IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 4 December 2023, accepted 6 January 2024, date of publication 11 January 2024, date of current version 22 January 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3352891

PERSPECTIVE

Resumability–A New Primitive for Developing Web Applications

JUHO VEPSÄLÄINEN [1], MIŠKO HEVERY [2], AND PETRI VUORIMAA [1]
[1]Department of Computer Science, School of Science, Aalto University, 02150 Espoo, Finland
[2]Builder.io, San Francisco, CA 94103, USA
Corresponding author: Juho Vepsäläinen (juho.vepsalainen@aalto.fi)

ABSTRACT World Wide Web was originally meant as a global information exchange but it has since then morphed into the largest available application platform. Especially during the past decade, mobile usage has been rising while the size of websites and applications has been steadily rising therefore making size an important target for optimization. In this article, we look into a new primitive called resumability. Resumability allows developers to avoid caveats of earlier approaches, such as hydration, by embedding some of the required data straight into HTML markup delivered to the client. Then the client resumes execution as an application becomes interactive. The technique allows frameworks to apply well-known techniques, such as code-splitting, automatically therefore reducing developer effort. By considering past developments and a couple of concrete examples, we propose resumability as a new primitive for web application development. Furthermore, we also discuss potential research directions for those wanting to understand the topic in greater detail.

INDEX TERMS Hydration, JavaScript, multi-page applications, page size, resumability, single page applications, software architecture, web application development, web performance, world wide web.

I. INTRODUCTION

Introduced in 1992, the World Wide Web was meant as a global information exchange [1]. Since then, the web has morphed into the largest application platform available, reaching roughly two-thirds of the global population [2]. At the same time, the size of websites and the share of mobile usage are constantly rising, as reported by [3] and [4]. It is telling that in [3] data, the median page weight on desktop grew from 669 kilobytes in March 2012 to 2060 kilobytes in March 2022. The change has been more drastic on mobile, and a nearly 600 % increase in page size has been reported [3]. JavaScript contributes roughly one-third of the size, making it the second biggest contributor after images and a clear optimization target [3].

A. COST OF JAVASCRIPT

The cost of JavaScript is two-fold. In addition to having a cost in terms of kilobytes to transfer, there is also a significant cost related to parsing and evaluating the code.[1] Especially in a mobile environment, the cost is felt in terms of reduced battery life due to increased processing. Due to lower computational capacities, application performance experienced by the user may be reduced. Due to the cost, any reduction to the size of JavaScript has a considerable impact on the end user, and even deferring the cost can be beneficial.

B. SINGLE PAGE APPLICATIONS

New solutions were needed as the web changed from a content platform into an application platform over time. The current mainstream option, Single Page Applications (SPAs), was motivated by the need to develop highly interactive solutions on top of the web [6] and SPAs came with many clear benefits for both developers and users compared to earlier models, such as Multi-Page Applications (MPAs). SPAs also came with new challenges related to Search Engine Optimization (SEO), reliance on JavaScript, and high cost

The associate editor coordinating the review of this manuscript and approving it for publication was Derek Abbott.

[1]This is particularly true within a mobile environment when JavaScript payloads are big enough [5].

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

of loading [7]. Especially the cost of loading has become an urgent issue.

C. NEED FOR RESEARCH AND SOLUTIONS IN THE SPACE

In [8], it was shown that most external resources are typically loaded as render-blocking code. In contrast, only a small portion of the code is used on the initial page load, implying significant potential for performance improvements [8]. The finding highlights the need for research and development in the space.

D. EARLY POTENTIAL OF DISAPPEARING FRAMEWORKS

As discussed by [7], so-called disappearing frameworks address challenges related to SPAs by addressing the problem of shipping as little JavaScript to the client as possible. It is not only frameworks that matter, but application code specifically, as there is a concrete cost related to loading the code. One of the possible techniques allowing developers to load less and later is resumability.

E. RESUMABILITY ALLOWS DEVELOPERS TO LOAD LESS JAVASCRIPT UPFRONT

Resumability addresses the cost of turning HTML markup sent to the client into an interactive application by performing some of the necessary work beforehand on the server, handing it over to the client, and then picking it up on demand. In contrast, the current mainstream JavaScript frameworks implement a hydration technique where JavaScript code is required to turn a page interactive; it has to be shipped and evaluated by the client, leading to a double cost [9]. Hydration can be optimized through approaches, such as islands architecture [10], but that does not solve the fundamental issue of hydration fully that resumability avoids by changing the axioms.

F. HOW CAN RESUMABILITY HELP TO ADDRESS THE PROBLEM OF GROWING WEBSITE WEIGHT

Since website weight is a significant issue in web development and resumability may be one of the key ways to address it, we have formed the following research question: *How does resumability address the problem of growing website weight?*

To understand how we arrived at the concept of resumability and what motivated its development, we consider the technical background in Section II. In the following Section at III, we delve into the concept through the examples of Sidewind and Qwik. Then, in Section IV, we consider the implications of resumability and its potential for web development. Finally, we conclude the article in Section V and consider potential research directions.

II. BACKGROUND

To highlight why resumability is an important topic, we go through the main ideas leading to the current mainstream approaches in this section from the perspective of interactive websites and applications.

A. EARLY STEPS TOWARDS INTERACTIVITY

The early web did not have a clear way to add interactivity to websites. In 1995, Netscape Communications, one of the early successful browser vendors, decided that the web needed a scripting language [11]. The task was given to Brendan Eich, who created an early version of JavaScript, which, despite the name, did not have much to do with the popular Java language [11]. The idea was that JavaScript code would be combined with Java applets and other components [12].

B. MULTI-PAGE APPLICATIONS

Early web applications were implemented using the MPA model in which the application state lives on the server, and each request from the client to the server reloads the page [6]. The model has several benefits as the initial cost of loading an individual page can be low [13], SEO is easy [6], [13], there is no dependency on JavaScript [6], [13], and security practices are well understood [13]. For a highly interactive application, MPA as a model is impractical due to its primary constraint of maintaining the state on the server and refreshing a page on state change.

C. SINGLE PAGE APPLICATIONS

In contrast to MPAs, SPAs do not rely on refreshing a page on state change [13]. Instead, they maintain the application state in the browser and update the user interface dynamically based on user interaction [13]. As only transaction-related data moves between the client and the server, the SPA model can save bandwidth compared to MPAs [6] while coming with an initial loading cost [13]. These characteristics make SPAs ideal for long-running and complex web applications where the initial loading time does not matter. SPAs have challenges related to SEO [6], [13], [14], security [6], and potentially routing [13] although the routing problem has been largely solved by 2023.

Generally, SPA frameworks implement the following concepts [7]: component abstraction, templating, and hydration. Components provide a way to encapsulate markup and potentially local state while templating solutions, such as JSX [15], capture markup. The challenge is how to turn static markup rendered by the server into an interactive application suitable for a user, and that is where hydration comes in. Hydration re-uses the existing DOM nodes, attaches event handlers, and executes component logic [16].

D. CHALLENGES OF HYDRATION

By definition, hydration requires application component-related code to run, as hydration is about re-running application code to learn about the system's state. In other words, there is code to download, parse, and execute. Each of these steps comes with an associated cost and is visible in the initial loading cost related to using SPA. Using code-splitting can defer the cost by allowing code to be loaded later [17]. It is important to note that this applies only to components

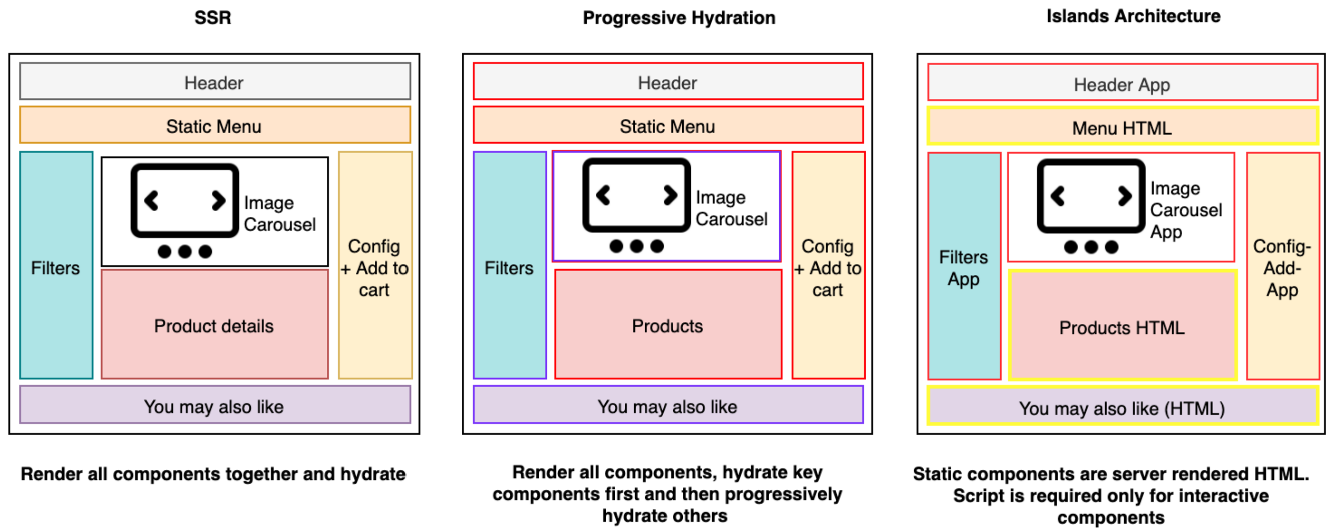

FIGURE 1. The figure shows a page comparable in three configurations: SSR, progressive hydration, and islands architecture. All the components are rendered together and then hydrated for the SSR option. In the case of progressive hydration, all the components are rendered, key components are hydrated first, and the rest are then progressively hydrated. For islands architecture, static components are server-rendered HTML; a script executed on the client side is required only for the interactive components [10].

not existing in the initial render tree, reducing the approach's usefulness. In other words, code-splitting may be used for cases requiring user interaction or some other trigger, such as an intersection observer.

Another option is to push work to build time, as in Static Site Generation (SSG), where component markup is generated ahead of time and then served to the user [18]. Markup generation can also occur during Server-Side Rendering (SSR) or incremental server regeneration (ISR) passes [18]. Regardless of the approach used to generate the markup, the hydration process remains the same, meaning earlier drawbacks remain. Next.js framework is a notable example as it implements these strategies and allows the developer to choose between them depending on the use case [19].

Progressive hydration allows hydrating the most critical components first while hydrating the rest later, although there is a caveat in the sense that hydration must begin from application root [10]. Islands architecture limits the problem to specific dynamic islands loaded using a specific strategy while treating the rest of the page as static [10]. It can be argued that islands are not without their challenges, as you have the problems of inter-island communication and page navigation to consider.[2] Figure 1 illustrates how the three basic approaches to hydration differ.

III. RESUMABILITY

Resumability approaches the problem solved by hydration from a different angle. Most importantly, resumability skips the re-execution portion of hydration, given the component boundaries, state, and event listeners are serialized into HTML, and the client will then pick up from there [21].

Both hydration and resumability have to provide front-end knowledge about the application. In hydration, the knowledge is passed via the execution of application components, while in resumability, the same is achieved by deserializing HTML. Through serialization, resumability avoids a significant cost related to hydration at the cost of having to provide the initial starting point as HTML [22].

A. IMPLEMENTING RESUMABILITY

Technically, resumability is surprisingly simple to implement as it boils down to how to resume state and execution from HTML markup. In any case, a JavaScript runtime is required. Simultaneously, a resumable approach can be designed with progressive enhancement [23] in mind so that the code will work even if JavaScript is disabled for the client. Implementation-wise, the question is how much work to do and how far to go. In the most straightforward implementations, it is enough to figure out how to read the state from HTML to a state container. More complex implementations can do optimizations related to event handling and code-splitting [17] for example, as those are possible to do in a novel manner on top of resumability.

1) SIDEWIND - A RUNTIME LEVERAGING RESUMABILITY

Sidewind[3] (2019) is an example of a library that leverages the idea of resumability. Sidewind is a light[4] state management solution designed to be used directly within HTML using specific directives modeled using HTML attributes and

[2]React Server Components address these problems in their way while depending on a server [20].

[3]https://sidewind.js.org/

[4]The minified version of Sidewind is roughly 16 kilobytes, and the cost can be reduced further to around 6 kilobytes by using gzip compression during transmission. The implementation can be optimized further as the current version includes all the functionality, and everything is loaded eagerly.

standard event handlers. Through resumability, a Sidewind-based state container can restore its state from the HTML markup. Sidewind fulfills the criteria for resumability by not re-executing the application on the client and by serializing state, listeners, and bindings to HTML.

The example below illustrates the syntax of Sidewind for a case where execution is resumed in the frontend as it constructs its state from HTML and then begins to operate on the state as visible in the **x** binding that shows the loaded data structure to the client. Due to the way state is included in the initial HTML markup, the page can be considered to follow the principle of progressive enhancement [24]. The benefit of progressive enhancement is that the page would work to some extent without JavaScript, making it more accessible, not to mention potentially having SEO benefits.

```
<div x-state=''{ todos: [] }''>
  <ul x-each=''state.todos'' x-ssr>
    <li x-template>
      <span x=''state.value.text''>Write</span>
      <ul x-each=''state.value.tags''>
        <li x-template x=''state.value''>
          chore
        </li>
      </ul>
    </li>
    <li x-template>
      <span x=''state.value.text''>Read</span>
      <ul x-each=''state.value.tags''>
      </ul>
    </li>
  </ul>
  <div x=''JSON.stringify(state.todos)''></div>
</div>
```

It is important to note that Sidewind has no opinions regarding how components should be created, so integrating Sidewind within existing environments or developing your abstractions to support it may be beneficial. There are also no strong opinions on how events should be handled or how more complex code should be loaded, as the developer can decide how to handle these concerns.

2) RESUMABILITY CAN BE DECOUPLED FROM EVENT HANDLING

Unlike in hydration-based approaches, resumability can be decoupled from event handling. For example, in Qwik,[5] one global event handler leverages browser event bubbling behavior, and events are then activated by user interaction [22]. Due to this feature, in Qwik, the amount of event handlers specified by a developer does not contribute to the application's running cost. In Sidewind, the situation is not as straightforward as the library leverages standard event handlers. It is an implementation detail of whether or not a single global event handler should be used.

Figure 2 illustrates the difference in the loading behavior of hydration against resumability when the user wants to use an interactive portion of a user interface, especially when a solution like Qwik is used. It is good to note that while the hydrated case is loaded eagerly, the resumable case can be loaded lazily on demand based on user interaction.

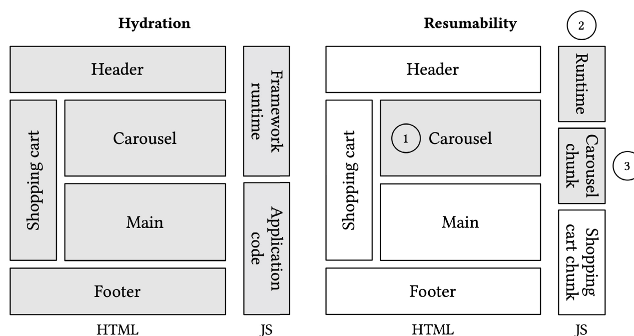

FIGURE 2. The figure illustrates how hydration and resumability differ in loading behavior. While in the naïve hydration case, framework and application code are loaded eagerly, in resumability, only runtime is loaded initially, and application code can be loaded on demand. In the image, the user intends to operate (1) the carousel component, and then (2) the runtime shipped to the client downloads (3) the related code. To provide more flexibility, a framework can let the developer adjust this behavior to consider different use cases, such as mobile and low bandwidth scenarios.

3) CODE-SPLITTING AS A FIRST-CLASS CITIZEN

To optimize application loading in hydration-based approaches, code-splitting [17] is used to defer loading based on interaction or some other trigger. In islands architecture [10], a similar effect is achieved on an architectural level. Resumability can enable more fine-grained code-splitting than earlier approaches. For example, Qwik leverages the dollar ($) character to signify a split point, as in the example below adapted from [25].[6]

```
import * as qwik from ''@builder.io/qwik'';

export default qwik.component$(() => {
  const count = qwik.useSignal(0);
  return (
    <div>
      <span>Count: {count.value}</span>
      <button onClick$={() => count.value++}>
        Click
      </button>
    </div>
  );
});
```

Due to its approach, Qwik's code-splitting boundaries can be small as they can exist for views, state, and event handlers [25], unlike in earlier approaches limited to module, typically component, boundaries. For more complex use cases, Qwik allows developers to tune the behavior through a Service Worker [26].

In the case of Sidewind, it is up to the developer to decide if they want to leverage code-splitting at an event handler level. Therefore, resumability can be seen as an enabler of granular

[5] https://qwik.builder.io/

[6] Examining different code extraction patterns would worth studying on its own in detail as what we cover here is only one option out of many.

code-splitting, and as a concept, it does not guarantee it as it is an implementation detail.

B. CHALLENGES OF IMPLEMENTING RESUMABILITY

As a new idea, implementing resumability comes with its challenges. To keep the approach ergonomic, at least some abstraction is required. In Sidewind runtime, the ergonomics are somewhat solved by leveraging standard HTML. For Qwik and Marko,[7] the frameworks have opted for leveraging a compiler that hides much of the complexity and enables functionality, such as granular code-splitting. It can be argued that the existence of a runtime or a compiler comes with its complexity as there is more for developers to debug and understand, as the code you write is not entirely what you may evaluate in the browser. However, the same problem applies to most mainstream frontend solutions as they rely on techniques, such as transpilation, that transform code to a form understandable by the browsers.

1) SERIALIZATION REQUIREMENTS CONSTRAIN A RESUMABILITY-BASED SYSTEM

Given that resumability relies on the ability of the server to serialize data into a format that the client can resume, that gives a significant constraint for implementations. In other words, data structures that cannot be serialized cannot live in a resumable system. The same constraint does not exist in hydration-based systems, and they can consume non-serializable data without problems.

2) PROGRAMMING LANGUAGE CONSTRAINTS FOR RESUMABILITY BASED SYSTEMS

When a resumability-focused library like Sidewind is used, the main constraint between the server and the client is that the server generates markup that the client-side runtime can parse and restore as a JavaScript structure. Therefore, in this case, the backend requirements are relatively light.

The situation is more complex for JavaScript frameworks like Qwik or Marko as they allow developers to implement backend logic. The situation is more constrained in these cases as the backend requires a JavaScript runtime to work. Simultaneously, developers can achieve more within the same framework and may be able to implement the most vital parts of web applications using these solutions.

C. POTENTIAL FOR ADOPTION BY FRAMEWORKS

For framework authors, resumability poses a challenge as it means a framework has to be reimagined from the ground up since underlying assumptions related to data loading differ from hydration-based approaches. The main difference concerns state management and how the state is loaded progressively over time as needed. The biggest shift concerns how a part of the state is serialized to the initial markup so the client can resume on it. As a novel idea, there are limited implementation examples in the wild. The biggest obstacle to larger framework adoption is the compiler requirement for ergonomics. However, a runtime-based alternative may be viable up to a point, as shown by the example of Sidewind.

[7] https://markojs.com/

D. FRAMEWORKS AND LIBRARIES LEVERAGING RESUMABILITY

Given that it is early days with resumability, possible ways to implement it have not been fully explored as only a few implementations exist. The main ones are Qwik, Wiz, and Sidewind. Out of these three, not much is known about Wiz as it is closed-source and not distributed publicly. In the case of Marko, resumability is planned [27] but has not been delivered yet. As a complete, compiled-based framework, Qwik can extract split points automatically to build on resumability. In contrast, Sidewind uses resumability only to capture state from HTML, leaving code-splitting to the user. Technically, it would likely be possible to build an entire framework around it, however. It is too early to tell about how Marko will implement resumability, but given they control both client and server-side code in their solution, they will likely find a good way to implement the idea.

To capture the currently available resumable frameworks and libraries, Table 1 lists the options while considering several attributes, including the streaming approach. The streaming approach is one of the factors where the current options differ, as their servers can stream content to the client either in or out of order. The same can be said for the templating approach.

E. EARLY EXPERIENCES WITH RESUMABILITY

As a new approach, there is little experience using resumability in practice. That said, early adoption is already visible, in addition to early empirical evidence that shows the strength of the approach.

1) EARLY ADOPTION OF RESUMABILITY IN APPLICATIONS

Early adoption of resumability has been visible in various instances, including Gmail (Wiz, [28]), Google Photos (Wiz, [28]), Google Search (Wiz, [28]), and eBay (Marko, [29]). Further, Marko [30] and Qwik [31] based examples are known to exist. It is safe to say that resumability is still far from a mainstream idea. More implementations will likely appear as resumability, and its benefits will be better understood.

2) EARLY EMPIRICAL EVIDENCE

In [32], it was found that using Qwik can decrease the amount of JavaScript shipped to the client compared to React. It was noticed, however, that this comes with a cost in terms of server rendering time and build time during deployments [32]. The early results of [33] imply that using Qwik gives benefits in terms of the initial loading performance of an application. It is good to note that both studies have limitations and may not give the full picture as further work is required within the domain, especially regarding continued application usage.

TABLE 1. Resumable web application frameworks and libraries.

Name	Type	Templating approach	Streaming approach	Version	Developers	Homepage
Asta	Framework	JSX	Unknown	0.0.0 (2023-01-31)	Yisar, an individual developer	https://github.com/yisar/asta
Marko	Framework	Custom DSL	In/out of order [27]	5.32.3 (2023-12-14) (beta for version 6 including resumability was planned for Summer 2023 but missed its deadline [27])	Developed as open source by eBay Inc. and other contributors	https://markojs.com/
Sidewind	Library	HTML	None	7.6.0 (2022-06-10)	Juho Vepsäläinen, an individual developer	https://sidewind.js.org/
Qwik	Framework	JSX	In order [27]	1.3.2 (2024-1-1)	Developed as open source by Builder.io, Inc. and other contributors	https://qwik.builder.io/
Wiz	Framework	Unknown	Unknown	Unknown	Developed as closed source by Google	Not available

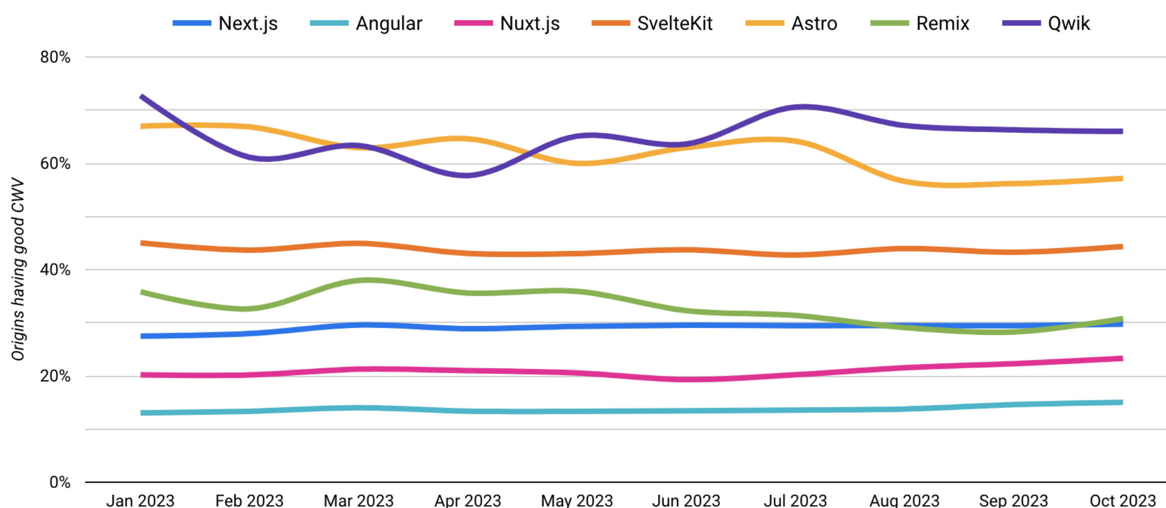

FIGURE 3. The figure shows how many websites using a specific JavaScript framework measured by Google have a portion of good Core Web Vitals (CWV) value while using a mobile client. Astro and Qwik perform above the rest, while Qwik has consistently been better than Astro, especially during the past few months. Similar results can be seen using a desktop client.

Google's Core Web Vitals (CWV) [34] technology report based on actual usage supports the initial findings and shows that Qwik-based websites outperform the rest as seen in Figure 3. It is good to note that there may be some bias in the numbers as Google evaluated around a hundred sites for Qwik while at least thousands for others [34]. The low figure tells about Qwik's early adoption; more accurate data may be gained as it becomes more popular.

Similar results can be seen for Google's median origin Lighthouse score and page weight, as Qwik is either leading or near leading in either case [34]. Astro[8] framework with its islands architecture can deliver pages with low page weight, or it is used for sites with a low page weight, and the same could be argued for Qwik. Based on the results alone, it is difficult to tell which case might be true, and as mentioned, there may be bias in the numbers due to the low sample size for Astro and Qwik relative to other options.

[8]https://astro.build/

F. MAIN DIFFERENCES BETWEEN HYDRATION AND RESUMABILITY

Table 2 captures the main differences between hydration and resumability to allow contrasting approaches and to recap the main points presented in this article.

IV. DISCUSSION

Although current mainstream solutions, such as SPAs, allow us to develop complex, interactive applications, they do not address the cost of website size and loading as their primary axioms. Rather, developers must try to address it independently through techniques like code-splitting [17]; even then, the results can be suboptimal. By changing the viewpoint, resumability provides another direction that may yield better results without additional effort for developers.

TABLE 2. Main differences between hydration and resumability.

Aspect	Hydration	Resumability
Client-side code loading strategy	Eager by default, lazy in limited contexts	Potentially lazy depending on the implementation
Client runtime	Library itself can be a runtime that hydrates application code delivered to the client	A runtime is generated to load application code that can be loaded separately
Code-splitting approach	Component and module-level	Potentially granular depending on the implementation
Implementation	May include a compiler (for example, JSX) but can be fully a runtime	Most likely implemented as a compiler to enable good development ergonomics, although a pure runtime-based solution is possible, as shown by Sidewind
Programming language requirements	None	Same language may have to be used on the server and client due to serialization requirements. In Sidewind, it is enough that the server can generate markup that can be deserialized by the client
Server-to-client data serialization	No constraints	Data passed from the server to the client has to be serializable

A. RESUMABILITY CAN HELP TO ADDRESS THE COST OF WEBSITE SIZE

Resumability can address the cost issue of websites at the tooling or library level while making websites faster to load. In resumability, the server serializes data to the markup, allowing the client to resume execution as needed. The main difference is that in a hydration-based approach, the client has to construct the state of the client independently, therefore adding cost to the approach. The shift enables code-splitting [17] as an architectural concern that the developers do not have to think about unless they prefer to, as prefetching can become a configurable aspect of an application [26].

B. RESUMABLE RUNTIMES CAN PROVIDE AN INTERMEDIATE STEP TOWARDS COMPILERS

As an intermediate solution, resumability can be adopted through a runtime-based approach at the cost of losing automatic code-splitting. A runtime may be easier to integrate into an existing project than a compiler-based solution, providing an alternative to a full refactor or an interim step towards one.

C. CAN MAINSTREAM FRAMEWORKS ADOPT RESUMABILITY?

One of the open questions is whether current mainstream frameworks can or should leverage resumability or will the concept be adopted solely by new frameworks and libraries. Technically, it is diametrically opposite to the concept of hydration implemented by many of the frameworks, meaning implementing resumability would represent a major change and shift in thinking while coming with the constraints related to serialization requirements. Therefore implementing resumability in an existing framework may represent a breaking change.

D. OTHER WAYS TO ACCESS THE COST OF WEBSITE SIZE

The cost of web application size can be potentially mitigated by leveraging external tools, as shown by [35]. The tools, Waiter and AUTRATAC[9] can apply code-splitting on existing codebases and therefore defer loading. In [35], it was shown that the tools improve First Contentful Paint (FCP), particularly at slower network speeds. The important point to make is that in [35] the total execution time matched the original JavaScript, but it was loaded with less render-blocking, which is a major improvement by itself.

The development of external tooling seems like another promising way to address the cost of websites. It can be argued that techniques like resumability may provide better results out of the box. However, adopting better techniques may not be feasible in many cases, meaning there is value in finding solutions that fit existing projects with minimal effort.

Although the directions of resumability and external tooling feel orthogonal, there may be some cases where both resumability and external tooling may be used together, as resumability does not imply automatic code-splitting. External tooling could bring the technique to use cases where only resumability is leveraged initially, especially when only a resumable runtime is used.

V. CONCLUSION

In this article, we sought to answer the question *How does resumability address the problem of growing website weight?*. Based on [3], we know that site weight, and consequently the weight of JavaScript execution, is a considerable problem. Although SPAs have enabled developers to build unprecedented, interactive experiences on top of the web platform, they do not address the size problem as a first-class citizen. Instead, the developers have to optimize their applications using techniques such as code-splitting [17] and memoization [36], and it can be argued that the results may not be ideal even then as they push the problem to developers and may not provide granular enough control. The problem is that optimization techniques do not address the eager nature of hydration. Resumability helps to address the site weight issue by avoiding eager execution and provides one potential direction for solving the problem of growing web application size.

A. POTENTIAL RESEARCH DIRECTIONS

As a new approach, many open research questions exist related to resumability. These can be categorized within approach, developer, and user-facing directions.

[9]Both tools are available through GitHub via https://github.com/waiter-and-autratac/WaiterAndAUTRATAC

1) APPROACH RELATED DIRECTIONS

When it comes to resumability as an approach, there are several levels of validation to be done:

1) Performance against hydration - especially performance against hydration using a real-life use case is interesting to study to see how the approach scales since hydration is the incumbent approach and the one to beat.[10]
2) Cost per request - given there is more work to be done by the server by definition, it may make sense to measure the cost per request against established approaches. Reference [32] provided initial insight into the topic, but more research is needed. It is possible that common techniques, such as caching, may mitigate some of the server costs.
3) Continued usage of an application and impact on the overall performance - [37] did an initial study on the topic, although the results were inconclusive and not easy to reproduce. While initial loading matters, so does long-term use since we are talking about web applications that may have a long lifecycle in the client browser.

2) DEVELOPER FACING DIRECTIONS

Even if the approach is technically sound, there are open questions related to developer ergonomics and adoption:

1) Developer eXperience (DX) - the underlying assumption is that it helps when familiar concepts, such as components, are leveraged. Still, it is unavoidable that adoption resumability implies learning and perhaps changing development practices. As seen by the examples in this article, some additional syntax may have to be learned, and possibly, the development mindset has to be adjusted.
2) Adoption in established projects - it is unclear how easy it is to adopt resumability in ongoing projects. The assumption is that runtimes can help, but not all benefits may be gained. That is where external tooling, such as [35], can come into play to defer loading as much as possible with fully resumable frameworks.
3) Migration strategies - migration is a related aspect as, in some cases, it may be preferable to move from the current framework to a resumable one, and a certain cost is involved. There are also open questions related to which migration strategies make sense and can or should resumable approaches be used next to contemporary ones.
4) Adoption for new projects - as shown by Qwik, the framework took care to build on existing technology, such as JSX. The question is, how easy is it to start a project with a resumable project, and is there a cost relative to established frameworks?
5) Adoption by new frameworks - assuming resumability is a worthwhile idea, more frameworks will likely adopt it. The question is, what are the obstacles related to implementing the idea? Furthermore, is it possible that established frameworks could implement resumability? Which are the directions where frameworks can develop unique advantages over others?
6) Scalability - given projects tend to grow over time, one of the main factors to consider is the scalability of the approach in terms of development and payloads delivered to the client. By definition, resumability should scale well as work can be deferred on an architectural level, but this should be validated to show what happens when a project grows.
7) Code complexity - as resumability brings new constraints to how code should be developed, it may affect code complexity. The complexity of resumable code may be worth considering.

3) USER-FACING DIRECTIONS

The user perspective is perhaps the most vital one, as it directly impacts web application usage and has a direct business impact. The main direction to study is User eXperience (UX) - the question is how adopting resumability affects UX. Early results by [34] imply that resumability improves UX, but further validation is needed.

[10]For example, https://www.builder.io/ could be compared against https://www.builder.io/?render=next as the site has been implemented using both Qwik and Next.js.

REFERENCES

[1] T. Berners-Lee, R. Cailliau, J. Groff, and B. Pollermann, "World-Wide Web: The information universe," *Internet Res.*, vol. 20, no. 4, pp. 461–471, Aug. 2010.

[2] *Internet and Social Media Users in the World 2023*. Accessed: Aug. 28, 2023. [Online]. Available: https://www.statista.com/statistics/617136/digital-population-worldwide/

[3] *Page Weight | 2022*. Accessed: Aug. 28, 2023. [Online]. Available: https://almanac.httparchive.org/en/2022/page-weight

[4] J. Howarth. 2023. *Internet Traffic from Mobile Devices (Sept 2023)*. Accessed: Aug. 28, 2023. [Online]. Available: https://explodingtopics.com/blog/mobile-internet-traffic

[5] T. Kadlec. (2014). *JS Parse and Execution Time*. Accessed: Sep. 13, 2023. [Online]. Available: https://timkadlec.com/2014/09/js-parse-and-execution-time/

[6] M. Kaluža, K. Troskot, and B. Vukelić, "Comparison of front-end frameworks for web applications development," *Zbornik Veleučilišta Rijeci*, vol. 6, no. 1, pp. 261–282, 2018.

[7] J. Vepsäläinen, A. Hellas, and P. Vuorimaa, "The rise of disappearing frameworks in web development," in *Proc. Int. Conf. Web Eng.* Cham, Switzerland: Springer, 2023, pp. 319–326.

[8] L. Vogel and T. Springer, "An in-depth analysis of web page structure and efficiency with focus on optimization potential for initial page load," in *Proc. Int. Conf. Web Eng.* Cham, Switzerland: Springer, 2022, pp. 101–116.

[9] X.-A. Cao, "Headless cms and qwik framework and their practicalities in the future of application development," M.S. thesis, Vaasa Univ. Appl. Sci., Vaasa, Finland, 2023.

[10] L. Hallie and A. Osmani. 2022. *Islands Architecture*. Accessed: Sep. 29, 2022. [Online]. Available: https://www.patterns.dev/posts/islands-architecture/

[11] J. Vepsäläinen, "ECMAScript—The journey of a programming language from an idea to a standard," in *Proc. Joint EURAS SIIT Standardisation Smart Syst.*, Jul. 2023, pp. 203–220.

[12] A. Wirfs-Brock and B. Eich, "JavaScript: The first 20 years," *Proc. ACM Program. Lang.*, vol. 4, pp. 1–189, Jun. 2020.

[13] V. Solovei, O. Olshevska, and Y. Bortsova, "The difference between developing single page application and traditional web application based on mechatronics robot laboratory onaft application," *Automat. Technol. Bus. Process.*, vol. 10, no. 1, pp. 4–8, 2018.
[14] T. F. Iskandar, M. Lubis, T. F. Kusumasari, and A. R. Lubis, "Comparison between client-side and server-side rendering in the web development," *IOP Conf. Ser., Mater. Sci. Eng.*, vol. 801, no. 1, May 2020, Art. no. 012136.
[15] S. Chen, U. R. Thaduri, and V. K. R. Ballamudi, "Front-end development in react: An overview," *Eng. Int.*, vol. 7, no. 2, pp. 117–126, Dec. 2019.
[16] A. Huotala, "Benefits and challenges of isomorphism in single-page applications: A case study and review of gray literature," M.S. thesis, Dept. Comput. Sci., Univ. Helsinki, Helsinki, Finland, 2021.
[17] B. Livshits and E. Kiciman, "Doloto: Code splitting for network-bound Web 2.0 applications," in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2008, pp. 350–360.
[18] J. Vepsäläinen, A. Hellas, and P. Vuorimaa, "Implications of edge computing for static site generation," in *Proc. 19th Int. Conf. Web Inf. Syst. Technol.* Cham, Switzerland: Springer, 2023, pp. 223–231.
[19] T. N. Nguyen, "Family in music platform," M.S. thesis, Vaasa Univ. Appl. Sci., Vaasa, Finland, 2022.
[20] Vercel. (2023). *Rendering: Server Components*. Accessed Oct. 4, 2023. [Online]. Available: https://nextjs.org/docs/app/building-your-application/rendering/server-components
[21] (2023). *Resumable vs. Hydration*. Accessed: Aug. 29, 2023. [Online]. Available: https://qwik.builder.io/docs/concepts/resumable/
[22] M. Hevery. (2022). *Resumability vs Hydration*. Accessed: Aug. 29, 2023. [Online]. Available: https://www.builder.io/blog/resumability-vs-hydration
[23] S. Champeon. (2003). Progressive Enhancement and the Future of Web Design. Accessed: May 15, 2023. [Online]. Available: http://www.webmonkey.com/03/21/index3a.html
[24] A. Gustafson, L. Overkamp, P. Brosset, S. V. Prater, M. Wills, and E. PenzeyMoog. (Oct. 2008). *Understanding Progressive Enhancement*. Accessed: Sep. 29, 2022. [Online]. Available: https://alistapart.com/article/understandingprogressiveenhancement/
[25] J. Vepsäläinen, A. Hellas, and P. Vuorimaa, "The state of disappearing frameworks in 2023," in *Proc. 19th Int. Conf. Web Inf. Syst. Technol.* Cham, Switzerland: Springer, 2023, pp. 232–241.
[26] (2023). *Prefetching*. Accessed: Aug. 29, 2023. [Online]. Available: https://qwik.builder.io/docs/advanced/prefetching/
[27] Marko Team. 2023. *Talking Points for Marko—HackMD*. Accessed: Oct. 4, 2023. [Online]. Available: https://hackmd.io/@markojs/BkW3fIze2
[28] Malte Ubl. (2022). *11 Years at Google*. Accessed: Oct. 4, 2023. [Online]. Available: https://www.industrialempathy.com/posts/11-years-at-google/
[29] (2023). *Marko*. Accessed: Oct. 4, 2023. [Online]. Available: https://markojs.com/
[30] Wappalyzer. (2023). *Websites Using Marko*. Accessed: Oct. 4, 2023. [Online]. Available: https://www.wappalyzer.com/technologies/web-frameworks/marko/
[31] Wappalyzer. (2023). *Websites Using Qwik*. Accessed: Oct. 4, 2023. [Online]. Available: https://www.wappalyzer.com/technologies/web-frameworks/qwik/
[32] T. Lonka, "Improving the initial rendering performance of react applications through contemporary rendering approaches," M.S. thesis, School Sci., Aalto Univ., Helsinki, Finland, 2023.
[33] BuilderIO. (2022). *GitHub—BuilderIO/Framework-Benchmarks*. Accessed: Oct. 3, 2023. [Online]. Available: https://github.com/BuilderIO/framework-benchmarks
[34] (2023). *Core Web Vitals Technology Report*. Accessed: Nov. 20, 2023. [Online]. Available: https://lookerstudio.google.com/u/0/reporting/55bc8fad-44c2-4280-aa0b-5f3f0cd3d2be/page/M6ZPC?s=iL-EeFbtDHg¶ms=%7B%22df44%22:%22include%25EE%2580%25800%25EE%2580%2580IN%25EE%2580%2580Next.js%25EE%2580%2580Angular%25EE%2580%2580Nuxt.js%25EE%2580%2580SvelteKit%25EE%2580%2580Astro%25EE%2580%2580Remix%25EE%2580%2580Qwik%22%7D
[35] L. Vogel and T. Springer, "Waiter and AUTRATAC: Don't throw it away, just delay!" in *Web Engineering*, I. Garrigós, J. M. M. Rodríguez, and M. Wimmer, Eds. Cham, Switzerland: Springer, 2023, pp. 278–292.
[36] I. Kainu, "Optimization in React.js: Methods, tools, and techniques to improve performance of modern web applications," B.S. thesis, Dept. Comput. Sci., Tampere Univ., Tampere, Finland, 2022.
[37] A. Lipiński and B. Pańczyk, "Performance optimization of web applications using Qwik," *J. Comput. Sci. Inst.*, vol. 28, pp. 197–203, Sep. 2023.

JUHO VEPSÄLÄINEN received the M.S. degree in information technology from the University of Jyväskylä, Finland, in 2011. He joined the Department of Computer Science, Aalto University, Finland, in 2022, to pursue a doctorate as a Doctoral Researcher. As a core team member, he has contributed to open-source projects, such as Blender and webpack, and publishes technical literature under the brand SurviveJS. His research interests include web development, web performance, and hybrid models for web application development.

MIŠKO HEVERY received the M.S. degree in computer engineering from the Rochester Institute of Technology, USA, in 2000, and the M.B.A. degree from Santa Clara University, in 2004. As a CTO, he oversees the technology division powering Builder.io applications. He is known for creating Angular and AngularJS frameworks and is the co-creator of the Karma testing framework. His latest project is Qwik, a resumable web application framework.

PETRI VUORIMAA received the M.S. degree in computer science from the Tampere University of Technology, Finland, in 1990, and the Doctor of Science degree from Tampere, in 1995. Currently, he holds the position of the Vice Head of education with the Department of Computer Science, Aalto University, Finland. His research and teaching areas include web applications, web technologies, and science. In addition to his academic roles, he actively contributes to the industry as a Supervisory Board Member of EIT Digital and a Board Member of Sofia Digital.