SA
Documentation

Swisscom Design System - Server-Side-Rendering

Semester: Autumn 2023

Version: 1.0
Date: 2023-12-22 14:52:12+01:00

Project Team: Natalia Gerasimenko
Tim Gamma
Project Advisor: Markus Stolze

School of Computer Science
OST Eastern Switzerland University of Applied Sciences

Contents

I Developer Documentation **1**

1 Developer Documentation **2**

1.1 Stencil SSR – Basic setup with Express 2
1.2 Workaround – Wrong order of elements after renderToString() 4
1.3 Old alpha release of StencilJS - SSR patches 8
1.4 Integrate Stencil Web Components into an Angular Application 9
1.4.1 Setup 9
1.5 Integration into Next.js 13
1.5.1 Tabs integration into Smartive company example 13
1.5.2 Additional integration test with mayerraphael example 16
1.6 Performance Analysis 17
1.7 Additional information 24

II Appendix **26**

Bibliography **27**

List of Figures **27**

Part I

Developer Documentation

Chapter 1

Developer Documentation

1.1 Stencil SSR – Basic setup with Express

Repository: basic-setup-with-express

Background

Our research and experimentation have led us to understand the significance of loading module scripts on the client side after utilizing `renderToString()`. This approach is crucial for enhancing the interactivity of StencilJS components.

Prerequisites:

- Builded Stencils component package
- NodeJS with Express.js as a server

Implementation Approach

Import hydrate script of stencil components into `app.mjs` located `sdx-ssr` project:

Listing 1.1: app.mjs

```
    import { renderToString } from 'mini-sdx/hydrate/index.js';
```

After importing the hydrate script the esm script must be integrated in the rendered HTML.

Options for Script integration:

Option 1: Appending scripts after render

Before delivering the results to the client, the `replace()` method can be utilised to add a script tag to the HTML rendered by the Hydrate script.

Listing 1.2: app.mjs

```
...
const results = await renderToString(tabs, {
  prettyHtml: true,
  removeScripts: false
});

const withStencilScripts = results.html.replace(
  '</head>',
  <script type="module" src="/scripts/stencil-starter-project-
     name.esm.js"></script>
  </head>`
);

res.send(withStencilScripts);
...
```

Option 2: Rendering HTML with already linked script

Another approach can be used to render HTML with the script already linked. This method involved rendering the page using the `renderFile()` method and ensuring that the ESM script tags are part of the template.

Listing 1.3: app.mjs

```
// Render page using renderFile method
ejs.renderFile('index.ejs', {},
    {}, async function (err, template) {
        if (err) {
            throw err;
        } else {
            const results = await renderToString(template, {
                prettyHtml: true,
                removeScripts: false
            });

            res.end(results.html);
        }
    });
```

Listing 1.4: index.ejs

```
<head>
    ...
    <script type="module" src="/path/to/stencil-esm-script.js"
       ></script>
    ...
</head>
```

Results

Integrating the ESM script using a script tag successfully transformed the components into interactive elements. Notably, the tabs component became clickable, and dynamically set styles functioned as intended. This approach was confirmed by our interactions with the Stencil Community, suggesting its effectiveness and reliability.

1.2 Workaround – Wrong order of elements after renderToString()

Repository: css-workaround-wrong-order-after-render

Background

When the renderToString() function was used for server-side rendering, it occasionally resulted in the incorrect ordering of elements in the DOM. The tabs component, for instance, displayed this issue prominently, causing a disruptive flashing effect as the client-side JavaScript took a moment to reorder the elements correctly.

renderToString result from server:

Listing 1.5: renderToString result

```

<div ... class="sc-sdx-tabs tablist">
    ...
    <button ... >
    <!--t.1.5.4.0-->
      Tab 6
    </button>
    <button ... >
    <!--t.1.7.4.0-->
      Tab 4
    </button>
    <button ... >
    <!--t.1.9.4.0-->
      Tab 2
    </button>
    ...
</div>
```

From our findings, the most effective workaround for this issue was the implementation of CSS. We utilized the CSS Grid model to control the order of elements visually. This approach allowed us to manipulate the display order of the elements without altering their actual DOM order.

Prerequisites:

- StencilJS web components project
- stencil/core 4.8.1

Implementation Approach

viable solution involves generating the `grid-template-areas` property by having prior knowledge of the IDs assigned to elements. Each element can then be assigned a `grid-area` based on its corresponding ID.

To ensure the correct order for `grid-template-areas`, a potential solution involves pre-knowing the IDs assigned to the tabs.

In this example the assigned ID for tab items is in the format "tab1", "tab2" etc. The prefix `tab` is a constant part of the ID, while the numerical value is determined dynamically.

The following steps were taken to achieve the correct order of the tabs.

1. Assining IDs to tabs elements in HTML

Listing 1.6: index.html

```html
<sdx-tabs sr-hint="Tabs with text.">
    <sdx-tabs-item  id="tab1"   label="Tab 1">This is the
        content of Tab 11111.</sdx-tabs-item>
    <sdx-tabs-item  id="tab2"   label="Tab 2">This is the
        content of Tab 2.</sdx-tabs-item>
    <sdx-tabs-item  id="tab3"   label="Tab 3 with very long
        text that will be truncated" selected>This is the
        content of Tab 3.</sdx-tabs-item>
    ...
</sdx-tabs>
```

2. Generating grid-template-areas

Since it is possible to get the number of child elements (tab items), the areas can be generated independently of the number of elements. As established earlier, the prefix for all Tab IDs is "tab," and it is hard-coded. By iterating through the array of tab elements, the loop index was appended to the prefix and pushed into the array of `grid-template-areas`.

Listing 1.7: tabs.tsx

```tsx
    private getTabIds(tabsItemEls: HTMLSdxTabsItemElement[]) {
// return tabsItemEls.map(element => element.id); // WONT work

    let gridAreas = [];
    for (let i = 1; i <= tabsItemEls.length; i++) {
        gridAreas.push('tab' + i)
    }
    return gridAreas;
}

public render() {

    const {
        tabsItemEls,
        selectedTabsItemEl
        } = this.state.get()

    ...
    const elementIdsForGrid = this.getTabIds(tabsItemEls)
    ...
}
```

3. Adding grid-template-area to the parent of tab items

The CSS Grid layout should be added to the parent container of the tabs component. By defining grid areas, it is possible to control the visual placement of child elements.

Listing 1.8: tabs.tsx

```
    return (
        ...
        <div
        class="tablist"
        role="tablist"
        aria-label={this.srHint}
        ref={(el) => (this.tablistEl = el)}
        style={{
          display: `grid`,
          gridTemplateAreas: `'${elementIdsForGrid.join(' ')}'
            `,
        }}
        >
        ...
    )
```

4. Assigning grid-area to each tab item

The style attribute should be set for each child, assigning the `grid-area` based on their respective IDs.

Listing 1.9: tabs.tsx

```
const Tag = tabsItemEl.href ? "a" : "button"

return (
    <Tag
        class={this.getClickableElClassNames(tabsItemEl)}
        style={{gridArea: tabsItemEl.id}}
        ...

    >
        {tabsItemEl.label}
    </Tag>
)
```

Results

This workaround successfully mitigated the visual disturbance caused by the incorrect ordering of elements. The tabs component now maintains a consistent visual order from the moment it loads, significantly enhancing the user experience. While this approach does not solve the root cause of the ordering issue, it provides a stable and effective temporary solution.

The corrected renderToString result from server:

Listing 1.10: renderToString result

```

<div ... class="sc-sdx-tabs tablist" style="display: grid;
   grid-template-areas: 'tab1 tab2 tab3 tab4 tab5 tab6';">
    ...
    <button ... style="grid-area: tab6;">
    <!--t.1.5.4.0-->
    Tab 6
    </button>
    <button ... style="grid-area: tab4;">
    <!--t.1.7.4.0-->
    Tab 4
    </button>
    <button ... style="grid-area: tab2;">
    <!--t.1.9.4.0-->
    Tab 2
    </button>

</div>
```

Future Steps

We recommend continued investigation into the behavior of the renderToString() function to address the root cause of the element misordering. Collaborative efforts with the broader StencilJS community or experimentation with alternative server-side rendering techniques could provide a more permanent solution.

1.3 Old alpha release of StencilJS - SSR patches

Background

In our exploration of StencilJS Server-Side Rendering (SSR), we have identified potential challenges, particularly in relation to the `renderToString()` function. Despite our efforts, we could not find any specific resources that precisely address the particular issues such as:

wrong order rendered by renderToString `select` attribute, initially set in HTML to the tab item, is lost during pre-render. 1.1

According to the StencilJS community, there are several ongoing issues with Server-Side Rendering (SSR) at the moment. An older alpha release, `@stencil/core@3.3.0-dev.1685496483.a31` was mentioned, which includes several SSR fixes. However, it has not yet been merged into the main version due to pending substantial pull requests.

Prerequisites: To experiment with this release, we used the following command:

```
npm install @stencil/core@3.3.0-dev.1685496483.a311f36
```

Experimenting with the Old Alpha Release

The objective was to determine if this version could resolve some of the SSR-related issues we were facing. After building web components with this specific release, we observed the following:

- **Select attribute behavior**: The previously encountered problem, where the select attribute was lost during pre-rendering, appeared to be resolved in this version. This was evident as the initially set HTML attribute for the tab component was now correctly retained post-rendering, as shown in Figure [fig:tab-selected].
- **Element order issue**: However, the issue of incorrect element order during initial rendering persisted. The "flashing" effect, where components momentarily displayed in the wrong order, was still present, indicating that this release did not fully address all the SSR issues.

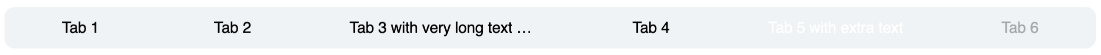
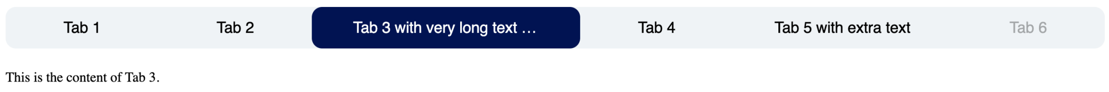


Figure 1.1: Initially selected tab not selected



This is the content of Tab 3.

Figure 1.2: Initially selected tab selected

Conclusion

The exploration of the old alpha release of StencilJS provided mixed results. While it resolved the issue with the select attribute, the element order problem during SSR remains unresolved. This highlights the complexity of SSR in web component frameworks and the need for continued research and collaboration with the StencilJS community to address these challenges fully.

1.4 Integrate Stencil Web Components into an Angular Application

The purpose of this guide is to show how it is possible to integrate Stencil web components into an Angular application, using Stencils output target and SSR.

Repository: Stencil web components SSR in Angular (repo)

Prerequisites:

- Stencil web component(s)
- Angular application

Overview
Stencil is able to generate Angular component wrappers for the web components. This makes it possible to use Stencil web components inside an Angular application.
It is also possible to integrate the web components without these wrappers, however using the wrappers comes with multiple benefits:

- Change detection detachment, this prevents unnecessary repaints of the web components
- Events will be converted to RxJS observables in order to align with Angular's @Output() and there are no emissions across component boundaries

ngModel can be used

1.4.1 Setup

To use these wrappers, changes to the Stencil web components and the Angular application have to be made.

Changes to Stencil Web Components

1. Install Angular output target

```
npm i @stencil/angular-output-target
npm i @stencil/sass
```

2. Adjust stencil.config.ts

Next, add the Angular output target and the hydrate output target to:

```
stencil.config.ts
```

The file should look like the following:

Listing 1.11: stencil.config.ts

```
1   import { angularOutputTarget, ValueAccessorConfig } from '@stencil/angular-output-target';
2   import { sass } from '@stencil/sass';
3   // existing imports
4
5   const angularValueAccessorBindings: ValueAccessorConfig[] = [];
6
7   export const config: Config = {
8     namespace: 'my-stencil-webcomponents',
9
10    outputTargets: [
11      // other output targets
12      {
13        type: 'dist-hydrate-script',
14      },
15      angularOutputTarget({
16        componentCorePackage: '@web-components/dist/components',
17        directivesProxyFile: './../stencil-angularapp-demo/src/libs/stencil-generated/proxies.ts',
18        valueAccessorConfigs: angularValueAccessorBindings,
19      })
20    ],
21    plugins: [sass()],
22  };
```

Remarks:

- Lines 1-2: New imports
- Line 5: Property binding
- Lines 12-14: Hydrate folder output target, that is needed later for SSR
- Lines 15-19: Newly defined Angular Output target
- Line 29: Points to the dist/components folder of the web component (wherever it is located)
- Line 30: This is a new folder, in the Angular application, not yet defined (used later to import the component into Angular)

- Line 21: Plugin sass, needed for correct styles in Angular application

With these changes, the Stencil web components can be imported into an Angular application.

Changes to Angular Application

1. Changes to tsconfig.json

First, add a relative path to the **dist directory of web-components** to the

```
tsconfig.json
```

Listing 1.12: Relative path to the dist directory of web-components

```
"paths": {
    "@web-components/*": ["../web-components/*"]
}
```

This will keep the stencil component as its dependency.

2. Export Component

The goal now is to export the component inside the libs directory.
For this, two things must be added to the Angular application inside the libs directory:

- stencil-generated (empty folder, will be automatically populated by the defined Angular output targets)
- web-components.module.ts (ts-file, to export module)

In the web-components.module.ts add the following code, to export the imported web components:

Listing 1.13: web-components.module.ts

```
import { NgModule, Inject, PLATFORM_ID } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MyWebcomponent } from "../../src/libs/stencil-generated/proxies";
import { isPlatformBrowser } from '@angular/common';

@NgModule({
    declarations: [MyWebcomponent],
    imports: [CommonModule],
    exports: [MyWebcomponent]
})
export class WebComponentsModule {
```

```
12          constructor(@Inject(PLATFORM_ID) private platformId:
                Object) {
13              if (isPlatformBrowser(this.platformId)) {
14              import('@web-components/loader').then(module => {
15                  module.defineCustomElements(window);
16              });
17              }
18          }
19      }
```

Replace **MyWebcomponent** with any web component that should be included.
Note that lines 12-18 are necessary, as in Angular, when employing server-side rendering (SSR) or pre-rendering, the server does not recognize browser-specific objects like window. This happens because server-side rendering in Angular is executed in a Node.js environment, which is different from the browser environment.
After this, the general setup is complete. Run

```
npm run build
```

inside the web-component folder, to populate the Angular output targets that were just defined.
Now, the imported web-components can be used in any Angular module. For it to work, import the following:

```
import { WebComponentsModule } from 'src/libs/web-components.module';
```

Now, the web-components can be used. However, they are not yet rendered server-side.

3. Enable SSR

To enable SSR (non-destructive), run:

```
ng add @nguniversal/express-engine
```

Import the provideClientHydration function as the provider of AppModule:

Listing 1.14: app.module.ts

```
import {provideClientHydration} from '@angular/platform-
    browser';
// ...

@NgModule({
    // ...
    providers: [ provideClientHydration() ],  // add this
        line
    bootstrap: [ AppComponent ]
})
export class AppModule {
    // ...
}
```

This will not yet work correctly, because the hydration process of Angular does not know what to do with the imported Stencil web components, as it is using custom tags.
For this reason, first hydrate the Stencil web components and then hydrate the Angular application. In the Angular application, in the server.ts file modify the app function in the following way:

Old:

Listing 1.15: server.ts (old)

```
    server.get('*', (req, res) => {
        res.render(indexHtml, { req, providers: [{ provide:
            APP_BASE_HREF, useValue: req.baseUrl }] });
  });
```

New:

Listing 1.16: server.ts (new)

```
    server.get('*', (req, res) => {
        res.render(indexHtml, { req}, (err, html) => {
          renderToString(html).then(({html}) => {
            res.send(html);
          });
        });
      });
```

This makes sure that the Stencil web component is loaded first.
Now everything is working, run

```
npm run dev:ssr
```

again to test the application.

1.5 Integration into Next.js

1.5.1 Tabs integration into Smartive company example

Repository: nextjs-smartive-integration

Background

In an effort to integrate Stencil web components with Next.js, including Server-Side Rendering (SSR), we encountered compatibility issues between the latest versions of Next.js (13 and 14) and Stencil. This was corroborated by insights from an external agency, Smartive, which has developed a tool for converting the shadow DOM to declarative, as seen in their GitHub repository.

The objective was to evaluate the feasibility of integrating Swisscom's components into the Smartive example of StencilJS and Next.js with SSR.
The experiment aimed to assess whether Swisscom could use the agency's tool or if there was a need to develop a custom solution.
Prerequisites:

- Smartive repository
- mini-sdx repository

Integration approach

The structure of the `smartive` repository:

- web-components – Stencil components
- web-components-react-wrapper
- app – Next.js application

1. Adding tabs components to smartive components

Tabs components including all styles and utilities shoudl be copied from the `mini-sdx` repository to the `smartive` components. 1.17
To align with Smartive's naming conventions, the `sdx` prefix was replaced with `abc`.

Listing 1.17: packages/app/src/components/tabs.tsx

```tsx
...
import { AbcTabs, AbcTabsItem } from "abc-web-components-react-wrapper";
import { AbcWrapper } from "abc-web-components-react-wrapper/client";
import { FC } from 'react';

export const Tabs: FC = () => (
  <AbcWrapper>
    <AbcTabs sr-hint="Tabs with text.">
      <AbcTabsItem id="tab1" label="Tab 1">
        This is the content of Tab 11111.
      </AbcTabsItem>
      <AbcTabsItem id="tab2" label="Tab 2">
        This is the content of Tab 2.
      </AbcTabsItem>
      <AbcTabsItem id="tab3" label="Tab 3 with very long text that will be truncated" selected>
        This is the content of Tab 3.
      </AbcTabsItem>
      <AbcTabsItem id="tab4" label="Tab 4">
        This is the content of Tab 4.
      </AbcTabsItem>
      <AbcTabsItem id="tab5" label="Tab 5 with extra text">
        This is the content of Tab 5.
      </AbcTabsItem>
      <AbcTabsItem id="tab6" label="Tab 6" disabled>
        This is the content of the disabled Tab 6.
      </AbcTabsItem>
    </AbcTabs>
  </AbcWrapper>
);
```

2. Importing react-wrapped Tabs component

Following the new `Tabs` component with React's `AbcWrapper` was added to the page in order to display it in frontend. 1.18

Listing 1.18: packages/app/src/app/page.tsx

```
import { Accordion } from '@/components/accordion';
import { Button } from '@/components/button';
import { Dropdown } from '@/components/dropdown';
import { Tabs } from '@/components/tabs';

const Page = () => (
  <main style={...}>
    <Button />
    <Dropdown />
    <Accordion />
    <Tabs />
  </main>
);

export default Page;
```

Results

Upon integration, we encountered a critical `TypeError`, initially believed to be related to the Redux store used by the tabs components. Further investigation pointed to a potential conflict between custom functions in `utils/webcomponents-helpers.ts` utilized by tabs component and React's AbcWrapper.

Technical Challenges and Observations

- TypeError: The critical `TypeError` encountered during integration highlighted some incompatibility between StencilJS components and the Next.js framework, particularly when using React's AbcWrapper.
- The functions `parent()` and `closest()` were identified as potential sources of the issue, causing `currentEl` to become undefined.

Development Implications

- This experiment underscores the need for a deep understanding of architectural and dependency-related challenges when integrating different web frameworks.
- The documentation of these findings provides a starting point for future efforts in integration and can guide the development of a tailored solution.

Conclusion

The complexities observed during this integration experiment highlight the significant compatibility challenges between StencilJS and Next.js, especially in the context of SSR. Given the unique challenges encountered, Swisscom may benefit from developing a custom tool specifically designed to meet its integration needs. Such a tool would provide more control over the integration process and be better suited to the specific requirements of Swisscom's components within the Next.js environment.

1.5.2 Additional integration test with mayerraphael example

Repository: nextjs-mayerraphael-integration

Background

Following our initial attempts to integrate tabs into Next.js using the Smartive example, we explored a different approach based on a discussion in the Ionic Stencil GitHub thread about "Declarative Shadow DOM with Hydrate".

Our attention was drawn to the nextjs-webcomponent-hydration repository, which presented a proof of concept for SSR and web components collaboration.

We downloaded and initialized the `nextjs-webcomponent-hydration` repository locally. The process involved integrating tabs components from the mini-sdx repository, similar to the previous integration.

The primary difference in this approach was in the method of wrapping the component with React's AbcWrapper, as documented in `/components/StencilWrapper.tex`. The wrapped tabs component was then integrated into `/pages/index.tsx` for frontend display.

Result

In this iteration, we encountered a TypeError, similar to our previous integration, however with a notable difference in its occurrence. The error manifested a few lines above the point where it occurred in the Smartive integration, indicating a persistent yet

slightly different compatibility issue between the Swisscom Tabs components and the Next.js SSR environment.
Also it is worse to note that the author of the nextjs-webcomponent-hydration example cautioned that their solution is not ideal, being CPU and memory-intensive. They suggested creating native components for the specific framework (React in the case of Next.js) and wrapping them to render the Declarative Shadow DOM (DSD).

Proposal

This insight leads us to consider alternative approaches, such as developing native components or wrappers, to achieve a more efficient and compatible integration of Stencil components within the Next.js framework.

1.6 Performance Analysis

Background

Server-Side Rendering (SSR) is posited to offer a performance edge over Client-Side Rendering (CSR), particularly in metrics like the **first contentful paint** and the **last contentful paint**. These improvements are more significant with SSR, as it allows for earlier rendering and content delivery from the server. It is crucial for optimizing the user experience, as the perceived load time of a webpage can be significantly reduced.

Purpose

The purpose of this experiment is to validate whether SSR indeed offers performance gains over CSR in different scenarios. This involves testing in both an Express.js and an Angular environment to ascertain the extent of these gains across different web development frameworks and conditions.

Method

Test Environment

Frameworks tested:

- Express: Basic Setup with Express (repo)
- Angular: Stencil web components SSR in Angular (repo)

Test environment:

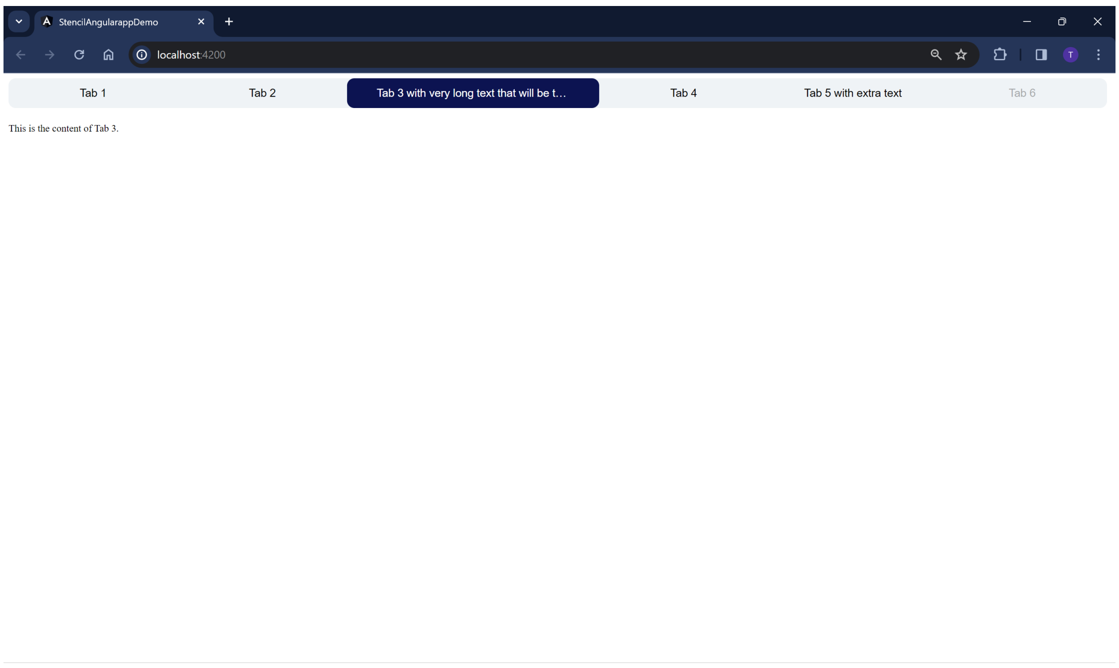
- Browser: Google Chrome
- Device Emulation: Desktop
- Network Conditions: OST W-Lan and simulated fast 3G network
- Test Application: Tabs component

Test Criteria

- First Contentful Paint (FCP): The time from when the page starts loading to when any part of the page's content is rendered on the screen.
- Largest Contentful Paint (LCP): The time taken for the largest content element visible in the viewport to be rendered.
- DOM Content Loaded (DCL): The time it takes for the HTML document to be completely loaded and parsed, without waiting for stylesheets, images, and sub-frames to finish loading.

Test Scenarios

The website tested is simple and consists of only one Stencil web component, the tabs component. Thus, the website looks like Figure 1.3.



This is the content of Tab 3.

Figure 1.3: Test Website

Scenario 1:

- No network throttling, no CPU throttling

Scenario 2:

- No network throttling, 4x CPU throttling

Scenario 3:

- Fast 3G network, no CPU throttling

Scenario 4:

- Fast 3G network, 4x CPU throttling

Each scenario was tested 10 times, with SSR and with no SSR. The results displayed here are average values.
As described in Section **??** Stencil uses the hydration app to enable SSR. In other words, an output target is defined that generates a module that can be used on the server to hydrate the document. That module can be used to use just one component or to bundle multiple components.
For instance, when a website uses five stencil web components, it is sufficient to generate one hydrate app that is able to hydrate all five web components. Similarly, this can be used to generate just one component or a whole library (many components).

Test 1: Express with minimal script (for just the tabs component)

In this test, the repo was used. The hydrate app only hydrates the tabs component that is needed on this webpage. In other words, the hydrate app contains code to just hydrate this one component, nothing else. This makes this script very small.

Test 2: Express with script of the entire SDX library

In this test, also the Basic Setup with Express repo was used. Instead of using the hydrate script that hydrates just the component present on the website, the script that hydrates the entire Swisscom SDX library was used. This, to determine whether the size of the script has a negative impact on the performance. Note the difference between the two scenarios is that with the script used in test 1 only the tabs component can be hydrated, while with the script in test 2 every component in the SDX library can be hydrated.

Test 3: Angular with minimal script

For this particular test, the Stencil web components SSR in Angular repository was utilized. This repository is configured to replicate the same website within an Angular framework, providing a platform to evaluate the performance of SSR in a more complex, application-oriented environment.

Results

Results Test 1

In general, the use of SSR tends to result in lower FCP, LCP, and DCL times compared to when SSR is not used. This suggests that SSR might be contributing to a faster rendering and loading experience in these scenarios. However, the time improvements are modest.

SSR	Scenario	FCP (in sec.)	LCP (in sec.)	DCL (in sec.)
Yes	1	0.12	0.12	0.11
No	1	0.17	0.17	0.19
Yes	2	0.34	0.41	0.38
No	2	0.53	0.53	0.71
Yes	3	0.67	0.68	1.92
No	3	0.67	0.67	2.07
Yes	4	0.86	0.86	2.07
No	4	0.99	0.99	2.26

Table 1.1: Results Test 1, average values

Particularly in scenario 3, where limited network speed was expected to amplify SSR benefits, the impact on FCP and LCP was not as significant. This outcome, especially in a small-scale test scenario, indicates that SSR's advantages might not be substantial for websites with limited content and functionality.
These graphs in Figure 1.4 show a side-by-side comparison of the load times, with SSR and without SSR:

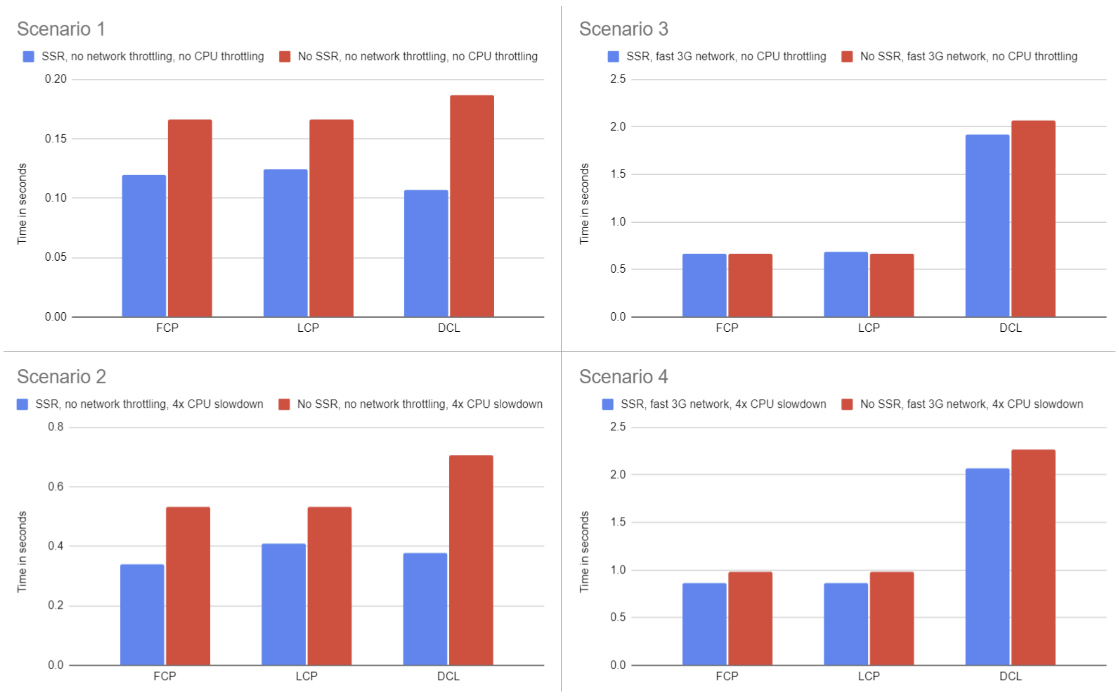


Figure 1.4: Performance comparison: SSR vs. No SSR, Test 1

SSR	Scenario	FCP (in sec.)	LCP (in sec.)	DCL (in sec.)
Yes	1	15.12	15.15	15.10
Yes	2	15.20	15.36	15.23
Yes	3	15.84	15.88	17.08
Yes	4	15.81	15.86	17.09

Table 1.2: Results Test 2, average values

Results Test 2

This test is only with SSR, as it is the same as test 1 with a different hydrate script. The results show that the hydrate script of the whole library introduces a large overhead. This overhead is present in all test scenarios. The difference between the different scenarios is not as big anymore, due to the overhead created by loading an enormous script.
These graphs in Figure 1.5 show a comparison of the load times when loading the minimal script of Test 1 compared to the full script.

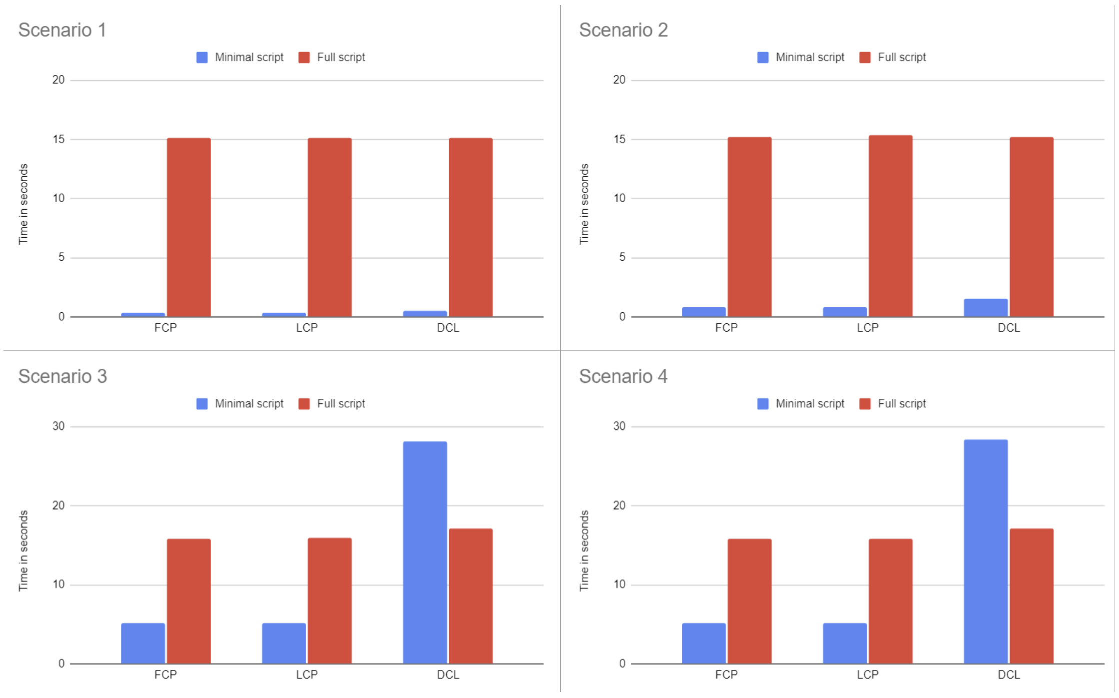


Figure 1.5: Performance comparison: Minimal script vs. Full script, Test 2

Results Test 3

Under optimal conditions (Scenario 1), both CSR and SSR displayed quick performance, with CSR slightly outpacing SSR. However, in Scenarios 3 and 4, which simulate more

SSR	Scenario	FCP (in sec.)	LCP (in sec.)	DCL (in sec.)
Yes	1	0.34	0.34	0.53
No	1	0.19	0.19	0.18
Yes	2	0.80	0.80	1.53
No	2	0.81	0.81	0.79
Yes	3	5.11	5.12	28.16
No	3	18.52	18.52	18.51
Yes	4	5.13	5.13	28.32
No	4	18.94	18.94	18.92

Table 1.3: Results Test 3, average values

constrained environments, the effectiveness of SSR became apparent, reducing FCP and LCP times by a factor of 3.5 compared to CSR.
Notably, the DCL metric was consistently slower for SSR across all scenarios. This slower DCL in SSR is attributed to the server's need to process requests, render pages, and then transmit them to the client, in contrast to CSR, which primarily retrieves necessary data. The extended DCL times are particularly pronounced due to the website's heavy use of styling mixins, requiring the import of multiple referenced files during server-side prerendering in Angular. This explains the longer DCL in all examples.
These graphs in Figure 1.6 show a side-by-side comparison of the load times, with SSR and without SSR:

Page Load Analysis

This section examines what the user observes during the time the page is loading, with and without SSR.

CSR

When a user requests a CSR-based website, the server sends a minimal HTML page along with JavaScript files. This HTML page is usually a skeleton of the page structure without the actual content. For the mini-sdx example, this looks the following Figure 1.7.
The visibility of the initial HTML page in Client-Side Rendering (CSR) varies depending on the framework being used. For instance, in Angular, this initial HTML page, which primarily serves as a structural skeleton, is typically kept hidden until the JavaScript renders the full content.
Once the browser executes the JavaScript, which is responsible for rendering the full webpage, there are no intermediate visible states.
The complete page becomes visible to the user immediately after the JavaScript execution, typically aligning with the LCP metric, marking the point where the main content of the page has been rendered:

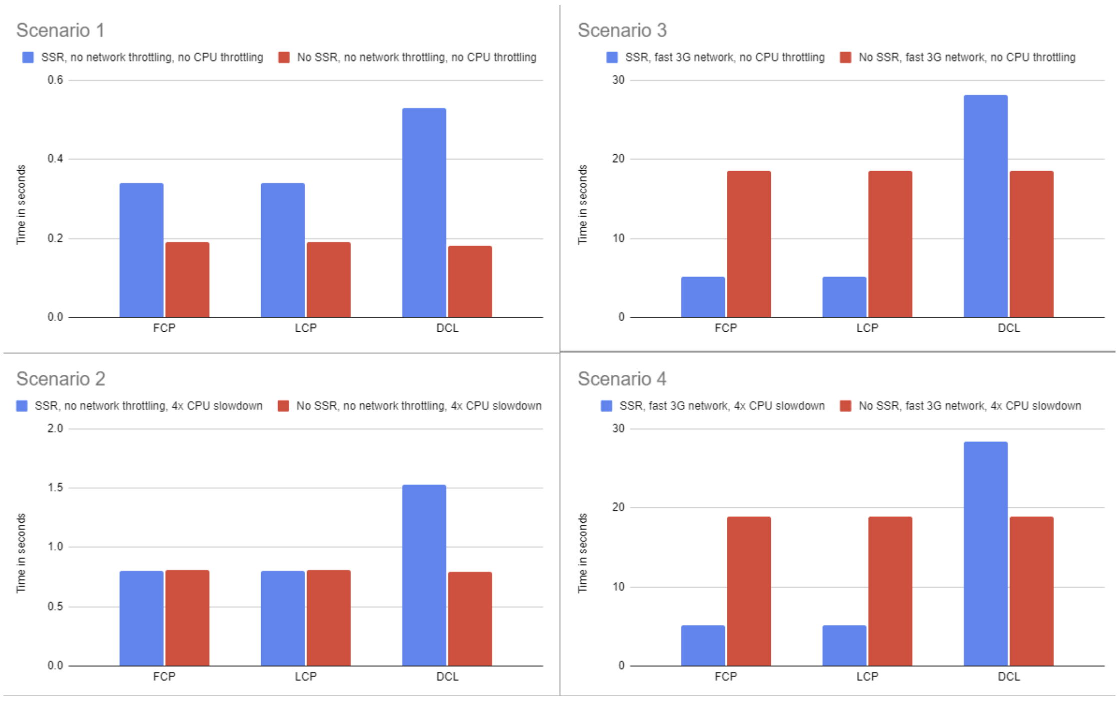


Figure 1.6: Performance comparison: SSR vs. No SSR, Test 3

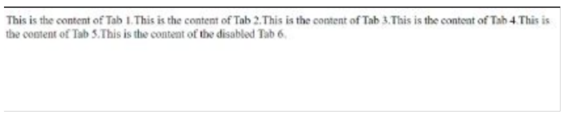
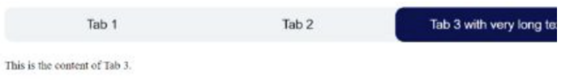

This is the content of Tab 1.This is the content of Tab 2.This is the content of Tab 3.This is the content of Tab 4.This is the content of Tab 5.This is the content of the disabled Tab 6.

Figure 1.7: Minimal HTML page

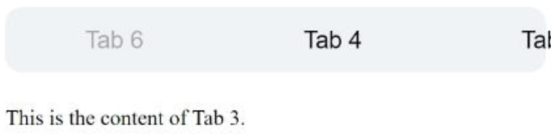


This is the content of Tab 3.

Figure 1.8: Fully loaded page

SSR

When a user requests a webpage that utilizes SSR, the server responds by sending a fully rendered HTML page. This page contains all the necessary content and structure, enabling the client to display it immediately upon receipt.



This is the content of Tab 3.

Figure 1.9: Rendered page from server

After the server sends a fully-rendered HTML page, the client-side JavaScript is executed to enable interactivity and dynamic features or styles. This process occurs after LCP, ensuring the page is displayed correctly and interactively, as illustrated in the Figure 1.8.

Conclusion

The results indicate that SSR generally offers improved performance metrics, particularly in FCP and LCP, under most test conditions. This advantage is more pronounced in scenarios with limited network capabilities, such as a simulated fast 3G network.
However, it's important to note that the benefits of SSR are not consistent across all scenarios. For instance, in tests with minimal network and CPU constraints, the performance gains of SSR were less significant, and CSR outperformed SSR in some cases. This variability implies that the choice between SSR and CSR should be tailored to the specific requirements of the application, considering factors such as the expected user environment and the complexity of the web content.
Exploring larger test scenarios, particularly within the Angular framework, would be a valuable extension of this study. This approach could offer deeper insights into the performance dynamics of SSR in more complex and demanding environments, further informing the optimal use of SSR and CSR in varied web development contexts.

1.7 Additional information

Repositories with react-wrapper solutions for Next.js

- Smartive company solution
- Proof of concept provided by mayerraphael

  Beware-Note from mayerraphael: Converting nodes to other formats is CPU heavy (and GC heavy because of the many objects created). I rather recommend creating native components for the framework you use (React in case of Next.js) and create wrappers around them to render the DSD.

- Porsche's solution, with Stencil patch

  Some details to Porsche's solution were published by Porsche's developer.

Could be useful for react-wrapper solution: React Integration

Bug Report

We opened a bug issue on Stencil's GitHub regarding the incorrect order after Stencil's renderToString. It can be found here.

Alternatives to StencilJS for Next.js

- Lit
- React native components

Part II

Appendix

Bibliography

List of Figures

1.1 Initially selected tab not selected 8
1.2 Initially selected tab selected 8
1.3 Test Website 18
1.4 Performance comparison: SSR vs. No SSR, Test 1 20
1.5 Performance comparison: Minimal script vs. Full script, Test 2 21
1.6 Performance comparison: SSR vs. No SSR, Test 3 23
1.7 Minimal HTML page 23
1.8 Fully loaded page 23
1.9 Rendered page from server 24