UDC 614.2+574/578+004.38
DOI https://doi.org/10.32782/IT/2023-4-5

Oleksandr LYTVYNOV
Candidate of Technical Sciences, Associate Professor, the Faculty of Physics, Electronics and Computer Systems, Oles Honchar Dnipro National University, 72, Haharina Ave, Dnipro, Ukraine, 49000, lytvynov@ffeks.dnu.edu.ua
***ORCID:** 0000-0001-7660-1353*

Dmytro HRUZIN
Master, Postgraduate Student, the Faculty of Physics, Electronics and Computer Systems, Oles Honchar Dnipro National University, 72, Haharina Ave, Dnipro, Ukraine, 49000, hruzin_dl@ffeks.dnu.edu.ua
ORCID: *0009-0004-8534-2559*

To cite this article: Lytvynov, O., Hruzin, D. (2023). Metody optymizatsii zavantazhennia ta onovlennia vebstorinok za dopomohoiu khmarnykh tekhnolohii [Methods for optimizing the loading and updating of web pages using cloud technologies]. *Information Technology: Computer Science, Software Engineering and Cyber Security,* 4, 40–50, doi: https://doi.org/10.32782/IT/2023-4-5

METHODS FOR OPTIMIZING THE LOADING AND UPDATING OF WEB PAGES USING CLOUD TECHNOLOGIES

An essential component of modern systems built on cloud technologies, which determines the performance and efficiency of system operation, is caching methods and technologies. For systems that provide static information to end-users, technologies such as SSG (Static Site Generation) and SSR (Server-side Rendering) are applied in building the client side. SSG is typically used for pages with a very low frequency of content changes, as modifying the content on one page requires rebuilding the entire site. To cache SSR application pages, the use of a cache proxy server is recommended. An example of implementing such an approach is Next.js Serverless. However, because the Next.js Serverless technology employs a weak caching model, delivering up-to-date data to end users often takes a significant amount of time, which can negatively impact user interaction.

This work is dedicated to addressing this problem. The paper proposes a flexible and efficient solution for generating and caching static pages using a cache proxy server and invalidation for data updates by the cache proxy server upon each change in the system's state. The integration of this solution into complex information systems is explored, and a performance comparison and evaluation of data delivery to end-users are conducted.

The results of the conducted experiment demonstrate that the proposed approach solves the identified problem without compromising the performance compared to the Next.js Serverless approach.

***Key words:** Cloud technologies, Server Side Rendering, Static Web Pages, Edge caching, Cache Proxy Server, Domain Driven Design.*

Олександр ЛИТВИНОВ
кандидат технічних наук, доцент, факультет фізики, електроніки та комп'ютерних систем, Дніпровський національний університет імені Олеся Гончара, просп. Гагаріна, 72, м. Дніпро, Україна, 49000, lytvynov@ffeks.dnu.edu.ua
***ORCID:** 0000-0001-7660-1353*

Дмитро ГРУЗІН
магістр, аспірант, факультет фізики, електроніки та комп'ютерних систем, Дніпровський національний університет імені Олеся Гончара, просп. Гагаріна, 72, м. Дніпро, Україна, 49000, hruzin_dl@ffeks.dnu.edu.ua
***ORCID:** 0009-0004-8534-2559*

Бібліографічний опис статті: Литвинов, О., Грузін, Д. (2023). Методи оптимізації завантаження та оновлення вебсторінок за допомогою хмарних технологій. *Information Technology: Computer Science, Software Engineering and Cyber Security, 4,* 40–50, doi: https://doi.org/10.32782/IT/2023-4-5

МЕТОДИ ОПТИМІЗАЦІЇ ЗАВАНТАЖЕННЯ ТА ОНОВЛЕННЯ ВЕБСТОРІНОК ЗА ДОПОМОГОЮ ХМАРНИХ ТЕХНОЛОГІЙ

Важливою складовою сучасних систем, побудованих на базі хмарних технологій, від якої залежить продуктивність та ефективність функціонування системи, є методи та технології кешування. Для ряду систем, які надають статичну інформацію кінцевому користувачеві, для побудови клієнтської частини застосовуються такі технології, як SSG (Static Site Generation), SSR (Server-side Rendering). SSG, як правило, застосовується для сторінок, частота зміни вмісту яких дуже низька, оскільки для зміни вмісту на одній сторінці потрібне перезбирання всього сайту. Для кешування сторінок SSR-додатків пропонується використовувати кеш-проксі-сервер. Прикладом реалізації такого підходу може служити Next.js Serverless. Проте через те, що технологія Next.js Serverless використовує слабку модель кешування, доставка актуальних даних кінцевим користувачам часто займає значну кількість часу, що може негативно вплинути на взаємодію з користувачем.

Вирішенню цієї проблеми присвячена дана робота. У роботі пропонується гнучке та ефективне рішення для генерації та кешування статичних сторінок з використанням кеш-проксі-сервера та інвалідації для актуалізації даних кеш-проксі-сервера при кожній зміні стану системи. Досліджено інтеграцію цього рішення в складні інформаційні системи, проведено порівняння продуктивності та оцінку доставки даних кінцевому користувачеві.

Результати проведеного експерименту показують, що запропонований підхід вирішує поставлену проблему, не уступаючи в продуктивності підходу Next.js Serverless.

Ключові слова: *хмарні технології, рендеринг на боці сервера, статичні вебсторінки, Едж Кешування, Кеш Проксі-сервер, Доменно-орієнтований Дизайн.*

The trend of migrating businesses to the cloud emerged long ago and has been resilient for quite some time. Correspondingly, cloud technologies continue to evolve. Currently, individuals are not just allowed to rent space for their servers but can gradually purchase it, dynamically scaling computational power both vertically and horizontally as needed.

The primary advantage of **cloud technologies** (Thakur, 2022) lies in their flexibility. Users no longer need to worry about physical hardware, its maintenance, and updates. They can focus on their business, leaving all technical aspects in the hands of the cloud provider. This also reduces operational costs and enhances efficiency, allowing companies to concentrate on the development and implementation of new products and services.

With each passing year, cloud technologies become increasingly accessible. The cost of services offered makes them viable even for novice startups. Moreover, major conglomerates such as Amazon (2), Google (3), and Microsoft (4) provide various discounts and free services up to specific credit amounts to attract potential clients. In addition, user interfaces are evolving, enabling non-technical users to leverage cloud services. While this, on one hand, reduces the skill requirements for DevOps personnel, on the other hand, it necessitates their familiarity with the specialized knowledge of using a specific console. Given the multitude of options available, training and even user certification programs are conducted to guide users in effectively utilizing cloud systems.

Talking about cloud services and web performance optimization, special attention is given to **data caching** (Chockler, 2011: pp. 1-11)(Choi, 2020: pp. 98-110)(Berger, 2014: pp. 2-23). Cloud platforms provide a range of caching mechanisms, including in-memory caches, distributed caches, and cache on the side of content delivery networks (CDNs). In-memory caches like Redis (8) and Memcached (Fukuda, 2014) prove particularly effective in accelerating data retrieval by storing frequently accessed data in RAM, enabling ultra-low-latency access. Distributed caching solutions, such as Amazon Elasticache (10), extend this capability across multiple nodes, further enhancing the speed and reliability of data access. CDNs, on the other hand, facilitate the efficient distribution of content and data to multiple geographic locations, reducing latency for users and researchers worldwide. These caching techniques are invaluable in scenarios where real-time access to data and services is critical.

Caching in cloud services not only enhances the speed and efficiency of scientific computations but also contributes to cost savings. By reducing the amount of data transfer and computational overhead, researchers can optimize resource utilization, ultimately leading to lower cloud service bills. Moreover, caching enables the reuse of previously computed results, minimizing redundant calculations and improving the sustainability of research projects.

Another technology that significantly increases cloud performance is **Edge caching** (Wu, 2021). The critical component of content delivery and data distribution networks is designed to enhance the speed and efficiency of content delivery to end-users. Unlike traditional caching, which is typ-

ically centralized on a single server or data center, edge caching decentralizes the caching process, bringing content closer to the end-users by placing cached data on edge servers distributed across various geographic locations. This approach significantly reduces the latency and congestion associated with serving content from a centralized location, leading to faster and more reliable access.

The best performance of edge caching is achieved when a website is developed using static web page technology. Pre-rendered pages are cached at the edge, ensuring rapid and efficient functionality. The concept of **pre-generating static web content** (Large, 2022)(Yang, 2022), often referred to as "**static web pages**", predates the modern static site generators (Jiang, 2010: pp: 588-591) that we use today. It's challenging to attribute this concept to a single individual, as it evolved gradually as the web itself developed. Static web pages have been used since the early days of the World Wide Web.

While the concept of static web pages was established early in the history of the web, the specific tools and technologies that we now associate with modern static site generators were developed later. Tools like Jekyll (15), Eleventy (16), Gatsby (17), and others were created to automate and simplify the process of generating static websites. The idea is to generate all possible web pages in advance and cache them. This provides significant performance improvement for read requests. However, this approach provides a low level of flexibility. The main con of the solution is when content does change, the site must be rebuilt to have these changes reflected. In practice, some hybrid approaches work. For example, as described in (Vepsäläinen, 2022), static and dynamic parts of the page are kept separately. The JSON for site definition is leveraged on the client side for editing, bridging the continuum's ends.

Another approach is the **Single Page Application (SPA)** (Fink, 2014), a web application that loads a single HTML page and dynamically updates its content as users interact, eliminating the need for a full page reload. This development paradigm has gained popularity due to its enhanced user experience, faster navigation, and reduced server load. However, a notable drawback of this approach is that the page renders after the initial load, introducing a certain delay before displaying data to the user. Additionally, this can impact page indexing by search engine robots, as dynamically generated content is ignored by some bots, negatively affecting Search Engine Optimization (Gudivada, 2015: pp. 67-76).

Another option for building web clients for complex information systems is **Server Side Rendering (SSR)** (Sun, 2019: pp. 191-217). When a user requests a page from a server, the server determines which page to render and processes the request, including any data fetching or computations needed to render the page. This may involve querying a database, making API requests, or other server-side logic. The server uses a template engine (e.g., Handlebars (22), Pug (23), etc.) to render the HTML template with the data retrieved in the previous step. The resulting HTML is a fully formed web page, often including the content, layout, and initial data. The server sends the complete HTML page, along with any associated assets like CSS and JavaScript files, to the client's browser. Once the initial HTML is loaded in the browser, any client-side JavaScript can take over and enhance the page's interactivity. The process repeats when a user navigates to another page within the application. A new request is sent to the server, which generates the HTML for the new page, and the client-side JavaScript updates the page content.

In practice, a common approach involves employing both methods for different parts of the system. Static pages, which do not require frequent updates, are generated using a generator and cached, while web pages in need of frequent updates are based on SSR or SPA approaches. Although these solutions work sufficiently, concerns regarding performance and content updates persist. Additionally, it is often a challenge for developers to find the right balance, determining which pages to generate and which to render in real time.

Methods exist to optimize page load time when using the SSR approach, such as employing a multi-level cache (Vilas, 2006: pp. 713-720) and a cache proxy server (Wang, 1999). The system's operation using these methods is illustrated in Fig. 1. When a client sends a request for a page, the request is received by the CDN service, and if an up-to-date version of the page is present, it is delivered to the client. If the page is not present or its time-to-live has expired, the request is redirected to the cache proxy server. This cache has a larger size, and the time-to-live for pages can be longer. If the cache proxy server also lacks an up-to-date version of the page, the page is requested from the server, which renders it and sends it to the client, also saving the updated version on the cache proxy server.

An example of implementing such an approach can be found in the Next.js Serverless solution (26) (27)(28). This framework is designed to create a

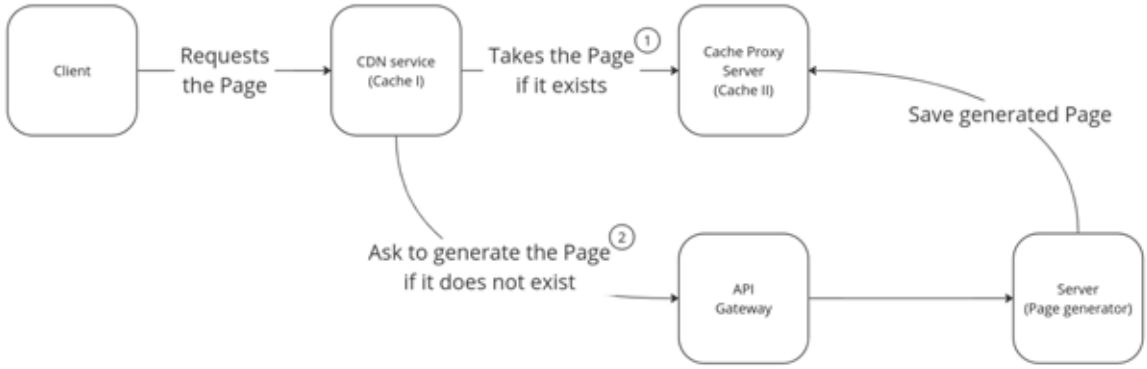

Fig. 1. Cache Proxy server approach

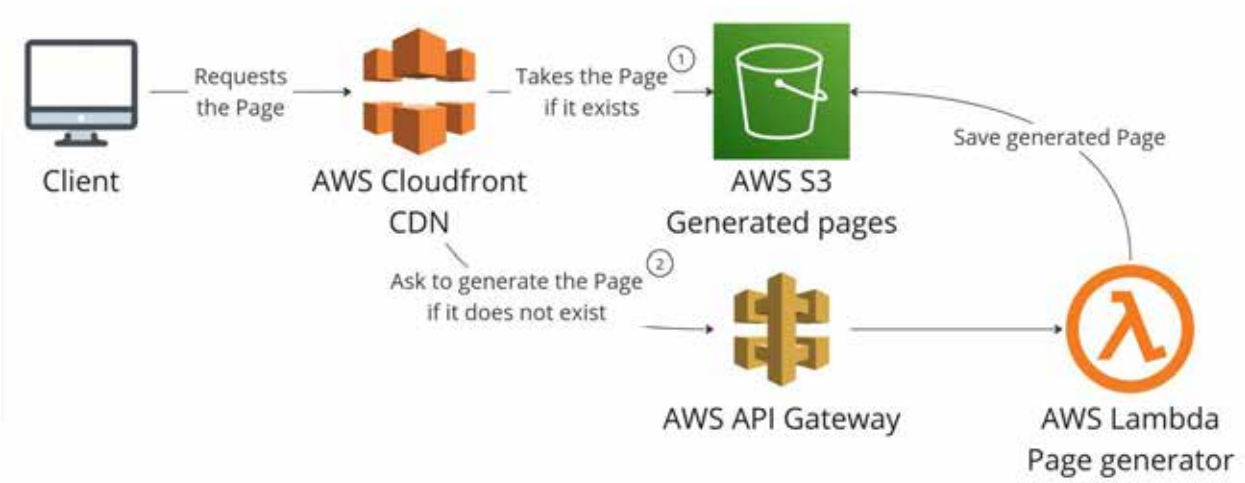

Fig. 2. The diagram of components for the Next.js Serverless approach

serverless architecture (Rajan, 2018), which offers certain advantages over the traditional server architecture for certain systems. The concept involves a resource distribution approach. In the case of the Next.js (30) SSR framework, the conventional server, which requires scaling and continuous operation with associated costs, is replaced with a storage solution for static files (AWS S3) and a stateless worker that can be invoked on-demand (AWS Lambda). The S3 server in this system essentially serves as a cache proxy server, with AWS CloudFront as the CDN service. Implementation of a system based on the Next.js Serverless framework is depicted in Fig. 2.

This approach ensures a high level of scalability, enabling the parallel rendering of a vast number of pages. However, the question of what cache model, weak or strong, to use remains. Cache invalidation operations on CDNs provided by cloud platforms can be relatively costly, especially with a large number of calls. Therefore, it is advisable to avoid cache invalidation unless necessary. Another consideration is data freshness in case a weak cache model is used.

Next.js Serverless approach utilizes a weak caching model, delivering up-to-date data to end-users often takes a considerable amount of time, which can negatively impact user experience.

Task definition. The solutions mentioned above are aimed at systems whose data state changes not too frequently but is not entirely static either.

The example can be illustrated by the health care system developed by a team from DBB Software company (31). The system aims to provide users with public profiles of doctors and their practices, along with reviews about these doctors. The system undergoes changes when content managers add or modify doctor profiles or their practices, and when patients add reviews. These are not real-time operations, and the update frequency of the page does not exceed tens of changes per hour.

One of the possible architectural solutions considered for building such a system was the generation of static pages. However, despite the high performance of this approach, especially when using edge caching, such a solution was rejected

at the planning stage. The challenge with the page generation approach lies in the need for rebuilding and redeploying all pages to make any changes. Using this approach in the current scenario would be inconvenient, as the rebuilding process takes time, and rebuilding the entire site multiple times a day would be resource-intensive.

The first solution implemented was the SSR Next.js server. SSR renders pages in real-time, and all system changes are reflected on the client almost immediately after the system data is updated. However, this approach comes at the cost of performance, as each page request involves one or more API requests, leading to increased page load time. Even with the use of CDN caching, the average page load speed for some pages reached up to 1700 ms.

Using the Cache Proxy Server with a weak caching model partially addresses this issue by introducing a second level of managed cache. However, on some pages, data may remain unchanged for months, while on others, it is constantly added.

For example, there is a doctor who obtained the last license several years ago and only treats his regular patients. Such a doctor's page will not be updated for months or even years. On the other hand, some doctors ask their patients to leave reviews and constantly receive new certificates. The pages of these doctors are updated multiple times per day. By default, the cache TTL is set the same for all pages. To ensure users receive up-to-date data, the TTL value should be set close to the update frequency of the most frequently updated pages.

One solution to this problem could be to implement logic on the cache proxy server that is responsible for grouping pages based on their update frequency. For each group, set a TTL that is most suitable for that specific group. However, with this solution, the dynamics of data changes need to be taken into account. For example, a doctor may start asking patients to leave reviews, and a page that used to change once a year will suddenly start changing several times a day. Considering the high complexity and low level of reliability, this solution is not considered effective.

This article aims to find an effective solution for the flexible response of the page generation module to changing data. It also provides a performance comparison and evaluates the delivery of updated data to the end user.

Main part. Similar to the Cache Proxy Server approach described above, we also propose rendering and storing pages in a static repository but using a strong caching model and storing not just already requested pages, but all possible pages in advance. The freshness of data in the storage is ensured by regenerating pages with each change (or some amount of changes) in the system that concerns these pages. One or more handlers subscribed to events about changes in the system's data, determine which pages are affected by these changes, and invoke the page renderer service to update these pages in the cache proxy server. After the pages are successfully rendered and saved, a request for invalidating the CDN cache is sent (Fig. 3).

For example, in Domain-Driven Design (DDD) systems (Evans, 2004) (Fig. 4), one of the handlers subscribed to changes in the aggregate's state can perform this function.

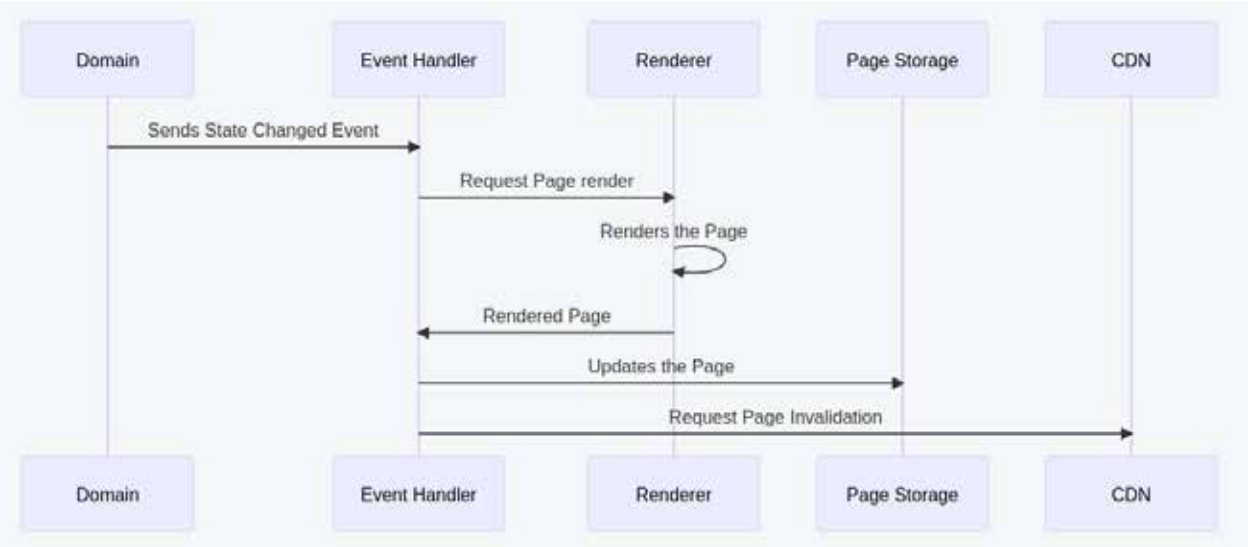

Fig. 3. Update the Page flow

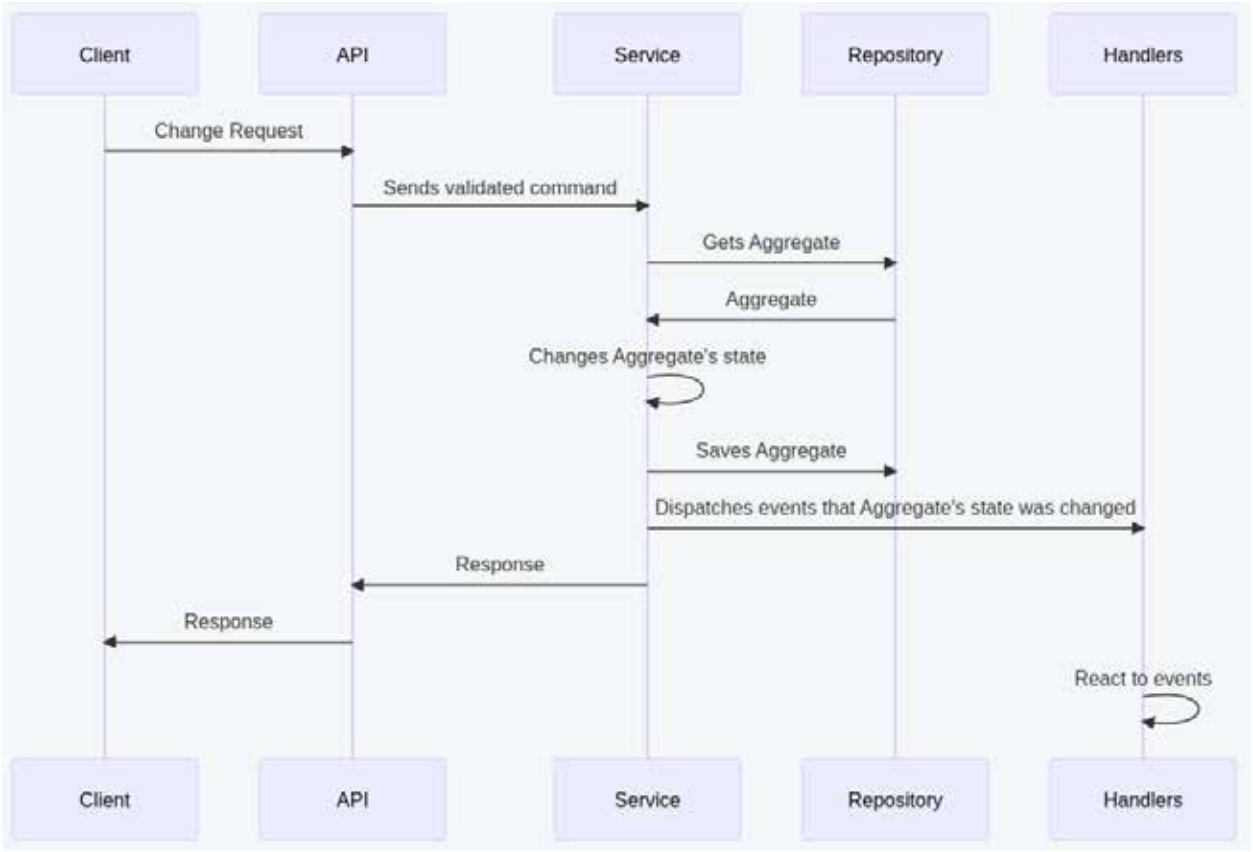

Fig. 4. Sequence diagram of successful DDD change request flow

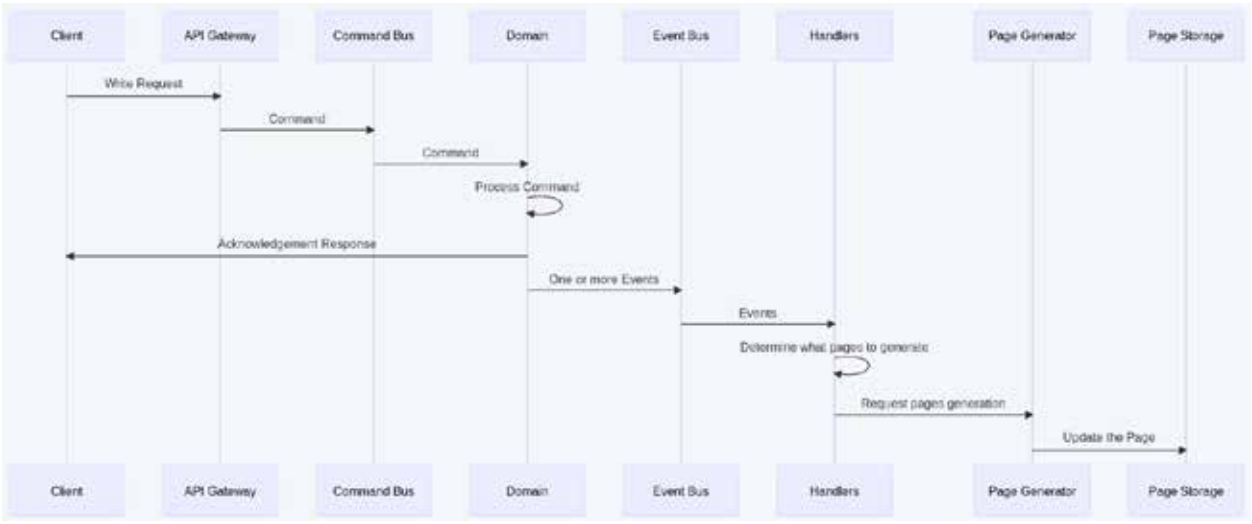

Fig. 5. Event source approach with Page Generator

This approach integrates particularly well with an Event Sourcing system (Chamberlain, 2017), as the implementation of sending events upon changing the state of an aggregate forms the foundation of such systems. The pre-rendered page essentially serves as a projection in this context.

In Fig. 5, the flow of state change in the event-sourced system is depicted. A change request transforms into an understandable message for Domain (command), which is validated and enters the domain through the command bus. The domain comprises command handlers, a repository, and an aggregate description. The command handler requests the repository for the aggregate based on the ID specified in the command. The repository, relying on the aggregate description, creates an empty object of the required aggregate type, queries the event store for all events

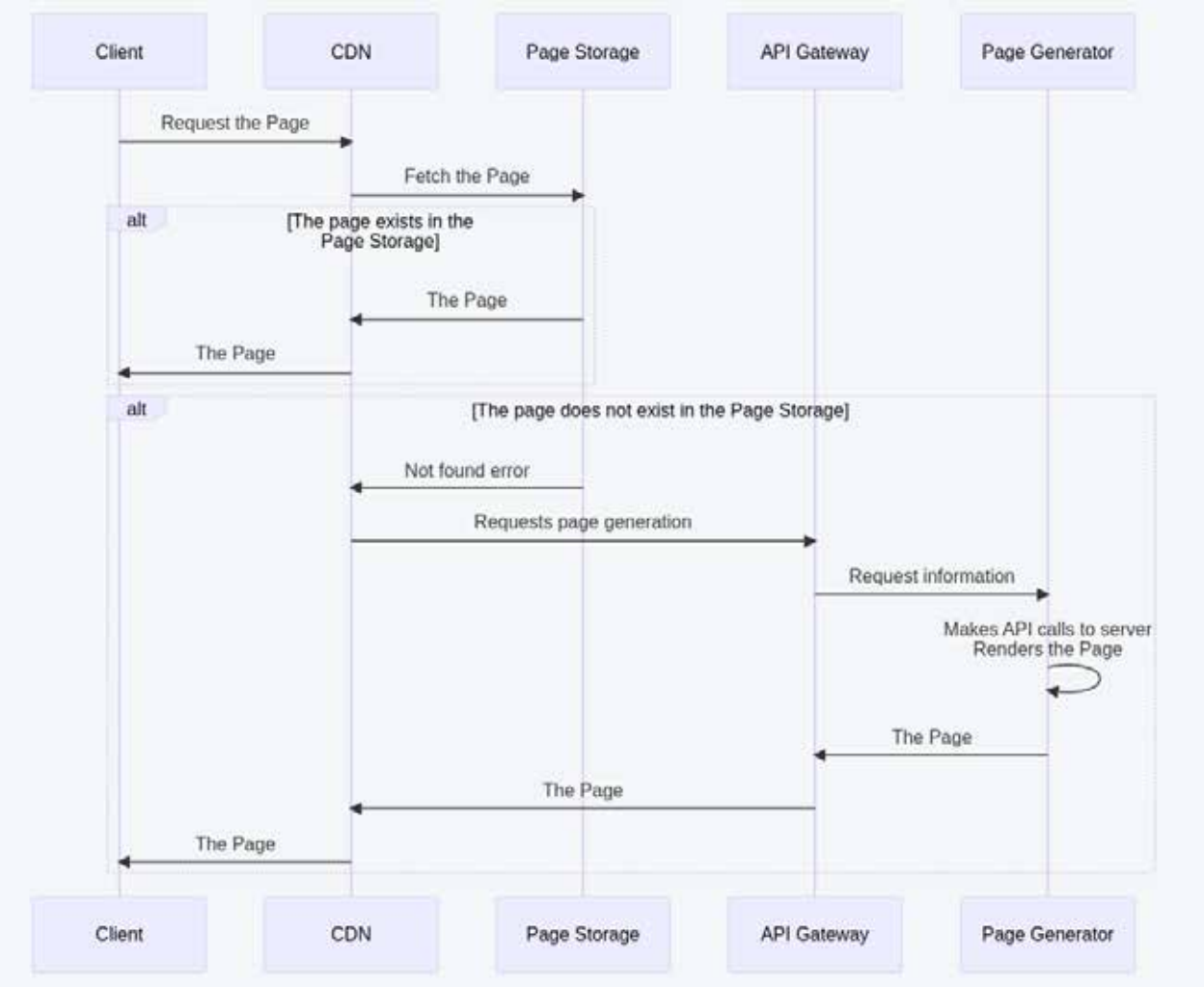

Fig. 6. Sequence diagram Request prerendered Page

related to the aggregate, replays them on the new object, and passes the resulting aggregate to the command handler. The command handler, using the aggregate's methods, modifies its state. Upon the state change of the aggregate, multiple events are generated. These events are sent to the repository, saved in the event store, and an acknowledgment response is sent to the client, indicating that the source of truth (event store) of the system has been successfully changed, and all other parts of the system will be eventually updated. Subsequently, the events are sent to the event bus, notifying numerous handlers about the system change. One such handler determines which pages need to be updated based on the dispatched events, triggers the page generator, and invalidates the cache when pages are updated in the Cache Proxy Server.

When a user's browser sends a request to retrieve a page, this request goes to the CDN, which in turn fetches data from the repository and serves the pre-rendered page to the client.

In case the requested page does not exist in the repository, the request is redirected to the generator, which renders the missing page, serves it to the client, and subsequently stores it in the storage (Fig. 6). Since the proposed architecture uses a strong caching model, such cases may occur only under normal system operation when requesting a new page that has not been rendered by the system since the last update.

The described system above has been implemented and deployed in a live project. Fig. 7 shows the component diagram of the implementation of this system based on AWS services and CQRS architecture.

This approach allows generating the necessary pages and invalidating the cache only in situations where it is justified by changes in the system. In scenarios where the system undergoes frequent changes, this approach can be resource-intensive. To enhance scalability, it is advisable to utilize an AWS Lambda function as a hosting environment for the renderer. This solution enables the paral-

Fig. 7. Component diagram

lel invocation of multiple renders when necessary, without incurring server maintenance costs during periods of frequent system changes.

The main concern with this solution could be the quantity and frequency of cache invalidation requests, as a large number of requests can become financially burdensome. This issue is addressed by an additional service that is invoked instead of direct invalidation. This service accumulates a predefined number of invalidation requests, aggregates them, removes duplicates, and invalidates all necessary pages with a single request. Such invalidation can be triggered either upon reaching a certain number of invalidation requests or when a specified timeout elapses if the request pool is not empty. This solution has also been implemented. A timeout of 1 hour and a pool size of 100 pages have been set. For the developed system, this results in approximately 200,000 CDN cache invalidations per month for a thousand pages under standard system load. Given the prices for CloudFront invalidation (34), this costs $1000 per month.

Methods. To assess the performance of the solutions in this study, the average read request time for pages with different amounts of content is measured. The measurements were conducted on a real system using an analytical algorithm that, upon each page load not from the browser cache, sent load duration statistics to the statistics server.

Experiment. The experiment was conducted based on the DBB Software company's (31) proprietary platform, which provided the necessary infrastructure and tools for data collection and analysis. This platform offered essential capabilities for our research, ensuring the accuracy and reliability of our experimental results. For a typical test system, as described above, the problem was solved using three different methods by two middle-level developers.

This approach significantly accelerates the delivery of up-to-date data to end users by invalidating the cache immediately after a system change rather than waiting for its expiration. In other words, pages of doctors who regularly receive new certificates or have new reviews from their patients will be updated regularly, while pages, where new data does not appear for months, will not undergo a rerender.

Another question is how the proposed modification affects performance compared to other approaches. The average page load time for a real website with varying amounts of content was measured using three considered approaches:

– SSR application using Next JS React framework, hosted on AWS Elastic Beanstalk (35) M2 instance.

– Cache Proxy Server with a weak caching model approach as Next JS Serverless realization, hosted on AWS Lambda function.

Table 1

Different update frequency issue solving

	SSR	Cache Proxy Server with a weak caching model	Suggested approach
Issue solved	Yes	No	Yes

Table 2

Average page load time

Page size	time, ms		
	SSR	Cache Proxy Server with a weak caching model	Suggested approach
9 Mb	1782	827	799
5 Mb	496	93	89
2 Mb	301	61	55

– Custom implementation of suggested approach on AWS Lambda function with a subscription to system events.

From Table 2, it can be seen that the proposed approach, and the Cache Proxy Server with a weak caching model, significantly reduce page load time compared to the pure SSR approach. This is logically explained by the fact that an SSR application renders the page every time if it is not in the CDN cache, and the CDN cache cannot store pages that quickly become outdated. The proposed approach is not inferior in speed to Cache Proxy Server with a weak caching model, even showing a slight performance improvement. This is because, using the weak cache model, the system requests page rendering upon user demand when the page's time-to-live is considered outdated. The proposed approach updates the data in the cache when the system state changes and assumes the presence of the page in the page storage. Thus, rendering upon user request occurs only when the data is not in the cache, which, under normal system operation, should not happen.

A drawback of the proposed approach is its considerable implementation complexity. The Cache Proxy Server with a weak caching model has a similarly high level of complexity, but Next.js Serverless provides a ready-made implementation that can be easily used for one's system. Similarly, the proposed approach offers a high degree of reusability. Once implemented for one system, it can be encapsulated into a separate module and reused for other systems with relatively small effort.

Summary. The page load time and timely delivery of up-to-date data significantly impact user experience and are critical indicators in assessing system performance from the end user's perspective. This article explored approaches to organizing the frontend part of the system and optimizing page load time for systems whose data state changes not too frequently but is not entirely static. Additionally, a modification to one of these methods was proposed, which does not lag in page load speed compared to existing approaches and speeds up the delivery of up-to-date data to end users. Examples of implementing the proposed approach for an event source and a standard DDD system were described. An experiment was also conducted, demonstrating that the proposed approach, for a specific type of system, performed no worse than existing ones.

BIBLIOGRAPHY:

1. Thakur N. Singh A. Sangal A.L. Cloud services selection: A systematic review and future research directions. Computer Science Review. Volume 46, 2022. Doi: 10.1016/j.cosrev.2022.100514.

2. Cloud computing services – Amazon Web Services. URL: https://aws.amazon.com.

3. Google Cloud console. URL: https://console.cloud.google.com.

4. Azure Cloud Services. URL: https://azure.microsoft.com/en-us/products/cloud-services.

5. Chockler G. Laden G. Vigfusson Y. Design and implementation of caching services in the cloud, 2011. Pages: 1 – 11. DOI:10.1147/JRD.2011.2171649.

6. Choi J. Gu Y. Kim J. Learning-based dynamic cache management in a cloud. Journal of Parallel and Distributed Computing. Volume 145, 2020. Pages: 98-110. DOI: 10.1016/j.jpdc.2020.06.013.

7. Berger D.S. Gland P. Singla S. Ciucu F. Exact analysis of TTL cache networks. Performance Evaluation. Volume 79, 2014. Pages 2-23. DOI: 10.1016/j.peva.2014.07.001.

8. Redis. URL: https://redis.io.

9. Fukuda E.S. Caching memcached at reconfigurable network interface. Conference: IEEE International Conference on Field Programmable Logic and Applications (FPL), 2014. DOI: 10.1109/FPL.2014.6927487.

10. Elasticache – AWS. URL: https://aws.amazon.com/elasticache.

11. Wu H. Fan Y. Wang Y. Ma H. Xing L. A Comprehensive Review on Edge Caching from the Perspective of Total Process: Placement, Policy and Delivery, 2021. Doi: 10.3390/s21155033.

12. Large D. What is a Static Site Generator? Cloud cannon, 2022. URL: https://cloudcannon.com/blog/what-is-a-static-site-generator/.

13. Yang K. An Introduction to Static Site Generators. Digital Ocean, 2022. URL: https://www.digitalocean.com/community/conceptual-articles/introduction-to-static-site-generators.

14. Jiang W.R. Yan J.H. Implementation of Static Web-Pages Generator Using JavaScript. Applied Mechanics and Materials, 2010. Pages. 588-591. DOI:10.4028/www.scientific.net/AMM.39.588.

15. Jekyll. Simple, blog-aware, static sites. URL: https://jekyllrb.com.

16. Eleventy. A simpler static site generator. URL: https://www.11ty.dev.

17. The Best React-Based Framework. Gatsby. URL: https://www.gatsbyjs.com.

18. Vepsäläinen J. Vuorimaa P. Bridging Static Site Generation with the Dynamic Web. In: Di Noia, T., Ko, IY., Schedl, M., Ardito, C. (eds) Web Engineering. ICWE 2022. Lecture Notes in Computer Science, vol 13362, 2022. DOI: 10.1007/978-3-031-09917-5_32. ISBN: 978-3-031-09916-8.

19. Fink G. Flatow I. Introducing Single Page Applications, 2014. DOI: 10.1007/978-1-4302-6674-7_1.

20. Gudivada V.N. Rao D. Paris J. Understanding Search Engine Optimization, 2015. Pages: 67-76. DOI: 10.1109/MC.2015.297.

21. Sun Y. Server-Side Rendering: Building Reliable, High-Performance Web Apps Using Elm-Inspired Architecture, Event Pub-Sub, and Components. Practical Application Development with AppRun, 2019. Pages: 191-217. DOI:10.1007/978-1-4842-4069-4_9.

22. Handlebars. URL: https://handlebarsjs.com.

23. Pug. URL: https://pugjs.org/api/getting-started.html.

24. Vilas J.F. Pazos-Arias J.J. Vilas A.F. Optimizing Web Services Performance Using Cache. Journal of Advanced Computational Intelligence and Intelligent Informatics, 2006. Pages: 713-720. DOI: 10.20965/jaciii.2006.p0713.

25. Wang J. A Scalable Efficient Robust Adaptive (SERA) Architecture for the Next Generation of Web Service, 1999.

26. Running Next.js applications with Serverless services on AWS. Serverlessland. URL: https://serverlessland.com/repos/nextjs-serverless-architecture.

27. Serverless Nextjs Plugin. URL: https://www.serverless.com/plugins/serverless-nextjs-plugin.

28. Pull-request with a description of how Next.js Serverless works inside. URL: https://github.com/serverless-nextjs/serverless-next.js/pull/1028.

29. Rajan A.P. Serverless Architecture – A Revolution in Cloud Computing, 2018. DOI: 10.1109/ICoAC44903.2018.8939081.

30. The React Framework for the Web. URL: https://nextjs.org.

31. DBB Software compare. Website URL: https://dbbsoftware.com/en.

32. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software, 2004. ISBN: 978-0321125217.

33. Chamberlain N. Applying Domain Driven Design with CQRS and Event Sourcing, 2017.

34. CloudFront – How AWS Pricing Works. URL: https://docs.aws.amazon.com/whitepapers/latest/how-aws-pricing-works/cloudfront.html.

35. Elasticbeanstalk – AWS. URL: https://aws.amazon.com/elasticbeanstalk.

REFERENCES:

1. Thakur, N., Singh, A., Sangal, A.L. (2022). Cloud services selection: A systematic review and future research directions. Computer Science Review. Volume 46. Doi: 10.1016/j.cosrev.2022.100514.

2. Cloud computing services – Amazon Web Services. Retrieved from https://aws.amazon.com.

3. Google Cloud console. Retrieved from https://console.cloud.google.com.

4. Azure Cloud Services. Retrieved from https://azure.microsoft.com/en-us/products/cloud-services.

5. Chockler, G., Laden, G., Vigfusson, Y. (2011). Design and implementation of caching services in the cloud. DOI:10.1147/JRD.2011.2171649.

6. Choi, J., Gu, Y., Kim, J. (2020). Learning-based dynamic cache management in a cloud. Journal of Parallel and Distributed Computing. Volume 145. DOI: 10.1016/j.jpdc.2020.06.013.

7. Berger, D.S., Gland, P., Singla, S., Ciucu, F. (2014). Exact analysis of TTL cache networks. Performance Evaluation. Volume 79. DOI: 10.1016/j.peva.2014.07.001.

8. Redis. Retrieved from https://redis.io.

9. Fukuda, E.S. (2014). Caching memcached at reconfigurable network interface. Conference: IEEE International Conference on Field Programmable Logic and Applications (FPL). DOI: 10.1109/FPL.2014.6927487.

10. Elasticache – AWS. Retrieved from https://aws.amazon.com/elasticache.

11. Wu, H., Fan, Y., Wang, Y., Ma, H., Xing, L. (2021). A Comprehensive Review on Edge Caching from the Perspective of Total Process: Placement, Policy and Delivery. Doi: 10.3390/s21155033.

12. Large, D. (2022). What is a Static Site Generator? Cloud cannon. Retrieved from https://cloudcannon.com/blog/what-is-a-static-site-generator/.

13. Yang, K. (2022). An Introduction to Static Site Generators. Digital Ocean. Retrieved from https://www.digitalocean.com/community/conceptual-articles/introduction-to-static-site-generators.

14. Jiang, W.R., Yan, J.H. (2010). Implementation of Static Web-Pages Generator Using JavaScript. Applied Mechanics and Materials. DOI: 10.4028/www.scientific.net/AMM.39.588.

15. Jekyll. Simple, blog-aware, static sites. Retrieved from https://jekyllrb.com.

16. Eleventy. A simpler static site generator. Retrieved from https://www.11ty.dev.

17. The Best React-Based Framework. Gatsby. Retrieved from https://www.gatsbyjs.com.

18. Vepsäläinen, J., Vuorimaa, P. (2022). Bridging Static Site Generation with the Dynamic Web. In: Di Noia, T., Ko, IY., Schedl, M., Ardito, C. (eds) Web Engineering. ICWE 2022. Lecture Notes in Computer Science, vol 13362. DOI: 10.1007/978-3-031-09917-5_32. ISBN: 978-3-031-09916-8.

19. Fink, G., Flatow, I. (2014). Introducing Single Page Applications. DOI: 10.1007/978-1-4302-6674-7_1.

20. Gudivada, V.N., Rao, D., Paris, J. (2015). Understanding Search Engine Optimization. DOI: 10.1109/MC.2015.297.

21. Sun, Y. (2019). Server-Side Rendering: Building Reliable, High-Performance Web Apps Using Elm-Inspired Architecture, Event Pub-Sub, and Components. Practical Application Development with AppRun. DOI:10.1007/978-1-4842-4069-4_9.

22. Handlebars. Retrieved from https://handlebarsjs.com.

23. Pug. Retrieved from https://pugjs.org/api/getting-started.html.

24. Vilas, J.F., Pazos-Arias, J.J., Vilas, A.F. (2006). Optimizing Web Services Performance Using Cache. Journal of Advanced Computational Intelligence and Intelligent Informatics. DOI: 10.20965/jaciii.2006.p0713.

25. Wang, J. (1999). A Scalable Efficient Robust Adaptive (SERA) Architecture for the Next Generation of Web Service.

26. Running Next.js applications with Serverless services on AWS. Serverlessland. Retrieved from https://serverlessland.com/repos/nextjs-serverless-architecture.

27. Serverless Nextjs Plugin. Retrieved from https://www.serverless.com/plugins/serverless-nextjs-plugin.

28. Pull-request with a description of how Next.js Serverless works inside. Retrieved from https://github.com/serverless-nextjs/serverless-next.js/pull/1028.

29. Rajan, A.P. (2018). Serverless Architecture – A Revolution in Cloud Computing. DOI: 10.1109/ICoAC44903.2018.8939081.

30. The React Framework for the Web. Retrieved from https://nextjs.org.

31. DBB Software compare. Website Retrieved from https://dbbsoftware.com/en.

32. Evans, E. (2004). Domain-Driven Design: Tackling Complexity in the Heart of Software. ISBN: 978-0321125217.

33. Chamberlain, N. (2017). Applying Domain Driven Design with CQRS and Event Sourcing.

34. CloudFront – How AWS Pricing Works. Retrieved from https://docs.aws.amazon.com/whitepapers/latest/how-aws-pricing-works/cloudfront.html.

35. Elasticbeanstalk – AWS. Retrieved from https://aws.amazon.com/ elasticbeanstalk.