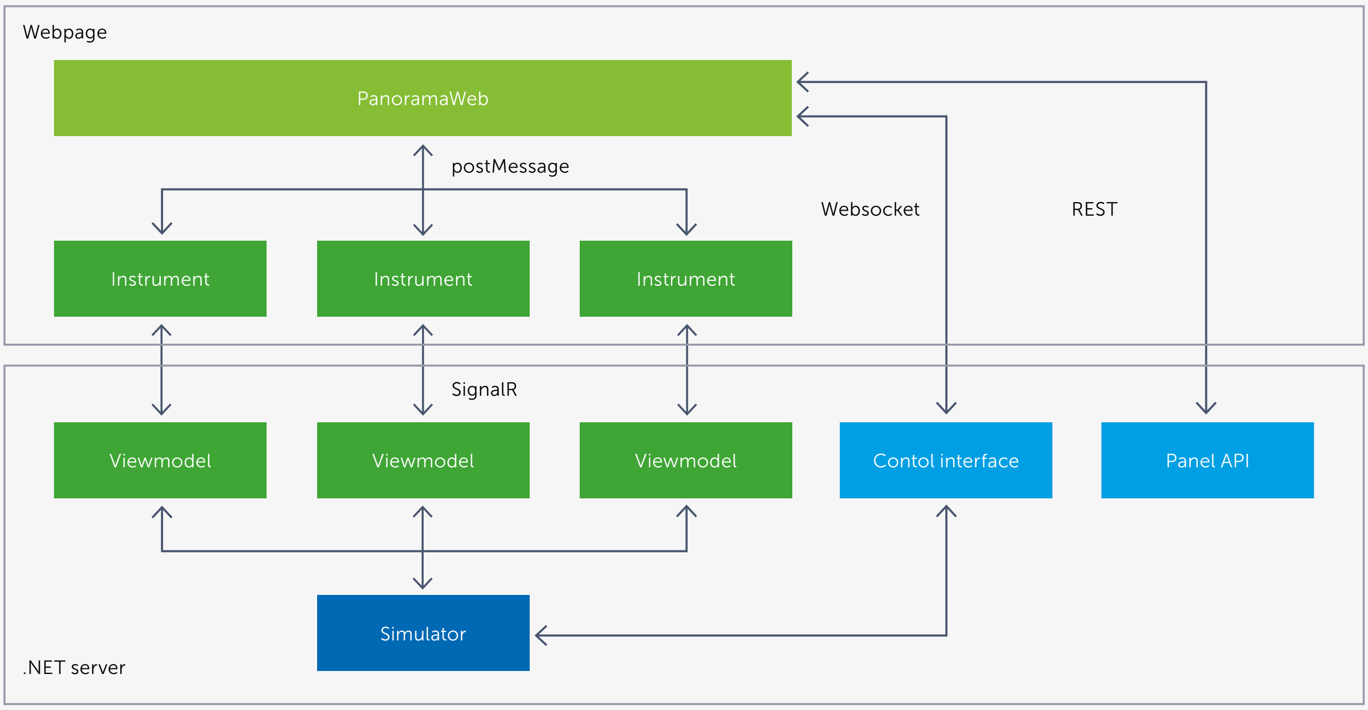
Many of the existing frameworks such as React, Vue.js and Angular offer a way to perform a special build that wraps components as custom elements. However, this comes at the cost of having to ship relatively large libraries, just to draw a single component. So we looked at alternative frameworks and libraries to create web components while adopting a modern approach, but without too much extra overhead.

The choice quickly became clear: we wanted to follow the recommendations from *Open Web Components*[2], a collective of web components enthusiasts. These recommendations provide a powerful and battle-tested setup for creating and sharing web components. It recommends the *LitElement*[3] library for building web components, the successor of the Polymer project, which pushed the Web Component standard initially.

Presenting the ship's bridge in a browser

While building the shared component library, work was underway to build a new version of the PanoramaWeb web application to show the overview of instruments to the user in a modern way. As PanoramaWeb is a single page app that shows the 'chrome' around instruments, it was not necessary to build this as a web component. Instead, I opted to use Vue.js, as it an easy to pick up framework for building large component-oriented user interfaces.

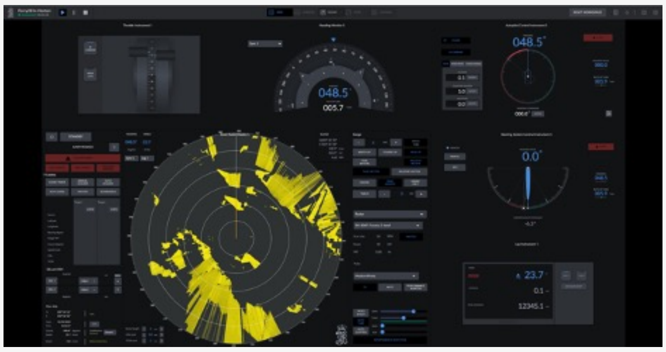
PanoramaWeb initially retrieves the instruments it needs to show via a panel API. When the instruments are loaded, the app has some high-level control over the simulator. It can start and stop the loaded exercise and show the simulated time, which is presented in the top bar. This communication is done over a bidirectional stream of events that is exposed via a Websocket connection. In addition, each instrument connects to its own server-side view model instance using SignalR.
And if that is not sufficient, each instrument can communicate with whatever service it wants, and with any protocol that is required for it. You can read about how the radar instrument uses a WebRTC stream for bringing the radar display to life in the article by Roy and Sander in this magazine.



One of the challenges was to build a view to show and interact with the various instruments, while also being able to resize them and reorder them using drag and drop. As the instruments are built as separate web pages, it made most sense to use plain iframes to show the contents. iframes have a long history, as they have been one of the first browser features. They allow you to load and show two or more different pages of content in a single view, which means that they are a good candidate to 'stitch' multiple instruments together in a unified view.

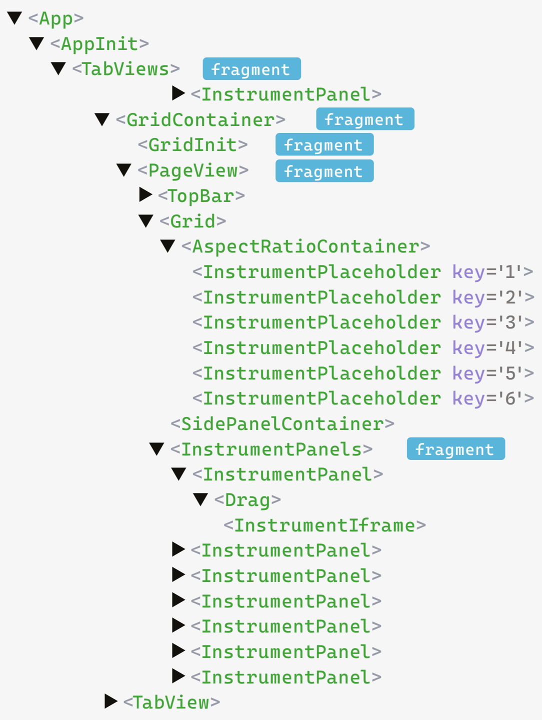
Of course, there are other ways of combining multiple elements on a single page. You can choose to create large components that are then loaded on a single page. You could even use custom elements as a boundary for communicating between components. However, you need to make sure that these components have separate styles and dependencies, otherwise one component could influence another component in unexpected ways. And given the output of the in-house tool (which you'll read about shortly), I opted to go for iframes.

It took some sweat and tears, but PanoramaWeb started to shape up nicely after some time.



Dragging and dropping iframes that are holding those instruments did become a hassle at some point. iframes are quite limited; partly because of security concerns (you can load a page from a different domain so a browser needs to be very careful in sharing information between both), partly due to standardization reasons (it's just an element that shows a page in another page and that's it). And for some historical reason, if you move an iframe element to another parent element (which I implemented as a naïve first approach), the iframe contents will be reloaded. Even though this was not really a functional problem (the page is initially synced with its server view model), I really wanted to fix this bad user experience issue.

After investigating it became clear that if you want to ensure that iframes don't reload when being dropped in a different place, you should not move them at all in the DOM. Instead, I went for another strategy: when an instrument is visually dropped at a certain position, an InstrumentPlaceholder component is drawn. This component constantly determines its visual size and position on the screen (using the modern ResizeObserver and MutationObserver web APIs) and updates the internal state of PanoramaWeb. Thanks to Vue's built-in reactivity, it was a breeze to let the component that holds the actual iframe to pick up this change and position itself on the placeholder location. This allows for iframes to be placed anywhere in the component tree. Nice!

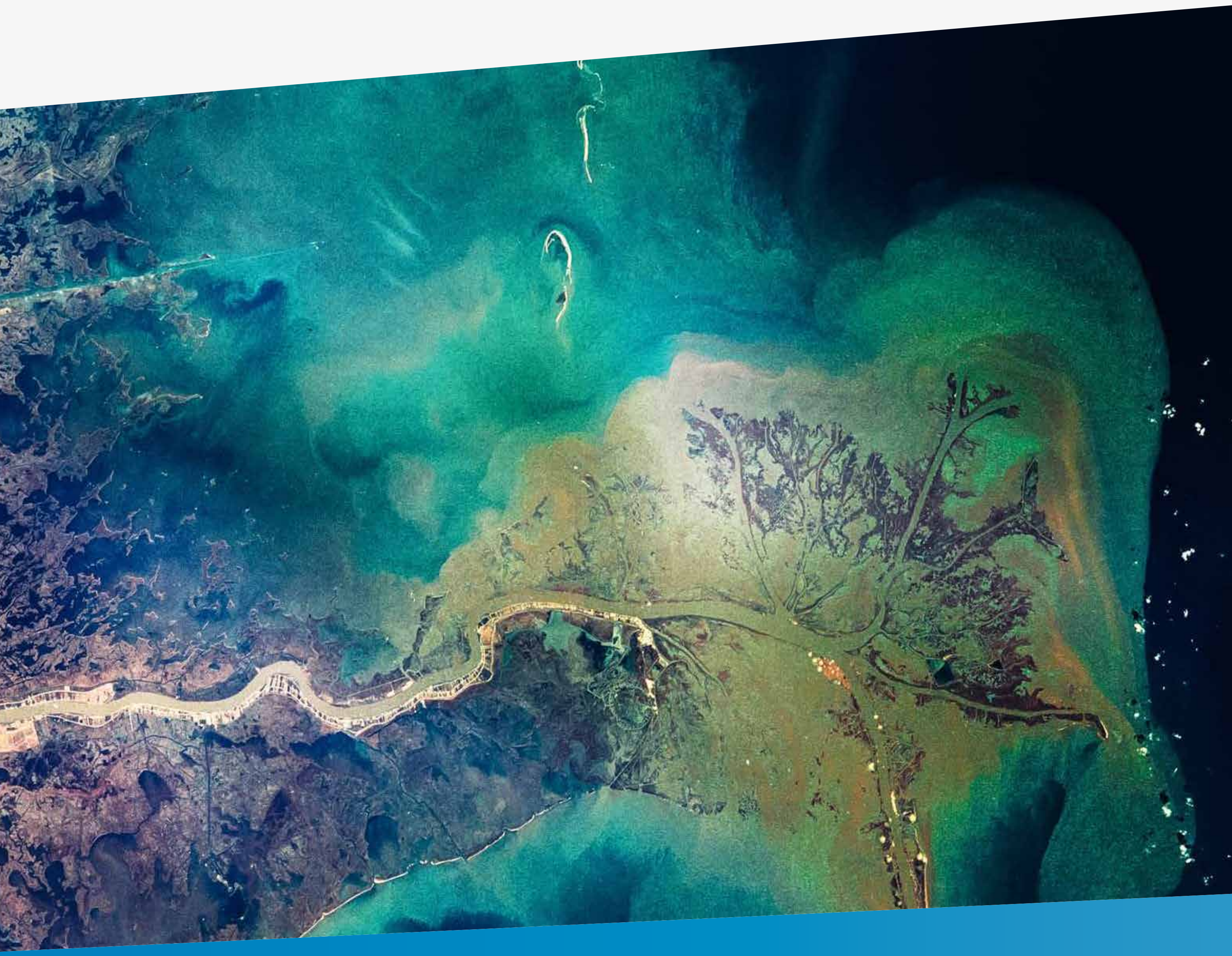


```
▼ <App>
  ▼ <AppInit>
    ▼ <TabViews>  fragment
          ▶ <InstrumentPanel>
        ▼ <GridContainer>  fragment
            <GridInit>  fragment
          ▼ <PageView>  fragment
            ▶ <TopBar>
            ▼ <Grid>
              ▼ <AspectRatioContainer>
                  <InstrumentPlaceholder key='1'>
                  <InstrumentPlaceholder key='2'>
                  <InstrumentPlaceholder key='3'>
                  <InstrumentPlaceholder key='4'>
                  <InstrumentPlaceholder key='5'>
                  <InstrumentPlaceholder key='6'>
                <SidePanelContainer>
              ▼ <InstrumentPanels>  fragment
                ▼ <InstrumentPanel>
                  ▼ <Drag>
                      <InstrumentIframe>
                ▶ <InstrumentPanel>
                ▶ <InstrumentPanel>
                ▶ <InstrumentPanel>
                ▶ <InstrumentPanel>
                ▶ <InstrumentPanel>
                ▶ <InstrumentPanel>
        ▶ <TabView>
```

The tool that ties everything together

While I was having a go at the PanoramaWeb application, the software architect was happily working on a tool that soon would become the official Kongsberg-endorsed way of creating user interfaces for instruments. Mind you, Kongsberg already created hundreds of different simulated instruments, and maintainability is a big concern. Many of these instruments differ widely in style, technology stacks, architecture, layers, initialization and communication. Only giving developers guidelines on how to build user interfaces was not enough to streamline and standardize this process.

A domain-specific language called 'Blueprint' was designed and it allows you to specify how your user interface is built up using components, binding them to certain inputs from the view model (even with complex expressions), and listen to output of these components. The tool, which is written in .NET Core, can load libraries of components and compile a Blueprint file to an actual web page (including CSS and JS dependencies) that is ready to be served as part of the extension for the web server application.



In theory this tool could be used to output something completely different: a native desktop user interface, or a virtual or augmented reality variant. The possibilities are pretty much endless, however that is a chapter that still needs to be written.

We proposed numerous enhancements such as file includes with parameterization that found its way into the tool. At some point I even created a Visual Studio Code extension to syntax highlight the Blueprint file contents. My fellow teammates who wrote a lot of Blueprint code were very happy with that, as code readability is improved a lot this way. And of course, you get pretty bored looking at grey code all day long...

autopilot.blueprint

```
angle $(Heading)
towards-angle $(HeadingOrder)
allow-drag $(InstrumentPower) and $(InCommand)

# heading
  group
    offset 0, -20

   label-text
     text 'HEADING'
     font-size $(FontSize)

   group
     offset 7.5, 4

     readout-text
       text $(HeadingAsString)
       horizontal-align 'right'
       font-size $(FontSizeXL)
       status 'highlight'

     readout-text
       offset 1, -2
       text 'O'
       font-size 3.5
       status 'highlight'

# heading command
  group
    offset 0, 17.5

    include "autopilot-field.blueprint-part"
      $(Disabled) = not $(InCommand)
      $(Label) = 'HEADING COMMAND'
      $(FieldOffsetX) = -1.5
      $(Flashing) = $(HeadingOrderFlashing) and $(BlinkSync)
      $(EditableText) = $(HeadingOrderReadout)
      $(EnterPushed) = $(EnterHeadingOrder)

    label-text
      offset 0.5, 2.5
       text 'O'
       font-size 2.5

# mode selectors
  group
    offset -37, -17.5
```

In conclusion

We Xebians have been trained to aim for the sky and see problems as opportunities, not as roadblocks. However, other developers might not have that mindset. Learning a new library such as LitElement or a tool as Blueprint takes time, and you need to constantly remind yourself to take a step back, keep explaining when something is unclear, and in the end let others learn by doing, and stop 'holding their hand'.

Luckily, the approach that we kickstarted is being picked up, and more and more teams are now investing in learning and embracing that modern stack. There will always be growing pains, but teams are pretty happy so far.

So there you have it, a 'blueprint' of the future of Kongsberg user interfaces. I honestly believe that thanks to the chosen modern standards such as Web Components and the effort that is going into the Blueprint tool, Kongsberg does not have to invest in rebuilding their user interfaces every two years.

And the future looks bright as well. The adoption of the cloud e-learning environment is rising and demand for more teaching scenarios is clearly visible. Who knows which products will see the light of day and set a high bar for what you can do with an 'ordinary' browser and the cloud? </>

Albert Brand
Core Development lead from
Xebia Software Development