

SA  
Documentation

# Swisscom Design System - Server-Side-Rendering

Semester: Autumn 2023

Version: 1.0

Date: 2023-12-22 14:50:18+01:00

**Project Team:** Natalia Gerasimenko  
Tim Gamma

**Project Advisor:** Markus Stolze



School of Computer Science  
OST Eastern Switzerland University of Applied Sciences

# Contents

<b>I</b>	<b>Management Summary</b>	<b>1</b>
<b>1</b>	<b>Abstract and Management Summary</b>	<b>2</b>
1.1	Abstract . . . . .	2
1.2	Management Summary . . . . .	3
1.2.1	Introduction to Swisscom’s Digital Experience . . . . .	3
1.2.2	Problem Identification . . . . .	4
1.2.3	Research Approach . . . . .	4
1.2.4	Results and In-Depth Analysis . . . . .	4
1.2.5	Conclusion and Future Directions . . . . .	4
1.2.6	Potential for Consecutive Projects . . . . .	5
<b>II</b>	<b>Introduction</b>	<b>6</b>
<b>2</b>	<b>Introduction</b>	<b>7</b>
2.1	Initial Situation . . . . .	7
2.2	Problem Description . . . . .	7
2.3	Approach . . . . .	9
2.4	Experiments Summary . . . . .	10
2.4.1	Problem Analysis . . . . .	10
2.4.2	Problem Solution . . . . .	10
2.4.3	Framework integration . . . . .	11
2.4.4	Performance Analysis . . . . .	11
<b>III</b>	<b>Product Documentation</b>	<b>12</b>
<b>3</b>	<b>Requirements</b>	<b>13</b>
3.1	Functional Requirements . . . . .	13
3.1.1	FR1: Documentation - SSR . . . . .	13
3.1.2	FR2: Root Cause Analysis . . . . .	13
3.1.3	FR3: Issue Resolution . . . . .	14
3.1.4	FR4: Extended Issue Resolution . . . . .	14

3.1.5	FR5: Analysis of Alternatives . . . . .	14
3.1.6	FR6: Documentation - StencilJS . . . . .	15
3.1.7	FR7: Visual user experience . . . . .	15
3.2	Non-Functional Requirements . . . . .	15
3.2.1	NFR1: Performance . . . . .	15
3.2.2	NFR2: Compatibility . . . . .	16
3.2.3	NFR3: Security . . . . .	16
3.2.4	NFR4: Testability . . . . .	16
3.2.5	NFR5: Documentation Quality . . . . .	17
3.2.6	NFR6: Compliance . . . . .	17
<b>4</b>	<b>SSR Theory</b>	<b>18</b>
4.1	Introduction to SSR . . . . .	18
4.1.1	Historical Context . . . . .	18
4.1.2	Definition of Rendering Patterns . . . . .	19
4.1.3	Key Terms in SSR . . . . .	19
4.1.4	Benefits of SSR . . . . .	19
4.1.5	Challenges of SSR . . . . .	21
4.1.6	SSR's in modern web development . . . . .	22
4.2	StencilJS . . . . .	23
4.2.1	Stencil: A Web Component Compiler . . . . .	23
4.2.2	Shadow DOM in Stencil . . . . .	24
4.2.3	StencilJS SSR . . . . .	25
4.3	Angular . . . . .	26
4.3.1	Overview . . . . .	26
4.3.2	SSR in Angular . . . . .	27
4.4	SSR in Next.js . . . . .	28
4.4.1	Implementation Strategies . . . . .	29
4.4.2	React Hydration in Next.js . . . . .	29
4.4.3	The Role of getServerSideProps . . . . .	30
4.4.4	Example: Basic Next.js SSR Usage . . . . .	30
4.4.5	Conclusion . . . . .	32
<b>5</b>	<b>Problem Analysis</b>	<b>33</b>
5.1	Nested Components . . . . .	33
5.2	Global Styles . . . . .	36
5.3	NPM Packages Update . . . . .	38
5.4	Chat GPT . . . . .	39
<b>6</b>	<b>Experiments</b>	<b>41</b>
6.1	Environment setup . . . . .	41
6.1.1	Used repositories . . . . .	41
6.1.2	Extra Projects . . . . .	41
6.2	Problem Solution . . . . .	42

6.2.1	Stencil ESM Scripts . . . . .	42
6.2.2	ESM Script from Full Library . . . . .	43
6.2.3	Contact with Stencil Community . . . . .	44
6.2.4	Hydrate Script Results vs. Hydrated HTML . . . . .	46
6.2.5	StencilJS SSR Issues – Community Help . . . . .	48
6.2.6	Old Alpha Release of StencilJS . . . . .	49
6.2.7	Workaround: Wrong Order – with Lifecycle . . . . .	50
6.2.8	Workaround: Wrong Order – with CSS . . . . .	53
6.3	Framework Integration . . . . .	56
6.3.1	Integrate Stencil Web Components into an Angular Application . . . . .	56
6.3.2	Tabs Integration into Next.js using Smartive Company Example . . . . .	61
6.3.3	Tabs Integration into Next.js using Mayerraphael Example . . . . .	64
6.4	Performance Analysis . . . . .	65
<b>7</b>	<b>Quality Measures</b>	<b>73</b>
7.1	Issue Tracking . . . . .	73
7.2	Development Environment . . . . .	73
7.3	Merge Requests . . . . .	73
7.4	Time Tracking . . . . .	73
7.5	Verification and Testing . . . . .	73
7.5.1	Testing Methodologies . . . . .	74
<b>IV</b>	<b>Results</b>	<b>75</b>
<b>8</b>	<b>Results</b>	<b>76</b>
8.1	Verification of Functional Requirements . . . . .	76
8.2	Verification of Non-Functional Requirements . . . . .	78
8.3	Conclusion . . . . .	82
<b>V</b>	<b>Developer Documentation</b>	<b>84</b>
<b>9</b>	<b>Developer Documentation</b>	<b>85</b>
9.1	Stencil SSR – Basic setup with Express . . . . .	85
9.2	Workaround – Wrong order of elements after renderToString() . . . . .	87
9.3	Old alpha release of StencilJS - SSR patches . . . . .	91
9.4	Integrate Stencil Web Components into an Angular Application . . . . .	92
9.4.1	Setup . . . . .	92
9.5	Integration into Next.js . . . . .	96
9.5.1	Tabs integration into Smartive company example . . . . .	96
9.5.2	Additional integration test with mayerraphael example . . . . .	99
9.6	Performance Analysis . . . . .	100
9.7	Additional information . . . . .	107

<b>VI Appendix</b>	<b>109</b>
<b>Bibliography</b>	<b>110</b>
<b>List of Figures</b>	<b>111</b>

**Part I**

**Management Summary**

# Chapter 1

## Abstract and Management Summary

### 1.1 Abstract

#### Introduction

Swisscom has its own design system, the Swisscom Digital Experience (SDX). It consists of a component library, UX principles, design guidelines, documentation, and rules. In this design system, Swisscom offers web components, reusable styled building blocks, build using StencilJS. This research project investigates the integration of Server-Side rendering (SSR) with these components. When built with StencilJS and combined with SSR, web components are not interactive and do incorporate incorrect CSS styling. The project focuses on addressing these issues. Additionally, it seeks solutions for integrating these web components within the frameworks Angular and Next.js.

#### Approach

The methodology involved a series of experiments to tackle the challenges in implementing SSR with StencilJS. These experiments included resolving interactivity deficits and rendering inconsistencies within Stencil's SSR framework, as well as exploring the integration of StencilJS web components in Angular and Next.js frameworks. The performance aspects of SSR were inspected in a simple Node.js environment using Express.js as a server and within an Angular application. The aim was to comprehensively evaluate the efficiency and responsiveness of SSR in different contexts.

#### Result

The findings revealed that manually adding ESM scripts post rendering resolved the interactivity issues. Incorrect CSS styling, linked to bugs in StencilJS, was addressed through a CSS-grid based workaround. The integration of StencilJS web components with Angular was achieved by defining Angular output targets and pre-rendering the

components. Integrating these components into Next.js presented significant challenges. The investigation revealed that, in its current state, Next.js is not compatible with StencilJS. This necessitates the development of react wrapper generator aimed at encapsulating Stencil components within React components, enabling seamless integration into the Next.js framework. Regarding performance, SSR generally demonstrated a faster response than clientside rendering, particularly in the first contentful paint. However, this performance advantage was not consistent across all scenarios.

The outcome of this project showed that SSR is possible with StencilJS web components. However, we identified a number of remaining issues with the current tooling for Stencil SSR support, necessitating the development of custom solutions and workarounds. While the integration of StencilJS web components within Angular and Next.js is achievable, it demands significant setup and configuration, especially for Next.js. Nevertheless, as framework compatibility improves, the efficiency and user experience offered by SSR with StencilJS web components are expected to become more pronounced, highlighting the promising future of this technology in web development.

## 1.2 Management Summary

### 1.2.1 Introduction to Swisscom's Digital Experience

Swisscom's Digital Experience (SDX) is a comprehensive design system featuring a StencilJS-based component library (Figure 1.1), aligned with user experience principles, design guidelines, and comprehensive documentation. The uniqueness of SDX lies in its offering of reusable, styled web components, which foster a cohesive digital experience across Swisscom's services.

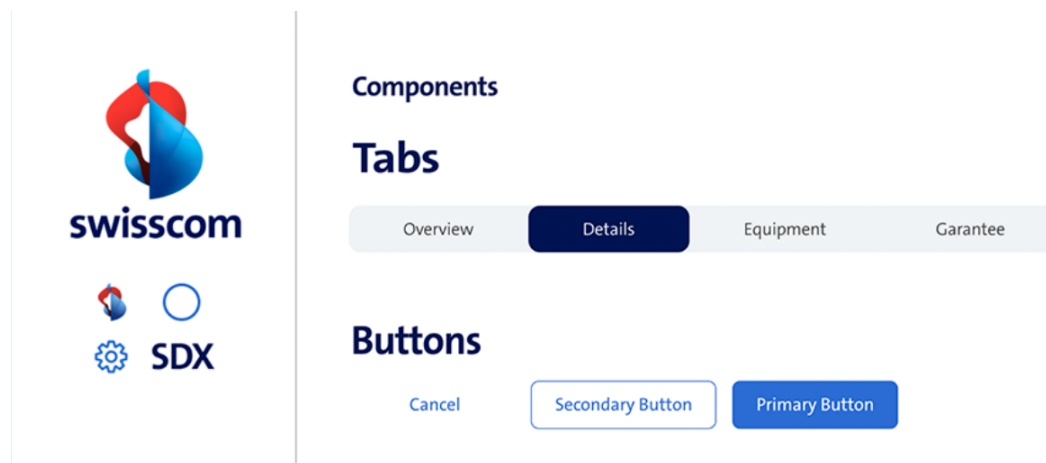


Figure 1.1: Example of Swisscom's web components from Swisscom SDX Documentation



### 1.2.2 Problem Identification

The integration of Server-Side Rendering (SSR) with StencilJS web components presented notable challenges. Specifically, when combined with SSR, these web components exhibited non-interactivity and incorrect CSS styling. Addressing these issues was pivotal for enhancing the user experience and maintaining the integrity of the design system.

### 1.2.3 Research Approach

Our methodology was comprehensive, involving multiple experimental phases to address the SSR integration challenges. These included:

- **Resolving Interactivity and CSS Issues:** Conducting targeted experiments within StencilJS SSR feature to overcome interactivity deficits and CSS inconsistencies.
- **Framework Integration Exploration:** Examining the integration of StencilJS web components within Angular and Next.js frameworks, each presenting unique challenges.
- **Performance Analysis:** Evaluating SSR's efficiency and responsiveness in a simple Node.js environment using Express.js, and within an Angular application, to understand the practical implications of SSR in different contexts.

### 1.2.4 Results and In-Depth Analysis

- **Interactivity Solutions:** The addition of ESM scripts post-rendering effectively restored interactivity in web components.
- **CSS Styling Fixes:** CSS-grid based workaround was developed to address styling issues, stemming from bugs in StencilJS.
- **Framework Integration:** Angular integration was successful through defining specific output targets and pre-rendering components. However, Next.js integration faced significant compatibility issues, necessitating a custom react wrapper generator for effective incorporation.
- **Performance Insights:** SSR generally improved response times, particularly in the first contentful paint. Importantly, there were specific scenarios where the performance with SSR was notably poor.

### 1.2.5 Conclusion and Future Directions

The project demonstrates that while SSR with StencilJS web components is feasible, it requires bespoke solutions and configurations. The integration within Angular and Next.js, though challenging, is achievable with significant setup, especially for Next.js.

The findings also highlight areas for future improvement, particularly in tooling for Stencil SSR support and framework compatibility. The variability in performance highlights the complexity of SSR integration and underscores the need for more refined solutions and optimizations in certain use cases.

### **1.2.6 Potential for Consecutive Projects**

The research opens avenues for further exploration in enhancing framework compatibility and refining SSR performance. Future projects could focus on optimizing the integration process, addressing the identified issues, and exploring new ways to leverage SSR's benefits in web development.

**Part II**

**Introduction**

# Chapter 2

## Introduction

### 2.1 Initial Situation

Swisscom has an own design system, the **Swisscom Digital Experience (SDX)**. It consists of a component library, UX principles, design guidelines, documentations and rules.

In this design system, Swisscom offers **components**, reusable styled building blocks using the basic elements. They are build using StencilJS.

One such components is the **tabs component**. It organizes content and allows navigation between groups of content that are related and at the same level hierarchy.

In Figure 2.1 an example is shown.

At Swisscom, there is an inclination towards implementing server-side rendering, necessitating that their components, built using StencilJS, support this feature. Given that StencilJS inherently facilitates Server-Side Rendering (SSR), this aligns with Swisscom's objective to enhance their web components with server-side rendering capabilities.

### 2.2 Problem Description

The problem lies in the fact, that the server-side-rendered version of these components has some visual differences and lacks interactivity compared to the client side rendered components.

This difference poses a significant challenge in achieving consistency and a seamless user experience between server-side and client-side rendering methods. The specific nature of these differences is illustrated in Figure 2.2 and Figure 2.3, where the tabs components are used to demonstrate the variance in their appearance and functionality across the two rendering techniques.

In Figure 2.2, the correctly rendered component is displayed, demonstrating how it should ideally appear. Conversely, Figure 2.3 presents the component as rendered using the SSR-version.

The SSR-version lacks interactivity, displays incorrect ordering and the styles are not applied correctly. For instance, the **Tab 3** should be selected, yet it is not.



## Product description

Pro. Beyond. A magical new way to interact with iPhone. Groundbreaking safety features designed to save lives. An innovative 48MP camera for mind-blowing detail. All powered by the ultimate smartphone chip.

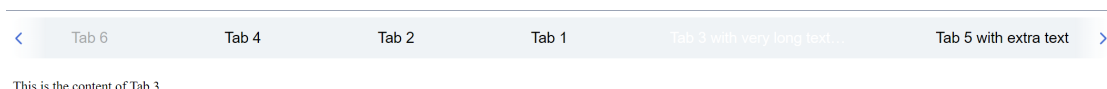
Meet the new face of iPhone. Introducing Dynamic Island, a truly Apple innovation that's hardware and software and something in between. It bubbles up music, sports scores, FaceTime, and so much more — all without taking you away from what you're doing. Your photo. Your font. Your widgets. Your iPhone. iOS 16 lets you customize your Lock Screen in fun new ways. Layer a photo to make it pop. Track your Activity rings. And see live updates from your favorite apps.

Figure 2.1: Example of tabs component from Swisscom SDX Documentation



This is the content of Tab 3.

Figure 2.2: Correct Tabs Component



This is the content of Tab 3.

Figure 2.3: Incorrect Tabs Component

## 2.3 Approach

The approach to the project is methodically divided into distinct phases:

The first phase is the **Problem Analysis Phase**, Section 5. The goal is to identify the problem of the non-interactive and the visual differences.

The next phase is the **Problem Solution Phase**, Section 6.2. In this phase, the problems identified during the problem analysis phase will get solved and related problems will be discussed.

In the **Framework Integration Phase**, Section 6.3, the solution will be tried to adapt to the frameworks Angular and Next.js, documenting any difficulties.

Finally, in the **Performance Analysis Phase**, Section 6.4, the performance of server-side rendering will get compared to the performance of not using server-side rendering.

In each phase, targeted experiments will be carried out, tailored to the specific problems identified and addressed within that phase.

In Figure 2.4 an overview of all the experiments can be seen.

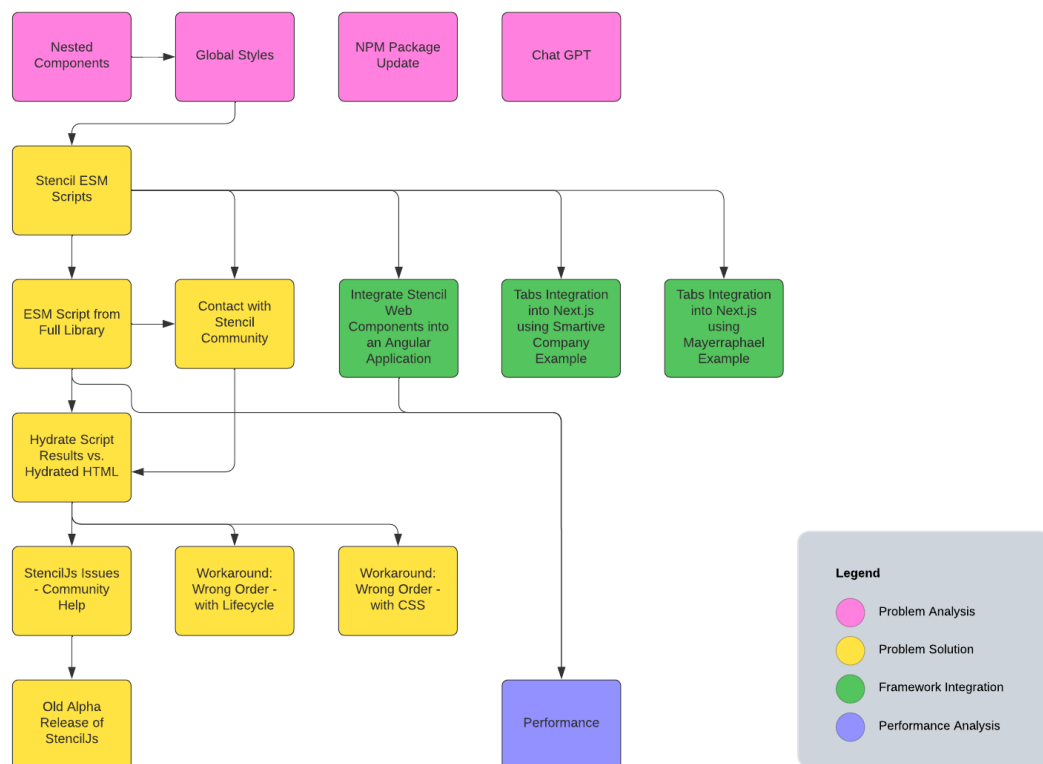


Figure 2.4: Experiments Conducted

## 2.4 Experiments Summary

### 2.4.1 Problem Analysis

The initial problem were the incorrect styles and the lack of interactivity on the tabs component.

**Nested components**, Section 5.1 tried to find out whether nested components in general were not correctly rendered by Stencils SSR functionality. The results indicated that such rendering problems did not exist.

**Global Styles**, Section 5.2 tackled to find out whether global styles sheets, used by the tabs component, lead to the incorrect display of the component. This was not the case, but interactive components and styles that were set in JavaScript were not working.

**NPM Packages Update**, Section 5.3 tried to update all NPM packages to be sure that the problem was not fixed in the meantime. This was not the case.

**Chat GPT**, Section 5.4 examined Chat GPT, to find out whether it was helpful solving the problems. It turned out Chat GPT knows little about Stencil and was thus no help.

With this experiments conducted, it was clear that the problem was that the component lacked interactivity, it was just a static component. The incorrect styles were due to missing JavaScript code that sets these styles.

### 2.4.2 Problem Solution

**Stencil ESM Scripts**, Section 6.2.1 Since only static HTML was sent to the client, it was tried to also send the necessary scripts for hydration. This resulted in an interactive component.

**Contact with Stencil Community**, Section 6.2.3 In order to be absolutely sure that the way scripts were added in scripts was correct, the Stencil Community was contacted. The community members confirmed that it was correct.

**ESM Script from Full Library**, Section 6.2.2 After rendering one component successfully, it was tried doing it for the whole library. This resulted in some weird behavior, such as code duplication. Additionally, the code was very slow.

**Hydrate Script Results vs. Hydrated HTML**, Section 6.2.4 As mentioned above, there were still some issues with the solution. Mainly some CSS-attributes which were not set and during the first few milliseconds, the order of some elements was wrong. The issues were documented.

**StencilJS SSR Issues – Community Help**, Section 6.2.5 The StencilJS community acknowledged potential bugs in the Server-Side Rendering (SSR) feature, particularly with the `renderToString()` function, suggesting an older alpha release for possible fixes, as ongoing efforts to address these issues continue.

**Old Alpha Release of StencilJS**, Section 6.2.6 tried out the above-mentioned alpha release. CSS-attribute problems were solved, while others persisted, especially the order issue.

To mitigate the order issue, two workarounds were created: **Workaround: Wrong Order – with Lifecycle**, Section 6.2.7 did not load the component until the order was correct. **Workaround: Wrong Order – with CSS**, Section 6.2.8 used CSS to always have the correct order, using CSS grid.

### 2.4.3 Framework integration

Now that SSR was running, and the issues were documented, the next step was to integrate Stencil components into frameworks, using SSR.

**Integrate Stencil Web Components into an Angular Application**, Section 6.3.1 tried to include Stencil web components into an Angular application using SSR and the Angular output target from Stencil. This was successful.

**Tabs Integration into Next.js using Smartive Company Example**, Section 6.3.2 **Tabs Integration into Next.js Using Mayerraphael Example**, Section 6.3.3 tried to include Stencil web components into a Next.js example, but were not successful. It was discovered that it would be necessary to develop react wrapper generator aimed at encapsulating Stencil components within React components, thereby enabling their seamless integration into the Next.js framework.

### 2.4.4 Performance Analysis

**Performance**, Section 9.6 evaluated the performance of SSR in comparison to CSR in an Express.js setting and an Angular setting. SSR generally demonstrated a faster response compared to client-side rendering, particularly in the first contentful paint. However, this performance advantage was not consistent across all scenarios.



**Part III**

**Product Documentation**

# Chapter 3

## Requirements

The primary objective of this project is to explore the feasibility of server-side rendering for Swisscom's SDX components.

Specifically, the project aims to fulfill certain requirements to achieve this goal effectively.

### 3.1 Functional Requirements

Functional requirements will be defined in plain text.

#### 3.1.1 FR1: Documentation - SSR

- An overview over the variety of different implementations of SSR should be provided.
- Both the positive and negative aspects of each method are highlighted.

**Acceptance Criteria:** The documentation should cover all aspects of SSR implementation and be easily understandable.

**Verification:** Do reviews and feedback sessions to assess clarity and completeness.

**Realization:** Develop a comprehensive documentation including case studies and examples.

#### 3.1.2 FR2: Root Cause Analysis

- Conduct a thorough analysis to identify the root cause of why the web components are not working as expected with SSR.

**Acceptance Criteria:** Clear identification and documentation of the root cause.

**Verification:** Peer review and cross-verification with system logs.

**Realization:** Systematic analysis involving expert consultations.

### 3.1.3 FR3: Issue Resolution

- Provide a solution on how to make the Swisscom web components compatible with StencilJS SSR.

**Acceptance Criteria:** Effective resolution of compatibility issues.

**Verification:** Manual testing of the implemented solutions in various scenarios to ensure they effectively resolve the compatibility issues with StencilJS SSR.

**Realization:** Find or implement solutions and monitor their performance.

### 3.1.4 FR4: Extended Issue Resolution

- Provide a solution on how to make a whole application compatible with StencilJS SSR.

**Acceptance Criteria:** Effective resolution of compatibility issues.

**Verification:** Manual testing of the implemented solutions in various scenarios to ensure they effectively resolve the compatibility issues with StencilJS SSR.

**Realization:** Find or implement solutions and monitor their performance.

### 3.1.5 FR5: Analysis of Alternatives

- Alternative solutions using different technologies should be considered.
- Positive and negative aspects of the alternative solutions should be given.

**Acceptance Criteria:** Identification of viable alternative solutions.

**Verification:** Comparative analysis of positive and negative aspects of the alternative solutions.

**Realization:** Explore and document alternative technologies and methods.

### 3.1.6 FR6: Documentation - StencilJS

- A detailed analysis of the StencilJS-SSR implementation should be provided.
- The compatibility issues between Swisscom web components and StencilJS SSR are attested.
- Step-by-step instructions, code examples, and explanations of the modifications made to achieve compatibility are documented.

**Acceptance Criteria:** Detailed and clear documentation of StencilJS-SSR implementation.

**Verification:** Peer review for thoroughness and clarity.

**Realization:** Create step-by-step guides with code examples.

### 3.1.7 FR7: Visual user experience

- Ensure that SSR web components provide correct visual user experience, e.g. all styles and scripts are loaded.

**Acceptance Criteria:** Correct display of SSR web components.

**Verification:** Conduct user interface testing to ensure that SSR web components display correctly with all styles and scripts loaded.

**Realization:** Continuous testing and refinement based on user feedback.

## 3.2 Non-Functional Requirements

### 3.2.1 NFR1: Performance

- Ensure that the SSR web components provide a fast user experience, with a specific requirement that the page should load within two seconds when using the web components. For example, loading a project dashboard should take no longer than two seconds to render.

**Acceptance Criteria:** Load time of no more than two seconds.

**Verification:** Load and stress testing.

**Realization:** Optimize performance through research and implementation of possible improvements.

### 3.2.2 NFR2: Compatibility

- Ensure that the web components are compatible with the latest versions of StencilJS and its SSR features.
- Verify that the web components are compatible with the latest versions of major browsers (Chrome, Firefox, Safari, Edge) and ensure that any browser-specific issues are addressed promptly.
- Ensure that the web components are compatible with all commonly user operating systems, such as Windows, MacOS and Linux.

**Acceptance Criteria:** Functionality across browsers and with latest StencilJS.

**Verification:** Cross-browser and version testing.

**Realization:** Regular updates and testing against new versions.

### 3.2.3 NFR3: Security

- Conduct a security review to identify and mitigate potential security risks associated, e.g. under NPM Packages

**Acceptance Criteria:** Strive for no known security vulnerabilities. Exceptions are permissible when using external projects with known issues, which should be clearly documented.

**Verification:** Conduct thorough security audits, including vulnerability assessments of external dependencies.

**Realization:** Document any exceptions and vulnerabilities in external dependencies to inform developers.

### 3.2.4 NFR4: Testability

- Ensure that the project is tested on a variety of systems and OSs, Windows, MacOS and Linux, to be precise.

**Acceptance Criteria:** Stable performance on multiple systems and OSs.

**Verification:** Testing on Windows, MacOS and Linux.

**Realization:** Manual testing.

### 3.2.5 NFR5: Documentation Quality

- Ensure that the documentation of compatibility solutions adheres to high-quality standards. It should be well-structured, easy to navigate, and free from grammatical errors. The documentation should be a valuable resource for Swisscom developers seeking to understand and implement the compatibility solutions.

**Acceptance Criteria:** Clear, well-structured, error-free documentation.

**Verification:** Reviews for quality and clarity.

**Realization:** Conduct regular reviews, use AI for corrections and improvements.

### 3.2.6 NFR6: Compliance

- Ensure that the project complies with any relevant legal and regulatory requirements, such as data protection and copyright laws.

**Acceptance Criteria:** Adherence to legal and regulatory standards.

**Verification:** Ensure the project meets all relevant legal and regulatory requirements, focusing on data protection and copyright laws.

**Realization:** Utilize reputable sources for code examples and snippets.

# Chapter 4

## SSR Theory

This chapter gives an overview of Server-Side Rendering (SSR). It outlines what SSR does, the key terms in SSR, the advantages and disadvantages, as well as its implementation in StencilJS, Angular and Next.js.

### 4.1 Introduction to SSR

#### 4.1.1 Historical Context

The historical development of Server-Side Rendering (SSR) in web development has been significantly shaped by the evolving landscape of web technologies and the changing needs of users and developers.

In the early days of the web, websites were predominantly static HTML documents served directly from servers, with minimal client-side processing. With the introduction of JavaScript in 1995, a new era started. Applications became increasingly more interactive, giving rise to Client-Side Rendering (CSR). However, CSR posed various other challenges, such as slow initial load times and poor search engine optimization (SEO), as search engines struggled to index JavaScript-loaded content.

To address these SEO concerns and enhance initial page load times, SSR gained popularity. SSR involved first rendering webpages on a server and then delivering a fully-formed HTML to clients, shortening initial load times and ensuring SEO-friendliness.

Google's support for SSR further encouraged its adoption, as sites using SSR ranked higher in Google's page-rank algorithm. SSR accessibility improved with the development of dedicated frameworks, such as Gatsby or Next.js and libraries for existing frameworks, for example Angular Universal for Angular. It became integral to Progressive Web Apps (PWAs) which were focused on rapid, reliable, and engaging web experiences. Throughout its history, SSR has evolved to emphasize performance optimization, incorporating techniques like code splitting and efficient caching.

Overall, SSR's historical journey reflects the dynamic evolution of web development paradigms, influenced by changing technologies, SEO considerations, and the pursuit of optimized user experiences.

## 4.1.2 Definition of Rendering Patterns

In web development, a rendering pattern describes how a web application or site processes and displays its HTML, CSS, and JavaScript content. The most common rendering patterns in web development are [Shi20]:

**Server-side Rendering (SSR):** Involves generating the full HTML for a page on the server in response to a user request, then sending it to the client's browser. This approach allows for immediate display of the content upon browser loading.

**Client-side Rendering (CSR):** The browser renders the HTML using JavaScript based on the server's response. The server typically sends a minimal HTML document with JavaScript that then renders the full page.

**Static Site Generation (SSG):** HTML pages are generated at build time and served as static files. It is a hybrid option of using both CSR and SSR allowing for pre-rendering of pages for performance benefits.

## 4.1.3 Key Terms in SSR

**Hydration:** Within the context of web development, hydration refers to the process of adding dynamic interactivity on the client side to an HTML document that was originally created on a server [LP23].

The browser initially performs HTML parsing and subsequently assembles a Document Object Model (DOM) tree. Subsequently, JavaScript code is employed to extract the initial state and properties associated with the HTML content generated on the server. These acquired data elements play a pivotal role in the process of component hydration, thereby transforming them into interactive elements.

**Pre-rendering:** When a user enters a web address in their browser, the server initially delivers a static HTML page along with JavaScript, which loads in the background. During this phase, users can view the static website, but interactivity is limited until the JavaScript is fully downloaded and executed (until initial HTML is "hydrated" with client-side JavaScript). In this context **pre-rendering** is a broader term that represents a trade-off between SSR and CSR.

**Initial Page Load (IPL):** The first load of a web page when a user visits it, often associated with SSR, as one of the main goals of SSR is to reduce the initial page load.

## 4.1.4 Benefits of SSR

When considering the use of SSR in web applications, it is essential to recognize the positive impacts it can have on initial load times, search engine visibility, and overall



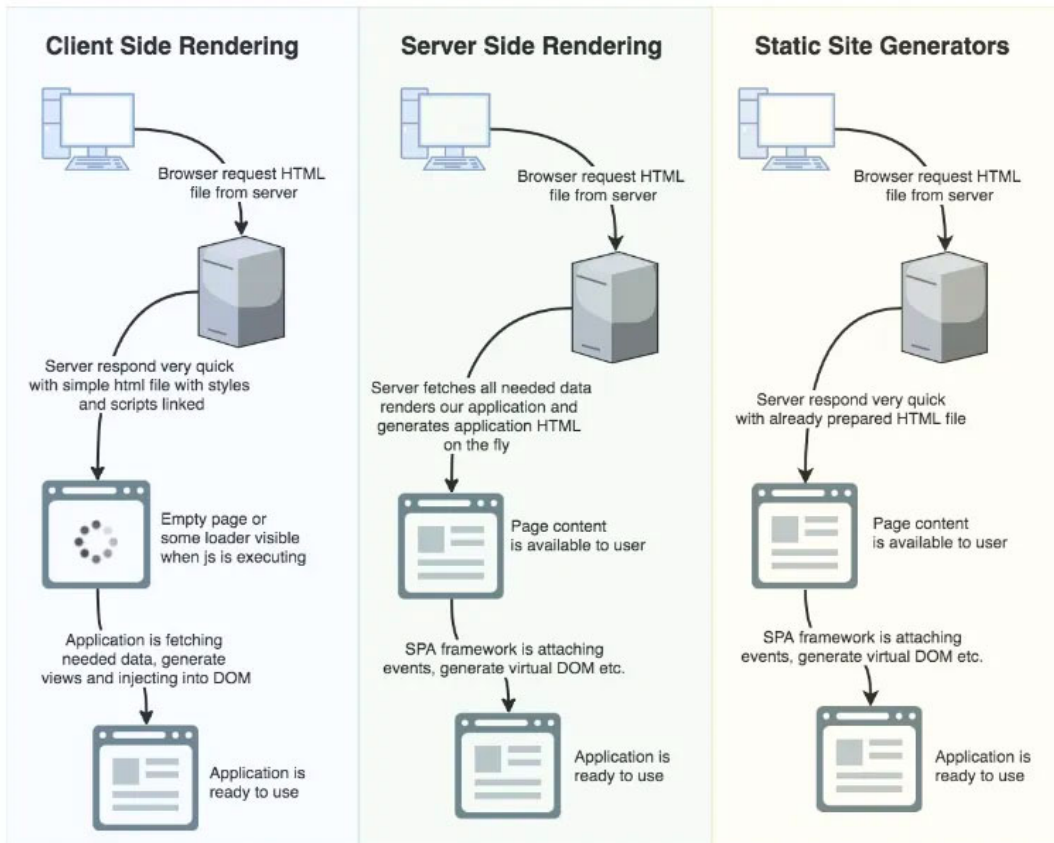


Figure 4.1: CSR vs SSR vs SSG [Mit22]

user engagement.

- **Improved Initial Page Load:** SSR significantly enhances initial page load performance by rendering the web page's HTML content on the server. Users can view content faster because the browser receives pre-rendered HTML, reducing the need for client-side rendering.
- **SEO Friendliness:** Because the content is already in the HTML when it reaches the browser, search engines can efficiently crawl and index the HTML content, resulting in improved search engine rankings and discoverability. This is crucial for websites aiming for high visibility.
- **Stable Performance on All Devices:** SSR tends to result in more stable performance across different devices because it reduces the reliance on client-side JavaScript execution, thereby minimizing the impact of variations in device hardware capabilities.
- **Enhanced Security:** SSR can help protect sensitive data by rendering it server-side, reducing the risk of exposing critical information in client-side code.
- **Accessibility:** SSR inherently provides better support for accessibility standards since the initial HTML is fully formed on the server, ensuring that content is accessible to screen readers and other assistive technologies.
- **Enhanced User Experience:** SSR provides a more responsive and user-friendly experience, as the page is visible and interactive more quickly.

#### 4.1.5 Challenges of SSR

While SSR offers numerous advantages, it is crucial to acknowledge the potential drawbacks and complexities that can arise when implementing this approach.

- **Increased Server Load:** SSR can place a higher load on servers, as they need to handle the rendering of HTML for each user request, potentially requiring more server resources.
- **Complexity:** SSR adds complexity to web development, as it requires SSR logic in addition to client-side code.
- **Slower Subsequent Page Transition:** While SSR speeds up initial loads, navigating between pages within an SSR application can be slower, as the server must be contacted for each page transition.
- **Limited Client-Side Interactivity:** Some complex client-side interactions, such as single-page app features, may be more challenging to implement in an SSR context.

## 4.1.6 SSR's in modern web development

### Trends and Future of SSR

Modern frameworks are increasingly adapting to improve SSR. Future developments in SSR technologies could significantly influence web development practices (web.dev, up-to-date articles on web trends).

Here are some trends and the potential future direction of SSR:

- **Hybrid Rendering:** A trend in SSR is the adoption of hybrid rendering, which combines SSR and Client-Side Rendering (CSR) to leverage the strengths of both approaches. These approaches aim to balance the SEO and initial loading benefits of SSR with the interactivity of CSR.
- **Performance Optimization:** Future SSR implementations will focus on further performance optimization to reduce server response times and enhance the user experience.
- **Serverless SSR:** Serverless computing<sup>1</sup> platforms are becoming more prevalent, and SSR may become more adaptable to serverless technologies, offering flexible scaling and cost-effective solutions
- **PWA<sup>2</sup> Integration:** SSR will continue to integrate with Progressive Web Apps, providing offline access, push notifications, and improved caching strategies
- **Standardization and Improved Tooling:** Efforts may be made to standardize SSR practices and improve developer tooling to simplify SSR adoption and enhance the developer experience.

### Frameworks for SSR:

- **Angular:** Angular Universal
- **ReactJS:** ReactJS with custom setup, Next.js
- **VueJS:** Nuxt.js

**Case Studies:** Real-world examples like Netflix and Amazon demonstrate the practical application and benefits of SSR.

---

<sup>1</sup>Serverless computing is a model for developing and executing cloud-based applications. It allows developers to create and operate application code without the need to manage servers or backend infrastructure.

<sup>2</sup>A Progressive Web App (PWA) is a web-based application that offers a native app-like experience. PWAs are cross-platform, installable, and capable of running offline, in the background, and integrating with device features and other apps.

## 4.2 StencilJS

### 4.2.1 Stencil: A Web Component Compiler

”Stencil is a compiler that generates web components” [Stef]. Or, to be more specific, it creates **custom elements**.

It was created by the Ionic Framework team, to build faster, more capable components, working across every major framework.

It uses the following technologies:

- TypeScript
- JSX <sup>3</sup>
- CSS

To further streamline the developer experience, Stencil provides framework-specific wrappers. These wrappers facilitate the integration of Stencil components, ensuring they align smoothly with the standards of the respective frameworks. [Stef]

#### Create a Stencil Web Component

In this section, a simple Stencil web component is created. The example is adapted from [Sted].

To develop Stencil web components, it is essential to have the latest Node.js installed on the development machine.

A new project can easily be initialized with

```
npm init stencil
```

This will provide a prompt, where it is possible to choose the type of project to start:

- component (collection of web components that can be used anywhere)
- app [ community ] (minimal starter for building a Stencil app or website)
- ionic-pwa [ community ] Ionic PWA starter with tabs layout and routes

For building a web component, select ”component”. After that the CLI will scaffold a project, with an example component already in it. It can be found under **src/component**. It is possible to edit this component or add new components.

Components are written in TypeScript and use JSX for rendering. Each component is thus a TypeScript class decorated with **@Component** that includes metadata and

---

<sup>3</sup>JSX (JavaScript Syntax Extension) is an extension of JavaScript. It allows XML-like syntax to create DOM trees

the component's functionalities and provides it to the compiler.

Here a simple example:

Listing 4.1: Basic Stencil web component

```
1 import { Component, Prop, h } from '@stencil/core';
2 import { format } from '../../utils/utils';
3
4 @Component({
5   tag: 'my-component',
6   styleUrls: 'my-component.css',
7   shadow: true,
8 })
9 export class MyComponent {
10   @Prop() first: string;
11   @Prop() middle: string;
12   @Prop() last: string;
13
14   private getText(): string {
15     return format(this.first, this.middle, this.last);
16   }
17
18   render() {
19     return <div>Hello, World! I'm {this.getText()}</div>;
20   }
21 }
```

Once created, this component can be seamlessly integrated into HTML files like any other tag:

```
1 <my-component first="Stencil" middle="'Don't call me a
   framework'" last="JS"></my-component>
```

The **@Prop()** decorator tells Stencil to re-render once one of them changes.

The **render()** function returns a `<div>` element, containing text to render.

The **shadow: true** tells Stencil to use the shadow DOM, more on the shadow DOM in Section 4.2.2.

## Useful Commands

- `npm start` to start a local development server.
- `npm run build` to create a production-ready version of the component.

### 4.2.2 Shadow DOM in Stencil

The **Shadow DOM** is a web standard that enables DOM as well as style encapsulation. It ensures that component's styles, markup and behaviour are isolated to the rest of the

application. An additional benefit is the simplified CSS scoping, as external interference is prevented.

In Stencil it can be used to "style clashes and external effects" [Steg]. It can be enabled by setting "shadow" to true, as seen in Listing 4.1, row 7.

When using shadow DOM, as previously mentioned, the elements are scoped. Only the styles of the respective component are used. Due to this, the CSS can be simpler. As it only effects the elements inside a component. Thus, it is not necessary to include selectors to scope styles to the component.

The Shadow DOM also create **shadow roots**, root elements for the scoped component. This root element can then be used to perform queries. [Steg].

Shadow DOM is supported by all major browsers:

- Chrome
- Firefox
- Safari
- Edge (verion 79 or newer)
- Opera

When a browser does not support shadow DOM, it is important to have a scoped CSS to fall back on.

### 4.2.3 StencilJS SSR

Stencil defines Server-Side Rendering as "The process of rendering content to a client based on an HTTP request. A client makes a request and the server processes it, returning rendered HTML back to the client. The Client then hydrates that HTML and bootstraps the client-side JS app" [Steb].

If a page must be rendered server-side. Stencil's **hydration functionality** can be used in any Node.js based server.

This is done with the **hydrate app**. It is a Stencil output target that generates a module which can be used on a NodeJs server to hydrate HTML and implement SSR [Stee].

#### How to Use the Hydrate App

First, define an output target in the stencil.config.ts:

```
    outputTargets: [  
  {  
    type: 'dist-hydrate-script',  
  },  
];
```

This will generate a hydrate app that can be imported by any Node server [Stee].

After publishing the component, it can be imported:

```
import { hydrateDocument, renderToString } from 'yourpackage/hydrate';
```

The hydrate app module has two export functions:

- `renderToString`
- `hydrateDocument`

**renderToString()** is designed for rendering HTML content into a string on the server side, with error handling and the option to clean up the rendering environment afterward. In other words, it outputs a static HTML string that reflects the state of the component at the time of rendering. The fact that the output is static also means that the component must be hydrated before it becomes interactive again.

**hydrateDocument** does the same, but not only for individual components but for the whole document.

## 4.3 Angular

This section provides a detailed overview of Server-Side Rendering (SSR) with Angular version 16.

### 4.3.1 Overview

Angular is a cross-platform JavaScript framework. It works seamlessly across web, server, mobile and desktop environments. It is able to craft high-performance and scalable web applications. Angular's architecture is hierarchical. An application consists of one or multiple components, each of which can consist of one or multiple components itself. These components represent and control a particular portion of a web page, called a view [Bam23].

### 4.3.2 SSR in Angular

Angular supports SSR through a library, called **Angular Universal**.

When an Angular application is rendered client side, all pages that are created dynamically, in the DOM of the browser. This happens during the usage of the application.

With Angular Universal these pages are created on the server, statically. This happens during application runtime. It thus creates a static version of the Angular application. This also means that it runs without having JavaScript enabled [Bam23].

#### Hydration in Angular

As previously discussed in section 4.1, hydration refers to the process of adding interactivity to a static HTML page.

Before the release of Angular 16, Angular utilized a technique known as **destructive hydration**. This involved a series of steps [Sun]:

- The browser makes a request to the server.
- The server responds by sending back the DOM structure of the website.
- The browser then renders this structure as a non-interactive page.
- Subsequently, JavaScript bundles are downloaded in the browser.
- The Angular client application starts up, loads these bundles, and begins its bootstrapping process.

As a result, the entire page was reloaded. This approach was called "destructive" because it necessitated the complete reloading of the page.

A key issue with this method was the potential for content "flashing." This flashing occurred when the markup rendered on the server-side was replaced by the client-side rendered content, as for a short period of time, the page is just blank.

In Angular 16 **non-destructive hydration** was introduced. In this approach, the DOM markup already rendered on the server-side is preserved and reused. Instead of discarding and reloading the entire page, Angular navigates through the DOM, it attaches event listeners and binds data as needed to finalize the rendering process [Sun]. Using non-destructive hydration has several benefits:

- Since event listeners can be reattached to the previous state of a component to enable interactivity without reloading the entire page, the application feels more **interactive** and **responsive**
- Since not the whole page must be reloaded, the **time to interactivity** is smaller



As mentioned in Section 4.3.1, Angular consists of components. A further advantage of non-destructive hydration is, that it is possible to exclude certain components from hydration. This can be done by adding the:

```
ngSkipHydration
```

attribute to a component's tag. It will then skip hydrating the entire component [Sun].

## Stencil web components in Angular with SSR

Stencil is able to generate **Angular component wrappers** for the web components.

Note that it is possible to integrate Stencil web components without using the Angular component wrappers. However, using this wrappers comes with multiple benefits over using them as just web components [Stea].

- 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

An overview of how to use how to set up a project using these component wrappers can be found in Section 9.4.

## Angular 17

With the release of Angular 17, hydration is stable and production-ready [Ang]. The Angular Universal package was moved to the Angular CLI.

In Angular 17, SSR can be added with:

```
ng add @angular/ssr
```

Compared to before:

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

## 4.4 SSR in Next.js

Next.js is a React-based framework, renowned for streamlining the development of modern web applications. It offers a comprehensive suite of built-in features, including routing, code splitting, and Server-Side Rendering. By providing these functionalities, Next.js enhances the development experience, offering a structured approach to building efficient web applications.

### 4.4.1 Implementation Strategies

Next.js provides two distinct methods for preparing pages [ima]: Static Generation and Server-Side Rendering as illustrated in Figure 4.2. Each method differs in how and when the HTML is generated, offering flexibility to developers based on the specific requirements of their applications.

Next.js advises choosing Static Generation for pages that can be pre-rendered before a user request. This approach suits pages without frequent content changes. Conversely, for pages with frequently updated data or where content changes with each request, Server-Side Rendering is recommended. Although slower<sup>4</sup>, it ensures up-to-date content. Alternatively, client-side JavaScript can be used for populating dynamic data. This guidance from Next.js helps developers decide the best pre-rendering strategy based on their specific page requirements. More details can be found in the Next.js documentation.

### Server-side Rendering

The HTML is generated on **each request**.

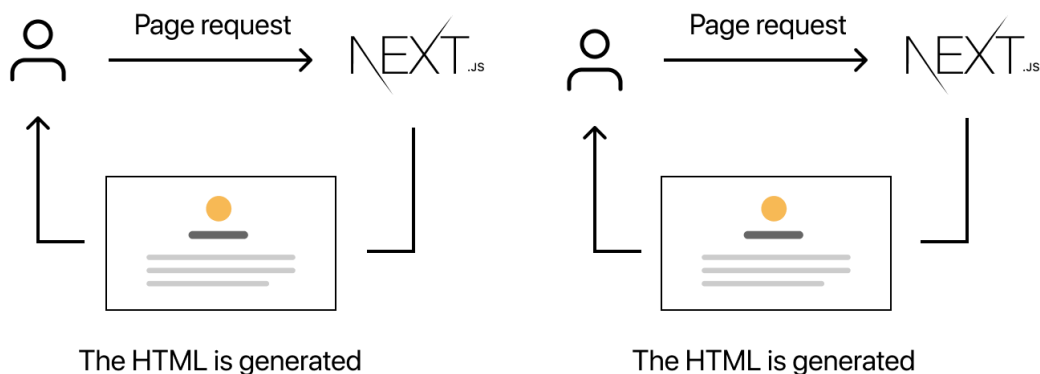


Figure 4.2: SSR in Next.js [ima]

### 4.4.2 React Hydration in Next.js

Next.js leverages React's hydration to maintain consistency between server-rendered content and client-side interactivity. This process allows React components to be integrated

<sup>4</sup>In Server-Side Rendering (SSR) with Next.js, the perception of being slower arises from the process where each page request involves generating the HTML on the server. This includes fetching data, executing JavaScript, and rendering components on the server for each request. While this ensures up-to-date content, it can take more time compared to serving static files or pre-rendered content. However, SSR improves Initial Page Load (IPL) time for users and benefits SEO.

seamlessly within the browser DOM, previously generated by `react-dom/server`. The concept of hydration in Next.js ensures that the initial HTML, rendered on the server, is enriched with dynamic functionalities on the client side, maintaining a balance between performance and interactivity.

The distinction between pre-rendering in Next.js (as shown in Figure 4.3) and a standard React.js application (illustrated in Figure 4.4) is starkly evident. These figures effectively demonstrate how Next.js enhances the pre-rendering process compared to a plain React.js setup.

## Pre-rendering (Using Next.js)

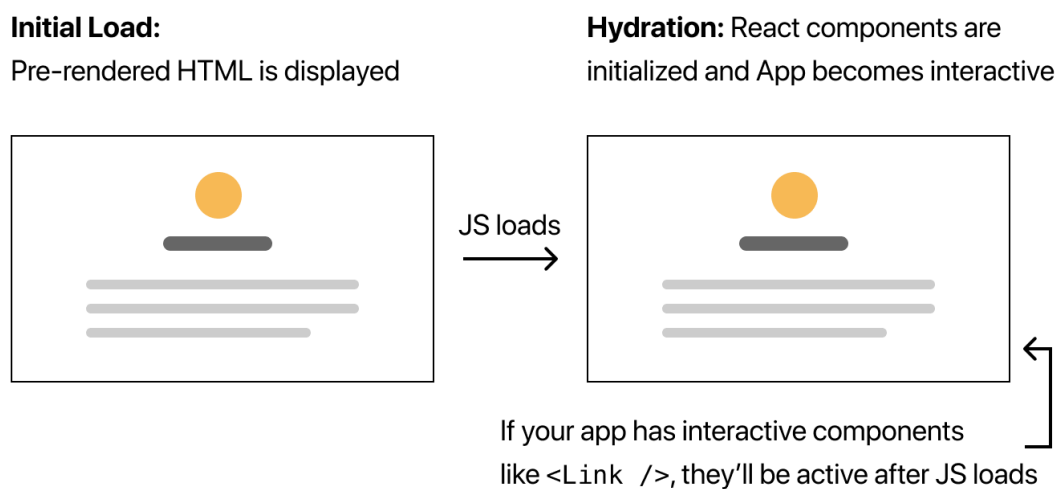


Figure 4.3: Pre-rendering using Next.js [ima]

### 4.4.3 The Role of `getServerSideProps`

The `getServerSideProps()` (Listing 4.2) function in Next.js is a pivotal aspect of implementing SSR. This asynchronous function is executed upon each request to the respective page, facilitating the dynamic retrieval of data, often from external APIs. This function exemplifies the ease with which Next.js integrates server-side processes into web development workflows.

### 4.4.4 Example: Basic Next.js SSR Usage

To illustrate the practical application of SSR in Next.js, consider a simple example that fetches and displays data from an external API:

## No Pre-rendering (Plain React.js app)

### Initial Load:

App is not rendered

### Hydration:

React components are initialized and App becomes interactive

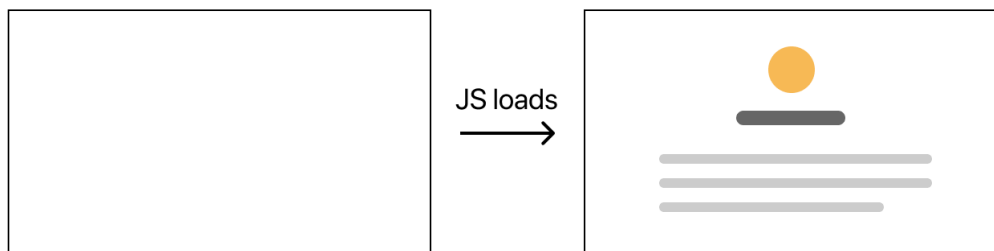


Figure 4.4: No Pre-rendering, Plain React.js app [ima]

Listing 4.2: Basic Next.js SSR Usage

```
1 import React from 'react'
2 import fetch from 'isomorphic-unfetch'
3
4 const Home = ({ data }) => {
5   return (
6     <main>
7       <h1>Documentary films</h1>
8       <ul>
9         {data.map(item => (
10           <li key={item.name}>{item.name}</li>
11         ))}
12       </ul>
13     </main>
14   )
15 }
16
17 export async function getServerSideProps () {
18   const response = await fetch('https://swapi.dev/api/films/')
19   const data = await response.json()
20
21   return { props: { data: data.results } }
22 }
23
24 export default Home
```

In this example, `getServerSideProps()` fetches data server-side, which is then rendered in the `Home` component. This approach demonstrates the effectiveness of SSR in

Next.js for building dynamic, data-driven web pages.

#### **4.4.5 Conclusion**

Next.js significantly simplifies the implementation of SSR in web development, making it an ideal choice for developers looking to balance performance, SEO, and interactivity. Its built-in features, coupled with the power of React, provide a robust foundation for building modern, efficient web applications.

# Chapter 5

## Problem Analysis

In order to narrow down the source of the error, we conducted a couple of general experiments.

### 5.1 Nested Components

#### Background

Swisscom is using nested Stencil web components in their SDX library. One such example is the **tabs component**. It can bundle multiple **tabs-item components**. A picture of the tabs component can be seen in Figure: 5.1. In Listing 5.1 is the html-code used to build the tabs component which is displayed on the picture.



Figure 5.1: Tabs Component

Listing 5.1: HTML Tabs

```
1 <sdx-tabs sr-hint="Tabs with text.">
2   <sdx-tabs-item label="Tab 1">
3     This is the content of Tab 1.
4   </sdx-tabs-item>
5   <sdx-tabs-item label="Tab 2">
6     This is the content of Tab 2.
7   </sdx-tabs-item>
8   <sdx-tabs-item label="Tab 3 with very long text that
9     will be truncated" selected>
10     This is the content of Tab 3.
11   </sdx-tabs-item>
12   <sdx-tabs-item label="Tab 4">
13     This is the content of Tab 4.
```

```

13     </sdx-tabs-item>
14     <sdx-tabs-item label="Tab 5 with extra text">
15         This is the content of Tab 5.
16     </sdx-tabs-item>
17     <sdx-tabs-item label="Tab 6" disabled>
18         This is the content of the disabled Tab 6.
19     </sdx-tabs-item>
20 </sdx-tabs>

```

Currently, when rendering this tabs component with Stencils SSR, the rendered component does not look as intended, as can be seen in Figure 5.2.

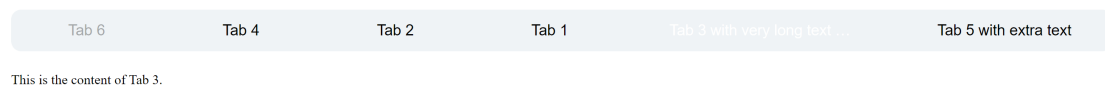


Figure 5.2: Tabs Component Rendered Falsely

## Purpose

The goal of this experiment is, to determine whether nested web components are rendered correctly using Stencils SSR functionality. In other words: is the tabs component rendered incorrectly because all nested components are rendered incorrectly. Or are nested components rendered correctly and the problem is something entirely different.

## Method

Two simple Stencil components were generated:

- innerComponent
- outerComponent

A detailed documentation of how to generate Stencil web components can be found here in Section 4.2.1.

The innerComponent is a submodule of the outer component. In Listing 5.2 is the html-code of these two components nested together:

Listing 5.2: outer-component.html

```

1     <outer-component name="Max">
2         <inner-component firstName="Max" lastName="Muster"></
           inner-component>
3     </outer-component>

```

In order to understand what these components should actually be outputting, here is the code of both components:

Listing 5.3: outer-component.tsx

```
1 import {Component, h, Prop} from '@stencil/core';
2
3 @Component({
4   tag: 'outer-component',
5   styleUrls: 'outer-component.css',
6   shadow: true,
7 })
8
9 export class OuterComponent {
10   @Prop() name;
11   private getName(): string {
12     return this.name;
13   }
14
15   render() {
16     return (
17       <div>
18         <slot></slot>
19         <p>Hello, World! I m {this.getName()}</p>
20       </div>
21     )
22   }
23 }
```

Listing 5.4: outer-component.css

```
1 :host {
2   display: block;
3   color: orange;
4 }
```

Listing 5.5: inner-component.tsx

```
1 import {Component, h, Prop} from '@stencil/core';
2
3 @Component({
4   tag: 'inner-component',
5   styleUrls: 'inner-component.css',
6   shadow: true,
7 })
8
9 export class InnerComponent {
10
11   @Prop() firstName: string;
12   @Prop() lastName: string;
13
14   private getName(): string {
15     return this.firstName + this.lastName;
16   }
17 }
```



```

17 |
18 |     render() {
19 |         return <div>Hello, World! I m {this.getName()}</
    |         div>
20 |     }
21 | }

```

Listing 5.6: inner-component.css

```

1 | :host {
2 |     display: block;
3 |     color: pink;
4 | }

```

So the expected output was first the innerComponents content "Hello World! I'm MaxMuster" in pink, then the outerComponents output "Hello World! I'm Max" in orange.

## Result

When hydrating the nested components, both components were displayed correctly and the styles are set correct as well. This can be seen in Figure 5.3.



The image shows a screenshot of a web component's output. It contains two lines of text. The first line is "Hello, World! I'm MaxMuster" and is rendered in a pink color. The second line is "Hello, World! I'm Max" and is rendered in an orange color. The text is centered and appears to be part of a larger container.

Figure 5.3: Nested Components Output

In Figure 5.4 it is visible that the component's DOM was hydrated properly.

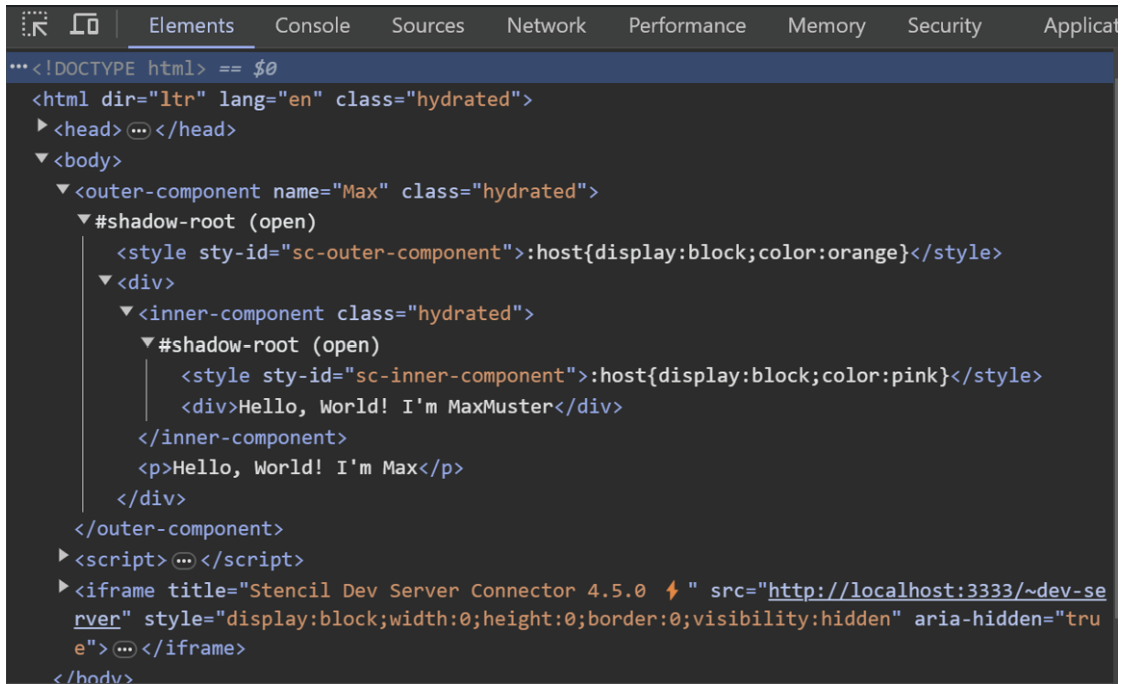
## Conclusion

This experiment shows that nested component work with Stencils SSR. The problem thus is not solved. It seems some other problem is present in the tabs component. Thus next, **Global Styles** will be inspected, as the application of those can also have a big impact on the style of the component.

## 5.2 Global Styles

### Background

Because nested components are not the reason for the incorrect styling of the web components, we have the suspicion that the global styles (or the lack of them) have an impact on the web components appearance after the hydrating done by Stencil. In Swisscom



```
...<!DOCTYPE html> == $0
<html dir="ltr" lang="en" class="hydrated">
  <head> ... </head>
  <body>
    <outer-component name="Max" class="hydrated">
      #shadow-root (open)
        <style sty-id="sc-outer-component">:host{display:block;color:orange}</style>
        <div>
          <inner-component class="hydrated">
            #shadow-root (open)
              <style sty-id="sc-inner-component">:host{display:block;color:pink}</style>
              <div>Hello, World! I'm MaxMuster</div>
            </inner-component>
            <p>Hello, World! I'm Max</p>
          </div>
        </outer-component>
      <script> ... </script>
      <iframe title="Stencil Dev Server Connector 4.5.0 ⚡" src="http://localhost:3333/~dev-server" style="display:block;width:0;height:0;border:0;visibility:hidden" aria-hidden="true"> ... </iframe>
    </body>
```

Figure 5.4: Nested Components DOM

SDX, global stylesheets are used for the styling of web components. In hydrated web components from Swisscom's SDX library, not all styles are applied correctly.

## Purpose

The purpose of this experiment is to evaluate whether global styles are correctly applied on hydrated web components.

## Method

This experiment was done, using the tabs component. This is the same component already described in Section 5.1. In a first step the component was rendered using global styles sheets. In a second step, the global stylesheets were removed and all global styles that would have been applied were moved into a local CSS-file.

## Results

The result was the same, whether global styles were used or not. It can be seen in Figure 5.5. The removal of global styles made no change in the appearance of the web components. Therefore global styles were not the problem, all that they did was correctly applied onto the web component.

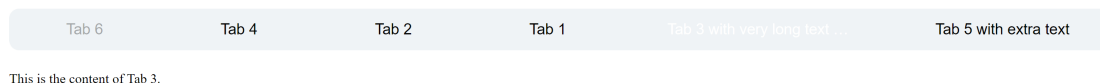


Figure 5.5: Styles applied SSR

## Conclusion

The problem here, seems to lie in the interactivity. As shown in the Figure 5.5 above, the general styles are applied. What is not applied, however, is the highlight of the currently selected tab. This is because the interactivity was lost. It is not known which tab should be highlighted and when to change it, as we have lost the interactivity. In other words: In the tabs component, some styles are determined dynamically by JavaScript. Exactly those styles are missing.

## 5.3 NPM Packages Update

### Background

In the world of modern web development, Stencil comes as a powerful tool for building efficient and reusable web components. Developed by Ionic, Stencil focuses on delivering high-performance components while providing developers with the flexibility to create web applications with ease. An important feature of Stencil is its support for Server-Side Rendering (SSR), an essential capability for optimizing search engine visibility and enhancing initial page load performance.

The incorporation of SSR in Stencil, however, presents developers with certain challenges. Issues such as rendering inconsistencies, performance bottlenecks, or unexpected behavior are reported by the developer community.

The hypothesis is that newer versions of dependencies might contain bug fixes or optimizations that could positively impact Stencil's SSR functionality.

### Purpose

The purpose of this experiment was to investigate whether updating npm packages within the project's environment could serve as a viable solution to the existing challenges associated with Stencil's Server-Side Rendering (SSR) feature.

### Method

The experiment involved a systematic update of npm packages in the project's environment. This process included identifying the current versions of dependencies used by the Stencil web components project and checking for newer versions available. The method aimed to assess whether a simple update of npm packages could have a noticeable impact on the identified issues with Stencil's SSR feature.

## Result

Following the update of npm packages, it was observed that no significant improvements were noticed in relation to the pre-existing issues with Stencil's SSR feature.

## Conclusion

Based on the results, it is concluded that the experiment does not yield the anticipated improvements, indicating that the issues in question may require more complex or alternative solutions. The experiment underscores the need for further research and investigation into the issues with Stencil's SSR feature, possibly exploring other strategies to achieve the desired functionality. In summary, this experiment is a valuable step in the ongoing research project, shedding light on the limitations of npm package updates in addressing the identified SSR issues within Stencil web components.

## 5.4 Chat GPT

### Background

In the world of web development, getting things set up right is important. We're focusing on making sure Node.js, a key tool, is correctly installed and configured. This is especially important when dealing with Stencil, a tool for creating web components, utilising Server-Side Rendering (SSR) to improve performance and SEO.

### Purpose

This experiment is all about checking if GPT Chat can give us reliable help in setting up Node.js for Stencil web components, including the tricky SSR feature.

### Method

To test this, we simply asked GPT Chat for guidance on how to install and configure Node.js for Stencil web components. We had to stick with GPT-3 because the fancier GPT-4 needed a premium subscription.

### Result

What we found wasn't perfect. GPT Chat's advice wasn't always right or reliable. There were times when it seemed unsure or gave wrong information, which could lead to setup mistakes. Also, GPT Chat's knowledge is a bit old, with the last update in January 2022, and it doesn't know what's happening in real-time.

### Conclusion

Based on the results, it can be concluded that while GPT Chat (GPT-3) can offer some guidance, its capabilities are limited and inconsistent, underscoring the need for caution

and verification. Given these limitations, it becomes evident that we need to explore other ways to achieve the SSR functionality we're aiming for.

# Chapter 6

## Experiments

### 6.1 Environment setup

??

We aimed to make StencilJS work better for Swisscom’s web components and to understand Server-Side Rendering (SSR). To do this, we organized our testing environment. This part explains the basic things we set up for our research.

#### 6.1.1 Used repositories

We relied on two main folders provided by Swisscom, which were the starting point for our experiments. These folders were given based on what others had learned before:

- **Mini-SDX repository**
  - StencilJS project for Swisscom’s web components
  - Trimmed-down version, featuring only SdxTabs and SdxTabsItem components
  - will be referred to as the `mini-sdx` repository
- **SSR-SDX repository**
  - Functioning as a server powered by NodeJS with ExpressJS
  - Here, the web components package from the `mini-sdx` repository, complete with its hydrate script, was imported
  - Serves as the SSR testing ground
  - Will be referred to as the `ssr-sdx` repository

#### 6.1.2 Extra Projects

Alongside these main folders, we also created a few separate projects to run specific experiments. As we get into each experiment, we will mention if there are any extra projects involved.

## 6.2 Problem Solution

### 6.2.1 Stencil ESM Scripts

#### Background

In our exploration of StencilJS, we notice the presence of styles but a lack of interactivity in our components. Our assumption is that the Hydrate Script generated by StencilJS provides HTML with styles but doesn't include accompanying scripts.

#### Purpose

The identified issue is a lack of interactivity despite having styles in our StencilJS components. Our working hypothesis is that the Hydrate Script might be missing the necessary scripts. The goal of this experiment is to determine if including the ESM script directly, similar to the `www` output target<sup>1</sup>, affects the interactivity of the components.

#### Method

The StencilJS components were built and then imported into the `sdx-ssr` project. Following the guidelines provided by Stencil's documentation, the Hydrate script was imported. After the `renderToString()` function was executed, however before delivering the results to the client, we utilized the `replace()` method to add a script tag to the HTML rendered by the Hydrate script. This script tag referenced to the ESM script within the web component's package.

Listing 6.1: Adding ESM scripts after `renderToString`

```
1  ...
2  const results = await renderToString(tabs, {
3    prettyHtml: true,
4    removeScripts: false
5  });
6
7  const withStencilScripts = results.html.replace(
8    '</head>',
9    '<script type="module" src="path/to/script.esm.js"></script>'
10   '</head>`
11  );
12
13  res.send(withStencilScripts);
14  ...
```

---

<sup>1</sup>The 'www' output target type in Stencil is designed specifically for web applications and websites that are hosted on an HTTP server. This type is particularly advantageous for sites that can utilize pre-rendering.

## Result

This approach resulted in a significant improvement in the interactivity of the components. Through the integration of the ESM script using a script tag, we successfully transformed the components into interactive elements. Notably, the tabs became clickable, and dynamically set styles are now functioning as intended.

## Conclusion

Based on the results, it becomes more evident that after using `renderToString()`, loading the modules script on the client side is important for achieving enhanced interactivity in StencilJS components. The experiment sheds light on a potential gap in the Stencil documentation regarding this aspect.

Currently, our understanding is growing, and we are becoming more certain that incorporating the ESM script directly contributes significantly to the desired interactivity. Looking ahead, it would be beneficial to engage with Stencil developers to validate our findings.

### 6.2.2 ESM Script from Full Library

#### Background

In this project, we are exploring the compatibility of the combination of hydrate script from `mini-sdx` version of Swisscom components with the ESM script from the full SDX Swisscom library.

The main objective is to satisfy our curiosity regarding the application's ability to seamlessly integrate the ESM script from the full Swisscom library while operating with a minimal version of hydrate script from `mini-sdx`.

Notably, the CDN link to the full script was graciously provided by Daniel Berkelmann from Swisscom.

#### Purpose

This experiment is driven more by curiosity than a specific problem-solving objective.

The primary purpose is to satisfy our interest in understanding whether the `sdx-ssr` application can handle the integration of the ESM script from the full Swisscom library while rendering is done by a minimal version of hydrate script (`mini-sdx`).

We found it interesting to test because `mini-sdx` is like a snippet of the full Swisscom library. It could also be that some functionalities are missing in `mini-sdx`, so we anticipate different behavior from what is expected, for example, that the selected tab is not marked in pre-rendered result.

The goal is also to observe the behavior and identify any potential issues that may arise from combining these scripts.



## Method

The experiment consisted of configuring the `sdx-ssr` application to use hydrate script from `mini-sdx`, while attempting to bind the ESM script from the full Swisscom library. The experiment was executed by rendering the `tabs` component to observe the behavior and potential conflicts arising from combining scripts.

## Result

The experiment revealed that attempting to bind the ESM script from the full Swisscom library while using the minimal version of hydrate script from `mini-sdx` led to the `tabs` component being rendered twice. The first instance worked as expected (even with navigation arrows), while the second instance exhibited malfunctioning behavior (see Figure 6.1), more precisely the second instance was not hydrated on client-side.

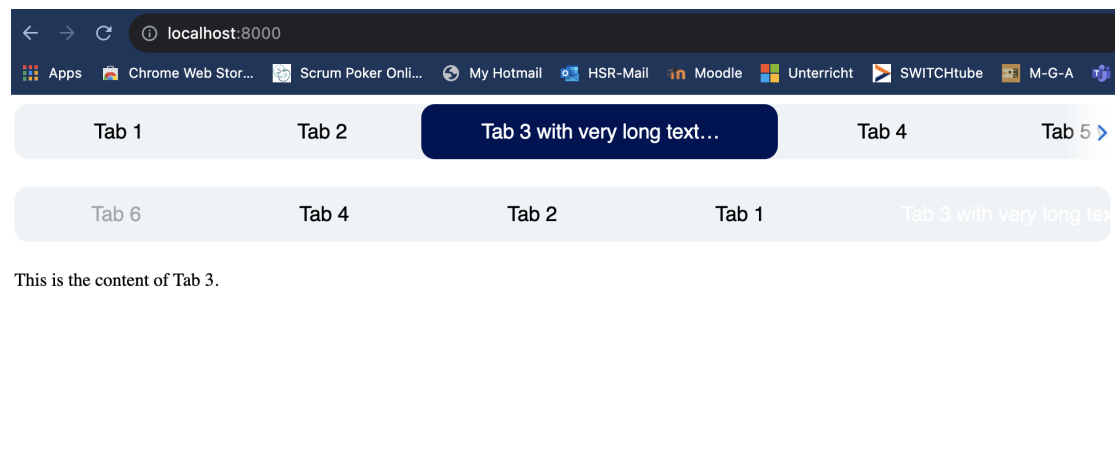


Figure 6.1: Result of mixing scripts

## Conclusion

Based on the results, it can be concluded that, even though the projects belong to the same components library, attempting to mix scripts from both of projects resulted in unexpected rendering behavior. This brings attention to the importance of careful consideration and potential adjustments when integrating scripts across different library versions.

### 6.2.3 Contact with Stencil Community

#### Background

After our investigation into the "ESM Scripts" experiment, our confidence in the understanding that the hydrate script exclusively provides static HTML without accom-

panying scripts is more robust. To consolidate this understanding and confirm our assumption, it is useful to take the proactive step of directly approaching the Ionic Team.

## Purpose

The purpose behind reaching out to the Stencil community was to confirm our growing understanding of the hydrate script.

This step is essential for gaining clarity on how to effectively hydrate components with JavaScript after using `renderToString()`. The goal was to seek recommendations from the community regarding the subsequent steps for achieving proper hydration. Additionally, we aimed to confirm if the manual addition of scripts was a suitable method for utilizing StencilJS components with Server-Side Rendering (SSR) and Hydration. We also wanted to find how the `hydrate/index.js` script works in general.

## Method

To engage with the Stencil community, we used the Discord platform and posted our queries in the StencilJS help channel, a link thoughtfully provided by a developer of Swisscom. In our post, we provided comprehensive details about the issue, step-by-step reproduction instructions, and a demo repository for reference.

In an effort to confirm the viability of our approach involving manual script inclusion, we posed three fundamental questions to the community:

- If `renderToString()` exclusively produces static HTML without accompanying scripts, what would be the recommended subsequent steps for effectively hydrating the components with JavaScript?
- Is the manual addition of scripts a suitable method for utilizing StencilJS components with SSR and Hydration?
- Could the community shed light on the inner workings of the `hydrate/index.js` script, providing a deeper understanding of its functionality?

Access to the entire discussion history by following [this link](#).

## Result

Certainly, adding scripts manually has been crucial to make the components work correctly. Also, a brief explanation of the hydrate script's purpose was given. This serves two main purposes:

- Ensuring that these components render seamlessly, eliminating any initial un-rendered appearance, thereby mitigating the "flashing" issue<sup>2</sup>

---

<sup>2</sup>Flashing issue, in this context, refers to a phenomenon similar to "Flash of Unstyled Content

- Enhancing the document’s search engine optimization (SEO) by improving accessibility and indexability for search engines, thereby increasing its overall SEO-friendliness

## Conclusion

Our interaction with the Stencil community validated the importance of manual script inclusion for effective component rendering. The insights gained contribute to a more comprehensive understanding of how StencilJS and its SSR function work.

### 6.2.4 Hydrate Script Results vs. Hydrated HTML

#### Background

After successfully incorporating the ”ESM Scripts into the rendered HTML”, it became apparent that certain issues persisted:

- Initially, there is an attribute `selected` configured to indicate one of the tabs as the pre-selected option upon the initial page load (see 6.2).

Listing 6.2: ”sdx-tabs” with pre-selected tab

```

1 <sdx-tabs sr-hint="Tabs with text.">
2 <sdx-tabs-item label="Tab 1">
3   This is the content of Tab 1.
4 </sdx-tabs-item>
5 ...
6 <sdx-tabs-item label="Tab 3 with ..." selected >
7   This is the content of Tab 3.
8 </sdx-tabs-item>
9 ...
10 </sdx-tabs>

```

However, it has come to our attention that this intended behavior is not currently being realized.

- Expected behavior: Tabs with `selected` attribute should be highlighted on initial load. 6.2
- Current behavior: Tabs with `selected` attribute are not highlighted and `selected` attribute is no more present if analysing displayed HTML 6.3
- Within the first few milliseconds until the ESM script loads on the client side, there is an occurrence of the tabs displaying in an incorrect order 6.4, which could be referred to as the ”flashing” issue.

---

(FOUC)”. FOUC occurs when a web page temporarily displays with the browser’s default styles before an external CSS stylesheet is loaded. This happens because the web browser renders the page before fully retrieving all necessary information. The page eventually displays correctly once the style rules are loaded and applied. However, this brief visual inconsistency can be noticeable and potentially distracting for users.

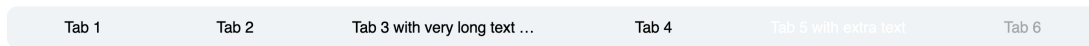


Figure 6.2: Initially selected tab not selected



This is the content of Tab 3.

Figure 6.3: Initially selected tab selected

**Initial Page Load:**



**Hydrated with ESM script:**



Figure 6.4: Incorrect order on Initial Page Load

## Purpose

We aim to investigate two primary issues with the tabs in our web application. The first is understanding why a tab, despite having the `selected` attribute, is not being pre-selected as expected. The second issue pertains to a brief visual inconsistency with ordering, observed in the tabs during the loading of the ESM script. We seek to determine whether this inconsistency is related to the "flashing" issue, or if it stems from a different cause.

In pursuit of enhancing the functionality, our objective is to identify the root causes of these issues. This understanding will enable us to develop effective strategies to mitigate and resolve the observed challenges.

## Method

We conducted a thorough examination of the source code within both the `mini-sdx` and `sdx-ssr` projects in the hopes of uncovering any noteworthy insights. We also explored the responses received from the server and conducted further research. To assess the impact of these issues on the user experience, we deliberately introduced script-loading delays by applying throttling.

## Result

The incorrect tab order appears to be a consequence of the hydrate script generated by Stencil, specifically the result of `renderToString`. As the script renders, it conducts some restructuring and renaming of the nodes, potentially leading to the incorrect order

in the output. Furthermore, upon the client script loading, a repaint operation occurs, which triggers the "flashing" issue. However, this issue is not related to "Flash of Unstyled Content".

Additionally, the initially configured "selected" attribute in the HTML appears to be getting lost in the rendering process.

When comparing the impact of these two issues, it becomes evident that the "flashing" issue carries a more significant risk. If the script loading is delayed, the user may observe the incorrect order of elements, which could potentially pose risks, particularly depending on the content contained within the tabs.

## Conclusion

To summarise, the investigation has provided insight into the causes of the `selected` attribute issue and the "flashing" issue when loading the ESM script. Fixing these issues is critical to improving the overall user experience with StencilJS components. Next steps involve researching existing fixes, if any, and developing solutions to address these issues to ensure the seamless functioning of tabs in various scenarios.

## 6.2.5 StencilJS SSR Issues – Community Help

### Background

In our ongoing exploration of StencilJS Server-Side Rendering (SSR), we have identified potential challenges, particularly in relation to the `renderToString()` function. Despite our efforts, we can not find any specific resources that precisely address the particular issue we are currently facing.

### Purpose

Based on our research, it appears that there could be potential challenges with the Server-Side Rendering (SSR) feature offered by StencilJS. However, it's worth noting that we haven't come across specific resources that precisely address the particular issue we are currently encountering.

We have chosen to reach out to the StencilJS community once more, seeking their assistance and guidance.

### Method

Since the issue remains associated with the `renderToString()` function, we have posted a new question within the same thread that we initially created in the "Contact with Stencil community" experiment. In this new post, we have provided a detailed description of the issue and asked if anyone might have insights into why it is occurring, including the possibility of it being a bug.

## Result

We have received a response that supports the notion that the issue related to the incorrect order has a significant likelihood of being a bug. According to the StencilJS community, there are several ongoing issues with Server-Side Rendering (SSR) at the moment. It's worth noting that a more comprehensive patch addressing these concerns is currently in a separate branch. Unfortunately, it cannot be merged into the main version yet due to some substantial pending pull requests (PRs).

Additionally the version with some bug fixes was mentioned. It is an older alpha release – `@stencil/core@3.3.0-dev.1685496483.a311f36`, which incorporates a series of fixes that may be relevant to addressing the issues we've encountered.

## Conclusion

While our exploration into StencilJS SSR issues has not yielded an immediate solution, the community's acknowledgment of potential bugs and ongoing efforts to address them is valuable.

The mention of an older alpha release with possible fixes provides a potential avenue for resolving our challenges.

The journey continues with a deeper dive into the suggested version and may include further communication with the Stencil community.

## 6.2.6 Old Alpha Release of StencilJS

### Background

Following a recommendation from a previous experiment "StencilJS SSR Issues – Community help", we were directed to explore the `@stencil/core` version, potentially containing numerous bug fixes related to Server-Side Rendering (SSR). In our pursuit to verify if an old alpha release of Stencil could address our SSR issues, we chose to build the `mini-sdx` web components using `@stencil/core@3.3.0-dev.1685496483.a311f36`.

### Purpose

The primary objective is to assess whether the utilization of the old alpha release of Stencil (`@stencil/core@3.3.0-dev.1685496483.a311f36`) would resolve the SSR issues identified in our previous experiments. Specifically, we aim to verify if this version addresses issues related to the `select` attribute and improves the overall functionality of our components within the `mini-sdx` project.

### Method

We are proceeding with the installation of `@stencil/core@3.3.0-dev.1685496483.a311f36` in the `mini-sdx` project. Following this installation, we initiated the build and publishing process for the components package, replicating the workflow from previous experiments. Subsequently, we imported

these components into `sdx-ssr` to evaluate the application's seamless functionality. This comprehensive approach aimed to provide insights into the impact of the new StencilJS version on the `mini-sdx` project.

## Result

The use of `@stencil/core@3.3.0-dev.1685496483.a311f36` has indeed resolved the issue with the initially set `select` attribute, ensuring that the selected tab is now properly marked and visible in the frontend. However, it is worth noting that the issue related to the wrong order and the flashing effect remains unaddressed.

This experiment has shed light on the fact that the hydrate script generated by Stencil exhibits several issues. This, in turn, calls for the implementation of a workaround to mitigate these challenges. It is also a possibility to wait for the fixes to be merged into the main version, although the `@stencil/core@3.3.0-dev.1685496483.a311f36`, published in June 2023, includes significant pull requests, which might take some time to be incorporated according to insights from the Stencil Community.

## Conclusion

In conclusion, the experiment with the old alpha release of Stencil (`@stencil/core@3.3.0-dev.1685496483.a311f36`) has successfully resolved one of the identified SSR issues. However, it has brought to light the persistence of other challenges, pointing to the need for alternative solutions. The decision to wait for main version fixes or explore alternative strategies remains in our ongoing efforts to enhance SSR functionality with hydrate script generated by Stencil.

### 6.2.7 Workaround: Wrong Order – with Lifecycle

#### Background

As already described, the order in which certain elements are loaded from Stencil, before the hydration, is incorrect. One example is the tabs component, seen in Figure 5.5. It is visible there that Tab6 is the leftmost, instead of Tab1. This is resolved once the JavaScript is loaded. But it is wrong for a short period of time (a few milliseconds) and can lead to "flashing" when corrected after this time. In other words, when the correct order is loaded, the component appears to "flash".

#### Purpose

The purpose of this experiment is to find a workaround of this "flashing" issue. Meaning it aims to provide a way such that the correct order appears instantly.

#### Method

The attention was turned to the **component lifecycle methods**. Two of these methods are used in the `mini-sdx` project.

- `componentWillLoad`
- `componentDidLoad`

The basic idea was the following: Once the whole component was loaded (including the JavaScript) the component will display the correct order. Thus, by setting the component "invisible" in CSS until after the component was loaded would resolve the issue with the "flashing".

`componentWillLoad` waits for one element to appear and then sets the default element. This meant one `tab-items` element in this case. In other words, as soon as something appears it recognizes that a component will be loaded here.

`componentDidLoad` sets the layout and subscribes on the store. After everything is loaded, the JavaScript is also active. Now the subscribe on the store can happen. The store was storing dynamic styles. Thus, only after this subscription all styles were active and the component was displayed correctly.

The idea now was to make the component "invisible" during the time it is loading and only set it to visible after it was loaded (after `componentDidLoad` was called).

To achieve this, an additional state needs to be introduced:

Listing 6.3: Add state

```
1 | State() isComponentLoaded: boolean = false;
```

This can be set to true once the component did load (in the `componentDidLoad` method):

Listing 6.4: hydrate/index.js

```
1 | public componentDidLoad() {
2 |     //... other code
3 |
4 |     this.isComponentLoaded = true;
5 | }
```

In the render function it is then checked if the component is loaded correctly or not:

Listing 6.5: hydrate/index.js

```
1 | const didLoad = this.isLayoutReady ? this.
   |   getComponentClassNames() : 'hidden';
```

Where 'hidden' was just a simple CSS-class:

```
.hidden {
  visibility: hidden;
}
```



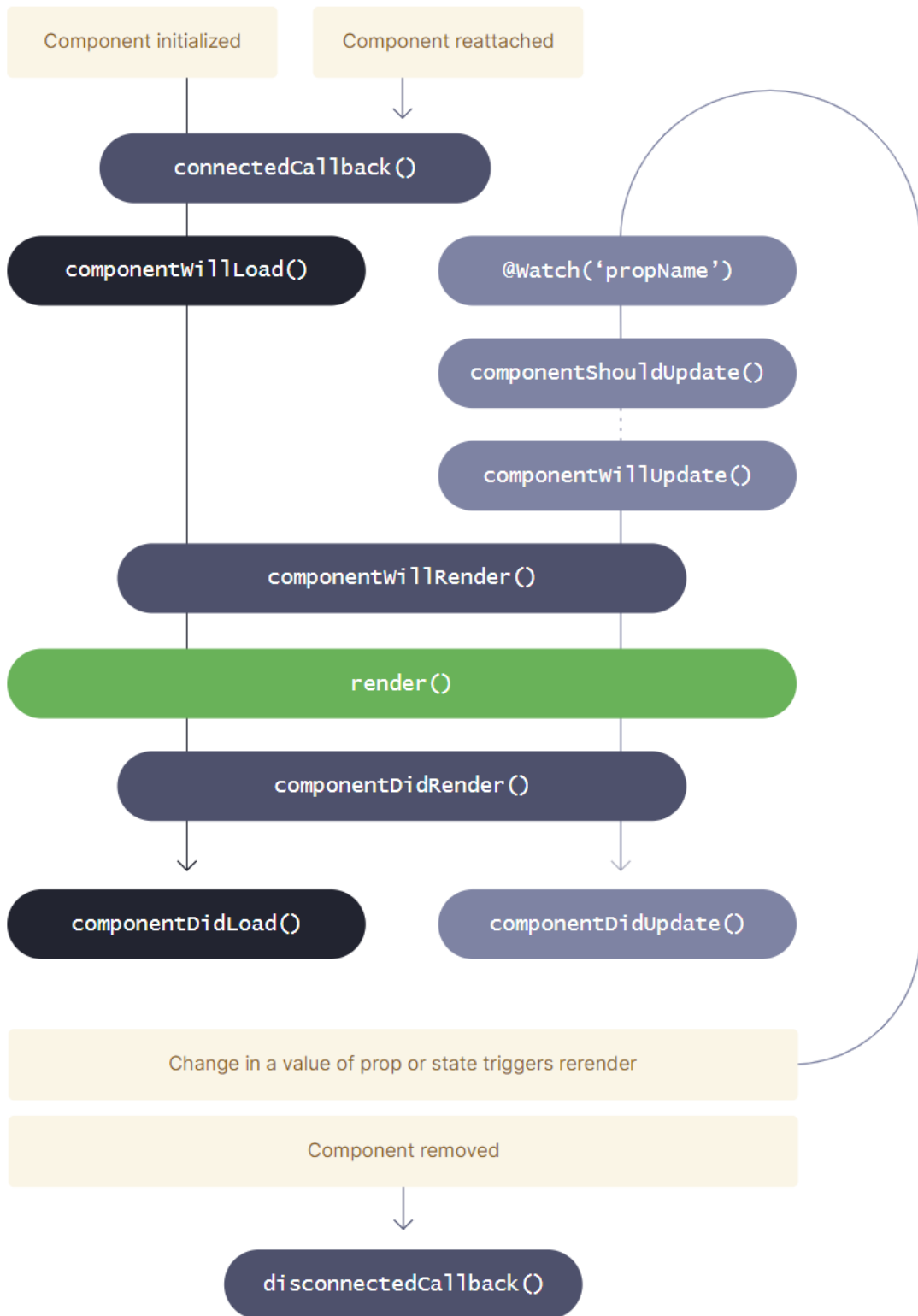


Figure 6.5: Component Lifecycle Methods (from [Stec])

## Results

These changes resulted in the correct order being displayed. The time that is needed to load a component was normally minimal and thus not noticeable. However, this is just a workaround and in case of long load times the component will not show up at all, resulting in a bad user experience.

## Conclusion

As mentioned in results, this works, however it also defeats part of the reason to do SSR. To "hydrate" the JavaScript later but already have something to show (the HTML). Thus, a solution only using CSS is preferable, see Section 6.2.8

### 6.2.8 Workaround: Wrong Order – with CSS

#### Background

The challenge of incorrect element order during the initial loading of certain components, as observed in the tabs component, leads to a brief but noticeable "flashing" effect. This occurs before the JavaScript corrects the order, impacting the visual experience of the component.

#### Purpose

This experiment seeks to address the challenge arising from Stencil's `renderToString()` function, which delivers static HTML with potential issues in element order. The primary goal is to investigate the feasibility of correcting the rendering order using CSS, given that `renderToString()` encapsulates stylings as well.

#### Method

##### First Attempt - Using `:key` Attribute:

It is not directly related to CSS, however it worse to note that the initial strategy involved setting the `:key` attribute<sup>3</sup> to the children of the tabs component. The intention was to leverage the uniqueness of keys on dynamic child elements, a common practice in frameworks like React and Vue. Outcome: Unfortunately, this attempt did not yield the desired impact, and no observable changes were noted.

##### Second Attempt - Flex with Order Property:

A secondary approach experimented with utilizing `flex` and setting the `order` property on tab-elements with help of map index. However, it became apparent that this strategy would be ineffective as the incorrect order was already established during the mapping process, causing the index to be applied to the wrong items. Realization: It became

---

<sup>3</sup>Keys tell which array item each component corresponds to, so that it can match them up later. More to Stencil's children.

clear that the initial approach was misguided after reviewing the outcomes.

### Third Attempt - CSS Grid:

The third attempt involved employing CSS Grid and using a unique identifier in the initial HTML. The experiment utilized the ID attribute of the tab-item as both the unique identifier and the definition for the grid area.

A 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.

In the second approach, it was noted that employing the map function posed challenges. When attempting to collect IDs directly from the HTML, the map iteration is already based on incorrect order.

As observed in second approach the map function. However in case the map function is used, collecting the IDs directly from html will not work. The end result would look like `grid-template-areas: 'tab6 tab4 tab2 ...'`.

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

In the course of this experiment, we decided to assign IDs in the format "tab1", "tab2" etc. to our elements, especially the tabs. 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.

In the "Tabs" component code, an array of `grid-template-areas` was generated:

Listing 6.6: `packages/app/src/components/tabs.tsx`

```
1 private getTabIds(tabsItemEls: HTMLSdxTabsItemElement[]) {
2   // return tabsItemEls.map(element => element.id); // WONT work
3
4   let gridAreas = [];
5   for (let i = 1; i <= tabsItemEls.length; i++) {
6     gridAreas.push('tab' + i)
7   }
8   return gridAreas;
9 }
10
11 public render() {
12
13   const {
14     tabsItemEls,
15     selectedTabsItemEl
16   } = this.state.get()
17
18   ...
19   const elementIdsForGrid = this.getTabIds(tabsItemEls)
20   ...
21 }
```

Since the array size was known, the areas could 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`.

Following that, the `grid-template-areas` is assigned to the parent element of the tab-elements:

Listing 6.7: packages/app/src/components/tabs.tsx

```
1  return (
2    ...
3    <div
4      class="tablist"
5      role="tablist"
6      aria-label={this.srHint}
7      ref={(el) => (this.tablistEl = el)}
8      style={{
9        display: `grid`,
10       gridTemplateAreas: ` '${elementIdsForGrid.join(' ')}' `,
11     }}
12    >
13      ...
14  )
```

The style attribute was then configured for each child, assigning the `grid-area` based on their respective IDs.

Listing 6.8: packages/app/src/components/tabs.tsx

```
1  const Tag = tabsItemEl.href ? "a" : "button"
2
3  return (
4    <Tag
5      class={this.getClickableElClassNames(tabsItemEl)}
6      style={{gridArea: tabsItemEl.id}}
7      ...
8    >
9      {tabsItemEl.label}
10   </Tag>
11 )
```

## Result

The implementation of the third strategy, utilizing CSS Grid and unique identifiers, successfully solved the issue of incorrect ordering initially caused by the `renderToString()` function.

## Conclusion

This experiment demonstrates a successful workaround for addressing the issue of incorrect rendering order in Stencil. Leveraging CSS Grid and utilizing unique IDs as grid areas effectively corrected the order, providing a solution that can be extended to other components experiencing similar challenges. This workaround ensures that even in scenarios where scripts may execute slowly, the correct rendering order is maintained on the client side.

## 6.3 Framework Integration

### 6.3.1 Integrate Stencil Web Components into an Angular Application

#### Background

Angular is a widely used framework in web development. Swisscom relies heavily on Angular. About 80 percent of their applications are using it. Angular is using their own implementation of server-side-rendering, more about Angular's SSR implementation can be found in Section 4.3. Angular provides dedicated output targets for the Angular framework. Note that it is also possible to integrate Stencil web components without using the output target.

#### Purpose

The purpose of this experiment is to determine if it is possible to integrate Stencil web components into an Angular application, using the from Stencil provided **Angular output target**, whilst using SSR.

#### Method

A project was set up, in order to test the possibility.

In this project, two main folders were needed:

- web-components (stencil web component(s))
- angular-app (angular application)

**Web-components** was a folder containing one or many Stencil web components. For this experiment the tabs component, already described in Section 2.2.

To set up the wrappers, the **Angular Output target** must get defined.

#### 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, the Angular output target was added to the hydrate output target:

```
stencil.config.ts
```

The file now looked like the following:

Listing 6.9: stencil.config.ts

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

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

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

With these changes, the Stencil web components were able to be imported into an Angular application.

## Changes to Angular Application

### 1. Changes to tsconfig.json

First, a relative path to the **dist directory of web-components** was added to the:

```
tsconfig.json
```

Listing 6.10: Relative path to the dist directory of web-components

```
1 | "paths": {
2 |     "@web-components/*": ["../web-components/*"]
3 | }
```

This kept the stencil component as a dependency of the Angular project.

### 2. Export Component

The goal now was to export the component inside the libs directory.

For this, two things had to 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 the following code was added, to export the imported web components:

Listing 6.11: web-components.module.ts

```
1 | import { NgModule, Inject, PLATFORM_ID } from '@angular/
   | core';
2 | import { CommonModule } from '@angular/common';
3 | import { SdxTabs, SdxTabsItem } from "../../src/libs/
   | stencil-generated/proxies";
4 | import { isPlatformBrowser } from '@angular/common';
5 |
6 | @NgModule({
7 |     declarations: [SdxTabs, SdxTabsItem],
8 |     imports: [CommonModule],
9 |     exports: [SdxTabs, SdxTabsItem]
10 | })
11 | export class WebComponentsModule {
```

```

12     constructor(@Inject(PLATFORM_ID) private platformId:
13         Object) {
14         if (isPlatformBrowser(this.platformId)) {
15             import('@web-components/loader').then(module =>
16                 {
17                     module.defineCustomElements(window);
18                 });
19         }
20     }

```

Lines 12-18 were necessary, as in Angular, when employing SSR or pre-rendering, the server did not recognize browser-specific objects like window. This happened because server-side rendering in Angular is executed in a Node.js environment, which was different from the browser environment.

After this, the general setup was complete.

```
npm run build
```

was run inside the web-component folder, to populate the Angular output targets that were just defined.

The imported web-components can be used in any Angular module. It can be imported:

```
import { SdxTabs, SdxTabsItem } from 'src/libs/web-components.module';
```

However, they were not yet rendered server-side.

### 3. Enable SSR

To enable SSR (non-destructive)

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

was run

The provideClientHydration function was imported as the provider of AppModule:

Listing 6.12: app.module.ts

```

1     import {provideClientHydration} from '@angular/platform-
2         browser';
3     // ...
4     @NgModule({
5         // ...

```



```

6     providers: [ provideClientHydration() ], // add this
      line
7     bootstrap: [ AppComponent ]
8   })
9   export class AppModule {
10     // ...
11   }

```

This did not work correctly, because the hydration process of Angular did not know what to do with the imported Stencil web components, as it was using custom tags.

For this reason, first the Stencil web components was hydrated and then the Angular application. In the Angular application, in the `server.ts` file the `app` function was modified in the following way:

Old:

Listing 6.13: `server.ts` (old)

```

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

```

New:

Listing 6.14: `server.ts` (new)

```

1   server.get('*', (req, res) => {
2     res.render(indexHtml, { req }, (err, html) => {
3       renderToString(html).then(({html}) => {
4         res.send(html);
5       });
6     });
7   });

```

This made sure that the Stencil web component was loaded first.

```
npm run dev:ssr
```

was run to see whether the application was working.

## Results

With the method described above, the integration of Stencil web components with the Angular output target into an Angular application was successful. It ran, however, it still had the same problems already seen in Section 6.2.1. Concretely this meant that some styles were not displayed correctly and the order upon reload was false.

## Conclusion

This experiment shows that using the Angular output target it is possible to integrate Stencil web components into an Angular application. As a next step it would be interesting to know how the performance differs between using SSR and not using SSR in this Angular application.

### 6.3.2 Tabs Integration into Next.js using Smartive Company Example

#### Background

Swisscom aimed to integrate Stencil web components with Next.js, including Server-Side Rendering (SSR). However, at the outset of this experiment, it was known that Next.js 13 / Next.js 14 is not compatible with the latest version of StencilJS. This is confirmed by developer from Swisscom, as he received an insight from an external agency called Smartive.

The agency's tool converts the shadow DOM to declarative, with the original source available here. This will be referred to as the `smartive` repository.

#### Purpose

The primary goal of this experiment was to analyze the Smartive example of integrating StencilJS and Next.js with SSR. Specifically, the focus is on assessing the possibility of integrating Swisscom's components into the provided Smartive example. Specifically, we aimed to assess the feasibility of integrating Swisscom's components into the Smartive example and evaluate whether Swisscom could utilize the agency's tool or develop a custom tool.

#### Method

The `smartive` repository was downloaded and initialized locally.

The structure of the `smartive` repository:

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

Our first step involved adding tabs components from the `mini-sdx` repository to the `smartive` components. To align with Smartive's naming conventions, the `sdx` prefix was replaced with `abc`. We also copied over all styles and utilities. Subsequently, a component with React's `AbcWrapper` was added to the Next.js application.

Listing 6.15: packages/app/src/components/tabs.tsx

```
1 | ...
2 | import { AbcTabs, AbcTabsItem } from "abc-web-components-react-
   | wrapper";
```

```

3 import { AbcWrapper } from "abc-web-components-react-wrapper/
  client";
4 import { FC } from 'react';
5
6 export const Tabs: FC = () => (
7   <AbcWrapper>
8     <AbcTabs sr-hint="Tabs with text.">
9       <AbcTabsItem id="tab1" label="Tab 1">
10        This is the content of Tab 11111.
11      </AbcTabsItem>
12      <AbcTabsItem id="tab2" label="Tab 2">
13        This is the content of Tab 2.
14      </AbcTabsItem>
15      <AbcTabsItem id="tab3" label="Tab 3 with very long text
16        that will be truncated" selected>
17        This is the content of Tab 3.
18      </AbcTabsItem>
19      <AbcTabsItem id="tab4" label="Tab 4">
20        This is the content of Tab 4.
21      </AbcTabsItem>
22      <AbcTabsItem id="tab5" label="Tab 5 with extra text">
23        This is the content of Tab 5.
24      </AbcTabsItem>
25      <AbcTabsItem id="tab6" label="Tab 6" disabled>
26        This is the content of the disabled Tab 6.
27      </AbcTabsItem>
28    </AbcTabs>
29  </AbcWrapper>
30 );

```

Following the new Tabs component with React's AbcWrapper was added to the page in order to display it in frontend. 9.18

Listing 6.16: packages/app/src/app/page.tsx

```

1 import { Accordion } from '@components/accordion';
2 import { Button } from '@components/button';
3 import { Dropdown } from '@components/dropdown';
4 import { Tabs } from '@components/tabs';
5
6 const Page = () => (
7   <main style={...}>
8     <Button />
9     <Dropdown />
10    <Accordion />
11    <Tabs />
12  </main>
13 );
14
15 export default Page;

```

## Result

Upon integration, we encountered a `TypeError` 6.17 that initially appeared to be linked to the Redux store utilized by the tabs components. A deeper analysis suggested that the error might stem from a conflict between custom-defined functions in `utis/webcomponents-helpers.ts` and the React's `AbcWrapper`, which now served as the parent of the tabs component.

Specifically, the functions `parent()` and `closest()` warrant more focused debugging efforts, as they resulted in `'currentEl'` becoming undefined during their execution. 6.18

Due to time constraints, a comprehensive analysis of this issue was not feasible, and as a result, the experiment had to be terminated prematurely.

Listing 6.17: `TypeError`

```
1 | ../web-components-react-wrapper/dist/esm/components/tabs-store.  
  | js 378:29 @storeKey  
2 | at new StoreConnection (../web-components-react-wrapper/dist/  
  | esm/components/tabs-store.js:331:35)  
3 | at new Tabs (../web-components-react-wrapper/dist/esm/  
  | components/abc-tabs.js:92:22)  
4 | ...  
5 | 378 |         this.store = currentEl[storeKey];  
6 | ...  
7 | Compiled /favicon.ico in 2.3s (496 modules)
```

Listing 6.18: `tabs/tabs-store.ts`

```
1 | ..  
2 | currentEl = closest(  
3 |   // Check if the parent was found (tagName match) or if a  
  |   store exists  
4 |   parent(currentEl)!,  
5 |   (el: IndexableElement) => el.matches(parentTagName) || el[  
  |     parentKey] === parentTagName) as IndexableElement  
6 | ...
```

## Conclusion

This experiment highlighted the complexities of integrating Stencil web components with Next.js. The encountered issues underline the challenges in ensuring compatibility between different frameworks, particularly concerning SSR. The findings emphasize the need for careful consideration of the underlying architecture and dependencies when attempting such integrations. Our findings point to the necessity of a deep understanding of the underlying architectures and dependencies for successful integration.

A crucial takeaway from this exploration is the potential need to develop a custom tool tailored to Swisscom's specific requirements. Given the unique challenges presented by the existing tools and frameworks, creating a bespoke solution could offer a more streamlined and efficient path to integration. This approach would allow for greater

control over the integration process and the ability to tailor the solution to the specific nuances of Swisscom's components and the Next.js environment.

The documentation of these initial findings lays a foundation for future explorations and potential solutions to bridge the compatibility gap between StencilJS and Next.js.

### 6.3.3 Tabs Integration into Next.js using Mayerraphael Example

#### Background

Following our previous attempts at integrating tabs into Next.js, we came across a recent discussion on the Ionic Stencil GitHub thread focusing on "Declarative Shadow DOM with Hydrate".

An intriguing example, noted in the comments, led us to the `nextjs-webcomponent-hydration` repository, which proposes a proof of concept for SSR and web components collaboration.

Owing to time constraints, we conducted a brief review rather than a deep dive.

#### Purpose

The experiment aimed to explore the functionality of this different repository, referred as `nextjs-webcomponent-hydration`. Specifically its approach to integrating Stencil components with Next.js.

#### Method

We downloaded and initialized the `nextjs-webcomponent-hydration` repository locally. Similar to our previous experiment, we integrated tabs components from the `mini-sdx` repository. While the steps were largely analogous to the previous Next.js integration, this experiment differed in the way the component was wrapped with React's `AbcWrapper`. The detailed wrapping method is documented in `/components/StencilWrapper.tex`. The wrapped tabs component was then imported into `/pages/index.tsx` to be displayed on the frontend.

#### Result

In this experiment, we once again encountered a `TypeError 6.19`, similar to the previous attempt but with a notable difference in its occurrence.

Listing 6.19: `TypeError`

```
1 | TypeError: Cannot read properties of undefined (reading '
  |   tagName')
2 | ...
3 | const parentTagName = getParentTagName(cmp.el.tagName.
  |   toLowerCase());
4 | |
```

```
5 | 429 | // Looking for existing store
6 | 430 | let currentEl = cmp.el;
7 | // previously TypeError occurred here
8 | 431 | this.store = currentEl[storeKey];
```

Interestingly, this error manifested a few lines above the point where it occurred in our previous experiment. This variation indicates a persistent, yet slightly different, compatibility issue between the Swisscom Tabs components and the Next.js SSR.

## Conclusion

Crucially, the two functions identified as potentially problematic in the previous experiment (`parent()` and `closest()`) remain under suspicion. In this case, the error seems to stem from an undefined `cmp.el` (which is assigned to `currentEl`). In the previous experiment, `currentEl` became `undefined`, potentially due to these functions. This pattern reinforces the hypothesis that these specific functions might be contributing to the `TypeError`, possibly due to how they interact with the component elements within the Next.js environment.

The author of the `nextjs-webcomponent-hydration` example highlighted a significant caveat: "The solution here is far from perfect and not what I would recommend doing currently. Converting nodes to other formats is CPU and GC heavy due to the many objects created. I rather recommend creating native components for the framework you use (React in the case of Next.js) and create wrappers around them to render the DSD."

## 6.4 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 6.6.

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



---

Figure 6.6: Test Website

Each scenario was tested 10 times, with SSR and with no SSR. The results displayed here are average values.

As described in Section 4.2.3 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

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 6.1: Results Test 1, average values

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. 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 6.7 show a side-by-side comparison of the load times, with SSR and without SSR:

### 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 6.8 show a comparison of the load times when loading the minimal script of Test 1 compared to the full script.

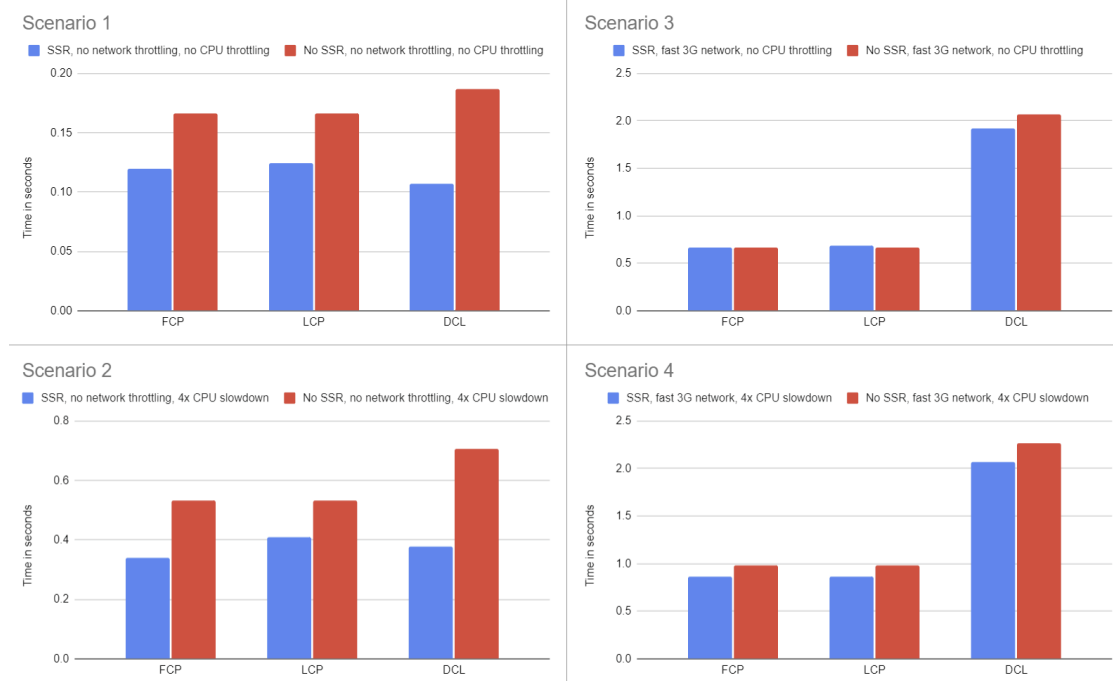


Figure 6.7: Performance comparison: SSR vs. No SSR, Test 1

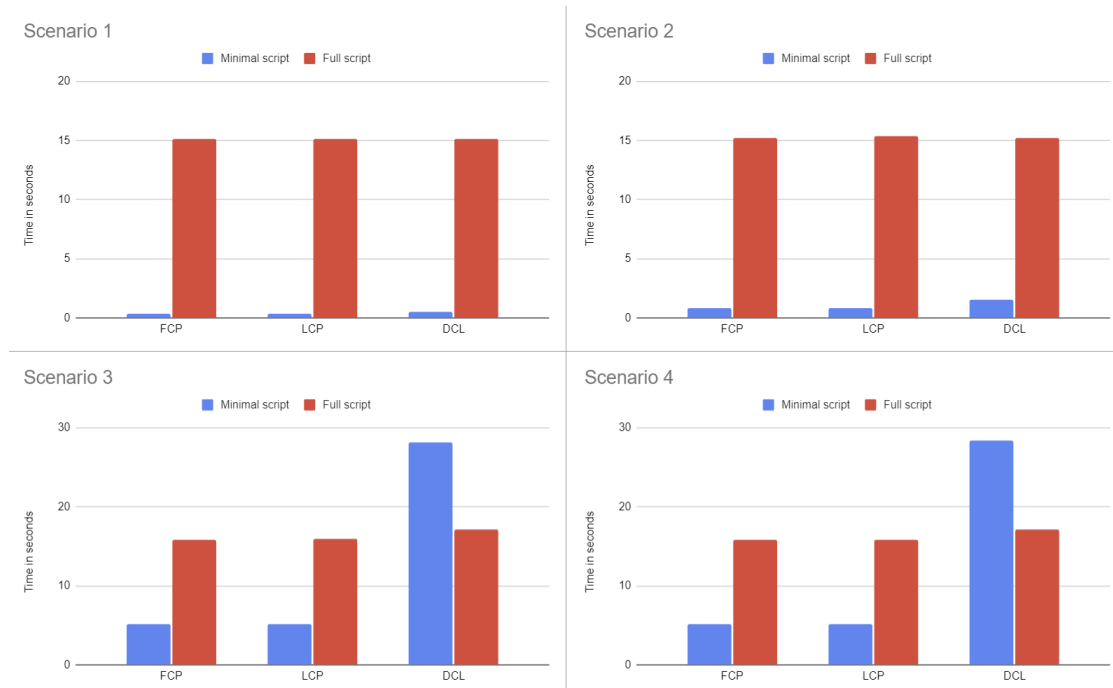


Figure 6.8: Performance comparison: Minimal script vs. Full script, Test 2

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 6.2: Results Test 2, average values

### Results Test 3

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 6.3: Results Test 3, average values

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 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 6.9 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

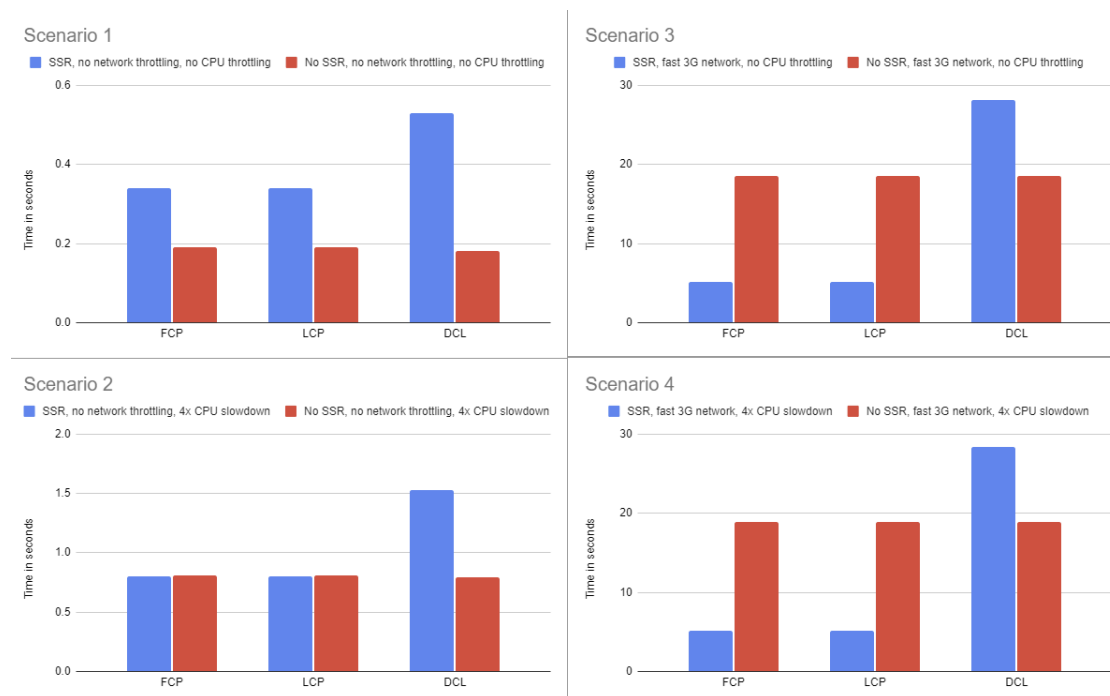


Figure 6.9: Performance comparison: SSR vs. No SSR, Test 3

without the actual content. For the mini-sdx example, this looks the following Figure 6.10.

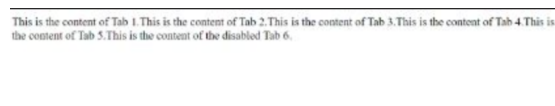


Figure 6.10: Minimal HTML page

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 6.11: 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.



Figure 6.12: 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 6.11.

## 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.

## Chapter 7

# Quality Measures

In order to maintain the high quality of our product and avoid mistakes, the following quality measures were defined.

### 7.1 Issue Tracking

Jira was selected for issue tracking. How Jira is used in combination with Scrum is described in Section ???. Issues are either assigned to a specific team member or left unassigned. If an issue is unassigned, both team members collaborate on resolving it.

### 7.2 Development Environment

GitLab is used for version control, while Jira handles issue tracking.

### 7.3 Merge Requests

A feature or change is always created on a separate branch and then merged into the main branch. A merge request always follows the four-eye principle; hence, the other team member will merge the branch into the main branch.

### 7.4 Time Tracking

It was decided that the time estimates were not required, as the focus is mainly on the research aspect of the project. Time tracking is performed directly within this documentation to avoid doubling the work. It can be found here ??.

### 7.5 Verification and Testing

As we approach the project's conclusion, it's crucial to verify and test the Functional Requirements (FRs) and Non-Functional Requirements (NFRs) outlined in Chapter 2.

This process ensures that the solutions developed meet the specified criteria and are effective in addressing the project's goals.

### **7.5.1 Testing Methodologies**

Given the nature of this project as a research endeavor, with a focus on exploring methodologies rather than developing extensive source code, our testing approaches are tailored to this context. We will not be performing extensive code-based testings, such as automated or system tests. Instead, our testing methodologies are confined to the following two approaches:

#### **Manual Testing**

This will be the primary methodology used in our project. Manual testing is especially crucial due to the project's research-oriented nature, where much of the work involves theoretical analysis, documentation, and exploration of concepts.

Manual testing includes reviewing documentation, assessing the implementation strategies discussed, and ensuring that theoretical concepts are accurately represented.

#### **Peer Review**

Peer review serves as an integral part of our testing process. In this approach, team members critically evaluate each other's work. This method is particularly effective in research projects like ours, where insights, analyses, and theoretical frameworks are central to the project's success.

The peer review process will focus on evaluating the accuracy, thoroughness, and clarity of the research and documentation. Team members will provide constructive feedback to ensure that all material produced meets high-quality standards and accurately reflects the project's objectives.

**Part IV**  
**Results**



# Chapter 8

## Results

### 8.1 Verification of Functional Requirements

<b>Requirement</b>	FR1: Documentation - SSR
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko, Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	The documentation covers SSR in detail, outlining the theory of SSR and its implementation in StencilJS, Angular and Next.js. All implementations have an example repository to facilitate understanding. Positive and negative aspects of SSR are covered. The language is technical, however, it is easy to understand for anyone with experience in software development.
<b>Conclusion</b>	Based on the tests findings, the documentation is sufficiently complete and understandable.

Table 8.1: Verification FR1: Documentation - SSR

<b>Requirement</b>	FR2: Root Cause Analysis
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko, Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	The initial search for the root cause was documented in 5 in detail. The root cause was found and was documented in 5. Verification that the root cause was solved was documented in 6.2.3.
<b>Conclusion</b>	Based on the test findings, a thorough analysis of the root cause was carried out. The root cause was found and the result were verified.

Table 8.2: Verification FR2: Root Cause Analysis

<b>Requirement</b>	FR3: Issue Resolution
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko, Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	In 6.2.2 it was tested if it is possible to include the whole SDX-library with the current fix. This was not the case. The issues are documented in 6.2.4. Workaround for the problems were proposed in 6.2.7 and 6.2.8. Implementations in Angular (6.3.1) and Next.js (6.3.2, 6.3.3) were tested.
<b>Conclusion</b>	Based on the test findings, it is possible to solve some of the compatibility issues with StencilJS. However, not all problems have been solved. Some issues persist. These issues are documented in 6.2.4.

Table 8.3: Verification FR3: Issue Resolution

<b>Requirement</b>	FR4: Extended Issue Resolution
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko, Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	There is no extended complexity when extending the problem from 8.3 to a whole application. The same points made there also hold for this functional requirement.
<b>Conclusion</b>	Based on the test findings, the same problems as in 8.3 persist. By solving FR3, this is also solved.

Table 8.4: Verification FR4: Extended Issue Resolution

<b>Requirement</b>	FR5: Analysis of Alternatives
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko, Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	Alternative frameworks, namely Angular and Next.js, and their advantages and disadvantages were looked at and documented in 4.3 and 4.4. Alternatives to StencilJS were not looked at.
<b>Conclusion</b>	Based on the test findings, StencilJS can be successfully implemented in multiple frameworks. As a next step, alternatives to StencilJS should be looked at.

Table 8.5: Verification FR5: Analysis of Alternatives

<b>Requirement</b>	FR6: Documentation - StencilJS
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko, Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	A developer guide was developed (V). It includes a general setup on how to achieve compability with Express and Angular. For Next.js the issues hindering compability were documented. StencilJS was documented (4.2).
<b>Conclusion</b>	Based on the test findings, StencilJS is well documented, and the developer guide provides good instructions on how to navigate the SSR-landscape.

Table 8.6: Verification FR6: Documentation - StencilJS

<b>Requirement</b>	FR7: Visual user experience
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko, Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	The test showed that SSR-rendered components were displayed, however, some minor bugs were still present.
<b>Conclusion</b>	Further investigation inside the visual bugs must be made. As already mentioned in the table of the verification of FR3 (8.3).

Table 8.7: Verification FR7: Visual user experience

## 8.2 Verification of Non-Functional Requirements

<b>Requirement</b>	NFR1: Performance
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko, Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	Performance tests (9.6) showed that under good conditions the 2 second mark is never exceeded. When loading larger files, the mark is higher. In general, the performance of SSR was better than with CSR especially when limiting the network speed.
<b>Conclusion</b>	Based on the test findings, the performance with SSR was reasonably good. When having bigger applications, it is important to execute tree shaking, in order to reduce load times.

Table 8.8: Verification NFR1: Performance

<b>Requirement</b>	NFR2: Compatibility
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko, Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	<ul style="list-style-type: none"> <li>• Successfully tested compatibility with the latest versions of StencilJS and its SSR features.</li> <li>• Verified full compatibility with major browsers including Chrome, Firefox, Safari, and Edge.</li> <li>• Confirmed compatibility with common operating systems: Windows, macOS, and Linux.</li> </ul>
<b>Conclusion</b>	All tests were successful. The web components are fully compatible with the latest versions of StencilJS, major browsers, and commonly used operating systems. No compatibility issues were identified.

Table 8.9: Verification NFR2: Compatibility

<b>Requirement</b>	NFR3: Security
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko, Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	<ul style="list-style-type: none"> <li>• Only used well known npm-packages</li> <li>• Kept packages up-to-date to avoid security risks</li> </ul>
<b>Conclusion</b>	As this was a research project, not much coding was done. The code that was written used only well known and reliable sources and were always kept up-to-date thus the security risk was minimized.

Table 8.10: Verification NFR3: Security

<b>Requirement</b>	NFR4: Testing
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko, Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	<ul style="list-style-type: none"> <li>• Successfully tested on Windows, stable performance.</li> <li>• Successfully tested on MacOS, stable performance.</li> <li>• Successfully tested on Linux, stable performance.</li> <li>• Successfully tested on multiple machines.</li> </ul>
<b>Conclusion</b>	There were no performance differences seen on different OSs.

Table 8.11: Verification NFR4: Testing

<b>Requirement</b>	NFR5: Documentation Quality
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko, Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	<ul style="list-style-type: none"> <li>• The documentation is well-structured, with clearly defined sections.</li> <li>• The documentation is easy to navigate with an easy to understand table of contents and many references, linking to relevant sections.</li> <li>• The documentation was thoroughly checked for grammatical errors.</li> <li>• The documentation provides a developer guide for a short overview for developers.</li> </ul>
<b>Conclusion</b>	The documentation meets high standards and is easy to use.

Table 8.12: Verification NFR5: Documentation Quality

<b>Requirement</b>	NFR6: Compliance
<b>Execution Date</b>	Milestone 5
<b>Executed By</b>	Natalia Gerasimenko and Tim Gamma
<b>Status</b>	Executed
<b>Assessment</b>	<ul style="list-style-type: none"> <li>• Discussed compliance with Swisscom.</li> <li>• It was agreed that all research materials and repositories should be made publicly available.</li> <li>• The project management documentation will not be publicly available.</li> </ul>
<b>Conclusion</b>	Compliance is ensured by making all documentation (without the project management documentation) public, no additional compliance measures required.

Table 8.13: Verification of NFR6: Compliance

## 8.3 Conclusion

### Findings

**Interactivity** was restored by adding the ESM scripts post rendering. The issue was that only static HTML was delivered to the client. Sending the script ensures that the components will be hydrated client-side and thus gain interactivity.

**CSS-styling** was addressed by a CSS-grid based workaround. The order of the elements can get messed up, resulting in a wrongly styled component during loading. By adding a CSS-grid, it can be secured that the user always sees the correct component.

**Angular integration** was achieved with Stencil's Angular output target, specific wrappers that allow Stencil web components to behave like native Angular components.

**Next.js integration** proved to be difficult. In order to use Stencil's web components in Next.js, custom react wrappers are necessary for an effective incorporation. These encapsulate Stencil components within React components, thereby enabling seamless integration.

**Performance** - vis SSR generally demonstrated a faster response compared to client-side rendering, particularly in the first contentful paint. However, this performance advantage was not consistent across all scenarios.

### Developer Documentation

Based on the findings, a developer documentation was developed. It includes:

- How to integrate Stencil web components into an Express.js setup
- How to integrate Stencil web components into an Angular application
- How to address the incorrect CSS styling (wrong order) with CSS-grid
- Possible fixes on a Stencil feature branch
- Findings on how it would be possible to use Stencil web components in Next.js (and why it is not simple)
- Performance analysis of SSR

It is aimed for developers to gain an overview of SSR with StencilJS web components and have some starting repositories to try them out.

### Future

The outcome of this project showed that SSR is possible with StencilJS web components. However, there are still several bugs present, requiring custom solutions and workarounds. While the integration of StencilJS web components within Angular and Next.js is achievable, it demands significant setup and configuration, especially for Next.js. Nevertheless, as framework compatibility improves, the efficiency and user

experience offered by SSR with StencilJS web components are expected to become more pronounced, highlighting the promising future of this technology in web development.



**Part V**

**Developer Documentation**

# Chapter 9

## Developer Documentation

### 9.1 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:

- Builted 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 9.1: app.mjs

```
1 | 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 9.2: app.mjs

```

1 ...
2 const results = await renderToString(tabs, {
3   prettyHtml: true,
4   removeScripts: false
5 });
6
7 const withStencilScripts = results.html.replace(
8   '</head>',
9   <script type="module" src="/scripts/stencil-starter-project-
   name.esm.js"></script>
10  </head>`
11 );
12
13 res.send(withStencilScripts);
14 ...

```

### 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 9.3: app.mjs

```

1 // Render page using renderFile method
2 ejs.renderFile('index.ejs', {},
3   {}, async function (err, template) {
4     if (err) {
5       throw err;
6     } else {
7       const results = await renderToString(template, {
8         prettyHtml: true,
9         removeScripts: false
10      });
11
12      res.end(results.html);
13    }
14  });

```

Listing 9.4: index.ejs

```

1 <head>
2   ...
3   <script type="module" src="/path/to/stencil-esm-script.js"
4     ></script>
5   ...
6 </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.

## 9.2 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 9.5: renderToString result

```
1 |
2 | <div ... class="sc-sdx-tabs tablist">
3 |   ...
4 |   <button ... >
5 |     <!--t.1.5.4.0-->
6 |     Tab 6
7 |   </button>
8 |   <button ... >
9 |     <!--t.1.7.4.0-->
10 |    Tab 4
11 |   </button>
12 |   <button ... >
13 |     <!--t.1.9.4.0-->
14 |     Tab 2
15 |   </button>
16 |   ...
17 | </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 9.6: index.html

```
1 <sdx-tabs sr-hint="Tabs with text.">
2   <sdx-tabs-item id="tab1" label="Tab 1">This is the
   content of Tab 11111.</sdx-tabs-item>
3   <sdx-tabs-item id="tab2" label="Tab 2">This is the
   content of Tab 2.</sdx-tabs-item>
4   <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>
5   ...
6 </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 9.7: tabs.tsx

```
1 private getTabIds(tabsItemEls: HTMLSdxTabsItemElement[]) {
2 // return tabsItemEls.map(element => element.id); // WONT work
3
4 let gridAreas = [];
5 for (let i = 1; i <= tabsItemEls.length; i++) {
6   gridAreas.push('tab' + i)
7 }
```

```

8     return gridAreas;
9 }
10
11 public render() {
12
13     const {
14         tabsItemEls,
15         selectedTabsItemEl
16     } = this.state.get()
17
18     ...
19     const elementIdsForGrid = this.getTabIds(tabsItemEls)
20     ...
21 }

```

### 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 9.8: tabs.tsx

```

1     return (
2         ...
3         <div
4             class="tablist"
5             role="tablist"
6             aria-label={this.srHint}
7             ref={(el) => (this.tablistEl = el)}
8             style={{
9                 display: `grid`,
10                gridTemplateAreas: `>${elementIdsForGrid.join(' ')}<
11
12                }}
13            >
14                ...
15        )

```

### 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 9.9: tabs.tsx

```

1 const Tag = tabsItemEl.href ? "a" : "button"
2
3 return (

```

```

4     <Tag
5         class={this.getClickableElClassNames(tabsItemEl)}
6         style={{gridArea: tabsItemEl.id}}
7         ...
8
9     >
10        {tabsItemEl.label}
11    </Tag>
12 )

```

## 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 9.10: `renderToString` result

```

1
2 <div ... class="sc-sdx-tabs tablist" style="display: grid;
   grid-template-areas: 'tab1 tab2 tab3 tab4 tab5 tab6';">
3     ...
4     <button ... style="grid-area: tab6;">
5         <!--t.1.5.4.0-->
6         Tab 6
7     </button>
8     <button ... style="grid-area: tab4;">
9         <!--t.1.7.4.0-->
10    Tab 4
11    </button>
12    <button ... style="grid-area: tab2;">
13        <!--t.1.9.4.0-->
14        Tab 2
15    </button>
16
17 </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.

## 9.3 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. 9.1

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.a311f36` 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.

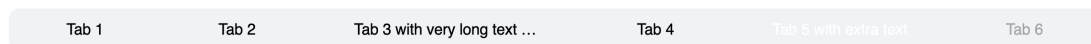


Figure 9.1: Initially selected tab not selected



This is the content of Tab 3.

Figure 9.2: 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.

## 9.4 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 Stencil's 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

### 9.4.1 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 9.11: 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 |
8 | export const config: Config = {
9 |   namespace: 'my-stencil-webcomponents',
10 |
11 |   outputTargets: [
12 |     // other output targets
13 |     {
14 |       type: 'dist-hydrate-script',
15 |     },
16 |     angularOutputTarget({
17 |       componentCorePackage: '@web-components/dist/components',
18 |       directivesProxyFile: '../stencil-angularapp-demo/src/libs/stencil-generated/proxies.ts',
19 |       valueAccessorConfigs: angularValueAccessorBindings,
20 |     })
21 |   ],
22 |   plugins: [sass()],
23 | };
```

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 9.12: Relative path to the dist directory of web-components

```
1 | "paths": {
2 |   "@web-components/*": ["../web-components/*"]
3 | }
```

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 9.13: web-components.module.ts

```
1 | import { NgModule, Inject, PLATFORM_ID } from '@angular/
   | core';
2 | import { CommonModule } from '@angular/common';
3 | import { MyWebcomponent } from "../../src/libs/stencil-
   | generated/proxies";
4 | import { isPlatformBrowser } from '@angular/common';
5 |
6 | @NgModule({
7 |   declarations: [MyWebcomponent],
8 |   imports: [CommonModule],
9 |   exports: [MyWebcomponent]
10 | })
11 | export class WebComponentsModule {
```

```

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

```

1     import {provideClientHydration} from '@angular/platform-
2         browser';
3     // ...
4     @NgModule({
5         // ...
6         providers: [ provideClientHydration() ], // add this
7             line
8         bootstrap: [ AppComponent ]
9     })
10    export class AppModule {
11        // ...

```

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 9.15: `server.ts` (old)

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

New:

Listing 9.16: `server.ts` (new)

```
1 server.get('*', (req, res) => {
2   res.render(indexHtml, { req }, (err, html) => {
3     renderToString(html).then(({html}) => {
4       res.send(html);
5     });
6   });
7 });
```

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.

## 9.5 Integration into Next.js

### 9.5.1 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 should be copied from the `mini-sdx` repository to the `smartive` components. 9.17

To align with Smartive's naming conventions, the `sdx` prefix was replaced with `abc`.

Listing 9.17: `packages/app/src/components/tabs.tsx`

```
1 ...
2 import { AbcTabs, AbcTabsItem } from "abc-web-components-react-
  wrapper";
3 import { AbcWrapper } from "abc-web-components-react-wrapper/
  client";
4 import { FC } from 'react';
5
6 export const Tabs: FC = () => (
7   <AbcWrapper>
8     <AbcTabs sr-hint="Tabs with text.">
9       <AbcTabsItem id="tab1" label="Tab 1">
10        This is the content of Tab 11111.
11      </AbcTabsItem>
12      <AbcTabsItem id="tab2" label="Tab 2">
13        This is the content of Tab 2.
14      </AbcTabsItem>
15      <AbcTabsItem id="tab3" label="Tab 3 with very long text
16        that will be truncated" selected>
17        This is the content of Tab 3.
18      </AbcTabsItem>
19      <AbcTabsItem id="tab4" label="Tab 4">
20        This is the content of Tab 4.
```

```

20     </AbcTabsItem>
21     <AbcTabsItem id="tab5" label="Tab 5 with extra text">
22         This is the content of Tab 5.
23     </AbcTabsItem>
24     <AbcTabsItem id="tab6" label="Tab 6" disabled>
25         This is the content of the disabled Tab 6.
26     </AbcTabsItem>
27 </AbcTabs>
28 </AbcWrapper>
29 );

```

## 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. 9.18

Listing 9.18: packages/app/src/app/page.tsx

```

1 import { Accordion } from '@components/accordion';
2 import { Button } from '@components/button';
3 import { Dropdown } from '@components/dropdown';
4 import { Tabs } from '@components/tabs';
5
6 const Page = () => (
7   <main style={...}>
8     <Button />
9     <Dropdown />
10    <Accordion />
11    <Tabs />
12  </main>
13 );
14
15 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.

### 9.5.2 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.

## 9.6 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 9.3.



Figure 9.3: 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 4.2.3 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 9.1: 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 9.4 show a side-by-side comparison of the load times, with SSR and without SSR:

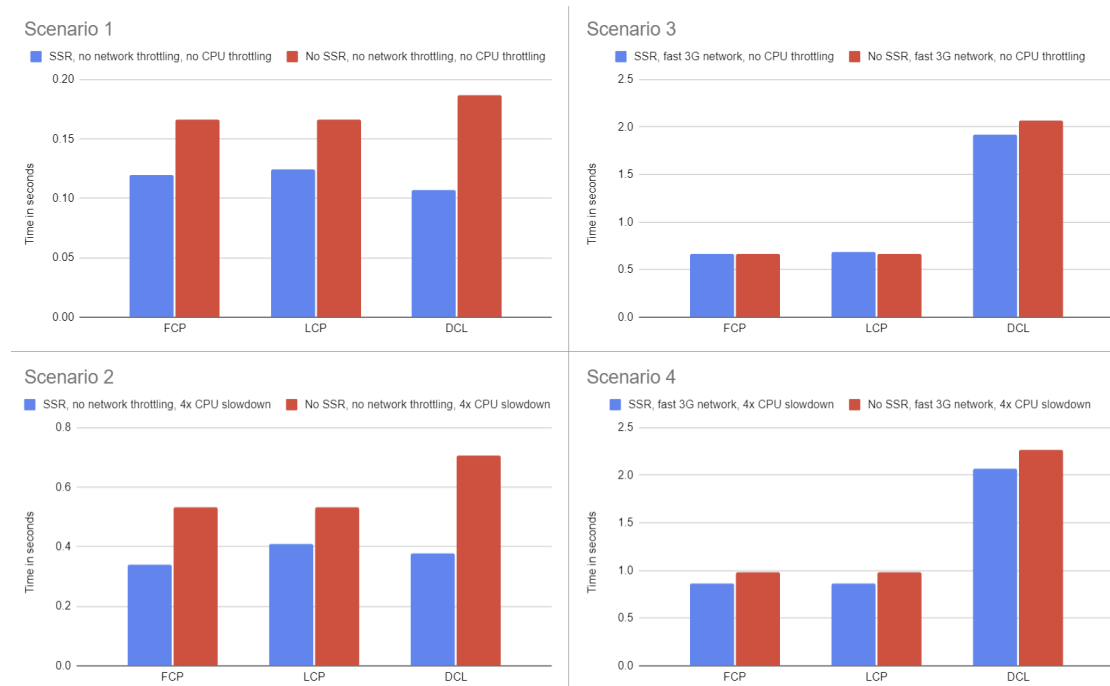


Figure 9.4: Performance comparison: SSR vs. No SSR, Test 1

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 9.2: 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 9.5 show a comparison of the load times when loading the minimal script of Test 1 compared to the full script.

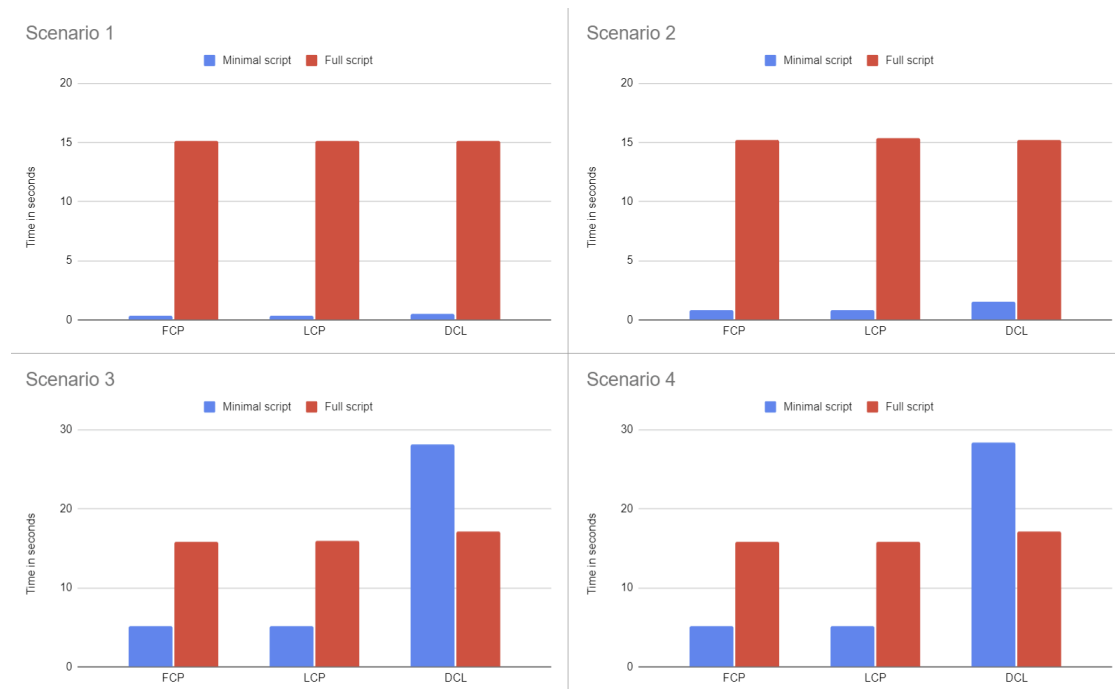


Figure 9.5: Performance comparison: Minimal script vs. Full script, Test 2

## 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 9.3: 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 9.6 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 9.7.

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:

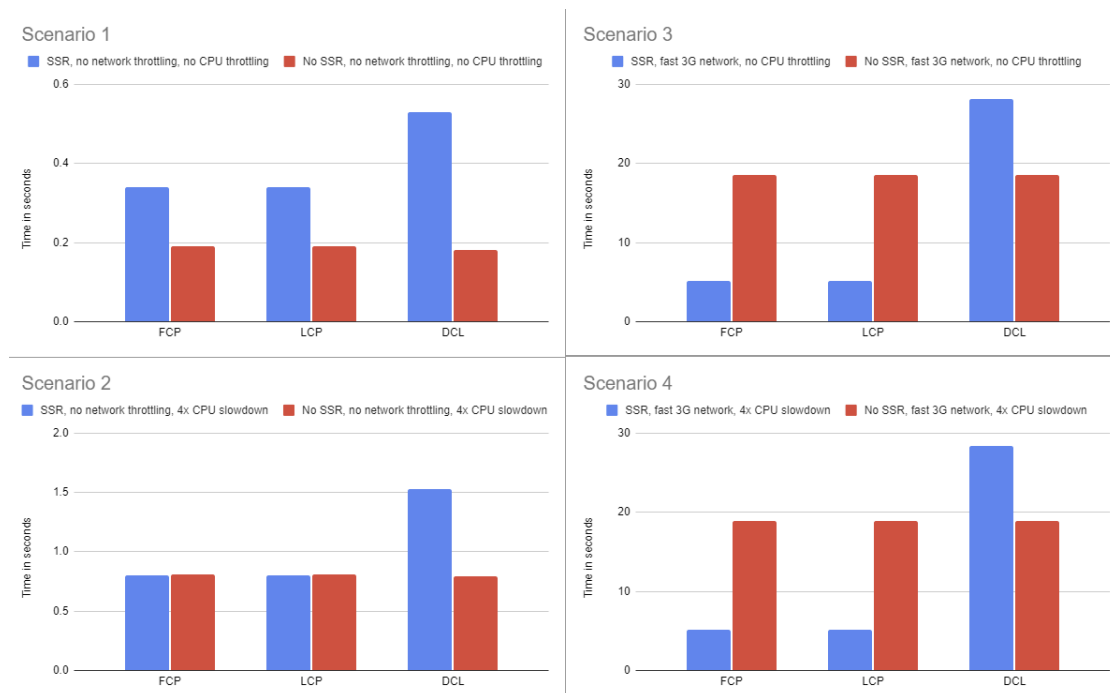


Figure 9.6: Performance comparison: SSR vs. No SSR, Test 3

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 9.7: Minimal HTML page

Tab 1      Tab 2      Tab 3 with very long list

This is the content of Tab 3.

Figure 9.8: 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.



Figure 9.9: 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 9.8.

## 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.

## 9.7 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 VI**  
**Appendix**

# Bibliography

- [Ang] Angular. Server-side rendering. <https://angular.io/guide/ssr>, (accessed on: 20.12.2023).
- [Bam23] Aristeidis Bampakos. *Angular Projects Build modern web apps in Angular 16 with 10 different projects and cutting-edge technologies*. Packt Publishing Ltd, 2023.
- [ima] Pre-rendering and data fetching. <https://nextjs.org/learn-pages-router/basics/data-fetching/two-forms>, (accessed on: 15.12.2023).
- [LP23] A. Lipiński and B. Pańczyk. Performance optimization of web applications using qwik. *Journal of Computer Sciences Institute*, 28:197–203, Sept 2023.
- [Mit22] Eishta Mittal. Different rendering modes, 2022. <https://eishta.medium.com/system-design-csr-vs-ssr-vs-ssg-8e26dbb20b1d>, (accessed on: 15.12.2023).
- [Shi20] Shailesh Kumar Shivakumar. *Modern Web Performance Optimization*. Apress, 2020.
- [Stea] Stencil. Angular integration. <https://stenciljs.com/docs/angular>, (accessed on: 20.12.2023).
- [Steb] Stencil. Combining server side rendering and static site generation. <https://stenciljs.com/docs/static-site-generation-server-side-rendering-ssr>, (accessed on: 20.12.2023).
- [Stec] Stencil. Component lifecycle methods. <https://stenciljs.com/docs/component-lifecycle>, accessed on: 20.12.2023.
- [Sted] Stencil. Getting started. <https://stenciljs.com/docs/getting-started>, (accessed on: 20.12.2023).
- [Stee] Stencil. Hydrate app. <https://stenciljs.com/docs/hydrate-app>, (accessed on: 20.12.2023).
- [Stef] Stencil. Overview stencil: A web components compiler. <https://stenciljs.com/docs/introduction>, (accessed on: 20.12.2023).

[Steg] Stencil. Styling components. <https://stenciljs.com/docs/styling>, (accessed on: 20.12.2023).

[Sun] Yan Sun. Server-side rendering in angular 16. <https://blog.logrocket.com/server-side-rendering-angular-16/>, (accessed on: 20.12.2023).

# List of Figures

1.1	Example of Swisscom’s web components from Swisscom SDX Documentation . . . . .	3
2.1	Example of tabs component from Swisscom SDX Documentation . . . . .	8
2.2	Correct Tabs Component . . . . .	8
2.3	Incorrect Tabs Component . . . . .	8
2.4	Experiments Conducted . . . . .	9
4.1	CSR vs SSR vs SSG [Mit22] . . . . .	20
4.2	SSR in Next.js [ima] . . . . .	29
4.3	Pre-rendering using Next.js [ima] . . . . .	30
4.4	No Pre-rendering, Plain React.js app [ima] . . . . .	31
5.1	Tabs Component . . . . .	33
5.2	Tabs Component Rendered Falsely . . . . .	34
5.3	Nested Components Output . . . . .	36
5.4	Nested Components DOM . . . . .	37
5.5	Styles applied SSR . . . . .	38
6.1	Result of mixing scripts . . . . .	44
6.2	Initially selected tab not selected . . . . .	47
6.3	Initially selected tab selected . . . . .	47
6.4	Incorrect order on Initial Page Load . . . . .	47
6.5	Component Lifecycle Methods (from [Stec]) . . . . .	52
6.6	Test Website . . . . .	67
6.7	Performance comparison: SSR vs. No SSR, Test 1 . . . . .	69
6.8	Performance comparison: Minimal script vs. Full script, Test 2 . . . . .	69
6.9	Performance comparison: SSR vs. No SSR, Test 3 . . . . .	71
6.10	Minimal HTML page . . . . .	71
6.11	Fully loaded page . . . . .	72
6.12	Rendered page from server . . . . .	72
9.1	Initially selected tab not selected . . . . .	91
9.2	Initially selected tab selected . . . . .	91
9.3	Test Website . . . . .	101

9.4	Performance comparison: SSR vs. No SSR, Test 1 . . . . .	103
9.5	Performance comparison: Minimal script vs. Full script, Test 2 . . . . .	104
9.6	Performance comparison: SSR vs. No SSR, Test 3 . . . . .	106
9.7	Minimal HTML page . . . . .	106
9.8	Fully loaded page . . . . .	106
9.9	Rendered page from server . . . . .	107